Analyses de sûreté de fonctionnement multi-systèmes

Romain Bernard

N° d'ordre: 3885

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

Par Romain Bernard

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : Informatique

Analyses de sûreté de fonctionnement multi-systèmes

Soutenue le : 23 novembre 2009
Après avis de : Hubert Garavel .... Rapporteurs
Antoine Rauzy ....

Devant la Commission d'examen formée de :

Pierre Bieber	Directeur de thèse, ONERA	Examinateurs
Jean-Philippe Domenger	Professeur, Université Bordeaux 1	
Alain Griffault	Maître de Conférence, Université Bordeaux 1	
Marc Zeitoun	Directeur de thèse, Université Bordeaux 1 ...	
Sylvain Metge	Ingénieur, Airbus	
François Pouzolz	Ingénieur, Airbus	
Hubert Garavel	Directeur de Recherche, INRIA	Rapporteur
Antoine Rauzy	Chargé de Recherche, CNRS	Rapporteur

2009

Analyses de sûreté de fonctionnement multi-systèmes

Résumé : Cette thèse se situe au croisement de deux domaines : la sûreté de fonctionnement des systèmes critiques et les méthodes formelles. Nous cherchons à établir la cohérence des analyses de sûreté de fonctionnement réalisées à l'aide de modèles représentant un même système à des niveaux de détail différents. Pour cela, nous proposons une notion de raffinement dans le cadre de la conception de modèles AltaRica : un modèle détaillé raffine un modèle abstrait si le modèle abstrait simule le modèle détaillé. La vérification du raffinement de modèles AltaRica est supportée par l'outil de model-checking MecV. Ceci permet de réaliser des analyses multi-systèmes à l'aide de modèles à des niveaux de détail hétérogènes : le système au centre de l'étude est détaillé tandis que les systèmes en interface sont abstraits. Cette approche a été appliquée à l'étude d'un système de contrôle de gouverne de direction d'un avion connecté à un système de génération et distribution électrique.

Mots clés : AltaRica, méthodes formelles, raffinement, conception/validation d'architecture, sûreté de fonctionnement des systèmes

Multi-system safety analyses

Abstract : This thesis links two fields : system safety analyses and formal methods. We aim at checking the consistensy of safety analyses based on formal models that represent a system at different levels of detail. To reach this objective, we introduce a refinement notion in the AltaRica modelling process : a detailed model refines an abstract model if the abstract model simulates the detailed model. The AltaRica model refinement verification is supported by the MecV model-checker. This allows to perform multi-system safety analyses using models with heterogeneous levels of detail : the main system is detailed whereas the interfaced systems remain abstract. This approach has been applied to the analysis of a rudder control system linked to an electrical power generation and distribution system.

Key words : AltaRica, formal methods, refinement, system design/validation, safety engineering

Équipe Méthodes formelles - Modélisation et Vérification
Laboratoire Bordelais de Recherche en Informatique
351 cours de la Libération F-33405 Talence cedex

REMERCIEMENTS

Je remercie Jean-Philippe Domenger d'avoir accepté de présider le jury, ainsi que mes rapporteurs Hubert Garavel et Antoine Rauzy (un merci particulier pour le soutien lors de ma présentation à Lambda-Mu).

Un grand merci à Pierre Bieber qui a su gérer ma *pugnacité*. Cette thèse doit beaucoup à Pierre et sa connaissance des mondes académiques et industriels. Merci d'avoir toujours trouvé du temps pour répondre à mes questions et de m'avoir soutenu, tout particulièrement durant les derniers mois. Merci d'avoir compris mon attirance pour le monde industriel et d'avoir composé avec pour mener à bien ces trois années. Tes qualités humaines et ton sens de l'humour ne peuvent transparaître dans ce mémoire mais ont compté pour moi dans les moments difficiles.

Je tiens à remercier Marc Zeitoun d'avoir accepté de diriger ma thèse. Tes qualités pédagogiques m'ont permis de comprendre des éléments théoriques qui auparavant pouvaient me poser problème. Ton sens du détail et de la vulgarisation scientifique ont contribué à rendre lisible le cœur théorique de mes travaux. Merci pour les entretiens à proximité du tableau blanc qui m'ont permis de comprendre ou de réfléchir à ces relations qui étaient souvent très abstraites à mes yeux.

Je tiens à remercier Alain Griffault pour avoir initié le master 2 Ingénierie des Systèmes Critiques qui m'a permis d'entrer en contact avec la société Airbus et de m'avoir toujours soutenu. Je me souviens t'avoir entendu dire que, à l'issue de ce master, nous serions des VRP responsables de promouvoir les méthodes formelles dans l'industrie et j'espère avoir contribué à cet objectif. Un grand merci pour les discussions dans ton bureau qui m'ont permis de mieux comprendre le monde de la recherche ou de parfaire ma connaissance d'AltaRica.

Je tiens à remercier Sylvain Metge de m'avoir donné l'opportunité d'effectuer mon stage de master puis ma thèse au sein de l'entreprise Airbus. Je garde en mémoire la rigueur avec laquelle tu as lu mon mémoire de master et que je cherche à appliquer dans mon travail depuis. Nos discussions sur le championnat de football ont apporté des moments de détente au bureau d'autant plus agréables que le titre bordelais approchait.

Je remercie François Pouzolz d'avoir pris la succession de Sylvain pour m'encadrer d'un point de vue industriel. Merci de m'avoir apporté ta connaissance de l'industrie pour décrypter les mécanismes de cette grande société qu'est Airbus. Merci pour toutes nos discussions sur des sujets variés au bureau et en dehors (sur le retour vers notre jolie région natale).

Je tiens à remercier Charles Zamora de m'avoir accueilli chez Airbus au sein de son équipe, pour effectuer mes premiers pas dans l'industrie. Je me suis très vite senti bien intégré dans une belle équipe aux accents français, italien, espagnol, allemand et bien entendu écossais. Je n'oublierai en particulier jamais le célèbre "I've a joke for you" de l'ami Chris BBHH Lacey, la bonne humeur d'Alessandro "esseptionnel" Landi et les anecdotes de Michel Tchorowski. Une pensée pour Delphine Johan, la plus pressée des membres de l'équipe mais avec qui j'ai apprécié de faire des pauses-chocolat. J'ai passé quatre belles années et je remercie tous ceux qui m'ont consacré une partie de leur temps.

Je remercie les membres du DTIM à l'ONERA et de l'équipe MF au LaBRI pour leur accueil. Plus particulièrement, je remercie Gérald (pour sa connaissance d'AltaRica et pour avoir bien voulu me fournir

ses sources LaTeX qui ont servi de modèle à la rédaction de ce mémoire) et Aymeric (pour son aide concernant MecV et sa réactivité à prendre en compte certains souhaits).

Un grand merci à ma famille qui m'a toujours soutenu malgré les difficultés diverses de la vie. Une pensée particulière pour ma mère (qui a stressé probablement plus que moi), pour mes sœurs Faustine et Manon, pour ma grand-mère Maïté (qui a suivi tous les travaux sans vraiment oser me dire quand elle ne comprenait pas) et mon grand-père Paul (qui pensait me voir un jour pilote mais qui, j'espère, serait fier de me voir docteur autant que je le suis de ses réalisations), pour ma grand-mère Odette (qui a trouvé la plus jolie description d'un docteur en informatique : celui qui soigne les ordinateurs), pour ma tante Nadine et mon oncle Michel d'être venus assister à ma soutenance, et pour mon père (qui m'a transmis l'amour des avions).

Un grand merci à mes amis Sylvie, Clément, Manu, Léo, Pib et Cec, Audrey et David, Steph, Rabah, Dine et ceux que j'oublie pour m'avoir soutenu et changé les idées afin de prolonger les hauts et de tenir le coup durant les bas.

Un merci argentin à mes amis Maria et Eduardo. Merci pour votre gentillesse et pour tous les souvenirs lors de ma dernière visite dans votre beau pays.

Je remercie les badistes du TOAC (Sylvie et Arno, Lili, Lolo, Guigui et Dédé ...) pour leur accueil et ces bons moments passés en particulier sur les tournois. Un merci particulier pour Elo : cette année ensemble m'a permis de grandir, de découvrir Toulouse et mon intégration au club n'aurait sûrement pas été la même sans toi.

Enfin un merci particulier pour mon cœur, ma Caro. Merci pour ton soutien durant toute la rédaction jusqu'à la soutenance.

TABLE DES MATIÈRES

INTRODUCTION

Depuis le début du XX^{ième} siècle, le transport aérien ne cesse de se développer. Tout d'abord réservé au transport de marchandises et de courrier, l'avion s'est depuis démocratisé permettant à chacun de voyager à travers le monde. Outre le nombre de vols, la capacité des avions a augmenté. Ces facteurs expliquent que le nombre de passagers transportés par avion augmente chaque année (+8,5% en 2005, +5% en 2006 et +7% en 2007). Cette évolution a deux conséquences :

- l'augmentation du nombre d'avions dans le ciel accroît les risques de collision en vol ;
- la capacité croissante des avions commerciaux implique un accroissement du nombre de victimes possibles en cas d'accident.

De plus, tout avion qui s'écrase peut causer des pertes humaines tant à bord (les passagers et membres d'équipage) qu'au sol en cas d'accident en zone habitée. Tous ces risques font que la sécurité est une préoccupation majeure de l'industrie aéronautique. Les grands constructeurs se sont entendus pour exclure la sécurité des domaines de concurrence et collaborent à l'amélioration des méthodes de travail dans ce domaine.

Afin d'établir et de garantir un niveau de sécurité acceptable, des autorités de *certification* définissent des exigences que chaque avion doit satisfaire pour obtenir une autorisation de voler. Chaque constructeur doit démontrer que l'avion fabriqué est conforme aux exigences des autorités en charge du pays de production, puis aux exigences des autorités du pays d'immatriculation.

Le souci constant d'amélioration de la sécurité a permis de réduire le nombre d'accidents (cf. figure 0.1) et a ainsi contribué à faire de l'avion un des moyens de transport les plus sûrs.

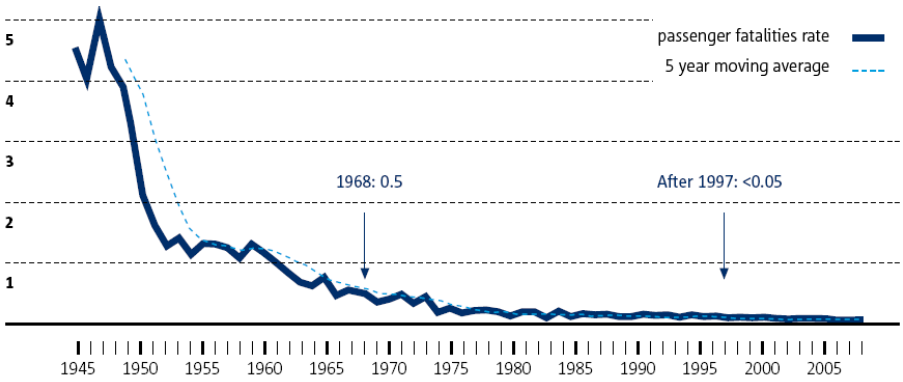


FIG. *0.1: Passagers décédés par an pour 100 millions de miles passager, sur des vols commerciaux, hors actes de malveillance intentionnelle (source : EASA Annual Safety Review 2008)*

Schématiquement, un avion est composé d'une structure, appelée *fuselage*, et d'équipements, dits *systèmes embarqués*, permettant de contrôler l'avion. Les principales causes d'accidents sont l'erreur humaine (d'un pilote ou du contrôleur aérien), les défaillances liées à la structure et les défaillances de systèmes embarqués. Nous nous intéressons à ces dernières qui font l'objet de différentes analyses.

Toute entreprise cherche constamment à optimiser ses coûts de conception pour améliorer les rendements. Dans le cas de la production d'avions, cela revient à :

- réduire les masses et coûts des systèmes embarqués afin d'augmenter la charge utile et de réduire les coûts de fabrication ;
- améliorer les processus pour réduire les temps de conception et, par conséquent, les délais de livraison aux clients.

Afin de limiter les équipements embarqués, certains systèmes, autrefois indépendants, peuvent aujourd'hui partager des ressources. Par exemple, le transfert des données peut désormais, sur les avions A380 et B787, être assuré par un réseau embarqué partagé par un grand nombre de systèmes. L'étude permanente de nouvelles solutions techniques justifie le fait que l'avion est un moyen de transport à la pointe de la technologie. Néanmoins, les systèmes, s'ils peuvent désormais remplir plusieurs fonctions, s'avèrent de plus en plus complexes et donc difficiles à analyser par les méthodes traditionnelles. Analyser des systèmes plus complexes en un délai plus court contraint les avionneurs à étudier de nouvelles méthodes pour supporter les analyses de sûreté de fonctionnement des systèmes.

Les modèles formels comme réponse au besoin d'évolution

L'utilisation des *modèles formels* dans l'industrie représente une évolution majeures des méthodes de travail. Un *modèle* est une représentation abstraite d'un objet : seules les informations pertinentes pour l'étude à mener sont prises en compte dans la spécification de l'objet.

Le développement de l'informatique a vu la création de *langages formels*, chacun défini par :

- une syntaxe : un lexique et une grammaire (règles d'écriture) permettent de décrire certains aspects (propres à chaque langage) d'un système de façon compacte et compréhensible par tout connaisseur du langage ;
- une sémantique : interprétation de la description faite par l'ordinateur.

Un modèle formel est une traduction de la spécification d'un objet dans un langage formel. Les outils informatiques associés au langage formel, permettent d'étudier certaines caractéristiques d'un objet sans avoir à le fabriquer. Ainsi, par exemple, un système peut être simulé pour en étudier son comportement.

Les modèles formels présentent de nombreux avantages pour l'industrie. Lorsque plusieurs interlocuteurs partagent des informations, formaliser ces dernières permet de prévenir toute interprétation pouvant conduire à une mauvaise compréhension. De plus, les outils de modélisation tirent profit de la puissance de calcul des ordinateurs pour traiter de grandes quantités de données, là où un ingénieur devrait choisir les calculs à effectuer.

Cependant, l'utilisation de modèles modifie l'organisation du temps de travail : la sélection des informations à modéliser et la validation du modèle constituent la part prépondérante du travail contrairement à la production de résultats qui, étant en partie automatisée, s'effectue plus rapidement que par les méthodes classiques. Dès lors, les modèles formels ne présentent un intérêt, pour motiver une évolution des pratiques, que s'ils garantissent un gain :

- de temps, tout en préservant la qualité des résultats ;
- de précision des résultats, sans allonger le temps de travail.

A ce jour, dans l'aéronautique, les modèles s'avèrent utiles pour étudier l'aérodynamique du fuselage ou l'installation des systèmes embarqués au sein de l'avion et sont intégrés au processus de développement de certains logiciels embarqués. Dans le domaine des analyses de sûreté de fonctionnement des systèmes, la société Dassault Aviation a, en 2005, démontré que les commandes de vol de son Falcon 7X étaient conformes aux exigences des autorités à l'aide de modèles réalisés en langage AltaRica. Ce langage, développé avec le Laboratoire Bordelais de Recherche en Informatique (LaBRI) pour supporter les analyses de sûreté de fonctionnement des systèmes, fait l'objet d'expérimentations internes par la société Airbus depuis 4 ans.

Des contraintes à résoudre pour une industrialisation de l'approche

La conception d'un système suit un processus incrémental : une spécification abstraite est progressivement enrichie jusqu'à obtenir un niveau de détail suffisant pour procéder à la fabrication des composants du

système et à leur assemblage. Afin de réduire les temps de conception, il est nécessaire de détecter au plus tôt tout détail contraire aux exigences des autorités de certification. En matière de sûreté de fonctionnement des systèmes, il faut donc analyser chaque nouvelle spécification. Or un modèle ne représente qu'une spécification. Procéder à une analyse de sûreté de fonctionnement des systèmes à l'aide de modèles implique de réaliser de nombreux modèles.

Valider un modèle constitue une des opérations les plus coûteuses du travail de modélisation. Durant la conception, une spécification est enrichie de façon à préserver les propriétés vérifiées jusqu'alors. Le modèle d'une spécification doit donc rester cohérent avec le modèle de la spécification qui le précède.

Une analyse multi-systèmes traite généralement un système, dit "central", en considérant les effets des défaillances des systèmes auxquels il est connecté, aussi appelés "systèmes en interface". Afin de préserver une certaine lisibilité et, dans le cas des analyses fondées sur les modèles, pour prévenir les risques d'explosion combinatoire, les différents systèmes ne sont pas tous décrits au même niveau de détail :

- le système central est détaillé ;
- les systèmes en interface sont décrits de façon plus abstraite.

Une modélisation multi-système se compose donc de modèles de niveaux de détail hétérogènes. En outre, chaque système en interface fait l'objet, comme tout système, d'une étude mono-système et nécessite donc un modèle détaillé.

Durant la conception d'un avion et pour les besoins des différentes analyses, chaque système dispose donc de plusieurs modèles. Il est donc primordial de s'assurer que ces différents modèles sont cohérents.

Objectifs de la thèse

Le développement du logiciel suit un processus comparable à la conception des systèmes embarqués. Confrontés aux mêmes difficultés de devoir valider chaque nouvelle spécification, des chercheurs ont défini la notion de *raffinement* : chaque nouvelle spécification présente un lien avec la précédente qui garantit la cohérence et la préservation d'exigences. Il nous a semblé pertinent de chercher à appliquer la notion de raffinement pour construire les modèles de systèmes embarqués.

Cette notion n'est définie ni pour la réalisation de modèles de sûreté de fonctionnement des systèmes, ni pour le langage AltaRica. Les objectifs de cette thèse sont donc :

- d'un point de vue théorique :
  - l'étude de relations de raffinement (simulation et bisimulation fortes et faibles),
  - la définition d'une notion de raffinement pour le langage AltaRica permettant de garantir la préservation des résultats d'analyse,
- d'un point de vue pratique :
  - l'implémentation des relations de raffinement en langage MecV (premier vérificateur de modèle capable de traiter nativement des modèles AltaRica) afin d'automatiser la vérification,
  - l'adaptation du raffinement AltaRica aux besoins des analyses de sûreté de fonctionnement des systèmes,
  - l'élaboration d'une méthodologie permettant de vérifier ce raffinement.

Organisation du document

Ce document est découpé en cinq chapitres : les trois premiers présentent les domaines concernés par cette thèse, puis les deux suivants détaillent nos travaux, tant théoriques que pratiques.

Chapitre 1 : La sécurité des systèmes aéronautiques

Ce chapitre présente le contexte industriel dans lequel se sont effectués nos travaux. Afin de bien comprendre comment le processus d'analyse de la sûreté de fonctionnement des systèmes d'un avion est élaboré, nous présentons la sécurité dans le domaine aéronautique mondial en décrivant ses acteurs, les liens qui existent entre eux et en introduisant les principaux textes réglementaires. Nous détaillons ensuite les principales analyses qui constituent le processus ainsi que les formalismes usuels.

Chapitre 2 : Le langage AltaRica et ses outils

Ce chapitre introduit le langage formel AltaRica, créé pour analyser la sûreté de fonctionnement des systèmes et qui est au cœur de nos travaux. Nous présentons les motivations qui ont conduit à sa création ainsi que les étapes majeures de son développement puis nous détaillons sa syntaxe concrète, i.e. les connaissances nécessaires et suffisantes pour pouvoir créer des modèles. Il existe en réalité deux langages AltaRica : le premier est plutôt utilisé dans le domaine académique, alors que le second est quasi-exclusivement employé dans l'industrie. Les deux langages ayant leur place dans nos travaux, nous les décrivons parallèlement puis nous présentons les outils qui leur sont associés. Nous terminons ce chapitre par une comparaison avec d'autres formalismes dédiés à la sûreté de fonctionnement.

Chapitre 3 : Les analyses de sécurité fondées sur les modèles

Ce chapitre présente deux types de modèles permettant de réaliser des analyses de sécurité : les modèles dédiés dits "modèles de propagation de défaillance" et les modèles enrichis dits "modèles étendus". Le premier type de modèle étant celui qui nous intéresse plus particulièrement, nous exposons en détail la méthodologie de modélisation d'une description du système de contrôle de la gouverne de direction. Nous terminons ce chapitre par des exemples d'approches d'un troisième type, couplant différents formalismes.

Chapitre 4 : Le raffinement pour AltaRica

Ce chapitre constitue le cœur théorique de cette thèse. Nous rappelons tout d'abord la syntaxe abstraite et la sémantique d'AltaRica. Nous présentons ensuite la relation de bisimulation, étudiée par Gérald Point dans sa thèse [Poi00] et pour laquelle il a défini un théorème de compositionnalité. La suite de ce chapitre détaille les travaux que nous avons menés sur les relations de simulation et de simulation quasi-branchante pour élaborer un théorème semblable. Chaque relation est implémentée dans le langage de spécification MecV.

Chapitre 5 : Utilisation du raffinement pour le développement de systèmes sûrs

Ce chapitre présente la démarche méthodologique visant à tirer profit du raffinement AltaRica pour faciliter la réalisation de modèles à différents niveaux de détail. Nous détaillons l'utilisation des relations décrites dans le chapitre précédent afin de garantir la préservation de propriétés globales ou restreintes, en suivant une approche orientée vers la vérification ou l'exploration des modèles. Nous illustrons nos apports sur différents exemples (calculateur des commandes de vol puis système de génération et distribution électrique) avant de comparer notre démarche à d'autres approches de raffinement pour la sûreté de fonctionnement des systèmes.

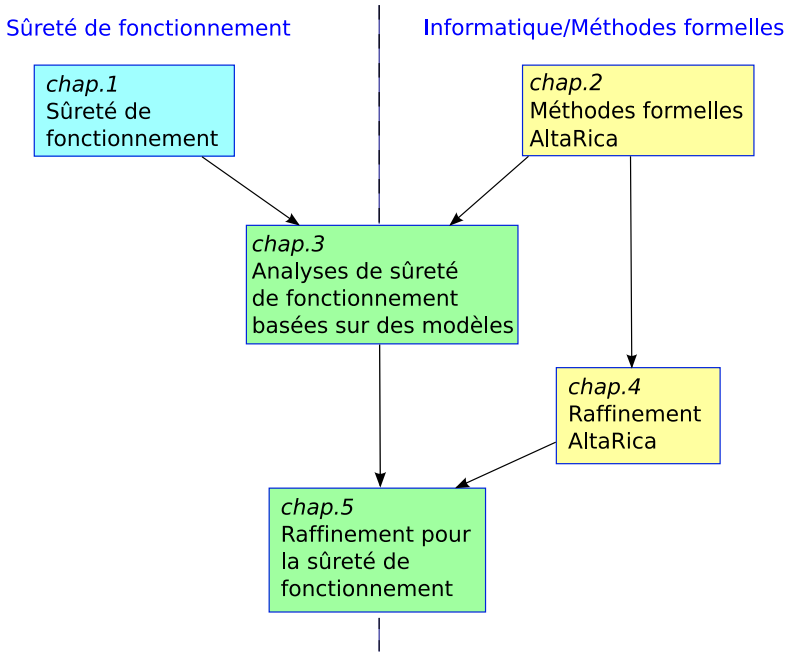


FIG. *0.2: Organisation du document*

1

LA SÉCURITÉ DES SYSTÈMES AÉRONAUTIQUES

L'avion est considéré comme l'un des moyens de transport les plus sûrs. Cette réputation perdure car la sécurité est une préoccupation majeure du monde aérien :

- tout avion qui décolle d'un aéroport répond à des exigences nationales, voire internationales (suivant son périmètre d'exploitation), garantissant l'intégrité des personnes, tant à bord qu'autour de l'appareil ;
- le contrôle aérien applique des procédures précises pour guider chaque avion afin qu'il effectue un vol en évitant toute collision.

Tout règlement impose des exigences. La démonstration de la tenue de ces exigences permet en particulier d'obtenir l'autorisation de faire décoller et voler un aéronef de forte capacité à but commercial : on parle communément de *certification*.

La société Airbus, en tant qu'avionneur européen, se doit de certifier, auprès des autorités européennes, chaque avion pour que la compagnie aérienne cliente puisse en prendre possession et l'exploite sur ses lignes. La société Airbus est concernée par toutes les réglementations liées aux aéronefs, durant chaque phase du cycle de vie d'un avion :

- la conception : avant d'être produit et assemblé à grande échelle, chaque nouvel appareil doit obtenir une *certification de type* ;
- la production :
  - chaque avion produit est réalisé spécialement pour une compagnie cliente ; des écarts par rapport aux spécifications certifiées durant la conception sont possibles mais ils doivent également répondre aux exigences des autorités européennes,
  - si la compagnie cliente réside en dehors d'Europe, il est également nécessaire de fournir des garanties aux autorités locales,
  - chaque avion obtient donc une *certification individuelle*,
- l'exploitation : chaque avion est suivi durant toute sa "vie" afin qu'il reste conforme à son état de certification :
  - chaque incident est signalé puis analysé pour démontrer la préservation de la conformité,
  - toute intervention (maintenance, réparation, modification) est enregistrée et contrôlée.

Durant la conception, en vue d'obtenir la certification de type, des analyses sont menées afin de chercher tout ce qui pourrait conduire l'avion à être dans une situation dégradée pouvant éventuellement porter atteinte à l'équipage, aux passagers ou à toute personne dans le périmètre de l'appareil : on parle d'*analyses de sûreté de fonctionnement des systèmes*.

Ce chapitre a pour but d'introduire les analyses de sûreté de fonctionnement des systèmes en présentant tout d'abord les différentes instances (autorités, groupes de travail, constructeurs) liées à la sécurité aérienne, ainsi que les différents textes dont elles ont la charge. La seconde partie de ce chapitre se focalise sur le processus actuel d'analyse de sûreté de fonctionnement des systèmes chez Airbus.

1.1 LA SÛRETÉ DE FONCTIONNEMENT DES SYSTÈMES DANS LE MONDE AÉRIEN

La sûreté de fonctionnement des systèmes constitue une partie de la sécurité d'un avion : seuls les systèmes et les fonctions qu'ils remplissent sont analysés ; la structure de l'avion et l'intégrité des données informatiques à bord sont traitées séparément.

Les acteurs de la sûreté de fonctionnement des systèmes aériens peuvent être classés en trois catégories :

- les autorités de certification définissent des règles et s'assurent de leur application ;
- les avionneurs, ainsi que les équipementiers et les motoristes, doivent appliquer ces règles et démontrer qu'elles sont correctement appliquées ;
- des groupes de travail, constitués à la fois de représentants des avionneurs et des autorités, s'entendent sur des pratiques permettant de satisfaire les règles et de répondre aux exigences.

Chaque catégorie d'acteurs élabore ses propres documents décrivant les pratiques permettant de satisfaire les règles.

Avant de présenter le processus actuel d'analyse de sûreté de fonctionnement des systèmes, cherchons à comprendre son élaboration en nous intéressant à chaque catégorie d'acteur et aux documents majeurs.

1.1.1 AUTORITÉS

La sécurité aérienne est assurée par une répartition précise des rôles en matière de réglementation : des règles sont définies au niveau mondial puis contrôlées par des autorités géographiquement liées aux avionneurs en vue d'attribuer les autorisations de vol.

L'Organisation de l'Aviation Civile Internationale (OACI), agence spécialisée des Nations Unies, établit les règles de l'air, pour l'immatriculation des aéronefs et la sécurité, ainsi que les droits et devoirs des pays signataires en matière de droit aérien relatif au transport international. L'OACI fut instaurée lors de la Convention relative à l'Aviation Internationale Civile, connue sous le nom de *Convention de Chicago*. Initialement signée le 7 décembre 1944 par 52 pays, elle compte à ce jour 190 nations signataires. Chaque pays signataire participe à l'élaboration des annexes, contenant les normes et recommandations pour le transport aérien international, et s'engage à les considérer dans ses lois nationales.

De nombreuses organisations de certification et de réglementation aéronautique civile existent dans le monde, néanmoins les deux principales agences sont :

- l'Agence Européenne de la Sécurité Aérienne (AESA) ou *European Aviation Safety Agency (EASA)* en anglais, créée en 2003 et succédant aux *Joint Aviation Authorities (JAA)*, association d'organisations européennes de réglementation aéronautique ;
- la *Federal Aviation Agency (FAA)* aux États Unis, créée en 1958.

L'EASA et la FAA définissent des exigences concernant, entre autres domaines : la conception des avions, leur exploitation commerciale ou les licences de pilotage. Chaque agence rédige ses propres documents mais les exigences étant proches, il est possible d'établir des correspondances entre documents européens et américains. Les principales exigences applicables à la sûreté de fonctionnement des systèmes sont définies dans le paragraphe numéroté 1309 à la fois dans le document baptisé *CS-25* [EAS08] *(Certification Specification)* de l'EASA pour l'Europe, et dans le document baptisé *FAR-25* [FAA03] *(Federal Aviation Regulation)* de la FAA pour les États-Unis. Les paragraphes décrivant des exigences sont généralement complétés d'annexes décrivant les moyens acceptés pour répondre à ces exigences, en anglais *Acceptable Means of Compliance (AMC)*.

1.1.2 GROUPES DE TRAVAIL

Afin d'améliorer constamment le niveau de sécurité des passagers, les principaux acteurs du monde aéronautique (avionneurs, motoristes, équipementiers) collaborent au sein de deux principaux groupes de travail :

- le groupe WG-63 *(Complex Aircraft Systems)* de la *European Organisation of Civil Aviation Equipment (EUROCAE)*, historiquement présidé par un représentant d'Airbus ;
- le groupe S-18 *(Aircraft and System Development and Safety Assessment Committee)* de la *Society of Automotive Engineers (SAE)*, historiquement présidé par un représentant de Boeing.

Ces groupes élaborent des recommandations, applicables aux processus, méthodes et outils, pour répondre aux exigences des autorités précédemment citées. La présence de représentants des autorités garantit la bonne interprétation des réglementations.

Le groupe S-18 travaille principalement à l'élaboration de deux documents dits *Aerospace Recommended Practice (ARP)* :

- l'ARP 4754 [SAE96b] *(Certification considerations for highly-integrated or complex aircraft systems)* fournit des recommandations pour démontrer que des systèmes complexes (i.e. dont le niveau de sécurité ne peut pas être démontré par test ou dont la compréhension du fonctionnement nécessite le support d'outils) ou hautement intégrés (i.e. qui réalise ou contribue à plusieurs fonctions) satisfont les exigences de navigabilité ;
- l'ARP 4761 [SAE96a] *(Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment)* propose des méthodes possibles pour évaluer la sûreté de fonctionnement des systèmes d'un avion en vue de sa certification (les différentes étapes sont présentées dans la suite de ce chapitre).

Le groupe WG-63 élabore des documents équivalents baptisés respectivement ED-79 et ED-135. Des réunions communes entre les deux groupes permettent une évolution cohérente des documents et une meilleure réflexion.

1.1.3 AVIONNEURS

Les avionneurs doivent se conformer aux exigences des autorités et, pour y répondre, disposent des recommandations SAE/EUROCAE afin d'établir leurs propres processus. Si quelques représentants participent aux groupes de travail, tous les ingénieurs contribuant aux analyses de sûreté de fonctionnement des systèmes doivent connaître les règles et méthodes à appliquer, ce qui nécessite l'élaboration de documents propres à chaque avionneur.

Concernant la sûreté de fonctionnement des systèmes et la fiabilité, la société Airbus élabore deux documents :

- l'*Airbus Directive (ABD)* 100 module 1.3, destiné aux équipementiers et fournisseurs, précisant les exigences applicables aux équipements réalisés, les informations à fournir et les études à conduire ;
- l'ABD 200 module 1.3, à usage interne, détaillant les différentes analyses et le processus à respecter.

Afin d'anticiper des exigences toujours plus nombreuses de la part des autorités, les avionneurs imposent généralement des contraintes supplémentaires de façon à prendre des marges de sécurité.

1.2 PROCESSUS ACTUEL

Conformément aux ARP 4754 [SAE96b] et 4761 [SAE96a], les avionneurs réalisent différentes analyses pour démontrer la tenue des exigences imposées par les réglementations internationales en matière de navigabilité aérienne. Le processus global se décompose selon les grandes étapes suivantes :

1. identifier les fonctions souhaitées ;
2. rechercher les conditions défaillantes liées à chaque fonction pour déterminer les possibles configurations de panne, en anglais *Failure Condition (FC)* ;
3. définir une criticité pour chaque FC en fonction de ses effets ;
4. déduire des exigences (principes de conception et probabilité d'occurrence maximum tolérée) pour prévenir chaque FC ou maintenir leurs conséquences à un niveau acceptable ;
5. démontrer que l'étude est complète et que les résultats sont en accord avec les réglementations en vigueur : la probabilité d'occurrence de chaque FC est inférieure ou égale à l'objectif déduit de la criticité.

Une fonction correspond à un service (abstrait) que doit rendre un objet physique ou un ensemble d'objets physiques (concrets). Les programmes aéronautiques considèrent la décomposition physique suivante :

- un *avion* est constitué d'une structure, appelée *fuselage*, de *moteurs* et de *systèmes* ;
- un système est composé d'*équipements*.

Plusieurs systèmes peuvent fonctionner conjointement afin de remplir une fonction, ces systèmes sont alors dits *interfacés* ou *en interface*. Par exemple, le système de génération électrique est interfacé à de nombreux systèmes tels que les commandes de vol du fait que les calculateurs de vol ne peuvent fonctionner sans énergie électrique.

Cette décomposition physique est transposée en décomposition fonctionnelle hiérarchique : les fonctions de niveau supérieur sont reliées à un ensemble de fonctions de niveaux inférieurs. La relation entre niveaux signifie que les fonctions de niveau inférieur contribuent à la réalisation de la fonction de niveau supérieur. Cette décomposition est utile pour guider les analyses de sûreté de fonctionnement des systèmes et allouer des exigences aux concepteurs des fonctions. Les trois principaux niveaux de décomposition utilisés dans les programmes aéronautiques sont :

- le niveau "avion", le plus haut dans la hiérarchie, qui regroupe les fonctionnalités principales de l'avion comme "commander l'avion", "naviguer" ;
- le niveau "système", qui regroupe des fonctions au sein de systèmes tels que le système des commandes de vol ;
- le niveau "équipement", dernier niveau dans la hiérarchie, qui regroupe des fonctions réalisées par un même équipement.

Les analyses à conduire concernent initialement l'avion dans son ensemble, puis les systèmes embarqués (considérés individuellement, liés fonctionnellement ou liés par leur placement à bord) et enfin les équipements qui composent un système. En nous référant aux ARP et ABD, nous avons résumé ci-après le processus actuel (cf. figure 1.1) que nous décomposons en trois catégories d'analyses :

- évaluation des risques fonctionnels, en anglais *Functional Hazard Assessment (FHA)* : étude des fonctions pour déterminer les défaillances fonctionnelles possibles et classifier les risques associés à des configurations de panne spécifiques ;
- évaluation de la sécurité des systèmes, en anglais *System Safety Assessment (SSA)* : évaluation de la conformité de l'architecture d'un système avec les exigences de sécurité déduites des FHA ;
- analyses des causes communes, en anglais *Common Cause Analysis (CCA)* : études complémentaires des risques liés à des causes communes à plusieurs systèmes.

Remarque

La figure 1.1 représente un canevas de processus d'analyse de sûreté de fonctionnement des systèmes. Chaque industriel est libre de s'en inspirer ou non pour mettre en place son propre processus d'analyse de sûreté de fonctionnement des systèmes.

1.2.1 FUNCTIONAL HAZARD ASSESSMENT (FHA)

Chaque nouvel avion est conçu pour une utilisation spécifique : transport de passagers, acheminement de marchandises, utilisation militaire. Chaque utilisation ajoute des fonctions spécifiques aux fonctions communes à tout avion (décoller, se mouvoir dans le ciel...). Or toute fonction peut être perdue ou remplie de façon incorrecte (partiellement, de façon intempestive, en retard...). L'examen systématique des fonctions permet de recenser toutes les Failure Conditions FC possibles. Pour chaque fonction et chaque défaillance fonctionnelle, une recherche systématique des scénarios opérationnels est réalisée. Ensuite, l'étude des répercussions de chaque scénario (sur l'avion, son équipage, ses occupants et les personnes évoluant autour de l'avion) permet d'établir, en se référant au tableau 1.1, sa criticité. Les scénarios sont ensuite regroupés par effet équivalent en FC. Cela permet d'associer à chaque FC :

- un objectif quantitatif : une probabilité par heure de vol à atteindre ;
- un niveau de confiance du développement, en anglais *Design Assurance Level (DAL)* : chaque catégorie de DAL impose des règles de conception tant sur les architectures physiques que sur les techniques d'implémentation logicielle.

La liste des fonctions, des FC et leur criticité constituent l'évaluation des risques fonctionnels (FHA).

1.2. PROCESSUS ACTUEL

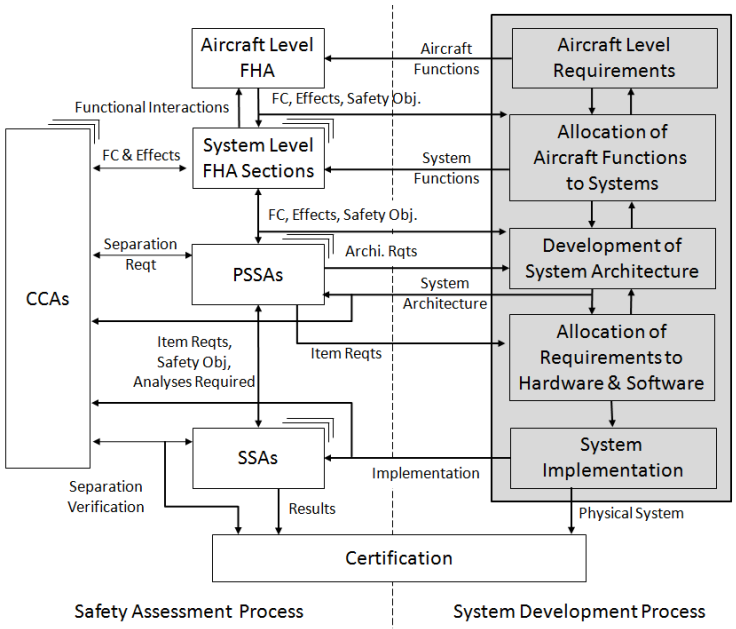


FIG. *1.1: Processus d'évaluation de la sécurité (extrait de l'ARP 4754)*

Criticité	Probabilité (par heure de vol)	Niveau de confiance du développement	Effets
Mineur *(Minor, MIN)*	N/A	D	- légère réduction des marges de sécurité - légère augmentation de la charge de travail équipage - dérangements pour les occupants
Majeur *(Major, MAJ)*	10^{-5}	C	- réduction significative des marges de sécurité ou des fonctionnalités - augmentation significative de la charge de travail équipage ou conditions dégradant l'efficacité de l'équipage - gênes pour les occupants
Dangereux *(Hazardous, HAZ)*	10^{-7}	B	- réduction importante des marges de sécurité ou des fonctionnalités - charge de travail plus élevée ou détresse physique telle que l'équipage n'est pas en mesure de réaliser des tâches précisément ou complètement
Catastrophique *(Catastrophic, CAT)*	10^{-9}	A	toute condition de panne qui empêche une poursuite du vol et un atterrissage sûrs

TAB. *1.1: Criticité des configurations de panne*

Outre les exigences quantitatives, des exigences qualitatives sont associées à chaque FC en fonction de sa criticité. Ces exigences sont de la forme “la FC de criticité C ne peut se produire en moins de N défaillances”. Le tableau 1.2 présente la relation entre le nombre N de défaillances nécessaires et la criticité C.

Classification	Catastrophic	Hazardous	Major	Minor	No Safety Effect
Qualitative requirement	3	2	2	1	0

TAB. *1.2: Criticité et exigences qualitatives associées*

Durant la conception d’un avion, une première FHA de niveau avion, en anglais *Aircraft level FHA(A/C FHA)*, considère l’avion dans sa globalité puis différentes FHA sont réalisées pour traiter chaque système embarqué individuellement, en anglais *System FHA*.

1.2.1.1 Aircraft Level FHA

A partir de la liste des fonctions attendues pour un avion et des choix de conception initiaux (nombre de moteurs...), une évaluation qualitative de haut niveau recense les risques potentiels liés aux fonctions basiques de l’avion.

Une fonction est réalisée par un système ou par une combinaison de systèmes. La A/C FHA doit donc permettre d’identifier des FC impliquant plusieurs systèmes à la fois : la criticité déduite des effets des défaillances combinées peut être supérieure à celle qui serait définie pour chaque système étudié individuellement.

Les exigences de sécurité, qualitatives (règles de ségrégation, DAL...) et quantitatives (objectif de probabilité) déduites de la A/C FHA sont allouées aux systèmes contribuant aux différentes fonctions et doivent être prises en compte durant leur conception.

1.2.1.2 System FHA

La FHA système est également une analyse qualitative et itérative. Elle s’appuie sur les résultats de la A/C FHA et traite les fonctions du système concerné en étudiant les conséquences d’une défaillance ou d’une combinaison de défaillances tant sur le système que sur l’avion. Lorsque les fonctions avion ont été affectées à des systèmes par le processus de conception, chaque système fait l’objet d’une FHA système.

1.2.2 SYSTEM SAFETY ASSESSMENT (SSA)

Les concepteurs de systèmes doivent élaborer une architecture pour réaliser les fonctions mentionnées précédemment. Compte tenu des FHA déjà réalisées, il est nécessaire d’évaluer chaque architecture pour s’assurer qu’elle répond aux exigences définies au préalable. Ces évaluations sont menées dans la continuité des FHA en suivant un processus itératif : chaque évaluation peut détecter une correction à effectuer qui nécessite une nouvelle évaluation.

Bien que l’évaluation d’une architecture s’intéresse principalement à un système, la forte interconnexion des systèmes dans un avion implique de considérer les systèmes en interface.

Dans un premier temps, des évaluations dites “préliminaires”, en anglais *Preliminary System Safety Assessment (PSSA)*, sont réalisées. Une PSSA permet, tout d’abord, de compléter la liste des FC et donc éventuellement d’ajouter de nouvelles exigences de sécurité. D’autre part, une PSSA permet de déterminer une liste préliminaire des actions pilotes à accomplir lorsqu’une FC se produit et des opérations de maintenance à effectuer. La PSSA vérifie également que l’architecture du système est cohérente avec les modes de défaillance envisagés. Pour cela, une recherche des combinaisons minimales de défaillances, appelées *coupes minimales*, est effectuée. Par souci de concision et de lisibilité, les coupes minimales sont présentées sous la forme d’un arbre de défaillances ou d’un diagramme de dépendance (cf. 1.2.4.1) par FC de criticité MAJ, HAZ ou CAT. La probabilité d’occurrence d’une FC est calculée à partir de ces coupes minimales,

de la probabilité individuelle de chaque défaillance ainsi que de paramètres complémentaires (temps de vol moyen, intervalle de temps entre 2 opérations de maintenance...).

Enfin, cette évaluation permet d'allouer au niveau équipement les exigences de sécurité du niveau système. Ces exigences sont soumises aux fournisseurs d'équipements.

Après plusieurs PSSA successives, les résultats des études menées par les fournisseurs sont intégrés aux résultats des PSSA afin de vérifier que l'architecture sélectionnée est conforme aux exigences qualitatives et quantitatives définies dans les FHA et PSSA.

1.2.3 CCA (PRA, ZSA, CMA)

Les exigences de sécurité peuvent imposer l'indépendance de plusieurs fonctions, systèmes ou équipements. Cette indépendance peut être dissemblance fonctionnelle (technologies différentes pour deux composants ayant même utilité, implémentation différentes pour deux logiciels...) ou physique (i.e. installation des composants redondants dans des zones différentes de l'avion). Le paragraphe 1309 du CS 25 insiste sur le besoin d'étudier les défaillances de cause commune (CCA).

Les CCA identifient les modes de défaillance individuels et les événements extérieurs potentiellement responsables de FC MAJ, HAZ ou CAT :

- dans le cas des FC CAT, de tels événements ne sont pas tolérés ;
- dans le cas des FC MAJ ou HAZ, la probabilité d'occurrence de tels événements doit respecter l'objectif de probabilité correspondant à la criticité de la FC.

Les analyses de causes communes sont constituées de trois types d'analyses : l'analyse des modes communs, en anglais *Common Mode Analysis (CMA)*, l'analyse des risques particuliers, en anglais *Particular Risk Analysis (PRA)* et l'analyse de la sécurité par zone, en anglais *Zonal Safety Analysis (ZSA)*.

1.2.3.1 CMA

L'analyse des modes communs fournit la preuve que des défaillances supposées indépendantes le sont réellement. Cette analyse étudie les effets d'erreurs de conception, de fabrication ou de maintenance et de défaillances de composants communs. Des composants constitués des mêmes pièces ou disposant du même logiciel peuvent subir la même défaillance et donc impacter tout le système malgré leur indépendance physique.

Cette analyse peut être effectuée sur les arbres de défaillances ou les diagrammes de dépendance en vérifiant que des événements liés à une même porte logique "ET" sont réellement indépendants : deux défaillances reliées par la porte "ET" sous-entendent la présence d'un principe de redondance, cependant, des erreurs d'implémentation, de fabrication ou de maintenance peuvent invalider ce principe.

1.2.3.2 ZSA

Une analyse de sécurité par zone étudie chaque zone physique de l'avion pour s'assurer que l'installation des équipements, les éventuelles interférences physiques avec les systèmes adjacents et les possibles erreurs de maintenance ne violent pas les exigences d'indépendance du système. Toute détection d'un risque potentiel pour la sécurité implique, sauf démonstration que ce risque reste acceptable, de concevoir une nouvelle architecture d'un système ou de revoir l'installation des équipements de ce système dans l'avion.

1.2.3.3 PRA

Une analyse des risques particuliers étudie qu'un événement extérieur au système concerné n'a pas d'influence sur les exigences d'indépendance. Cette analyse s'intéresse aux effets simultanés d'un risque ainsi qu'aux effets résultants. Il existe différents risques à prendre en compte parmi lesquels :

- le feu ;
- la glace, la grêle, la neige ;
- les fuites de liquides ;

– le péril aviaire ;
– la foudre ;
– l'éclatement pneu et moteur.

Les risques particuliers étudiés peuvent avoir un impact sur plusieurs zones alors que le périmètre d'étude d'une analyse de type ZSA est restreint à une zone.

1.2.4 TECHNIQUES, MÉTHODES

Afin d'étudier les pannes simples ou combinaisons de défaillances conduisant à chaque FC identifiée dans la FHA d'un système, l'ARP 4761 propose trois techniques d'analyse descendante :

– l'arbre de défaillances, en anglais *Fault Tree (FT)* ;
– le diagramme de dépendance, en anglais *Dependence Diagram (DD)* ;
– l'analyse markovienne, en anglais *Markov Analysis (MA)*.

Les outils basés sur ces techniques facilitent le calcul de la probabilité d'occurrence de la FC étudiée. Cela permet durant les évaluations préliminaires de la sécurité d'un système (i.e. la phase PSSA) d'allouer des objectifs de taux de défaillance ainsi que des DAL à des composants physiques, puis, en intégrant les taux réels fournis par les équipementiers, de vérifier la tenue des objectifs qualitatifs et quantitatifs (i.e. la phase SSA). Un objectif non tenu peut impliquer une modification d'architecture et donc, généralement, la mise à jour des FT/DD/MA concernés.

Afin d'illustrer les techniques suivantes, considérons le système de génération et de distribution hydraulique, illustré par la figure 1.2, composé d'un réservoir alimentant deux pompes similaires et d'un système de distribution fournissant de la puissance hydraulique tant qu'une pompe fonctionne.

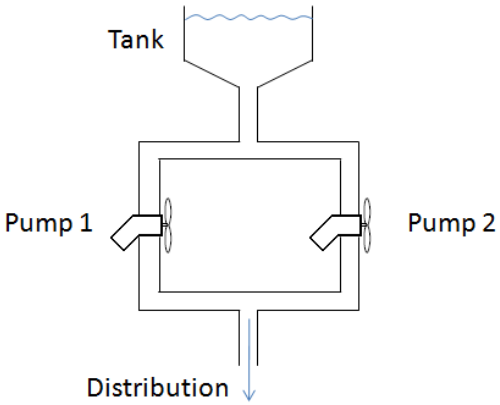


FIG. *1.2: Système de distribution hydraulique*

Remarque

Dans la suite de ce mémoire, les expressions *arbre de défaillances* et *graphe de Markov* désignent respectivement un *arbre de défaillances statique* et un *graphe de Markov à temps discret*. Ces derniers sont majoritairement utilisés dans l'industrie aéronautique.

1.2.4.1 Arbres de défaillances et diagrammes de dépendance

Les arbres de défaillances et les diagrammes de dépendance, présentées notamment dans [Vil88], facilitent la lecture et l'analyse en ne mettant en évidence que les défaillances qui contribuent à la FC étudiée et en hiérarchisant la présentation des défaillances : des événements de haut niveau peuvent être décomposés en combinaisons d'événements de niveaux inférieurs.

Un arbre de défaillances est une représentation hiérarchique de la formule logique décrivant les combinaisons de défaillances pouvant conduire un système dans l'état redouté étudié. Essentiellement, les portes logiques "et" et "ou" sont utilisées dans un arbre de défaillances :

- si deux défaillances sont liées par une porte “et”, alors elles doivent se produire simultanément pour que la situation redoutée soit atteinte (une seule défaillance ne représente qu’une dégradation sans effet tant que la seconde défaillance ne s’est pas produite) ;
- si deux défaillances sont liées par une porte “ou”, il suffit qu’une défaillance se produise pour que la situation redoutée soit atteinte.

Un arbre de défaillances est composé de défaillances élémentaires et d’états défaillants résultant de la combinaison de défaillances. La création d’un arbre commence par la sélection de l’événement cible, dit de haut niveau, à étudier. L’analyste élabore ensuite chaque niveau de l’arbre en développant les causes de chaque état défaillant.

La réalisation d’un arbre de défaillances nécessite une bonne connaissance du comportement du système. De plus, pour pouvoir effectuer des calculs sur un arbre, la structure de celui-ci doit être telle que toute défaillance primaire n’apparaît qu’une fois afin de garantir une réelle indépendance des causes potentielles de deux états défaillants distincts.

La figure 1.3 représente l’arbre de défaillances de l’analyse de la perte de distribution hydraulique.

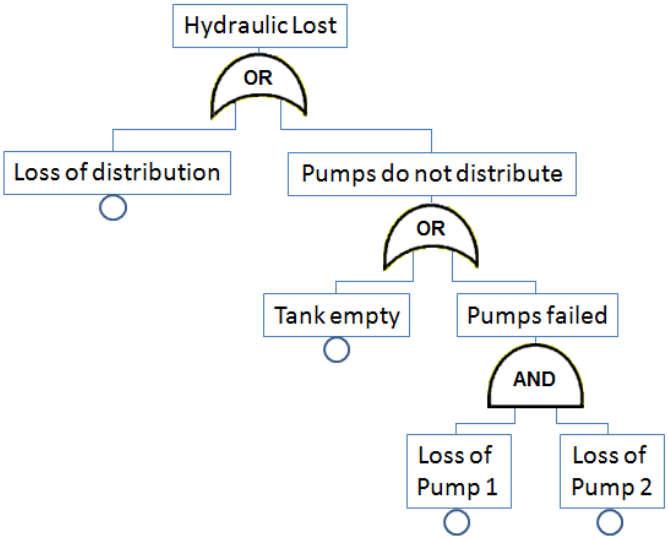


FIG. *1.3: Arbre de défaillances*

Le formalisme des diagrammes de dépendance est une alternative aux arbres de défaillances. Un diagramme de dépendance substitue les portes logiques par des chemins pour montrer les relations entre défaillances : des chemins parallèles sont équivalents à une porte “et”, tandis que des chemins en série sont équivalents à une porte “ou”. Tout comme un arbre de défaillances, un diagramme de dépendance est composé de défaillances conduisant à l’événement redouté et peut contenir d’autres événements redoutés, analysés séparément à l’aide d’un diagramme de dépendance indépendant, ou des événements extérieurs à l’avion tels que des conditions givrantes.

La figure 1.4 est la transposition sous forme de diagramme de dépendance de l’arbre de défaillances 1.3.

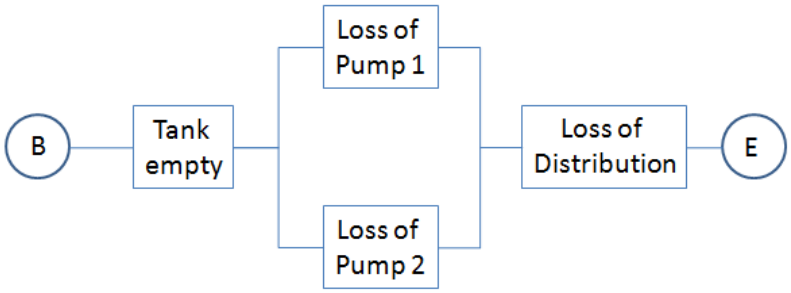


FIG. *1.4: Diagramme de dépendance*

Un arbre de défaillances (ou un diagramme de dépendance) est une représentation graphique d'une formule logique. De cette formule f sous-jacente sont extraits les implicants premiers [DR97], produits logiques de taille minimale impliquant la formule f. Les implicants premiers peuvent contenir des littéraux positifs (qui représentent la survenue de pannes) et des littéraux négatifs (qui représentent leur non-survenue). Les analyses de sécurité s'intéressant uniquement à la survenue de pannes, le concept de *coupe minimale* [Rau01] a été introduit : une coupe minimale est une partie positive minimale d'une affectation satisfaisant f. En d'autres termes, les coupes minimales correspondent aux plus petites combinaisons de défaillances conduisant à l'événement redouté étudié. De plus, la taille d'une coupe minimale est baptisée *ordre*.

La perte du système de distribution hydraulique présente trois coupes minimales :

- la fuite du réservoir ;
- la perte du système de distribution ;
- la perte des deux pompes.

Dans son mémoire de thèse [Tho02], Philippe Thomas présente comment, connaissant la probabilité d'occurrence de chaque défaillance d'un système ainsi que les coupes minimales liées à un événement redouté, il est possible de calculer la probabilité de chaque coupe : la probabilité d'une coupe minimale est le produit des probabilités des défaillances qui la constituent. Les défaillances étant des événements indépendants, la probabilité d'occurrence de l'événement redouté peut être calculée [1] par sommation des probabilités des différentes coupes minimales. Cette approche présente l'inconvénient de nécessiter la génération des coupes minimales, opération pouvant s'avérer coûteuse au delà d'un certain ordre. En pratique, seules les coupes d'ordre réduit (4 voire 5) sont considérées, la probabilité d'occurrence n'étant alors qu'une approximation.

Certains systèmes, de par leur taille ou leur complexité, ne peuvent être traités par des méthodes analytiques. Une technique, appelée *simulation de Monte-Carlo*, permet alors de réaliser des estimations avec peu de moyens informatiques. Le principe est de définir un temps de mission puis de créer aléatoirement des scénarios composés de défaillances et éventuellement de réparations, si ces dernières sont considérées dans le modèle, tels que la somme des durées des différents événements est inférieure ou égale à la durée de la mission. Ces scénarios sont ensuite simulés et l'état résultant du système est observé. Soit N le nombre total de scénarios simulés et n le nombre de scénarios conduisant dans l'état redouté, le rapport n/N fournit, lorsque N tend vers l'infini, une estimation de l'indisponibilité du système.

Enfin une troisième approche [Rau93], développée par Antoine Rauzy, calcule la probabilité d'occurrence d'un événement redouté directement sur une arbre de défaillances encodé sous la forme d'un diagramme de décision binaire (en anglais *Binary Decision Diagram, BDD*), sans générer préalablement les coupes minimales.

[1] Les algorithmes de recherche des coupes minimales et de calcul de la probabilité d'occurrence d'un événement redouté sont implémentés dans des outils industriels de traitement d'arbre tels qu'Aralia [DR97] afin d'automatiser une partie de l'analyse.

1.2.4.2 Graphes de Markov

Un graphe de Markov est une représentation du comportement d'un système via un automate : un état du modèle représente un état du système, opérationnel ou non, alors qu'un arc entre deux états représente une défaillance ou un événement fonctionnel (ex : reconfiguration après détection d'une panne). La structure d'un graphe de Markov permet d'écrire la probabilité d'être dans chaque état sous la forme d'une équation en fonction des états origine (resp. destination) des arcs entrant (resp. sortant).

Ce formalisme présente l'avantage de pouvoir inclure des changements de phase de vol ou représenter naturellement une large gamme de comportements tels que les événements dépendant de l'état courant et des événements passés. Cependant, la complexité et la taille d'un graphe de Markov augmente très rapidement lorsque le nombre de composants augmente, entraînant des problèmes d'explosion combinatoire.

Le système de distribution hydraulique 1.2 contient 4 composants. En ne considérant comme seuls événements que les défaillances visibles sur le diagramme de dépendance 1.4, il n'existe que 4 événements possibles. Malgré le nombre restreint de composant et d'événements à considérer, le graphe de Markov correspondant à ce système est composé de 16 états et 32 arcs. Cherchant à étudier la perte de distribution de puissance hydraulique, il s'avère qu'une, voire deux défaillances suffisent à atteindre l'état redouté, les défaillances suivantes n'ayant pas d'effet visible.

Par souci de lisibilité, les états de perte (rouge) ont été considérés comme des états absorbants sur le graphe de Markov, illustré sur la figure 1.5, décrivant le comportement du système depuis l'état nominal (vert) jusqu'aux états de perte accessibles suite à une défaillance ou via un état de défaillance partielle (violet).

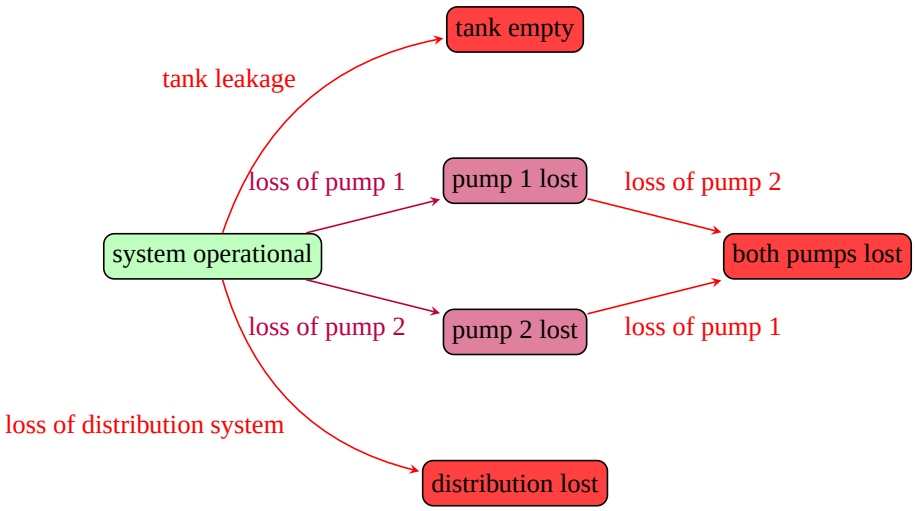


FIG. *1.5: Graphe de Markov du système de distribution hydraulique*

Un analyste expérimenté aurait tendance à réduire ce graphe en représentant seulement trois états : le système opérationnel, le système ayant perdu une pompe et enfin le système perdu (cf. graphe 1.6).

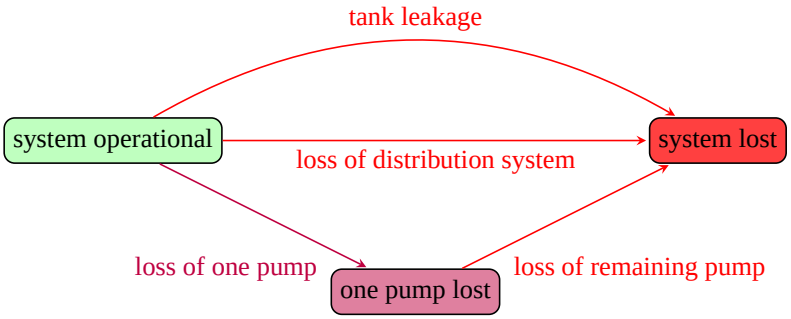


FIG. *1.6: Graphe de Markov optimisé du système de distribution hydraulique*

1.2.4.3 Constat

Les arbres de défaillances et les diagrammes de dépendance sont très répandus dans l'aéronautique car ils sont particulièrement lisibles à la fois pour les ingénieurs et pour les autorités de certification.

Néanmoins, ils présentent un certain nombre de limitations :

- dans le cas d'un système très complexe, la taille d'un arbre de défaillances peut être particulièrement grande : au delà d'une certaines profondeur, l'arbre est alors difficilement lisible ;
- afin de garantir une analyse indépendante, l'analyse de sécurité est menée par un ingénieur spécialisé qui réalise le FT/DD puis chaque FT/DD doit être validé par le concepteur afin de s'assurer que le fonctionnement du système pris en compte par l'ingénieur spécialisé est correct : chaque FC nécessitant un FT/DD, la validation s'avère être une phase coûteuse du processus ;
- la représentation du comportement d'un système est statique : la chronologie des défaillances dans une coupe minimale n'est pas prise en compte ;
- toute modification de l'architecture d'un système nécessite la vérification et éventuellement la mise à jour de tous ses FT/DD.

Les graphes de Markov nous semblent moins répandus dans l'aéronautique. Cela s'explique probablement par le fait que réaliser un graphe de Markov manuellement n'est réaliste que pour de très petits systèmes. La taille d'un tel graphe augmente très vite. Néanmoins, pour prévenir les risques d'explosion combinatoire, des méthodes de simulation des graphes de Markov ont été élaborées.

2

LE LANGAGE ALTARICA ET SES OUTILS

2.1 INTRODUCTION

2.1.1 MOTIVATIONS DU PROJET

AltaRica est un langage formel né de la volonté d'industriels et de chercheurs d'établir divers ponts entre :

- les méthodes formelles et la sûreté de fonctionnement ;
- les analyses quantitatives des dysfonctionnements et les analyses qualitatives des comportements fonctionnels ;
- les différents outils et méthodes d'aide à la modélisation.

Le but du projet est de fournir aux concepteurs un environnement de travail pour réaliser des analyses qualitatives ou quantitatives de systèmes complexes et critiques.

2.1.2 APERÇU DU LANGAGE, CHOIX INITIAUX

AltaRica fut initialement créé pour décrire des systèmes à événements discrets représentables par des Système de Transitions (ST) composés :

- d'états représentant les états globaux du système modélisé ;
- de transitions étiquetées représentant les changements d'état.

Dans le cas des systèmes à événements discrets, le temps s'écoule dans les états tandis que les changements d'état sont instantanés.

Dans le cas de systèmes critiques réels, la taille du ST pouvant être très importante, il n'était pas envisageable de le décrire en AltaRica comme un unique ST. Les responsables académiques du projet choisirent dès le début de généraliser le produit synchronisé de ST, défini par André Arnold et Maurice Nivat [AN82] et présent dans l'outil Mec4 [Cru89]. Un système est donc décrit de manière hiérarchique comme une arborescence d'objets appelés *nœuds*. On distingue deux types de nœuds :

- les nœuds feuilles qui décrivent un ST à l'aide de variables qui représentent les configurations, et d'événements pour passer d'une configuration à une autre ;
- les autres nœuds qui décrivent également un ST et qui contrôlent les interactions entre leurs nœuds fils par par synchronisation d'événements ou partage de variable (on parle alors de *variables de flux*, par opposition aux *variables d'état* qui décrivent l'état interne d'un nœud).

Un autre choix pour éviter la redondance de ST à décrire fût d'adopter le principe d'*instanciation* : un objet décrit une seule fois peut être dupliqué/instancié. Toutes les instances ont le même comportement théorique mais chacune évolue indépendamment des autres.

2.1.3 HISTORIQUE

De novembre 1996 à octobre 1997, une première phase visant à étudier les possibilités de connexion de différents outils a permis de définir un langage de description de systèmes complexes. Des expérimentations sur différents cas d'études ont permis de valider ce langage. Cette première phase comprenait la compagnie

d'ingénierie IXI, la société pétrolière Elf Aquitaine ainsi que deux laboratoires de Bordeaux : le LaBRI et le Laboratoire d'Analyse des Dysfonctionnements de Système (LADS).

De novembre 1997 à octobre 1999, une seconde phase a permis de formaliser la sémantique du langage AltaRica [GLP+98, GLP+99, PR99a, PR99b, AGPR99a], de définir une méthodologie de modélisation [AGPR99b] et de réaliser des prototypes d'outil restreignant AltaRica aux domaines finis : un simulateur basé sur la sémantique définie, un compilateur vers les outils Aralia (moteur de calcul d'arbres de fautes) et Mec (vérificateur de modèles). Aux participants initiaux se sont ajoutés Dassault Aviation, Schneider Electric, IPSN (Institut de Protection et de Sûreté Nucléaire) et Thomson Detexis.

Depuis ces premières étapes de la création du langage AltaRica d'origine, que nous baptiserons AltaRica LaBRI, deux principales variantes, dites "à flot de données" contrairement au langage "par contraintes" d'origine (cf. section suivante présentant les différences), ont été créées :

- AltaRica Data-Flow [Rau02], supporté par les outils Combava (anciennement Toolbox) d'Arboost Technologies et Simfia développé par EADS APSYS ;
- AltaRica OCAS (Outil de Conception et d'Analyse Système), supporté par l'outil Cecilia OCAS développé par Dassault Aviation.

Le langage AltaRica OCAS, qui peut être vu lui-même comme une variante d'AltaRica Data-Flow, est développé pour répondre au mieux aux besoins industriels de Dassault Aviation en matière d'analyse de sûreté de fonctionnement. L'outil développé par Dassault apporte une interface graphique permettant une prise en main rapide du langage et ajoute une représentation graphique au comportement textuel en langage AltaRica. Ces apports ont permis d'étendre le périmètre des utilisateurs d'AltaRica :

- l'Office National d'Études et de Recherches Aérospatiales (ONERA) et Airbus collaborent depuis les projets de recherche européens EnHance Safety Assessment for Complex Systems (ESACS) [ABB+03] et Improvement of Safety Activities on Aeronautical Complex systems (ISAAC) [ABB+06] ;
- la société Thalès, après évaluation, a rejoint le projet européen More Integrated and cost efficient System Safety Assessment (MISSA) ;
- d'autres sociétés (automobiles, pétrolières...) procèdent à des évaluations.

Si la société Dassault Aviation a réalisé un modèle AltaRica dans un cadre industriel pour certifier le système des commandes de vol du Falcon 7X, la plupart des sociétés précédemment citées réalisent, pour le moment, des expérimentations dans un cadre de recherche et développement afin, par exemple, d'intégrer les modélisations AltaRica dans les processus classiques d'analyse de sûreté de fonctionnement.

Le langage AltaRica a depuis 1996 donné lieu aux thèses suivantes :

- Gérald Point, dans [Poi00], a formalisé la sémantique d'AltaRica LaBRI ;
- Aymeric Vincent, dans [Vin03], a conçu l'outil de vérification de modèle MecV ;
- Claire Pagetti, dans [Pag04], a formalisé une extension temporisée à AltaRica LaBRI ;
- Christophe Kehren, dans [Keh05], a défini des motifs d'architecture (construits en AltaRica OCAS) pour aider à la conception de systèmes sûrs ;
- Sophie Humbert, dans [Hum08], a défini une méthodologie d'aide à la définition d'exigences de sécurité basée sur la complémentarité d'AltaRica, pour une modélisation globale d'un système, et Scade (outil basé sur le langage formel Lustre), pour une modélisation détaillée d'un logiciel embarqué ;
- Laurent Sagaspe, dans [Sag08], a défini une méthodologie et conçu un outil d'aide à l'allocation (fonctionnelle et géométrique) dans le cadre des systèmes aéronautiques.

A ce jour, les thèses suivantes sont en cours :

- Hayssam Soueidan définit le langage BioRica, extension d'AltaRica LaBRI pour une approche de rétro ingénierie de la modélisation de processus biologiques, exploitant des formalismes discrets, stochastiques et dynamiques, [SSN07] ;
- Fares Chucri étudie la vérification de modèles AltaRica par l'approche CEGAR (Counter-Example Guided Abstraction Refinement [CGJ+03]) et implémente une extension CEGAR à l'outil MecV.

Parallèlement à l'élargissement de la communauté des utilisateurs industriels d'AltaRica et aux différentes thèses, des projets académiques ont conduit à l'évolution du langage AltaRica LaBRI. Une nouvelle version dite *AltaRica extended* dispose désormais :

- d'attributs de visibilité,
- de la prise en compte de paramètres,

- de la possibilité de créer :
  - des structures de données
  - des tableaux
  - des types de données abstraits

Les types abstraits de données ont été introduits pour permettre aisément la traduction de modèles avec données infinies vers les outils dédiés. Les autres apports peuvent être perçus comme des macros facilitant le travail de description et ne changent en rien le calcul de la sémantique d'un modèle.

Les types de données abstraits étant en dehors du champ d'étude de cette thèse et les autres apports n'étant que des macros, nous désignons par AltaRica LaBRI le langage d'origine dans la suite de ce mémoire.

2.2 LE LANGAGE ALTARICA

Comme mentionné précédemment, il existe en réalité plusieurs langages AltaRica. Cependant les différences sont relativement mineures : tout comportement écrit en AltaRica Data-Flow ou OCAS peut être transposé en AltaRica LaBRI, ce dernier étant plus expressif. D'autre part, bien que les travaux de raffinement présentés dans le chapitre suivant s'appuient sur le langage AltaRica LaBRI défini dans [Poi00], cette thèse est réalisée en collaboration avec la société Airbus qui réalise des expérimentations en langage AltaRica OCAS. Afin de faciliter la compréhension des expérimentations et résultats exposés plus loin dans cette thèse, il est donc nécessaire de présenter les deux langages.
Nous parlerons :

- de *sémantique* pour désigner l'interprétation mathématique d'un nœud ;
- de *syntaxe abstraite* pour désigner la description d'un nœud utilisée pour calculer sa sémantique ;
- de *syntaxe concrète* pour désigner la description d'un nœud écrite par l'utilisateur. La syntaxe concrète est plus naturelle que la syntaxe abstraite afin de faciliter la saisie par l'utilisateur.

Afin de fournir les connaissances suffisantes pour réaliser un modèle AltaRica, la suite de ce chapitre présente la syntaxe concrète tandis que la syntaxe abstraite et la sémantique d'AltaRica LaBRI sont présentées dans le chapitre 4.

Nous décrivons donc dans la suite de cette section les éléments communs à ces deux variantes (nous parlerons alors d'AltaRica, sans précision) et nous précisons les nuances entre le langage AltaRica LaBRI, présenté sous forme textuelle et le langage AltaRica OCAS, présenté sous forme tabulée.

Le code de tout nœud AltaRica est encapsulé par les mots-clés `node` et `edon`.

Un nœud simple est composé :

- de déclarations :
  - de variables d'états : champs `state` et `init` (pour préciser la valeur initiale de chaque variable),
  - de variables de flux : champ `flow`,
  - d'événements : champ `event`,
- de définitions :
  - des transitions : champ `trans`,
  - d'assertions : champ `assert`.
- de déclarations d'informations externes à la sémantique d'AltaRica mais utiles pour certains outils complémentaires : champ `extern`.

Le langage étant hiérarchique, un nœud peut contenir des sous-nœuds. Il est alors possible de déclarer, outre les informations citées ci-dessus :

- les sous-nœuds contenus : champ `sub` ;
- des synchronisations d'événements : champ `sync`.

Pour illustrer chaque élément de syntaxe, nous utiliserons, pour exemple support, un composant `generator`, fournissant de la puissance lorsqu'il est allumé (position on), que l'on peut allumer lorsqu'il est éteint (événement $start$) et inversement (événement $stop$).

2.2.1 VARIABLES D'ÉTAT ET DE FLUX

Toute variable dispose d'un identificateur et d'un type :
- booléen `bool`, i.e. `true` et `false` ;
- ensemble énuméré, e.g. `failType={ok,err,lost}` ;
- intervalle d'entiers, e.g. `pos=[0,3]`.

Les variables d'état sont des variables internes au nœud pour lesquelles il est nécessaire de préciser une valeur initiale.

En AltaRica LaBRI, le champ `state` permet de déclarer les variables d'état et le champ `init` permet de déclarer la valeur initiale de chaque variable.

En AltaRica OCAS, la variable d'état `on` est déclarée et initialisée à l'aide du tableau ci-dessous.

```
state on : bool;
init  on := true;
```

Variable d'état	Type	Valeur initiale
on	bool	true

FIG. *2.1: Déclaration et initialisation d'une variable d'état*

Les variables de flux permettent d'interconnecter des nœuds entre eux. Elles ne sont jamais initialisées.

En AltaRica LaBRI, les flux ne sont pas orientés.

En AltaRica OCAS, les flux sont orientés : toute variable de flux est définie comme étant une entrée, une sortie ou une variable privée (sans orientation mais non visible de l'extérieur et principalement utilisée pour factoriser des expressions dans le code d'un nœud).

```
flow power : bool;
```

Variable de flux	Type	Orientation
power	bool	out

FIG. *2.2: Déclaration d'une variable de flux*

AltaRica OCAS offre la possibilité de regrouper plusieurs flux "simples" en un seul flux dit "structuré". Cet apport permet principalement de réduire le nombre de flux visibles sur la représentation graphique d'un modèle. Ex : Soit le type structuré `T_s` contenant 2 booléens `a` et `b`, la variable de sortie `S` de type `T_s` symbolise en réalité 2 variables de sorties booléennes `S^a` et `S^b`.

2.2.2 ÉVÉNEMENTS

Un événement AltaRica est une action qui étiquette une transition entre deux configurations. Une configuration est un état global d'un nœud qui associe une valeur à toutes les variables d'état et de flux.

Le langage AltaRica LaBRI autorise les priorités entre événements : si 2 événements peuvent se produire depuis un même état, seul le plus prioritaire sera possible.

Le champ `event` permet de déclarer les événements d'un nœud et les priorités entre événements déclarés. Cependant les aspects quantitatifs étant dissociés de la sémantique d'AltaRica, les probabilités associées aux événements sont spécifiées dans le champ `extern` (la syntaxe du champ `extern` étant liée à l'outil complémentaire ciblé, l'exemple de la figure 2.3 est purement théorique, sans lien avec un outil particulier).

D'autre part, en AltaRica LaBRI, un événement ϵ[1] est automatiquement déclaré, représentant l'absence de changement d'état.

Le langage AltaRica OCAS n'autorise pas les priorités entre événements (un contournement au problème est présenté dans 2.2.3). Cependant les modèles écrits en AltaRica OCAS étant généralement réalisés pour des analyses de sûreté de fonctionnement, les nœuds créés contiennent des défaillances auxquelles il faut associer une probabilité d'occurrence.

[1] ϵ en AltaRica n'a pas le même sens que dans la théorie classique des automates déterministes.

Outre les lois de probabilités habituelles dans les analyses quantitatives, AltaRica OCAS dispose du type $Dirac(0)$ qui influe sur le comportement et les résultats des analyses qualitatives. Un événement de type $Dirac(0)$, aussi appelé "événement instantané" :

– se produit automatiquement dès qu'il dispose d'une transition dont la garde est satisfaite ;
– n'apparaît pas dans les scénarios générés (cf. section outils).

D'après la sémantique d'AltaRica OCAS, si plusieurs événements instantanés peuvent se produire depuis un état, tous les scénarios contenant uniquement des événements instantanés doivent conduire au même état "stable" (duquel aucun événement instantané ne peut se produire).

```
event start,stop;
extern
 law <stop> = exp(1e-5);
```

Événement	Type de loi	Paramètres
start		
stop	exponential	1e-5

FIG. *2.3: Déclaration d'événements*

2.2.3 TRANSITIONS

Une transition définit les conditions d'occurrence et les conséquences d'un événement. Une transition est composée :

– d'une garde : expression booléenne (composée d'(in)égalités, des opérateurs `and`, `or` et `not`) fonction des variables d'état et de flux, représentant les conditions d'occurrence ;
– d'un identifiant : nom de l'événement, déclaré dans le champ `event`, qui étiquette la transition ;
– d'un ensemble d'affectations de nouvelles valeurs aux variables d'état pour décrire les conséquences d'un événement.

Il est possible de définir plusieurs transitions pour un même événement.

Un événement est qualifié de *tirable* s'il existe une transition pour cet événement dont la garde est vraie.

En AltaRica LaBRI, une transition est de la forme :

```
garde |- identifiant -> affectations ;
```

Les règles d'écriture d'une affectation sont précisées dans l'annexe C de [Vin03] présentant la grammaire d'AltaRica LaBRI. Les affectations constituent une partie (les assertions représentant l'autre partie) des *post-conditions* à satisfaire pour que la transition soit possible (ex : si l'affectation incrémente une variable s de 1, la transition ne sera possible que si $s + 1$ appartient à l'ensemble des valeurs possibles du type de s).

AltaRica LaBRI autorise le non-déterminisme, il est donc possible de déclarer, pour un même événement, plusieurs transitions dont les gardes ne sont pas disjointes. A l'opposé, si un événement déclaré n'apparaît dans aucune transition, de par la sémantique du langage, une transition de la forme suivante est implicitement définie :

```
false |- identifiant -> ;
```

Enfin, de même que l'événement ϵ est automatiquement déclaré, la transition associée est implicitement définie.

```
true |- epsilon -> ;
```

AltaRica OCAS n'autorise ni le non-déterminisme, ni les transitions sans affectation (i.e. une transition sans conséquence) comme c'est le cas pour la transition ϵ en AltaRica LaBRI. Tout événement déclaré doit apparaître dans au moins une transition et si plusieurs transitions sont définies pour un même événement, leurs gardes doivent être disjointes. L'absence de transition et le non-déterminisme sont vérifiés par des outils existants.

```
trans
 not on |- start -> on := true;
     on |- stop  -> on := false;
```

Événement	Garde	Affectation
		on
start	not on	true
stop	on	false

FIG. *2.4: Définition de transitions*

Le comportement exprimé ci-dessus peut être représenté par l'automate présenté sur la figure 2.5.

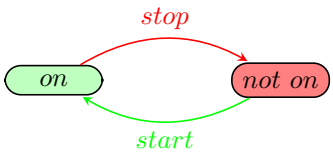


FIG. *2.5: Sémantique du nœud* `Generator`

Il est possible en AltaRica LaBRI de déclarer deux événements e_1 et e_2, ayant pour garde respectivement g_1 et g_2, tels que e_1 est plus prioritaire que e_2. Autrement dit, si g_1 et g_2 sont satisfaites en même temps, seule la transition associée à e_1 sera possible.

On peut contourner la limitation d'AltaRica OCAS qui ne propose pas les priorités entre événements. Sous certaines conditions de "complétude" (toutes les variables de sortie doivent être valuées quelles que soient les valeurs des variables d'entrée et d'état), il suffit de compléter la garde g_2 par la négation de g_1, i.e. g_2 devient g_2 *and not* g_1. Ainsi l'événement e_2 ne sera possible que lorsque la garde g_2 sera vraie et que l'événement e_1, plus prioritaire, ne sera pas possible.

2.2.4 ASSERTIONS

Les assertions servent à restreindre les configurations possibles et en particulier :
- pour un nœud simple : à définir les valeurs des variables de flux ;
- pour un nœud contenant des sous-nœuds : à définir les connexions entre nœuds.

En AltaRica LaBRI, les assertions sont des contraintes sur les variables de flux. Du fait que les flux ne sont pas orientés, il est possible de définir des contraintes sur toute variable. Les assertions constituent une partie (les affectations des transitions étant l'autre partie) des *post-conditions*.

Les règles d'écriture d'une assertion sont précisées dans l'annexe C de [Vin03] présentant la grammaire d'AltaRica LaBRI.

En AltaRica OCAS, chaque variable de sortie ou privée est définie par une assertion en fonction des variables d'état, d'entrée ou privées (sans récursion).

Contrairement au langage AltaRica LaBRI qui accepte différentes constructions dont celle de type "if ... then ... else ...", le langage AltaRica OCAS tabulé présenté ici se restreint à la construction de type "case ... else ...". Cette construction se transpose aisément sous forme tabulée et permet une présentation extrêmement claire des situations considérées. Le cas "else", englobant toute situation n'ayant pas été définie préalablement, garantit que toute variable de sortie ou privée dispose d'une valeur à tout instant (l'absence de valuation d'une variable dans une configuration n'est pas détectée par les outils de contrôle mais représente une situation bloquante pour OCAS pour effectuer une simulation ou une analyse).

```
assert
  power = on;
```

Variable de flux	Case	Affectation
power	on	true
	else	false

FIG. *2.6: Définition d'assertions*

2.2.5 NŒUD SIMPLE

Le code AltaRica d'un nœud simple est composé des champs mentionnés ci-dessus. Le nœud `generator`, dont la sémantique est illustrée sur la figure 2.5, s'écrit en AltaRica comme présenté ci-dessous :

```
node generator
  state on : bool;
  init  on := true;
  flow  power : bool;
  event start,stop;
  trans
    not on |- start -> on := true;
        on |- stop  -> on := false;
  assert
    power = on;
  extern
    law <stop> = exp(1e-5);
edon
```

En AltaRica LaBRI, l'initialisation des variables d'état est facultative : en l'absence d'initialisation, tout état est initial. En revanche, en AltaRica OCAS, toute variable d'état doit être initialisée.

2.2.6 NŒUD HIÉRARCHIQUE, COMPOSÉ DE SOUS-NŒUDS

De par la hiérarchie d'AltaRica, un nœud peut contenir un ou des sous-nœuds. Plusieurs sous-nœuds peuvent être d'un même type : nous parlerons d'instances. D'autre part, les variables de flux, utilisées pour relier des nœuds entre eux (chaque lien étant défini par une assertion), servent également à relier les sous-nœuds aux nœuds environnant.

Afin d'illustrer les concepts liés à la notion de hiérarchie, considérons un nœud `GenSystem` contenant deux sous-nœuds `Gen1` et `Gen2` de type `generator`.

En AltaRica LaBRI, tout nœud hiérarchique peut disposer de ses propres variables d'état et de ses propres événements. Les instances sont déclarées dans le champ `sub`, chaque instance étant composée d'un identifiant et d'un type de nœud.

```
node GenSystem
  sub Gen1,Gen2 : generator;
  ...
edon
```

En AltaRica OCAS, un nœud contenant des sous-nœuds, baptisé *équipement* (par opposition à *composant* pour un nœud sans sous-nœud), ne dispose que de ses propres variables de flux : les variables d'état et les événements sont ceux des sous-nœuds. Afin de faciliter la création d'un équipement, l'instanciation se fait de manière graphique par glisser/déposer d'un type de nœud dans la zone de contenu, chaque instance étant ensuite munie d'un identifiant unique.

2.2.6.1 Connexion de variables

Deux variables de flux peuvent être connectées si elles sont de même type. Cette connexion permet de représenter la communication :

- entre deux sous-nœuds ;
- entre le nœud père et un de ses sous-nœuds, nécessaire pour qu'un sous-nœud puisse communiquer avec un nœud extérieur.

Un nœud père contient le détail des connexions entre deux variables de flux appartenant à deux sous-nœuds ou une variable de flux d'un sous-nœud et une de ses variables de flux.

Supposons que `GenSystem` contient deux variables de flux `power1` et `power2` reliées respectivement à la variable `power` de `Gen1` et `Gen2`.

En AltaRica LaBRI, toute connexion est définie par une assertion. Les connexions du nœud `GenSystem` à ses sous-nœuds s'écrivent alors :

```
node GenSystem
  sub    Gen1,Gen2 : generator;
  flow   power1,power2 : bool;
  assert power1 = Gen1.power;
         power2 = Gen2.power;
edon
```

En AltaRica OCAS, le contenu d'un équipement (nœud contenant des sous-nœuds) est présenté graphiquement comme un modèle. Chaque variable de flux d'un équipement est représenté graphiquement dans le contenu de l'équipement : il est ainsi possible de connecter toute variable de flux de l'équipement ou d'un de ses composants à une autre variable de flux par simple sélection des variables à connecter (l'outil vérifie automatiquement la compatibilité).

2.2.6.2 Synchronisation d'événements

Il est possible en AltaRica de contraindre plusieurs événements à se produire en même temps : on parle de synchronisation d'événements. Une synchronisation concerne des événements de nœuds différents et doit être définie au niveau du nœud "père" contenant les sous-nœuds concernés. Une synchronisation se déclare dans le champ `sync` par une liste de vecteurs de synchronisation. En l'absence de cette rubrique, les sous-nœuds sont indépendants (asynchrone) et la liste des vecteurs de synchronisation implicites est obtenue par l'union des événements des sous-nœuds.

Afin d'illustrer les différents types de synchronisation possibles, considérons le nœud `GenSystem` mentionné précédemment. La garde de l'événement `start` de `Gen1` (resp. `Gen2`) est désignée par g_{Gen1} (resp. g_{Gen2}). La synchronisation de `Gen1.start` et `Gen2.start` est étiquetée `start`.

Nous représentons par un automate le comportement correspondant au code AltaRica décrit en adoptant les conventions suivantes :

- chaque état est de la forme x/y où x (resp. y) représente la valeur de `Gen1.on` (resp. `Gen2.on`) ;
- un état est de couleur verte si les 2 sorties de `GenSystem` sont vraies, de couleur bleue si une seule sortie est vraie, de couleur rouge si les deux sorties sont fausses ;
- l'état $true/true$ de couleur verte est l'état initial de chaque automate ;
- toute transition résultant d'une synchronisation est de couleur noire ;
- une transition verte représente la transition d'un événement `start` non synchronisé ;
- une transition rouge représente la transition d'un événement `stop` non synchronisé ;
- toute transition est étiquetée par une synchronisation constituée d'un événement du nœud `GenSystem` suivi d'un événement, défini par l'utilisateur, d'un (ou de chaque) sous-nœud ;
- tout vecteur ne contenant pas un événement pour chaque sous-nœud est implicitement complété par l'événement ϵ pour ce sous-nœud.

En AltaRica LaBRI, tout événement d'un sous-nœud est, par défaut et automatiquement, synchronisé avec ϵ et non visible des autres nœuds. Pour rendre un événement d'un sous-nœud visible, il est nécessaire de le synchroniser avec un événement de même nom déclaré dans le nœud "père"[2].

L'étiquette d'une synchronisation est déclarée comme un événement. De même que pour tout événement d'un nœud sans sous-nœud, il est nécessaire de définir une transition pour un événement d'un nœud avec sous-nœud. Cette transition permet éventuellement d'ajouter des contraintes aux gardes des événements synchronisés.

Par défaut, une synchronisation n'est possible que si toutes les gardes et post-conditions sont satisfaites. Cependant il existe dans AltaRica LaBRI une notion de synchronisation avec diffusion : un

[2] Dans le langage AltaRica extended, un événement déclaré "public" est visible dans le nœud père.

événement marqué d'un ? n'est pris en compte que si sa garde et ses post-conditions sont satisfaites. De plus, le nombre d'événements marqués tirables peut être contraint : à titre d'exemple, le vecteur `<repair,A.repair?,B.repair?> =1` exprime le fait que lorsque la synchronisation `repair` est tirée, un seul des événements du vecteur de synchronisation sera tiré.

D'après la sémantique d'AltaRica LaBRI, un vecteur de synchronisation avec diffusion représente en réalité l'ensemble des vecteurs de synchronisation pour lesquels le nombre d'événements marqués d'un ? respecte la contrainte définie (l'absence de contrainte signifie que le nombre d'événements dans le vecteur est compris entre 0 et le nombre total d'événements marqués). Cependant la sémantique d'AltaRica LaBRI introduit une notion de priorité entre les vecteurs possibles : depuis chaque configuration, seul les vecteurs contenant un maximum pour l'inclusion d'événements tirables sont possibles (les post-conditions d'AltaRica LaBRI influant sur les priorités entre vecteurs d'une synchronisation avec diffusion).

Pour illustrer les notions énoncées, considérons tout d'abord un système sans synchronisation. Le code correspondant est celui présenté dans le paragraphe 2.2.6.1. En l'état, chaque générateur agit indépendamment de l'autre générateur. Le comportement est représenté par la figure 2.7.

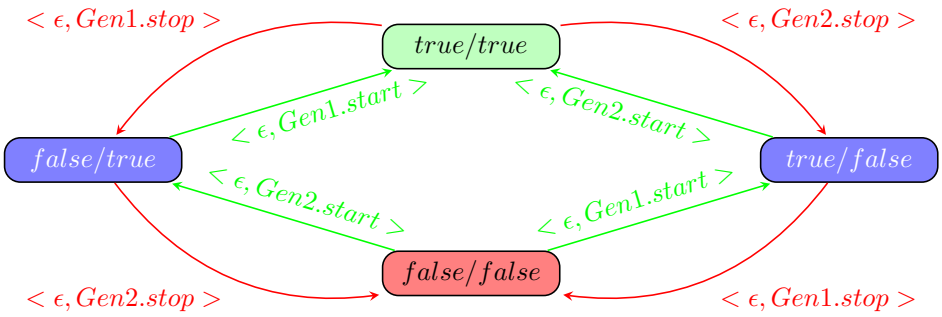


FIG. 2.7: *Sémantique du nœud `GenSystem` sans synchronisation*

Afin de considérer une synchronisation $start$, ajoutons tout d'abord un événement `start` et la transition correspondante au nœud `GenSystem`. Le code précédemment présenté devient alors :

```
node GenSystem
  sub    Gen1,Gen2 : generator;
  flow   power1,power2 : bool;
  assert power1 = Gen1.power;
         power2 = Gen2.power;
  event  start;
  trans  true |- start -> ;
edon
```

L'événement `start` étant déclaré dans le nœud `GenSystem`, l'étape suivante consiste à définir le vecteur de synchronisation. Dans un premier temps, considérons la synchronisation sans diffusion, exprimée en AltaRica LaBRI comme écrit ci-dessous :

```
node GenSystem
  sub    Gen1,Gen2 : generator;
  flow   power1,power2 : bool;
  assert power1 = Gen1.power;
         power2 = Gen2.power;
  event  start;
  trans  true |- start -> ;
  sync   <start, Gen1.start, Gen2.start>;
edon
```

Cette synchronisation signifie que les deux générateurs ne peuvent être activés que lorsque les deux sont éteints.

En comparant la figure 2.8 du comportement avec synchronisation avec la figure 2.7, on note :
- l'ajout d'une transition $start$ correspondant à la synchronisation ;
- la suppression de toutes les transitions ϵ correspondant à l'événement $start$ d'un sous-nœud.

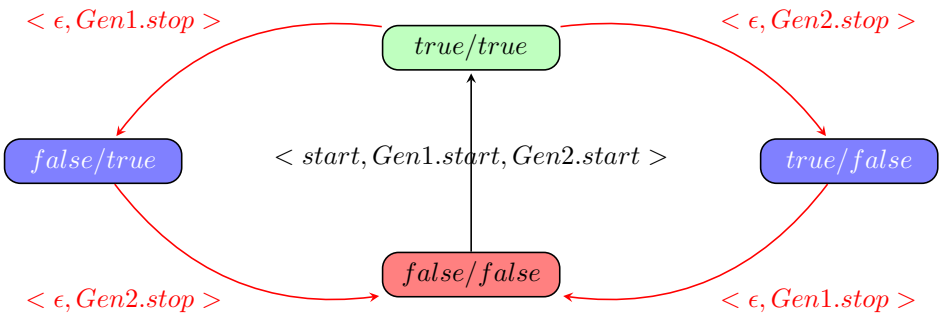


FIG. 2.8: *Sémantique du nœud* `GenSystem` *avec synchronisation, sans diffusion*

Dans un second temps, considérons la synchronisation avec diffusion et contrainte faible, exprimée en AltaRica LaBRI comme écrit ci-dessous (la précision '>=1' est nécessaire pour interdire que l'événement $start$ du nœud $GenSystem$ soit simultanément synchronisé avec l'événement ϵ de chaque sous-nœud) :

```
sync <start, Gen1.start?, Gen2.start?> >=1;
```

Cette synchronisation équivaut aux trois vecteurs suivants, le premier vecteur étant plus prioritaire que les deux suivants qui ont même niveau de priorité entre eux :

```
<start, Gen1.start, Gen2.start>;
<start, Gen1.start>;
<start, Gen2.start>;
```

Cette synchronisation signifie que :
- si un seul générateur est éteint, il peut être activé seul ;
- si les deux générateurs sont éteints, ils ne peuvent être activés que simultanément (il n'est pas possible d'activer un seul générateur).

En comparant la figure 2.9 du comportement avec synchronisation avec la figure 2.7, on note l'ajout de trois transitions $start$ correspondant à la synchronisation et la suppression de toutes les transitions ϵ représentant l'événement $start$ d'un sous-nœud.

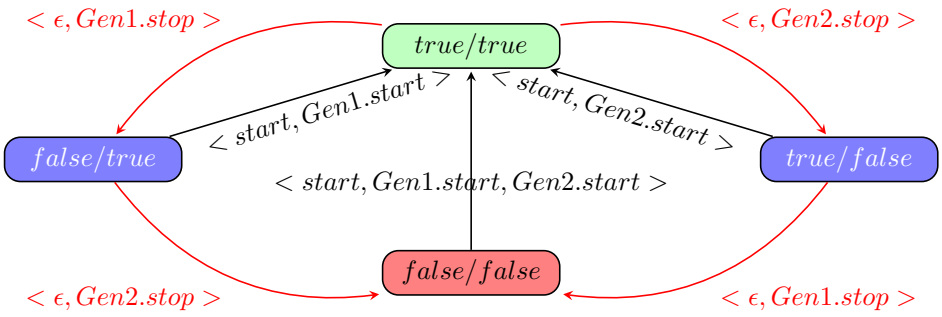


FIG. 2.9: *Sémantique du nœud* `GenSystem` *avec synchronisation et diffusion*

Enfin, considérons la synchronisation avec diffusion et contrainte forte, exprimée en AltaRica LaBRI comme écrit ci-dessous :

```
sync <start, Gen1.start?, Gen2.start?> =1;
```

Cette synchronisation équivaut aux deux vecteurs suivants de même priorité :

```
<start, Gen1.start>;
<start, Gen2.start>;
```

Cette synchronisation signifie que tout générateur ne peut être activé que seul.

En comparant la figure 2.10 du comportement avec synchronisation avec la figure 2.7, on constate que les comportements semblent identiques mais on note que les transitions contenant un événements $start$ sont étiquetées $start$ dans la figure 2.10 et ϵ dans la figure 2.7.

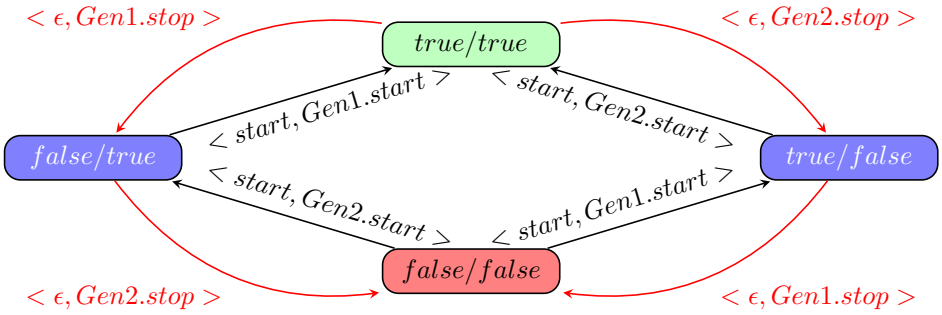


FIG. 2.10: *Sémantique du nœud* `GenSystem` *avec synchronisation, diffusion et contrainte*

Pour clarifier la *maximisation pour l'inclusion*, considérons 3 événements nécessitant chacun des ressources :

- `A.a` nécessite 2 ressources ;
- `B.b` et `C.c` ne nécessitent qu'une ressource.

La synchronisation `<sync, A.a?, B.b?, C.c?>` modélise la volonté d'effectuer les 3 événements simultanément. Cependant, le nombre de ressources disponibles, précisé via une assertion, est limité :

- si une seule ressource est disponible, `A.a` ne peut jamais être effectué et la synchronisation revient à considérer deux vecteurs de même priorité, `<sync, B.b>` et `<sync, C.c>` ;
- si deux ressources sont disponibles, la synchronisation revient à considérer deux vecteurs de même priorité, `<sync, A.a>` et `<sync, B.b, C.c>` ;
- ...

En AltaRica OCAS, une synchronisation dispose d'une étiquette unique pour identifier un ensemble d'événements.

Tout événement d'un sous-nœud est, par défaut, visible des autres nœuds (i.e. synchronisé avec un événement de même nom déclaré dans le nœud "père"). La visibilité d'un événement peut changer, s'il est déclaré dans une synchronisation, en fonction du type de synchronisation choisi.

On distingue trois types de synchronisation :

- la synchronisation :
  - $start$ n'est possible que si g_{Gen1} et g_{Gen2} sont satisfaites,
  - les événements $Gen1.start$ et $Gen2.start$ ne sont pas visibles,
- la diffusion :
  - $start$ est possible si au moins une garde est satisfaite (i.e. dès que g_{Gen1} ou g_{Gen2} est satisfaite),
  - seules les affectations définies pour les événements dont les gardes sont satisfaites sont réalisées,
  - les événements $Gen1.start$ et $Gen2.start$ ne sont pas visibles,

– la Défaillance de Cause Commune (DCC)[3] :
  – $start$ est possible si au moins une garde est satisfaite (i.e. dès que g_{Gen1} ou g_{Gen2} est satisfaite) ;
  – seules les affectations définies pour les événements dont les gardes sont satisfaites sont réalisées,
  – les événements $Gen1.start$ et $Gen2.start$ sont visibles indépendamment de $start$.

Pour résumer, la synchronisation nécessite que toutes les gardes soient satisfaites contrairement à la diffusion et au type DCC qui ne nécessitent qu'une garde satisfaite ; d'autre part la synchronisation et la diffusion masquent les événements contenus contrairement au type DCC.

Considérons l'implémentation en AltaRica OCAS de l'équipement $GenSystem$. La sémantique sans synchronisation est présentée par la figure 2.11. Les figures 2.12, 2.13 et 2.14 présentent respectivement les sémantiques si la synchronisation est de type synchronisation, diffusion et DCC.

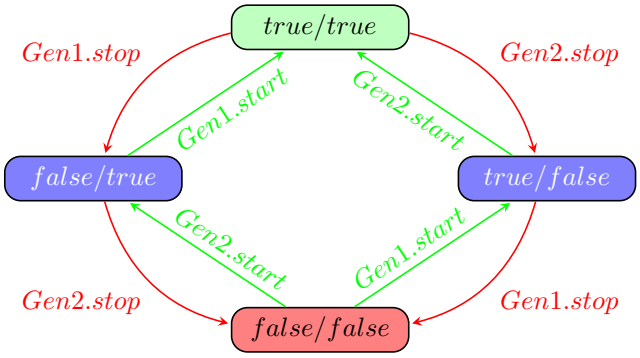


FIG. *2.11: Sémantique de l'équipement* `GenSystem` *sans synchronisation*

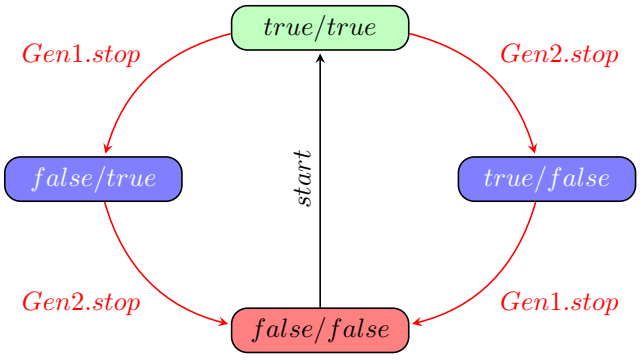


FIG. *2.12: Sémantique de l'équipement* `GenSystem` *avec synchronisation de type synchronisation*

[3]La notion de défaillance de cause commune, en sûreté de fonctionnement des systèmes, désigne le fait que des composants physiques considérés comme différents peuvent, en réalité, avoir un point commun et qu'une défaillance liée à ce point commun peut impacter simultanément ces composants : des composants dont les architectures internes sont différentes mais qui contiennent un même sous-composant ou un même logiciel...

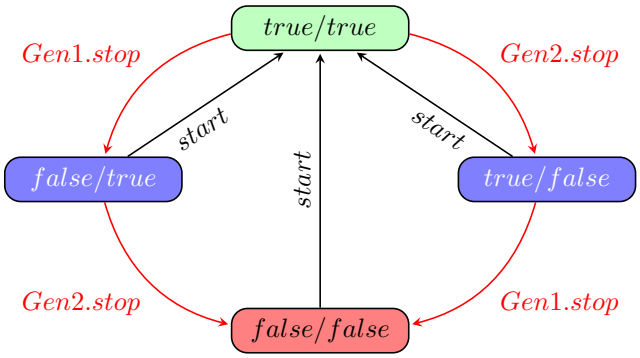


FIG. 2.13: *Sémantique de l'équipement* `GenSystem` *avec synchronisation de type diffusion*

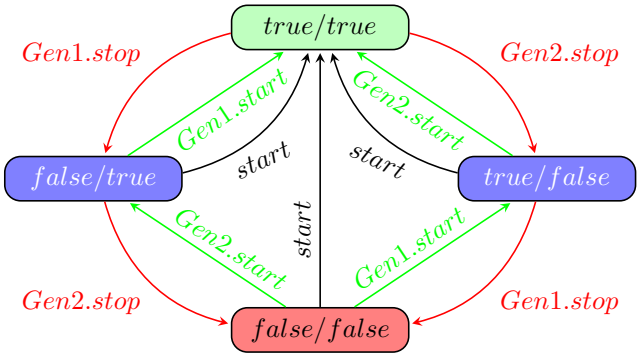


FIG. 2.14: *Sémantique de l'équipement* `GenSystem` *avec synchronisation de type DCC*

2.3 LES OUTILS DE MODÉLISATION

Si l'analyse d'un système de petite taille peut être réalisée manuellement, l'utilisation d'un modèle formel et des outils associés se révèle pertinente pour analyser des systèmes complexes : les interactions entre composants d'un système sont généralement très nombreuses et le cerveau humain n'a rapidement pas la capacité de maîtriser la combinatoire des événements.

L'activité de modélisation se décompose, en réalité, en trois activités :

– la réalisation d'un modèle à partir d'une spécification d'un système ;
– la validation du modèle vis-à-vis de la spécification ;
– l'analyse (qualitative et quantitative) du système à l'aide du modèle.

Chaque activité est supportée par un logiciel ou une fonctionnalité d'un logiciel.

Historiquement, le LaBRI a développé, d'une part, des outils regroupés sous le nom d'Altatools et, d'autre part, un vérificateur de modèles Mec dont l'évolution MecV [Vin03] accepte directement les modèles AltaRica LaBRI. Depuis 2005, le LaBRI conçoit l'outil AltaRica Checker (ARC), basé à la fois sur les Altatools et sur MecV, afin de proposer un unique logiciel réunissant tous les services des outils historiques. MecV ayant permis les expérimentations liées au raffinement présenté dans les chapitres suivants, cette section détaillera plus particulièrement cet outil.

De son coté, Dassault Aviation développe l'Outil de Conception et d'Analyse Système (OCAS) permettant à la fois la réalisation d'un modèle et sa validation, intégrant des outils d'analyse qualitatives et générant des fichiers compatibles avec des outils complémentaires pour des analyses quantitatives.

2.3.1 ÉDITION

Réaliser un modèle formel consiste à traduire la spécification du comportement d'un système dans un langage formel donné. De même que ce système peut être composé d'objets de niveaux hiérarchiques différents (équipements, composants, ...), son modèle AltaRica peut être composé de nœuds simples et de nœuds composés de sous-nœuds. Les nœuds simples sont à écrire en priorité car nécessaires pour écrire les nœuds de niveau hiérarchique supérieur.

Un modèle en AltaRica LaBRI est seulement composé d'une description textuelle (cf. section précédente). Tout éditeur de texte (ex : Emacs) permet donc d'écrire un modèle en AltaRica LaBRI.

Un modèle en AltaRica OCAS contient à la fois une description textuelle (cf. section précédente) et une représentation graphique :

- chaque nœud dispose de sa propre représentation graphique (figure 2.16) :
  - par défaut un nœud est représenté par un carré ; il est cependant possible de définir un icône, voire plusieurs, associés à des configurations différentes,
  - les ports sont visibles autour de l'icône d'un nœud : un carré vide (resp. plein) pour une variable d'entrée (resp. sortie),
- tout nœud créé est automatiquement stocké dans une librairie de nœuds (figure 2.15) pour être ensuite instancié graphiquement dans un nœud composé (figure 2.17) ou un modèle (un modèle peut être perçu comme un nœud composé "fermé", i.e. sans variable de flux),
- les instances de nœuds sont reliées graphiquement.

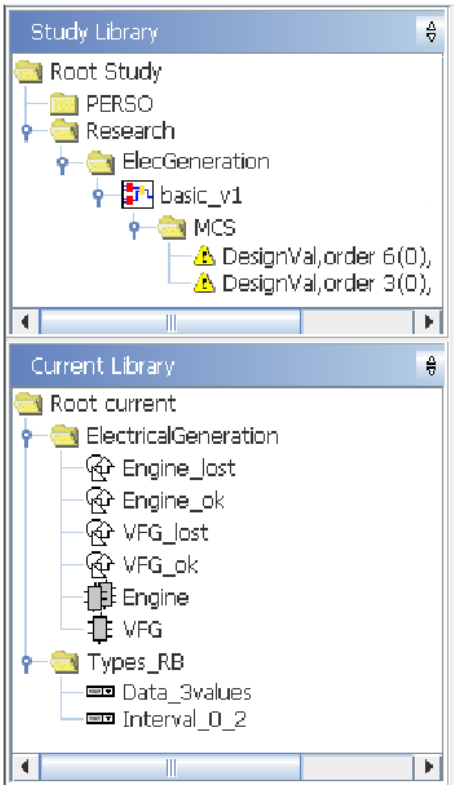


FIG. 2.15: *Librairies de modèles (study library) et d'objets (current library, contenant composants, icones, types définis par l'utilisateur)*

FIG. *2.16: Interface de saisie d'un nœud "simple" : déclaration des variables de flux*

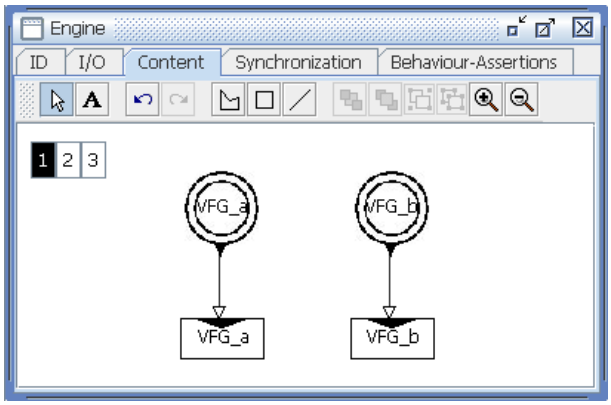


FIG. *2.17: Interface de saisie d'un nœud composé : instanciations de nœuds et définition du contenu*

2.3.2 VALIDATION

Avant de pouvoir analyser un modèle, il est nécessaire de s'assurer que :
- le modèle respecte les règles d'écriture du langage formel choisi ;
- le comportement décrit par le modèle est fidèle au comportement spécifié/réel du système.

Un **contrôleur syntaxique** vérifie que le code respecte les règles de syntaxe : les mots clés, séparateurs, parenthèses sont correctement écrits, toute variable qui apparaît dans une transition ou un assertion a bien été déclarée et typée, ...

Cette opération est effectuée automatiquement lors du chargement d'un modèle en AltaRica LaBRI dans les outils MecV et ARC.

Cette opération est appelée manuellement dans OCAS par l'utilisateur.

Un **contrôleur de cohérence** permet de s'assurer que toutes les informations nécessaires à la description d'un nœud (domaines, sous-noeuds...) sont spécifiées et que le comportement décrit respecte les exigences du langage.

Cette opération est effectuée automatiquement lors du chargement d'un modèle en AltaRica LaBRI dans les outils MecV et ARC.

Cette opération, appelée manuellement dans OCAS par l'utilisateur, vérifie également, entre autres particularités d'AltaRica OCAS, l'absence :
- d'événement non-déterministe (un événement dont plusieurs transitions sont possibles depuis un même état et dont les affectations conduisent à des états différents) ;

– de boucle : une variable de sortie pour laquelle une définition circulaire peut être établie : la valeur dépend de l'entrée qui elle-même dépend de la sortie (directement ou indirectement via des nœuds interconnectés) ;
– de transition pour un événement déclaré ;
– d'assertion pour une variable de sortie ;
– de lien conduisant à une entrée d'un composant.

Un **simulateur interactif** permet, à partir d'une situation initiale décrite dans le code AltaRica ou définie par l'utilisateur et mémorisée dans un outil, de sélectionner manuellement les actions que doit effectuer le système et d'observer son évolution. Cet outil permet de tester le comportement du modèle sur un ensemble fini d'exécutions d'événements afin de s'assurer que le comportement modélisé est fidèle à celui du système réel.

Dans l'outil ARC, le simulateur présente respectivement sous forme de tableaux : la configuration courante, les événements possibles et le détail de la transition de l'événement sélectionné. Observer l'évolution du système revient à observer les changements de valeur des différentes variables.

Outre les nuances de langage, AltaRica OCAS se différencie d'AltaRica LaBRI par l'ajout de considérations graphiques en complément du comportement fonctionnel/dysfonctionnel. Ainsi l'outil OCAS permet :

– de déclarer plusieurs icônes à chaque composant, chacun étant associé à une configuration ;
– de définir une couleur pour chaque valeur d'un type d'une variable de flux.

Le simulateur permet d'observer graphiquement l'évolution du système modélisé : la sélection un événement s'accompagne d'une mise à jour des icônes et liens.

2.3.3 ANALYSE

Les analyses qualitatives de modèles se focalisent sur la recherche des scénarios qui conduisent à une configuration donnée du modèle.

Un **générateur de scénario** fournit pour un modèle AltaRica donné, à partir d'une situation initiale sélectionnée et en précisant la taille maximale des scénarios, toutes les combinaisons d'événements qui permettent d'atteindre une situation cible définie.

La chronologie des événements peut avoir son importance. Il peut donc être nécessaire de la faire apparaître dans les combinaisons : on parle alors de "séquences d'événements", ou plus simplement de "séquences".

Si quelque soit l'ordre dans lequel les événements surviennent, l'état atteint est le même, les combinaisons d'événements peuvent alors être représentées sous la forme compacte de "coupes". Soient e_1 et e_2 deux événements, la coupe "e_1 et e_2" traduit le fait que les séquences "e_1 puis e_2" et "e_2 puis e_1" conduisent à la situation cible.

Une coupe ou une séquence est qualifiée de "minimale" lorsqu'il s'agit de la plus petite combinaison d'événements pouvant conduire à la situation redoutée : si un événement d'une coupe minimale ne se produit pas, la situation redoutée n'est pas atteinte.

OCAS permet de générer directement depuis un modèle AltaRica OCAS des coupes minimales ou des séquences minimales.

L'arbre de défaillances [Eri99] est un formalisme courant dans la sûreté de fonctionnement. OCAS propose un **générateur d'arbre de défaillances** [Rau02] qui effectue une analyse exhaustive d'un modèle donné pour identifier toutes les combinaisons de défaillances, sans limite de taille, conduisant à une situation redoutée. L'arbre généré peut alors être traité par l'outil Aralia afin d'en extraire toutes les coupes minimales et de calculer la probabilité d'occurrence de la situation redoutée.

Un **model-checker** [QS83, CES86, CGP00, SBB+99], littéralement **vérificateur de modèle**, effectue une analyse exhaustive automatique du modèle formel d'un système pour vérifier qu'une propriété formalisée attendue est satisfaite.

Dans la plupart des cas, un modèle et une propriété exprimée dans une logique temporelle (linéaire ou arborescente) sont fournis au model-checker qui cherche un exemple d'exécution du système violant la propriété. L'analyse s'arrête :

- dès qu'un contre-exemple est trouvé ;
- lorsque tous les scénarios ont été analysés (seulement possible si le modèle a un nombre d'états accessibles fini : toute succession d'événements conduit à un état à partir duquel aucun événement n'est possible), auquel cas la propriété est satisfaite.

Si le modèle a un nombre d'états accessibles infini, l'analyse peut ne pas se terminer et il est alors impossible de conclure sur la satisfaisabilité de la propriété.

Contrairement à la génération de scénarios qui fournit un ensemble de combinaisons d'événements, le model-checking "se contente" de trouver une combinaison d'événements. Néanmoins, une approche itérative permet, par model-checking, de générer un ensemble de combinaisons d'événements comparable à celui obtenu à l'aide d'un générateur de scénario :
Soient S_c la situation cible que l'on souhaite étudier et P_{S_c} la traduction de S_c en propriété de la forme "il n'existe aucun scénario conduisant à la situation S_c". Le model-checker identifie un premier contre-exemple cex_1. La propriété P_{S_c} est alors enrichie pour devenir "il n'existe pas un scénario conduisant à la situation S_c autre que cex_1". On ajoute itérativement les contre-exemples jusqu'à ce que la propriété enrichie soit satisfaite.

Le premier model-checker capable de traiter directement des modèles AltaRica finis est **MecV** [Vin03, GV03, GV04]. MecV permet de calculer tout type de relation, y compris des relations complexes entre modèles, et de générer des graphes. MecV dispose de son propre langage de spécification, directement transposé du μ-calcul [Koz83, Arn92] (logique du premier ordre étendue avec des définitions de points fixes).

Le principe du calcul d'un plus petit point fixe, noté `'+='` en MecV, est le suivant : soit S l'ensemble des solutions du calcul par point fixe, partant de S égal à l'ensemble vide, tout élément qui vérifie la condition (généralement exprimée sous forme disjonctive) est ajouté à S ; cette opération de vérification et d'ajout est répétée depuis le nouvel ensemble S jusqu'à ce qu'aucun élément ne puisse plus être ajouté.

De façon complémentaire, le principe du calcul d'un plus grand point fixe, noté `'-='` en MecV, est le suivant : soit S l'ensemble des solutions du calcul par point fixe, partant de S contenant tous les éléments, tout élément qui ne vérifie pas la condition (généralement exprimée sous forme conjonctive) est supprimé de S ; cette opération de vérification et de retrait est répétée depuis le nouvel ensemble S jusqu'à ce qu'aucun élément ne puisse plus être retiré.

Lorsqu'un nœud AltaRica `nA` est chargé, MecV crée automatiquement :

- les types :
  - `nA!c` : ensemble des configurations de `nA`,
  - `nA!ev` : ensemble des vecteurs d'événements de `nA`,
- les relations :
  - `nA!t` : ensemble des transitions de `nA`,
  - `nA!init` : ensemble des configurations initiales de `nA`.

Tout autre objet, créé par l'utilisateur, doit être typé : les domaines considérés sont les booléens (`true`, `false`), les intervalles d'entiers (`[0,9]`) et les énumérations de constantes (`ok, lost`).

Enfin les relations MecV sont écrites à l'aide des opérateurs suivants :

- la conjonction : `&` ;
- la disjonction : `|` ;
- la négation : `~` ;
- la modalité existentielle "il existe un x t.q. ..." : `<x>` ;
- la modalité universelle "pour tout x, ..." : `[x]`.

Exemple 2.1

Le calcul des états accessibles d'un nœud est un calcul de plus petit point fixe :

- *les premiers états accessibles sont les états initiaux ;*
- *on ajoute les états destination d'une transition depuis un état initial ;*

– *on répète l'opération depuis le nouvel ensemble d'états.*

Soit `Reach` *la relation calculant les états accessibles du nœud* `generator`*, la traduction en langage MecV est :*

```
Reach(s) += generator!init(s)
          | (<u>(Reach(u) & <e>(generator!t(u,e,s))));
```

Les configurations calculées par cette relation sont :

```
({ on,  power})
({~on, ~power})
```

Remarque

MecV peut extraire un scénario violant une propriété mais nécessite l'écriture, dans un premier temps, de la relation booléenne permettant de vérifier la propriété puis de la relation retournant le scénario invalidant la propriété.

En complément des analyses qualitatives qui viennent d'être présentées, il faut mentionner des analyses qui visent à vérifier des exigences quantitatives. Par exemple, la **simulation de Monte Carlo** a pour principe de simuler "l'histoire" du système par tirage aléatoire en tenant compte des probabilités associées aux événements (cf.2.2.2) du modèle AltaRica. Des exigences quantitatives, comme le taux de défaillance du système, sont obtenues statistiquement après la simulation d'un grand nombre d'histoires.

2.4 LES LANGAGES POUR LES ANALYSES DE SÛRETÉ DE FONCTIONNEMENT DE SYSTÈMES

Comme nous l'avons dit en introduction de ce chapitre, le langage AltaRica a été créé dans le but de supporter les analyses de sûreté de fonctionnement de systèmes en tirant profit des avancées dans le domaine des méthodes formelles. Ainsi, les inconvénients identifiés des formalismes traditionnels, décrits dans 1.2.4.3, sont solutionnés par l'automatisation de la création d'arbres de défaillances ou de graphes de Markov.

Néanmoins AltaRica n'est pas le seul langage formel dédié à la sûreté de fonctionnement. Bien que le langage AltaRica soit au cœur de cette thèse, nous nous sommes intéressés aux travaux récents (ultérieurs à 1996, année de la dernière version officielle des ARP 4754 [SAE96b] et 4761 [SAE96a], i.e. documents de recommandations applicable dans le domaine aéronautique) spécifiquement orientés sur la modélisation fonctionnelle et dysfonctionnelle d'un système. Nous avons retenu :

– le langage Figaro[BS06] et l'environnement SyRelAn [RMR+07] (System Reliability Analysis) afin de réaliser des modèles à la fois fonctionnels et dysfonctionnels,
– les langages East-ADL2 [DJL+08] et AADL [FR07] afin d'enrichir des modèles fonctionnels de considérations dysfonctionnelles.

Ces langages permettent de modéliser différents types de systèmes contrairement au formalisme FTDF [MEP+05] qui permet de générer des arbres de défaillances pour des systèmes à taches communicantes.

2.4.1 LE LANGAGE FIGARO

Le langage Figaro a été créé par EDF (Electricité de France) en 1990 [Bou90]. L'article [BS06] compare les langages AltaRica et Figaro. Si ces deux langages partagent un but commun (modéliser la propagation de défaillances au sein d'un système), ils considèrent des principes de modélisation différents :

– le langage Figaro est orienté objet : une base de connaissances définit des types et des comportements génériques puis, par des mécanismes d'héritage et de surcharge, les objets d'un système sont créés ; ces objets n'étant ni organisés hiérarchiquement, ni encapsulés ;

– le langage AltaRica est basé sur la compositionalité : des objets sont définis puis instanciés pour créer des objets plus complexes de niveau supérieur (le niveau le plus haut étant la description du système dans sa globalité).

D'autre part, l'approche de modélisation en langage Figaro tend à écrire des modèles probabilistes quantifiables pouvant être simplifiés en modèles qualitatifs, contrairement au langage AltaRica utilisé pour créer des modèles qualitatifs pouvant être complétés pour des analyses quantitatives.

Dans l'article [BS06], certaines affirmations mettant en avant le langage Figaro sont à nuancer : le langage Figaro et l'outil support KB3 sont étroitement liés à EDF contrairement au langage AltaRica, créé initialement au LaBRI (Laboratoire Bordelais de Recherche en Informatique) et dont les outils industriels, basés sur la variante data-flow, sont depuis développés par différentes sociétés (Dassault Aviation, Dassault Systèmes, EADS-APSYS...). La syntaxe de Figaro permet effectivement d'écrire des expressions proches du langage naturel contrairement au langage AltaRica, dont la présence de certains éléments syntaxiques pourrait rebuter des "non-informaticiens". Néanmoins, notre expérience a montré que les ingénieurs, familiers à la lecture de tableaux de données, assimilent aisément les modèles AltaRica via la présentation tabulaire, illustrée dans ce chapitre.

La conclusion de l'article [BS06] oriente le choix du langage à utiliser suivant le besoin : le langage Figaro s'adresse à des concepteurs d'outils de modélisation basés sur des schémas de principes ou des formalismes abstraits, alors que le langage AltaRica, plus facilement accessible, s'adresse directement aux ingénieurs chargés d'étudier des systèmes.

2.4.2 L'ENVIRONNEMENT SYRELAN

L'institut d'ingénierie des systèmes aéronautiques de l'Université de Hambourg a également étudié, en collaboration avec Airbus Allemagne, la formalisation du comportement d'un système en cas de défaillance, autrement dit la prise en compte des principes de reconfiguration. Contrairement aux principaux cas d'études réalisés en AltaRica ou en Figaro, le but des modélisations était l'analyse de la dégradation de performances en cas de défaillances de composants d'un système. Cette étude à conduit à la réalisation de l'environnement SyRelAn, logiciel fortement graphique permettant de réaliser un modèle sans nécessiter l'apprentissage d'un langage. Par commodité, dans la suite de ce chapitre, un modèle (resp. un composant) réalisé dans cet environnement sera qualifié de modèle (resp. composant) SyRelAn.

Un modèle SyRelAn est composé d'un modèle d'architecture du système et d'un modèle de comportement par composant du système. Ces deux types de modèles s'appuient sur des formalismes différents :

– le modèle d'architecture se présente sous la forme d'un diagramme de succès, aussi appelé diagramme de fiabilité ;
– le comportement d'un composant est représenté à l'aide d'automates (ex : si le comportement à prendre en compte varie suivant la phase de vol, il est alors possible de définir un automate par phase de vol).

Un diagramme de succès est visuellement similaire à un diagramme de dépendance mais contrairement à ce dernier qui s'intéresse à un événement redouté, un diagramme de succès décrit un état du système en logique positive.

Un modèle SyRelAn se focalisant sur les reconfigurations, seuls cinq états sont proposés pour constituer l'automate d'un composant :

– "actif" : le composant est en fonctionnement et a un rôle actif ;
– "actif chaud" : le composant est en fonctionnement et prêt à remplacer le composant "actif", son taux de défaillance est égal à celui du composant "actif" ;
– "passif chaud" : le composant est soumis à moins de contraintes que le composant "actif" jusqu'à ce que celui-ci soit perdu et que le composant "passif chaud" soit sollicité, son taux de défaillance est positif mais inférieur au taux du composant "actif" ;
– "passif froid" : le composant n'est soumis à aucune contrainte, son taux de défaillance est nul ;
– "isolé" : le composant est perdu.

Tant que le composant n'est pas dans l'état "isolé" duquel il ne peut sortir, tous les changements d'états sont possibles et réversibles.

Dans l'approche AltaRica, le comportement d'un composant ou d'un système n'est pas représenté directement sous forme d'automate mais obtenu via une description textuelle dont la sémantique est un automate. Contrairement à l'approche SyRelAn, la liberté de description offerte par AltaRica se traduit par une multitude d'automates possibles sans limite du nombre d'état possibles. Décrire un composant SyRelAn revient à sélectionner les états et transitions possibles, les conditions à satisfaire pour qu'une transition soit possible.

D'un point de vue sûreté de fonctionnement, la seule défaillance prise en compte est la perte du composant, les cas de comportement erroné sont exclus. En effet, la représentation sous forme de diagramme de succès ne prend pas en compte la propagation d'une donnée erronée mais seulement l'absence de donnée.

Des calculs de performance comparables à ceux réalisés par SyRelAn sont proposés par les outils industriels traitant des modèles écrits en AltaRica data-flow comme l'illustre l'article [BDRS06].

Les possibilités de modélisation offertes par l'environnement SyRelAn sont moins importantes que celles offertes par le langage et les outils AltaRica. Cependant il répond bien à une utilisation précise et semble conçu pour être particulièrement accessible du fait de l'interface graphique qui guide l'utilisateur.

2.4.3 EAST-ADL2

Le langage graphique Unified Modeling Language (UML), créé en 1997 et destiné à réaliser des modélisations objet utilisées en génie logiciel, a inspiré d'autres langages dédiés à d'autres domaines : System Modeling Language (SysML) créé en 2001 pour l'ingénierie de systèmes et EAST-ADL pour le développement de systèmes électroniques automobiles. L'évolution de ce dernier langage, EAST-ADL2, également dédié à la description d'architectures de systèmes embarqués pour les automobiles, a servi à l'étude décrite dans [DJL+08]. Cet article part du constat qu'une approche systématique dans la gestion des informations, la conception et la vérification d'architectures est absente dans l'automobile, et que des méthodes basées sur des modèles sont nécessaires. L'approche proposée considère un méta-modèle en EAST-ADL2 composé d'un modèle d'erreur et d'un modèle fonctionnel d'architecture, réalisé selon AUTOSAR (Automotive Open System Architecture, technologie propriétaire d'un consortium d'industriels automobiles, incluant un modèle de conception, une méthode et un outil).

Tout comme AltaRica, l'approche EAST-ADL2 considère les défaillances de composants et la propagation de ces défaillances, et peut intégrer des informations complémentaires sur chaque défaillance (sévérité, exposition...). Cependant, le comportement dysfonctionnel est séparé du comportement nominal. Cette séparation est justifiée dans [DJL+08] par de possibles problèmes durant la compréhension et la gestion de l'architecture nominale, la réutilisation de modèles ou la génération de code. L'approche AltaRica ne fait pas cette séparation. Les modèles AltaRica réalisés pour le domaine aéronautique n'ont pas pour finalité la génération de code. De plus, les modes de défaillances pris en compte pour un composant (entre deux systèmes ou entre deux avions), étant généralement les mêmes, il n'est pas nécessaire de distinguer le fonctionnel du dysfonctionnel.

Tout comme les langages graphiques qui ont progressivement contribué à sa création, EAST-ADL2 permet de représenter une grande quantité d'informations reliées par des symboles à la signification codifiée. La clarté de la représentation d'une telle approche est au détriment de son accessibilité : en effet, il est nécessaire de disposer de la légende des types de liens pour pouvoir lire un modèle et en comprendre sa globalité.

2.4.4 AADL

La SAE, mentionnée précédemment dans le chapitre 1 au sujet des ARP, est à l'origine d'Architecture Analysis and Design Language (AADL). Une annexe à la norme AADL définit la notion de modèle d'erreur pour enrichir les modèles d'architectures : un modèle d'erreurs est un automate composé d'états d'erreur reliés par des transitions. L'utilisation de modèles AADL enrichis est présentée comme un support aux méthodes citées dans 1.2.4 en réponse à [SAE96a].

Le rapport [FR07] développe les possibilités d'utilisation d'un modèle d'erreur et propose une méthodologie de modélisation d'architectures de systèmes embarqués. Cette méthodologie se décompose

en deux étapes majeures : la création de bibliothèques de modèles d'erreur puis l'instanciation de ces modèles sur des modèles d'architecture.

Tout comme en AltaRica, un modèle d'erreur d'un système est obtenu par composition des modèles d'erreurs des composants du système.

En réalité, un type de modèle d'erreur est distingué d'une implémentation d'un modèle d'erreur :

- un type de modèle d'erreur déclare les états possibles, les événements et les flux entrant/sortant ;
- une implémentation définit les transitions et les probabilités d'occurrence des événements.

La description d'un composant AltaRica est composée d'éléments comparables mais réunis dans un même objet appelé "nœud". Le rapport [FR07] justifie cette distinction entre type et implémentation par la possibilité de définir plusieurs implémentations pour un même type de modèle d'erreur.

La syntaxe d'une transition d'un modèle d'erreur rappelle celle d'une transition AltaRica sans pour autant être identique :

- la notion de transition d'un modèle d'erreur associe les changements d'état et les traitements de flux : un changement d'état correspond soit à l'occurrence d'un événement, soit à la réception d'une donnée, soit à l'émission d'une donnée (une boucle d'un état vers lui-même est néanmoins possible pour émettre une donnée sans influer sur l'état courant) ;
- le langage AltaRica sépare la notion des flux sortant, définis en fonction de l'état courant et des données reçues, de celle de changement d'état.

La syntaxe des modèles d'erreur, en apparence plus compacte, offre des possibilités de modélisation moins importantes que la syntaxe AltaRica.

Une fois la bibliothèque de modèles d'erreurs constituée, il est possible de définir un système en déclarant ses sous-composants, i.e. des instances de composants, puis en leur associant un modèle d'erreur.

3

LES ANALYSES DE SÉCURITÉ FONDÉES SUR LES MODÈLES

De nos jours, le terme *modèle* est couramment utilisé dans des domaines aussi différents que l'économie, la météorologie ou la conception aéronautique. Si les objets désignés et les formalismes diffèrent, le concept sous-jacent reste le même : un *modèle* est une représentation abstraite du réel.

Tout modèle est réalisé dans un but précis : effectuer des prévisions météorologiques, tester différents placements d'équipements à bord d'un avion... La description de l'objet à modéliser ne nécessite que les informations pertinentes au regard de l'utilisation attendue du modèle. La notion d'*abstraction* dans un modèle induit l'idée de filtrage des informations selon un besoin précis et un angle de vue spécifique.

La sûreté de fonctionnement se focalise sur la capacité d'un système à remplir une fonction en cas de défaillance et, de façon complémentaire, étudie les scénarios de pannes qui conduisent à l'incapacité de remplir cette fonction. Un modèle pour la sûreté de fonctionnement représente donc à la fois le comportement fonctionnel et dysfonctionnel d'un système.

Les projets européens de recherche ESACS [ABB+03] puis ISAAC [ABB+06] ont étudié deux approches distinctes de réalisation d'un modèle de sûreté de fonctionnement :

- la réalisation d'un modèle spécifique, dédié, dit *modèle de propagation de défaillances*, en anglais Failure Propagation Model (FPM) ;
- l'extension d'un modèle fonctionnel existant.

Ces modèles facilitent le travail d'analyse en permettant, par exemple, de simuler des défaillances, d'observer leurs conséquences, et de rechercher de façon automatique les combinaisons de défaillances qui conduisent le système dans un état redouté.

De façon générale, les FPM, comme les modèles étendus, s'inscrivent dans un processus d'évaluation de la sûreté de fonctionnement d'une architecture de système basée sur un modèle, en anglais *Model Based Design Safety Assessment (MBDSA)*.

Il est intéressant de noter que les model-checkers, dont l'utilisation se généralise avec les approches basées sur des modèles formels, ont initialement été développés afin de procéder à des analyses fonctionnelles. Les travaux menés dans des projets tels qu'ESACS et ISAAC, ont montré que le model-checking pouvait s'appliquer à des analyses dysfonctionnelles pour les besoins de la sûreté de fonctionnement des systèmes. D'autres démarches comparables ont été menées dans des cadres tels que l'optimisation ou l'évaluation de performances [GH02].

Dans un premier temps, ce chapitre présente l'approche FPM pour laquelle le langage AltaRica est tout particulièrement adéquat et propose une méthodologie de modélisation. Les concepts de la seconde approche seront présentés dans un second temps.

3.1 MODÈLE DE PROPAGATION DE DÉFAILLANCES

Un modèle de propagation de défaillance (FPM) est un modèle formel dédié à la sûreté de fonctionnement, garantissant ainsi une bonne lisibilité et un contenu pertinent. Un FPM décrit à la fois :

– le comportement dysfonctionnel : les défaillances de chaque composant et leur propagation (l'envoi de données erronées ou l'absence d'envoi, le comportement de chaque composant en fonction de données reçues incorrectes...) ;
– certains aspects du comportement fonctionnel liés aux défaillances : les reconfigurations, les surveillances permettant de détecter des erreurs...

L'université de York, dans [FMNP94], définit les principes d'écriture d'un FPM. Ces principes sont appliqués au cours du processus de modélisation suivant, illustré par la figure 3.1 :
– à partir d'une description complète du système à étudier, une analyse préliminaire permet de définir le contenu du système, d'identifier les types de composants présents à modéliser et les objectifs de l'étude (i.e. les situations à analyser via le modèle) ;
– une sélection des informations pertinentes pour l'étude (abstraction) permet d'établir, pour chaque type de composant, une spécification qui est ensuite implémentée afin d'obtenir une classe de composant par type de composant ;
– les classes de composants sont instanciées puis interconnectées pour obtenir le modèle du système, enrichi d'un observateur implémentant les situations à analyser.

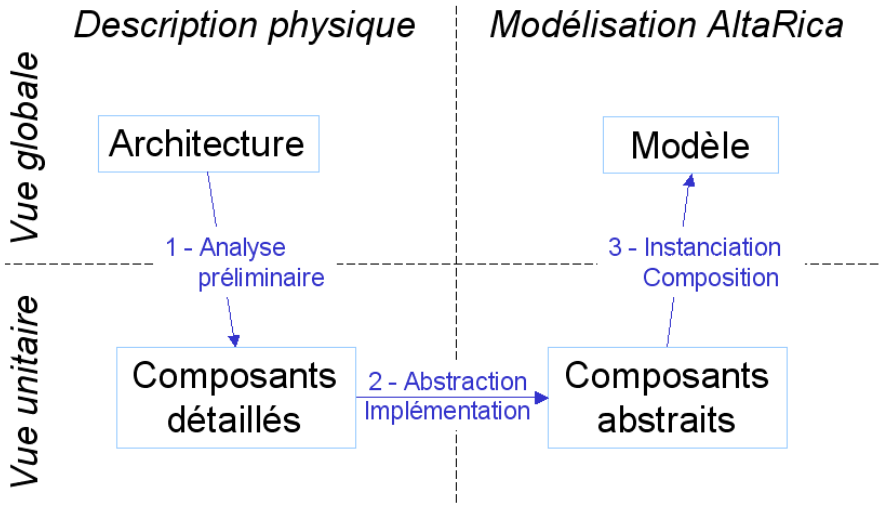


FIG. *3.1: Étapes majeures d'une modélisation*

Dans la suite de cette section, nous illustrons chacune de ces étapes du processus de modélisation avec le système de contrôle de la gouverne de direction d'un avion civil de type Airbus A340.

Exemple 3.1

Considérons le système de contrôle de la gouverne de direction. La figure 3.2 présente très simplement l'architecture du système et les liens entre composants.

Deux systèmes, absents du schéma, alimentent en énergie les différents composants du système de contrôle de la gouverne : la génération électrique et la génération hydraulique. Ils sont à considérer dans l'étude et la modélisation du système.

Pour résumer le fonctionnement de ce système :
– *trois Servocommande (SC) agissent sur la gouverne de direction, simultanément en situation nominale,*
– *chaque servocommande est nominalement contrôlée par un calculateur primaire (PRIM),*
– *lorsque chaque ligne "calculateurs primaires + servocommande" est perdue, un calculateur secondaire (SEC) prend le relais sur une servocommande,*
– *un module autonome, composé de deux équipements capables de fournir de la puissance électrique à partir de puissance hydraulique, en anglais Back-Up Power Supply (BPS), à un module de contrôle,*

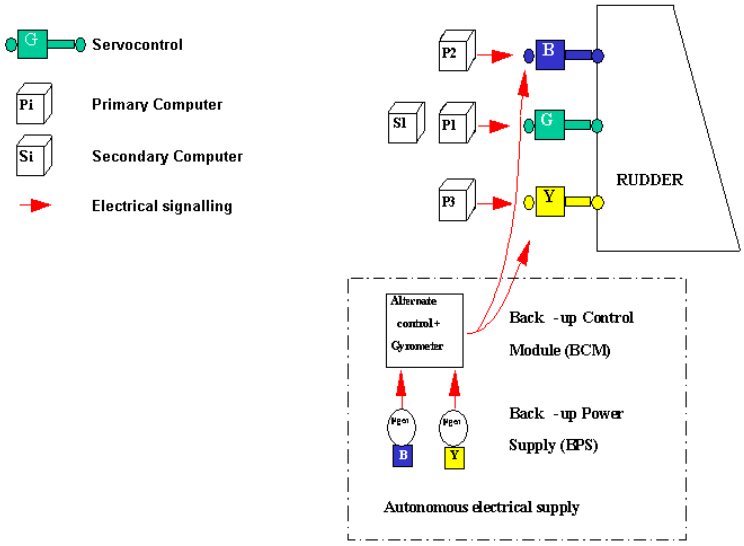


FIG. *3.2: Schéma de principe du système de contrôle de la gouverne de direction*

en anglais Back-Up Control Module (BCM), est activé lorsque tous les calculateurs, primaires et secondaires, ne contrôlent plus les servocommandes.

3.1.1 ANALYSE PRÉLIMINAIRE DU SYSTÈME : CONTENU D'UN SYSTÈME ET TYPES DE COMPOSANTS PRÉSENTS

L'architecture d'un système est un ensemble de composants physiques interconnectés. Le modèle a pour finalité de représenter fidèlement l'architecture du système proposée par le concepteur. Il est nécessaire de **recenser les composants du système réel ainsi que les liens entre composants**.

Les systèmes critiques présentent des redondances : un composant de secours peut suppléer le composant initialement actif. Pour préciser la nature des liens entre composants, les composants connectés sont différenciés selon qu'ils :

- fournissent de la puissance pour alimenter et faire fonctionner le composant observé ;
- fournissent des données en vue de l'activation du composant observé ;
- fournissent des données nécessaires au fonctionnement (donnée d'un capteur pour un calcul, ordre pour un déplacement...) ;
- reçoivent un flux du composant observé.

Le tableau 3.1 est une présentation tabulaire du contenu du système illustré sur la figure 3.2.

En observant les aspects fonctionnels/dysfonctionnels de chaque composant d'un système individuellement et la nature des données échangées sans tenir compte des composants en interface, on constate généralement que plusieurs composants sont similaires. L'approche FPM et la modélisation en langage AltaRica profitent de ce constat en s'appuyant sur le principe d'*instanciation* : un objet créé une fois peut ensuite être répliqué/instancié à volonté, chaque instance étant autonome. Ce principe permet de restreindre le nombre de composants à modéliser et donc de faciliter à la fois la validation du comportement des composants et la mise à jour des modèles (corriger une erreur sur un composant corrigera toutes les instances).

Profiter au mieux de ce principe revient à réduire autant que possible le nombre de composants à modéliser tout en conservant un comportement fidèle à la réalité. La seconde étape consiste à effectuer

Components	Power from	Activation from	Inputs from	Outputs to
Prim1	Elec.Gen.		Prim2	Servo-ctl G
			Prim3	Prim2
				Prim3
				Sec1
Prim2	Elec.Gen.		Prim1	Servo-ctl B
			Prim3	Prim1
				Prim3
				Sec1
Prim3	Elec.Gen.		Prim1	Servo-ctl Y
			Prim2	Prim1
				Prim2
				Sec1
Sec1	Elec.Gen.	Prim1		Servo-ctl G
		Prim2		
		Prim3		
Servo-ctl B	Hyd.Gen.		Prim2	
			BCM	
Servo-ctl G	Hyd.Gen.		Prim1	
			Sec1	
Servo-ctl Y	Hyd. Gen.		Prim3	
			BCM	
BCM	BPS B	Prim2		Servo-ctl B
	BPS Y	Prim3		Servo-ctl Y
BPS B	Hyd.Gen.	Prim1		BCM
		Sec1		
BPS Y	Hyd.Gen.	Prim1		BCM
		Sec1		

TAB. *3.1: Contenu du système de contrôle de la gouverne de direction*

une rapide analyse du système pour **identifier les composants à modéliser** parmi tous les composants physiques du système.

Par définition, un modèle n'est qu'une représentation abstraite du réel. Il est donc possible de faire des choix de modélisation permettant de simplifier la description de certains composants pour aboutir à des composants dits *génériques*.

Une astuce/technique de modélisation utile pour simplifier les composants impliqués dans une redondance en vue d'obtenir des composants génériques consiste à **extraire les règles d'activation et de reconfiguration** des composants impliqués et à **les centraliser dans un composant** artificiel/virtuel, uniquement présent dans le modèle. Cette centralisation est d'autant plus pratique que les règles de reconfiguration sont sujettes à évolution durant la conception d'un système.

Tout modèle est réalisé dans un but précis et la spécification de chaque composant en dépend. Il est donc nécessaire d'**identifier les situations redoutées** que le modèle doit permettre d'analyser pour pouvoir ensuite spécifier chaque composant individuellement.

Appliquons ces principes à l'architecture du système de contrôle de la gouverne de direction. Une simple observation de la figure 3.2 permet de constater que :

– les trois servocommandes peuvent être modélisées par un même composant `Servo-ctl` recevant deux ordres : l'un d'un calculateur primaire, l'autre d'un calculateur secondaire ou du BCM ;

– les deux BPS ne diffèrent que par leur source hydraulique (B pour l'un, Y pour l'autre) ; il est donc naturel de les modéliser par un même composant.

Dans le système réel, tous les calculateurs communiquent entre eux et échangent leur état de fonctionnement pour que chacun ait connaissance des calculateurs actifs, le calculateur secondaire s'activant lorsque les calculateurs primaires ne sont plus actifs. Au niveau de détail que nous souhaitons considérer, l'information importante concerne l'activation du calculateur secondaire. La communication entre calculateurs primaires n'étant pas indispensable, ces derniers peuvent être modélisés par un composant générique `Prim` ne recevant pas l'état des autres calculateurs mais émettant son état. Un composant virtuel `SecActivationCtlr` reçoit l'état de chaque calculateur primaire et transmet un seul ordre d'activation au calculateur secondaire `Sec`.

L'activation du BCM et celle des BPS sont analogues, conditionnées par la perte de deux calculateurs. La création d'un composant `BCM/BPS ActivationCtlr` recevant l'état de deux calculateurs et transmettant un ordre d'activation permet d'extraire l'activation des composants BCM et BPS et de ne considérer pour chacun qu'une entrée pour l'ordre d'activation.

La synthèse des composants physiques à modéliser est présentée dans le tableau 3.2, les composants virtuels étant présentés dans le tableau 3.3.

Components	Inputs	Outputs
Prim	Power from Elec.Gen.	Order to Servo-ctl
		Status for activation
Sec	Power from Elec.Gen.	Order to Servo-ctl
	Status for activation	
Servo-ctl	Power from Hyd.Gen.	
	Order from Prim	
	Order from Sec/BCM	
BCM	Power from BPS B	Order to Servo-ctl B
	Power from BPS Y	Order to Servo-ctl Y
	Activation order	
BPS	Power from Hyd.Gen.	Power to BCM
	Activation order	

TAB. *3.2: Composants génériques du système de contrôle de la gouverne de direction*

Components	Inputs	Outputs
SecActivationCtlr	Status from first Prim	Activation order to Sec1
	Status from second Prim	
	Status from third Prim	
ActivationCtlr	Status from first Prim	Activation order
	Status from second Prim	

TAB. *3.3: Composants virtuels de contrôle de l'activation du Sec, des BPS et du BCM*

3.1.2 ABSTRACTION DES COMPOSANTS PHYSIQUES ET IMPLÉMENTATION DES COMPOSANTS ABSTRAITS

Une fois les composants à modéliser identifiés, chaque composant doit être abstrait pour établir une spécification à implémenter en langage AltaRica. Plus répandu industriellement et plus intuitif du fait des flux orientés, le langage AltaRica OCAS est ici choisi pour décrire l'implémentation.

Pour faciliter la compréhension des concepts présentés dans la suite de cette section, nous illustrerons la création progressive d'un composant en nous appuyant sur l'exemple du calculateur secondaire, présenté dans le tableau 3.2, les points de suspension représentant les zones à compléter. Le tableau 3.2, issu d'une analyse préliminaire, montre que le calculateur secondaire dispose de deux entrées et d'une sortie. Il est donc possible, avant même de procéder à l'abstraction du comportement, de déclarer ces variables de flux en ne précisant que leur orientation et d'écrire :

```
node SEC
  ...
  flow
    PowerFromElecGen : ... : in;
    ActivationOrderReceived : ... : in;
    OrderToSc : ... : out;
  ...
edon
```

Un modèle de type FPM permet d'observer l'évolution d'un système suite à une (ou plusieurs) défaillance(s) : cela sous-entend que chaque composant peut être dans un état de fonctionnement dégradé, appelé mode défaillance, qui influe sur l'état des flux échangés entre composants. La première étape dans la réalisation d'une spécification d'un composant est donc l'**identification des modes de défaillance de chaque type de composant**.

L'Analyse des Modes de Défaillance et de leurs Effets (AMDE), ou Failure Mode and Effect Analysis (FMEA) en anglais, est une démarche classique en sûreté de fonctionnement des systèmes. Celle-ci consiste à identifier les modes de défaillance de chaque composant d'un système et leurs effets sur les autres composants de celui-ci. Cette analyse, dont le niveau de détail est très fin, est généralement associée dans l'aéronautique à un document de synthèse appelé Failure Mode and Effect Summary (FMES) en anglais.

Profitant d'années d'application de l'approche FMEA/FMES, l'université de York a proposé trois critères de classification des modes de défaillance :

- la fourniture du service ;
- la valeur produite ;
- l'aspect temporel de la fourniture du service.

Tout d'abord, il existe deux types de défaillances, contraires, impactant la fourniture du service :

- la *perte* : le service n'est pas fourni alors qu'il devrait l'être ;
- la *fonctionnement intempestif* : le service est fourni alors qu'il ne devrait pas l'être.

Lorsque le service est rendu, la qualité du service peut différer de l'attente. On considère alors que le service rendu est *erroné* et on différencie :

- l'erreur grossière : la valeur produite est hors du domaine de valeur nominale donc facile à détecter par les moyens usuels ;
- l'erreur subtile : la valeur produite reste dans le domaine des valeurs nominales, elle est donc plus difficile à détecter.

Un modèle de type FPM peut intégrer des aspects temporels. Il est alors possible de différencier le service rendu en retard du service rendu en avance. Nous n'avons pas considéré ce type de mode de défaillances dans nos travaux car ils ne sont pas pris en compte dans les analyses de sécurité réalisées chez Airbus.

En AltaRica, l'état d'un composant est modélisé par une (ou plusieurs) variable(s) d'état (champ `state`, cf. 2.2.1). Un mode de fonctionnement/défaillance correspond donc à une valeur possible pour une variable d'état. Si le niveau de granularité recherché est fin, il peut être plus pratique de définir plusieurs variables d'état (ex : une variable pour la capacité à effectuer un traitement, une variable pour l'état des surveillances...), l'état global du composant est alors une combinaison de valeurs des variables d'état. Le nombre d'états possibles pour un composant étant fini, les types respectifs des variables d'état sont des énumérations de valeurs, i.e. des ensembles finis.

Considérons l'état perdu comme seul mode de défaillance. Le composant Sec nécessite donc une variable d'état `status` pouvant prendre pour valeur `correct` ou `lost` et initialisée à `correct`. Après déclaration de cette variable d'état, le code AltaRica du composant est :

```
node SEC
  state
    status : {correct, lost};
  flow
    PowerFromElecGen : ... : in;
    ActivationOrderReceived : ... : in;
    OrderToSc : ... : out;
  ...
  init
    status := correct;
edon
```

Les variables de flux (champ `flow`, cf. 2.2.1) permettent de connecter des composants entre eux et donc de propager des défaillances :

- une donnée en entrée peut être dégradée et alors impacter le fonctionnement du composant ;
- tout mode de défaillance peut s'accompagner de valeur non nominale pour les variables de sortie propageant ainsi la défaillance.

Du fait des limitations imposées par les outils (limitations compensées par les performances qu'elles rendent possibles), les types possibles pour les variables de flux sont des ensembles finis (booléen, intervalle d'entiers, énumération). Il est donc nécessaire de discrétiser tout type : une valeur pouvant varier entre un seuil a et un seuil b sera représentée par une énumération de valeurs symbolisant les différents intervalles possibles entre a et b.

D'après le tableau 3.2, le composant Sec ne dispose que d'une sortie `OrderToSc` symbolisant un ordre envoyé à une servocommande. Cette sortie est de même type que la variable d'état `status`. Ce composant dispose en revanche de deux entrées : l'une symbolisant l'alimentation et l'autre représentant l'ordre d'activation. Ces deux entrées peuvent être de type booléen. Une fois que les types des variables de flux précédemment déclarées ont été définis, le code du composant est :

```
node SEC
  state
    status : {correct, lost};
  flow
    PowerFromElecGen : bool : in;
    ActivationOrderReceived : bool : in;
    OrderToSc : {correct, lost} : out;
  ...
  init
    status := correct;
edon
```

Une défaillance est un événement stochastique modélisé en langage AltaRica par un événement (champ `event`, cf. 2.2.2).

Des événements fonctionnels, tels que les surveillances, sont interprétés comme des changements de mode de fonctionnement suite à la réception d'un certain signal. Ces événements, de type Dirac(0), sont qualifiés d'*update* ou *événements instantanés*. Un événement instantané peut être utilisé pour introduire un délai dans une boucle (cf. opérateur *Pre* en lustre permettant de considérer la valeur au toc d'horloge précédent). La notion de Dirac est propre au langage AltaRica OCAS, cependant il devrait être possible de transposer cette notion en AltaRica LaBRI à l'aide de priorités.

L'état perdu étant le seul mode de défaillance, le composant Sec ne nécessite qu'un événement `loss` pour atteindre l'état `lost`. Déclarer cet événement revient à écrire le code AltaRica suivant :

```
node SEC
  state
    status : {correct, lost};
  flow
    PowerFromElecGen : bool : in;
    ActivationOrderReceived : bool : in;
    OrderToSc : {correct, lost} : out;
  event
    loss;
  ...
  init
    status := correct;
edon
```

A chaque événement correspond au moins une transition (champ `trans`, cf. 2.2.3), précisant les conditions d'occurrence, définie à l'aide d'une expression booléenne appelée *garde*, et les conséquences sur les variables d'état.

Le créateur du modèle peut décider qu'une défaillance ne se produit que depuis un état de bon fonctionnement, auquel cas une seule défaillance est possible.

Dans le cas d'un update, la garde précise la variable de flux et la valeur particulière considérées.

La garde d'une transition peut également permettre :

- de différencier un événement pouvant se produire avant la réception d'un signal d'activation d'un événement ne pouvant se produire qu'après réception ;
- de définir des groupes de défaillances ; par exemple par phase de vol, en définissant une variable d'état mémorisant la phase de vol courante ;
- ...

La transition la plus simple pour l'événement `loss` ne tient compte que de l'état interne représenté par la variable d'état `status` : cet événement, possible uniquement lorsque le composant est dans un état correct, conduit ce dernier dans un état perdu. Une fois cette transition définie, le code AltaRica du composant Sec est :

```
node SEC
  state
    status : {correct, lost};
  flow
    PowerFromElecGen : bool : in;
    ActivationOrderReceived : bool : in;
    OrderToSc : {correct, lost} : out;
  event
    loss;
  trans
    status = correct |- loss -> status := lost;
  ...
  init
    status := correct;
edon
```

Pour qu'un composant propage une défaillance, il doit produire une donnée symbolisant cette défaillance et la transmettre à d'autres composants.

Dans le cas d'un composant AltaRica, les variables de flux sortant jouent ce rôle. La valeur de chaque variable de flux sortant est définie à l'aide d'une assertion (champ `assert`, cf. 2.2.4) en fonction des

variables d'état et de flux entrant du composant.

L'unique variable de flux sortant, `OrderToSc`, dépend de l'état interne du composant. La transition la plus simple s'écrit alors :

```
node SEC
  state
    status : {correct, lost};
  flow
    PowerFromElecGen : bool : in;
    ActivationOrderReceived : bool : in;
    OrderToSc : {correct, lost} : out;
  event
    loss;
  trans
    status = correct |- loss -> status := lost;
  assert
    OrderToSc = status;
  init
    status := correct;
edon
```

Certaines défaillances peuvent concerner plusieurs composants simultanément. Le langage AltaRica permet à l'aide de synchronisations de modéliser ces défaillances :

- un événement ainsi que la transition correspondante sont définis pour chaque composant concerné ;
- une synchronisation est définie au niveau supérieur (composant père ou modèle) comme la conjonction des événements définis dans les composants concernés ;
- cette synchronisation ne peut se produire que si les gardes respectives des transitions définies dans les composants concernés sont satisfaites.

Ce principe de synchronisation peut s'appliquer également dans des cas non dysfonctionnels tels que la prise en compte des phases de vol : tous les composants d'un système doivent se trouver dans la même phase.

Pour affiner la description de composant, il est possible de tenir compte des flux entrants. Un calculateur ne peut émettre un ordre que s'il est alimenté, autrement dit l'assertion d'`OrderToSc` peut être enrichie. Le code du composant Sec devient alors :

```
node SEC
  state
    status : {correct, lost};
  flow
    PowerFromElecGen : bool : in;
    ActivationOrderReceived : bool : in;
    OrderToSc : {correct, lost} : out;
  event
    loss;
  trans
    status = correct |- loss -> status := lost;
  assert
    OrderToSc = case {PowerFromElecGen : status;
                      else lost};
  init
    status := correct;
edon
```

De même, il est possible de considérer que le calculateur secondaire ne peut défaillir que s'il est alimenté. Autrement dit, la garde de la transition de l'événement `loss` peut à son tour être enrichie. Le code est alors :

```
node SEC
  state
    status : {correct, lost};
  flow
    PowerFromElecGen : bool : in;
    ActivationOrderReceived : bool : in;
    OrderToSc : {correct, lost} : out;
  event
    loss;
  trans
    PowerFromElecGen and status = correct |- loss -> status := lost;
  assert
    OrderToSc = case {PowerFromElecGen : status;
                      else lost};
  init
    status := correct;
edon
```

Du fait de la variable d'activation (symbolisant la demande de sollicitation par les autres calculateurs), il serait possible de différencier la perte avant/après activation en remplaçant l'événement `loss` par deux événements `sleeping_loss` et `acting_loss` munis des transitions adéquates.

```
node SEC
  state
    status : {correct, lost};
  flow
    PowerFromElecGen : bool : in;
    ActivationOrderReceived : bool : in;
    OrderToSc : {correct, lost} : out;
  event
    sleeping_loss,acting_loss;
  trans
    PowerFromElecGen and status = correct
             and not Activation |- sleeping_loss -> status := lost;
    PowerFromElecGen and status = correct
             and Activation |- acting_loss -> status := lost;
  assert
    OrderToSc = case {PowerFromElecGen : status;
                      else lost};
  init
    status := correct;
edon
```

3.1.3 CRÉATION DU MODÈLE : INSTANCIATION DES COMPOSANTS ABSTRAITS

Une fois les composants abstraits implémentés, l'utilisateur dispose d'une librairie de composants instanciables pour composer le modèle d'un système.

Si la première étape d'analyse préliminaire permet de définir un unique composant générique comme représentation commune à plusieurs composants réels, la création du modèle commence par l'opération inverse, appelée **instanciation des composants abstraits** : un composant abstrait, implémentation d'un

composant générique, est dupliqué/instancié dans le modèle autant de fois qu'il y a de composants réels, représentés par le même composant générique, dans le système.

Il suffit ensuite de **relier les composants** entre eux pour obtenir la représentation du système physique.

Pour que le modèle soit fidèle au système, il est nécessaire d'y **intégrer les règles d'activation/reconfiguration**. Comme indiqué précédemment, ces règles peuvent être extraites des composants du système puis centralisées et implémentées dans un composant dit *contrôleur* ou, comme c'est le cas dans l'exemple du système de contrôle de la gouverne de direction, réparties dans plusieurs contrôleurs.

Un contrôleur dispose généralement d'une entrée par composant actif contribuant à l'activation d'un composant de secours, pour recevoir l'état ou un signal des composants actifs, et une sortie booléenne pour chaque composant à activer.

Enfin, pour pouvoir exploiter les outils disponibles (plus particulièrement les générations automatiques de coupes/séquences), il est nécessaire d'**implémenter les situations redoutées** identifiées durant l'analyse préliminaire du système dans un composant appelé *observateur*, défini de la manière suivante.

Une situation redoutée peut généralement être exprimée comme une combinaison de sorties de composants du système. Un observateur dispose donc d'une entrée par composant contribuant à une situation redoutée et d'une sortie booléenne par situation redoutée : la sortie a pour valeur *vrai* lorsque la situation est atteinte, *faux* dans le cas contraire.

Dans le cas du système de contrôle de la gouverne de direction, la perte totale de contrôle se traduit par la perte de contrôle des trois servocommandes, tandis que la perte partielle se traduit par la perte d'une ou deux servocommandes. Sur le modèle AltaRica correspondant, le composant `Rudder` décrit ci-dessous joue le rôle d'observateur et modélise ces situations redoutées.

```
node Rudder
  flow
    OrderFromG, OrderFromB, OrderFromY : {correct, lost} : in;
    ControlLost, ControlPartiallyLost : bool : out;
  assert
    ControlLost = ((OrderFromG = lost)
                   and (OrderFromB = lost)
                   and (OrderFromY = lost));

    ControlPartiallyLost = (not ControlLost
                            and ((OrderFromG = lost)
                                 or (OrderFromB = lost)
                                 or (OrderFromY = lost)));
edon
```

La figure 3.3 est la représentation graphique du modèle AltaRica du système de contrôle de la gouverne de direction en simulation : les liens verts symbolisent les flux ayant pour valeur `vrai` ou `correct`, tandis que les liens de couleur rouge symbolisent les flux ayant pour valeur `faux` ou `perdu`.

Un modèle AltaRica se compose du code de chaque nœud et du code du nœud `main` suivant, contenant le détail des instances de chaque nœud et les liens entre instances.

Le nœud `main` du modèle illustré sur la figure 3.3 est de la forme :

```
node Rudder
  sub
    P1, P2, P3 : Prim;
    S1 : Sec;
    G, B, Y : Sc;
    Rudder : RudderObs;
    ...
```

```
  assert
    Rudder.OrderFromG = G.Order;
    Rudder.OrderFromB = B.Order;
    Rudder.OrderFromY = Y.Order;
    ...
edon
```

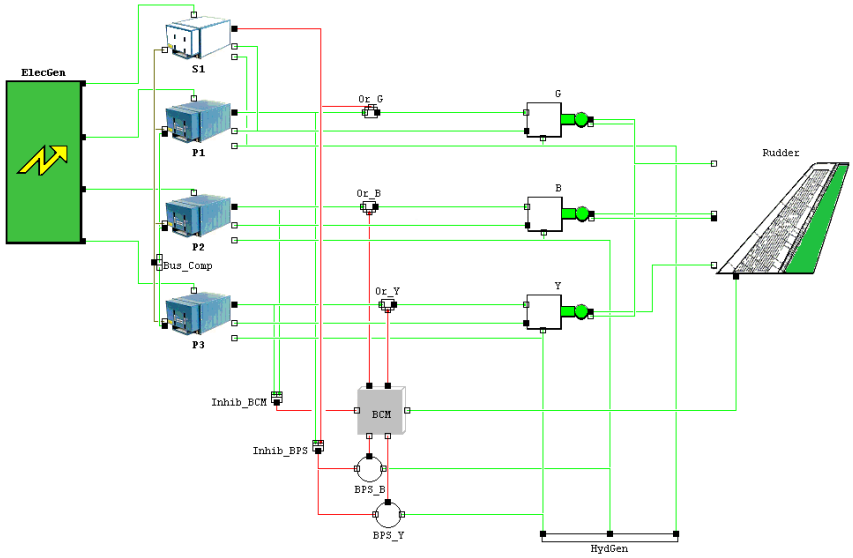


FIG. *3.3: Modèle AltaRica du système de contrôle de la gouverne de direction*

3.2 MODÈLE ÉTENDU

La conception de certains systèmes, tels que les commandes de vol, s'appuient sur des modèles formels fonctionnels. Or, tout particulièrement dans le cas de systèmes complexes, valider un modèle est une tâche coûteuse. Afin de profiter d'un modèle dont le comportement fonctionnel a été validé par un concepteur, une approche différente du FPM propose d'ajouter les aspects dysfonctionnels pour obtenir un modèle étendu [BCC+03].

Cette approche, appliquée à des modèles réalisés en Scade [DA04a, DA04b], en StateMate [HBB+04] et en Simulink [JH05, JWH05], s'appuie sur l'ajout d'un composant appelé *failure mode node* permettant d'injecter des défaillances. Ce composant dispose :

- d'une entrée pour recevoir le flux nominal ;
- d'une entrée booléenne par défaillance pour représenter l'occurrence (ou l'absence) de cette défaillance et, donc, éventuellement influer sur la valeur du flux étendu par rapport au flux nominal ;
- d'une sortie pour transmettre le flux étendu (après une éventuelle défaillance).

La société Prover a développé un model-checker, baptisé Prover Plug-In [SSS00] qui vérifie si un modèle satisfait une propriété et fournit, s'il existe, un contre-exemple invalidant cette propriété. Ce contre-exemple est une succession d'états globaux du modèle, chaque état global associant une valeur à chaque variable du modèle. Dans le cas d'un modèle étendu, l'observation des variables correspondantes aux défaillances permet d'extraire de la succession d'états un scénario de défaillance ou coupe minimale.

De leur côté, Marco Bozzano et Adolfo Villafiorita ont développé, dans le cadre du projet ESACS, la plate-forme FSAP/NuSMV-SA [BV03] basée sur le model-checker NuSMV2 [CCJ+]. Cet outil permet de

construire un modèle étendu à partir d'un modèle SMV importé en l'enrichissant à l'aide de défaillances prédéfinies dans une librairie puis d'effectuer différentes opérations : simulation, injection de défaillances, génération automatique d'arbre de défaillance...

Les valeurs des flux étant généralement des grandeurs physiques (par opposition aux valeurs abstraites de l'approche FPM), un flux sortant d'un composant ne peut être perçu comme erroné qu'en comparant sa valeur avec celles des flux entrants. Il est donc nécessaire d'ajouter un composant d'observation recevant les flux entrants et sortants de ce composant et effectuant la comparaison.

Le modèle initial étant un modèle de conception, il comporte généralement des informations plus détaillées que nécessaire pour des analyses de sûreté de fonctionnement. La lisibilité du modèle s'en trouve alors impactée.

De plus, l'ajout des nœuds "failure mode" augmente la taille du modèle et diminue les performances des model-checkers. S'ils sont trop complexes, les modèles de conception peuvent ne pas être vérifiables par model-checking. En revanche, ces modèles sont généralement simulables. Dans ce cas, une approche fondée sur la simulation de modèle étendu (cf. [BFFG04]) semble tout à fait pertinente.

Les expérimentations réalisées dans le cadre des projets ESACS et ISAAC ont montré qu'il était nécessaire de simplifier les modèles fonctionnels pour pouvoir réaliser des analyses de sécurité sur les modèles étendus.

3.3 COMBINAISON DE MODÈLES

Contrairement aux approches de type FPM ou modèle étendu qui sont basées sur un unique modèle, certains travaux s'appuient sur plusieurs modèles indépendants ou plusieurs méthodes. Parmi ces approches, nous avons retenu :

- l'approche Hierarchically Performed Hazard Origin and Propagation Studies (Hip-HOPS) [PMSH01] ;
- le formalisme Safe-SADT [CBR06] accompagné de sa méthodologie.

Dans la suite de ce chapitre, nous présentons les idées majeures de chaque approche puis évaluons leur apport en comparaison des techniques précédemment décrites.

3.3.1 HIP-HOPS

Les départements d'ingénierie des systèmes et d'informatique de l'Université de York étudient, au contact d'industriels de l'aéronautique ou de l'automobile, de nouvelles méthodes pour réaliser des analyses de sûreté de fonctionnement des systèmes. Les techniques existantes sont ainsi améliorées et coordonnées afin d'assurer la cohérence des résultats. Yiannis Papadopoulos, dans [PMSH01], propose la méthode Hip-HOPS pour automatiser la génération d'arbres de défaillances. Cette méthode est composée de trois étapes :

1. les défaillances fonctionnelles d'un système sont analysées sur un modèle fonctionnel ; ce modèle, initialement abstrait, est progressivement raffiné au fur et à mesure que le système est décomposé en sous-systèmes, puis en composants de base, jusqu'à constituer un *modèle hiérarchique* du système ;
2. le comportement de chaque composant du système est analysé : une table, qualifiée de Modèle du Comportement Défaillant Local (MCDL), recense les modes de défaillance observables sur les sorties du composant étudié, puis en détermine les causes (défaillance interne ou réception de flux non nominaux) ;
3. l'arbre de défaillance de chaque défaillance fonctionnelle est automatiquement créé en s'appuyant sur la structure du modèle hiérarchique et des informations du MCDL de chaque composant.

Le modèle hiérarchique est présenté graphiquement à l'aide de diagrammes de flux, tandis que le MCDL s'appuie sur une description textuelle organisée sous forme de table.

Un outil [PM01] a été implémenté permettant d'importer un modèle Simulink, offrant directement les informations nécessaires au modèle hiérarchique, de le compléter par les informations de type MCDL puis

de générer des arbres de fautes compatibles avec l'outil commercial Fault Tree +.

Le modèle hiérarchique ne nécessitant que des informations sur l'architecture du système (composants et connexions), il serait envisageable d'étendre les types de modèles importables à d'autres langages que Simulink. Cette possibilité d'exploiter un modèle existant représente un gain de temps. Néanmoins, l'étude de chaque composant reste une étape importante et coûteuse qui doit être effectuée manuellement. De plus, les aspects dysfonctionnels étant totalement décorrélés du modèle d'architecture, cette méthode n'a d'autre but que de générer des arbres de défaillances et ne permet pas, par exemple, de simuler le système afin d'observer une défaillance.

3.3.2 SAFE-SADT

Le Laboratoire d'Automatique, de Mécanique et d'Informatique industrielles et Humaines (LAMIH) de Valenciennes s'est intéressé aux phases et aux méthodologies existantes de conception de systèmes complexes. Estimant que l'ingénierie des systèmes doit s'intéresser à l'optimisation de la conception plutôt qu'à la minimisation de problème, le LAMIH a élaboré une méthode de conception [CBR06] intégrant les études quantitatives de fiabilité et la gestion de risques. Cette méthode se décompose en six étapes :

1. la réalisation d'un modèle fonctionnel ;
2. la décomposition des fonctions en sous-fonctions élémentaires ;
3. le choix des équipements ;
4. la projection/allocation des fonctions sur les équipements ;
5. l'amélioration des choix d'équipements suivant des critères de fiabilité ;
6. la validation globale.

Ne trouvant aucun formalisme répondant à toutes leurs attentes, les chercheurs du LAMIH ont créé leur propre formalisme, baptisé *Safe-SADT* en s'inspirant de la méthode d'analyse fonctionnelle baptisée Structure Analysis and Design Technique (SADT). Ce langage graphique tire profit des techniques de décomposition fonctionnelle et s'appuie sur la simulation de Monte Carlo pour étudier l'évolution des paramètres de fiabilités du système modélisé. Un diagramme Safe-SADT, composé de boîtes connectées par des flèches, considère quatre types de données d'entrée :

1. les fonctions à réaliser ;
2. les équipements disponibles ;
3. les événements (contrôlables ou non) ;
4. des contraintes/critères (ex : objectifs de coût ou temps de réponse).

Un opérateur de projection permet de formaliser l'allocation des fonctions sur les équipements. Les données en sortie d'une boîte Safe-SADT sont les résultats de la projection/allocation choisie :

- le mode nominal de fonctionnement ;
- le mode dégradé de fonctionnement ;
- le temps de réponse en mode nominal et en mode dégradé ;
- le coût ;
- les événements non contrôlés après projection/allocation dont la propagation peut impacter la fiabilité d'autres fonctions.

L'approche Safe-SADT part de l'hypothèse qu'un logiciel ou un équipement est, soit opérationnel, soit défaillant : l'automate sous-jacent à un logiciel ou un équipement est donc constitué de deux états reliés par deux arcs, respectivement paramétrés par un taux de défaillance et un taux de réparation. Une fois l'allocation définie, l'automate global d'une boîte Safe-SADT est implicitement calculé de façon à pouvoir utiliser la simulation de Monte Carlo.

Cette approche, illustrée sur un système hydraulique, décompose clairement les données à modéliser (fonctions, composants et architecture, allocation). Cependant, l'allocation/projection reste une opération manuelle alors que des techniques [Sag08] permettant de produire automatiquement des allocations en fonction de contraintes existent.

D’autre part, l’hypothèse qu’un logiciel ou un composant ne dispose que de deux états réduit le comportement pris en compte à un unique mode de défaillance. En pratique, les composants des systèmes auxquels nous nous sommes intéressés peuvent disposer de plusieurs modes de défaillances. Le langage AltaRica a l’avantage de ne pas limiter le nombre de modes de défaillance considéré.

4

LE RAFFINEMENT POUR ALTARICA

Le développement d'un système industriel suit un processus incrémental : des exigences initiales sont transposées en une architecture préliminaire simple qui est progressivement affinée jusqu'à obtenir une architecture à implémenter/réaliser/installer.

Dans le chapitre 1, nous avons présenté les analyses de sûreté de fonctionnement menées sur les architectures des systèmes. De plus, nous avons montré dans le chapitre 3 que ces analyses peuvent être supportées par des méthodes formelles permettant de vérifier de manière rigoureuse des propriétés via un modèle formel.

Cependant, un modèle ne représente qu'une architecture : toute modification d'architecture nécessite la réalisation d'un nouveau modèle. Effectuer une analyse de la sûreté de fonctionnement d'un système à l'aide d'un modèle, en anglais MBDSA, à chaque étape du processus de développement requiert de créer un modèle par étape, ce qui implique une forte charge de validation individuelle de chaque modèle. Pour rendre l'approche MBDSA réellement industrialisable, il est nécessaire de réduire la charge de validation en complétant les méthodes existantes précédemment présentées de techniques permettant de faciliter les validations successives.

Les domaines du génie logiciel et des méthodes formelles (particulièrement le langage B) disposent d'une technique baptisée *raffinement* analogue au processus incrémental de développement de système industriel : une spécification abstraite est détaillée progressivement en respectant certaines règles qui garantissent la cohérence vis-à-vis de la spécification précédente et, par récurrence, de la spécification initiale. Le langage AltaRica pouvant être utilisé pour les analyses de type MBDSA mais aucun raffinement n'étant jusqu'alors défini pour ce langage, notre objectif initial fut d'en définir un.

Le langage AltaRica offre la possibilité de définir un nœud N contenant d'autres nœud N_i : N est *composé* des différents N_i. La notion de *compositionnalité* désigne le fait que si deux nœuds N_i et N_i' sont en relation, alors le nœud N', obtenu en remplaçant N_i par N_i' dans N, et le nœud N sont également en relation. Cette notion permet de raisonner sur les composants N_i puis de déduire des propriétés sur N.

Pour affiner cet objectif initial, nous avons cherché à préciser le besoin en tenant compte de l'existant en matière de raffinement et des problématiques propres aux pratiques industrielles.

Les recherches initiales, ainsi que l'article [GFL05], ont permis de distinguer le raffinement algorithmique, dont l'ajout progressif de détails depuis une formulation abstraite permet d'obtenir une description proche du code, et le raffinement de données (ou dit de détails) permettant l'ajout de nouveaux événements ou de nouvelles variables d'état.

Dans le cas du langage AltaRica, nous considérons également deux formes majeures d'enrichissement de modèle :

- l'ajout de détails à granularité constante : les transitions et assertions sont progressivement détaillées, sans ajouter de nouveau nœud ;
- le raffinement par ajout de hiérarchie, donc de nouveaux nœuds, impliquant à la fois l'ajout de nouvelles variables et de nouveaux événements.

Le raffinement établit une relation entre 2 modèles. Les systèmes de transitions ST étant sous-jacents aux modèles AltaRica, nous avons choisi d'étudier des relations entre ST. Dans son mémoire de thèse

définissant la sémantique du langage AltaRica [Poi00], Gérald Point s'intéresse à la relation de bisimulation forte [Par81]. Cette relation très contraignante est trop restrictive pour notre approche, néanmoins elle a permis à Gérald Point d'établir un théorème de compositionnalité permettant d'utiliser dans un modèle indifférent un nœud $\mathcal{A}$ ou le nœud $\mathcal{A}'$, dès lors qu'ils sont bisimilaires, tout en conservant certaines propriétés. La compositionnalité étant très présente dans les modèles AltaRica, cette propriété est donc particulièrement intéressante. C'est pourquoi nous avons cherché des relations garantissant cette propriété de compositionalité mais moins contraignantes que la bisimulation.

D'un point de vue applicatif, la sûreté de fonctionnement s'intéresse aux séquences d'événements possibles sur le ST d'un modèle. La génération de séquences pouvant être coûteuse, nous avons souhaité que le raffinement AltaRica préserve ces séquences ou du moins permette de calculer les séquences du modèle raffiné à partir des séquences du modèles abstrait.

Dans un premier temps, nous nous sommes intéressés à la relation de simulation forte, relation généralement associée à la notion de raffinement. Cependant, la simulation établit un lien d'événement à événement qui n'est pas naturel lorsqu'un nœud est remplacé par une hiérarchie de nœuds (introduire de la hiérarchie s'accompagne généralement d'une décomposition d'un événement initial abstrait en une séquence d'événements de plus bas niveau). Nous avons donc souhaité étudier également une relation dite faible : certains événements sont considérés comme non observables, les équivalences sont établies sur des chemins et non des événements unitaires. Différentes relations [GW96, BES94] existent selon que les événements non observables précèdent un événement observable et/ou lui succèdent. Nous avons choisi d'étudier une relation de ce type qui se nomme *simulation quasi-branchante*.

La simulation quasi-branchante tient compte de la visibilité d'un événement. La notion de visibilité des événements a précédemment été mentionnée comme une information formalisée par un attribut dans le langage AltaRica extended. Cependant, cet attribut a été ajouté dans le but d'enrichir la description pour des outils dédiés, et non pour changer la sémantique d'un modèle. L'étude de la simulation quasi-branchante nous conduit à proposer une sémantique intégrant cette notion. La définition d'une nouvelle extension d'AltaRica nécessiterait la mise à jour des outils existants. Ce travail dépasse les objectifs de cette thèse. Néanmoins, étant convaincus de l'intérêt de cette relation, nous avons choisi de compléter la description AltaRica par des relations MecV, pour valider notre étude.

Pour résumer, nous avons cherché à définir un raffinement AltaRica utile pour les analyses de sûreté de fonctionnement et répondant aux besoins suivants :

- garantir la propriété de compositionnalité établie par Gérald Point pour AltaRica,
- préserver les séquences d'événements ou permettre de calculer les séquences du modèle raffiné à partir des séquences du modèle abstrait.

Ce chapitre présente, tout d'abord, le langage AltaRica et la bisimulation tels que définis formellement par Gérald Point [Poi00]. Nous détaillons ensuite nos travaux sur la relation de simulation puis sur la relation de simulation quasi-branchante.

4.1 DÉFINITIONS INITIALES

Le langage AltaRica permet de modéliser hiérarchiquement des systèmes. Dans la suite de cette section, le terme *composant* désigne l'objet le plus simple en AltaRica tandis que le terme *nœud* désigne un objet contenant des composants. Tout objet est un nœud AltaRica dont la sémantique est un système de transitions interfacé.

4.1.1 SYSTÈMES DE TRANSITIONS INTERFACÉS

Le langage AltaRica autorise les communications entre objets. Pour cela, des flux entre objets, sur lesquels il est possible d'instaurer des contraintes, sont considérés. D'autre part, le langage est dit événementiel : chaque transition permettant de faire évoluer l'état d'un objet est étiqueté par un événement. Ces

deux particularités (notions de flux et d'événements) conduisent à considérer des systèmes de transitions interfacés et non des systèmes de transitions étiquetés "simples" [Mil80].

Définition 4.1.1 (Système de transitions interfacé)
Un système de transitions interfacé *(STI) est un quintuplet* $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ *où :*

- $E = E_+ \cup \{\epsilon\}$ *est un ensemble fini d'événements où* ϵ *est un événement distingué n'appartenant pas à* E_+*.*
- F *est un ensemble de valeurs de flux.*
- S *est l'ensemble des états du STI*
- $\pi : S \rightarrow 2^F$ *est une application qui associe à tout état l'ensemble des valeurs de flux qu'il autorise. De plus, nous supposons que pour tout* $s \in S$, $\pi(s) \neq \emptyset$*.*
- $T \subseteq S \times F \times E \times S$ *est la relation de transition supposée vérifier :*
  - $(s_1, f, e, s_2) \in T \Rightarrow f \in \pi(s_1)$
  - $\forall s \in S, \forall f \in \pi(s), (s, f, \epsilon, s) \in T$*.*

Une configuration *d'un STI est un couple* $(s, f) \in S \times F$ *tel que* $f \in \pi(s)$*.*

□

Le langage AltaRica offre la possibilité d'établir des priorités entre événements. Il est possible que 2 transitions correspondant à 2 événements distincts aient pour origine une même configuration. Cependant, si l'un d'eux est défini comme plus prioritaire que le second, alors seule la transition étiquetée par l'événement prioritaire sera réellement possible. Cela est défini par la notion d'ordre partiel introduite ci-dessous.

Définition 4.1.2 (STI avec priorités)
Soit $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ *un STI et* $<$ *un ordre partiel strict sur* E*. Nous notons* $\mathcal{A}\restriction<= \langle E, F, S, \pi, T\restriction< \rangle$ *le STI dont l'ensemble des transitions est défini par* $(s_1, f, e, s_2) \in T\restriction<$ *si et seulement si* $(s_1, f, e, s_2) \in T$ *et pour tout* $s_k \in S$ *et pour tout* $e_k \in E$, $(s_1, f, e_k, s_k) \in T \Rightarrow e \not< e_k$*.*

□

4.1.2 SÉMANTIQUE DES COMPOSANTS

Dans la suite nous considérons que les expressions et formules utilisées dans le langage sont des termes ou des formules d'un langage du premier ordre ; nous notons $\mathbb{F}$ l'ensemble des formules et $\mathbb{T}$ l'ensemble des termes. Pour tout élément φ de $\mathbb{F} \cup \mathbb{T}$ nous notons $vlib(\varphi)$ l'ensemble de ses variables libres. Nous supposons que $\mathbb{F}$ contient deux symboles de prédicats d'arité nulle *tt* et *ff* qui s'interprètent comme les constantes booléennes, respectivement, *vrai* et *faux*.

Intuitivement, la formule, appelée garde, d'une transition doit être vraie pour que celle-ci soit franchissable. Les variables considérées dans la garde sont interprétées dans un domaine concret. Ce domaine constitue le domaine d'interprétation de la garde.

Définition 4.1.3 (Composants – Syntaxe abstraite)
Un composant *est un septuplet* $\mathcal{C} = \langle V_S, V_F, V_L, E, A, M, < \rangle$ *où*

- V_S*,* V_F *et* V_L *sont des ensembles de variables deux à deux disjoints, appelés respectivement, ensembles des variables d'*états*, de* flux *et* flux non observables *; nous notons* $V_\mathcal{C}$ *l'union de ces ensembles.*
- $E = E_+ \cup \{\epsilon\}$ *est un ensemble d'*événements *où* ϵ *est un événement n'appartenant pas à* E_+*.*
- $A \in \mathbb{F}$ *est l'*assertion *du composant ; elle vérifie* $vlib(A) \subseteq V_\mathcal{C}$ *;*
- $M \subseteq \mathbb{F} \times E \times \mathbb{T}^{V_S}$ *est un ensemble de* macro-transitions (g, e, a) *telles que :*
  - $g \in \mathbb{F}$ *est une formule appelée* garde *; elle doit vérifier* $vlib(g) \subseteq V_\mathcal{C}$ *;*
  - $e \in E$ *est l'événement déclenchant la transition ;*
  - $a : V_S \rightarrow \mathbb{T}$ *est une application qui associe à toute variable d'état un terme* $a(s)$ *tel que* $vlib(a(s)) \subseteq V_\mathcal{C}$*.*

  L'ensemble M *contient implicitement la macro-transition* $(g_\epsilon, \epsilon, a_\epsilon)$ *où* $g_\epsilon =$ *tt et* a_ϵ *est l'identité sur* V_S*.*

– $<$ *est une relation d'ordre partielle stricte sur* E *telle que* ϵ *soit incomparable à tout autre événement (i.e. pour tout* $e \in E$, $e \not< \epsilon$ *et* $\epsilon \not< e$*).*

□

Exemple 4.1

Soit $n\mathcal{A}$ *un nœud AltaRica défini par le code ci-dessous :*

```
node n
  state s:[0,2];
  flow  f:[0,2];
  event e1,e2;
  trans
    s = 0 |- e1 -> s := 1;
    s = 0 |- e2 -> s := 2;
  assert
    s = 0 => (f = 0);
    s = 1 => (f = 1);
    s = 2 => (f != 1);
  init  s := 0;
edon
```

La syntaxe abstraite de ce nœud est :
- $V_S = \{s\}$ *et* $D_s = \{0, 1, 2\}$ *;*
- $V_F = \{f\}$ *et* $D_f = \{0, 1, 2\}$ *;*
- $V_L = \emptyset$ *;*
- $E = \{e_1, e_2, \epsilon\}$ *;*
- $A = (s = f)|(s = 2 \ \& \ f = 0)$ *;*
- $M = \{((s = 0), e_1, (s = 1)), ((s = 0), e_2, (s = 2)), (true, e_2, a_\epsilon)\}$.

Désignons par s_{ij} *les états de* n *tels que* $s = i$ *et* $f = j$*. La sémantique du nœud* $n\mathcal{A}$ *est représentée par le STI ci-après (par souci de lisibilité, les transitions* ϵ *d'un état sur lui même ne sont pas représentées) :*

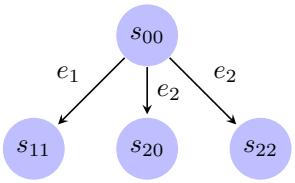


FIG. *4.1: Sémantique du nœud* n

Dans la suite de ce chapitre, nous nous contenterons de représenter la syntaxe concrète (code AltaRica) et la sémantique des nœuds dans les exemples à suivre.

L'événement ϵ est supposé incomparable aux autres événements du composant. Cette hypothèse se justifie par la sémantique intuitive de cet événement : il modélise une évolution de l'environnement alors que le composant ne change pas d'état. Concrètement, l'événement ϵ ne doit ni bloquer d'autres événements, ni être bloqué. Pour un même système, un modèle de composants peut être utilisé dans des contextes différents ; une "bonne" modélisation d'un type de composants devrait alors être élaborée sans hypothèse particulière sur son contexte. Supposer qu'un événement soit plus (ou moins) prioritaire que ϵ va à l'encontre de ce principe.

Pour faciliter la définition de la sémantique des composants et des nœuds AltaRica, nous supposons que toutes les variables prennent leurs valeurs dans un même domaine $\mathcal{D}$. Une valuation d'un ensemble de variables X est une application de X dans $\mathcal{D}$ et l'ensemble des valuations de X est noté $\mathcal{D}^X$. Toute

formule $\varphi \in \mathbb{F}$ s'interprète par rapport à un ensemble de variables X tel que $vlib(\varphi) \subseteq X$. La sémantique d'une formule φ est alors un sous-ensemble $[\![\varphi]\!]$ de $\mathcal{D}^X$; nous posons $[\![tt]\!] = \mathcal{D}^X$ et $[\![ff]\!] = \emptyset$. De manière analogue, un terme t s'interprète par une fonction $[\![t]\!]$ de $\mathcal{D}^X$ dans $\mathcal{D}$.

La sémantique d'un composant est définie en deux étapes : sans priorité puis avec priorités.

Définition 4.1.4 (Composants sans priorité - Sémantique)
La sémantique d'un composant $\mathcal{C} = \langle V_S, V_F, V_L, E, A, M, \emptyset\rangle$ est un système de transitions interfacé $[\![\mathcal{C}]\!] = \langle E, F, S, \pi, T\rangle$ construit de la manière suivante :

- $F = \mathcal{D}^{V_F}$
- $S = \{s \in \mathcal{D}^{V_S} \mid \exists f \in \mathcal{D}^{V_F}, l \in \mathcal{D}^{V_L}, (s, f, l) \in [\![A]\!]\}$
- $\pi : S \to 2^F$ *est définie par* $\pi(s) = \{f \mid \exists l \in \mathcal{D}^{V_L}, (s, f, l) \in [\![A]\!]\}$
- $T \subseteq S \times F \times E \times S$ *est telle que* $T = [\![M]\!]$ *où* $[\![M]\!] = \bigcup_{t\in M} [\![t]\!]$, *et où* $[\![t]\!]$ *est défini comme suit :* $[\![(g, e, a)]\!] = \{(s_1, f, e, s_2) \mid \exists l \in \mathcal{D}^{V_L}, (s_1, f, l) \in [\![A \wedge g]\!] \wedge s_2 = a[s_1, f, l]\}$ *où* $a[s_1, f, l]$ *dénote la valuation des variables d'état telle que pour tout* $v \in V_S$, $a[s_1, f, l](v) = [\![a(v)]\!](s_1, f, l)$.

□

Dans la définition 4.1.4, la sémantique $[\![(g, e, a)]\!]$ d'un triplet (g, e, a) est l'ensemble des transitions :

- pour lesquelles la garde g est vraie ;
- qui sont étiquetées par l'événement e ;
- dont les affectations a respectent les domaines des variables d'états concernées.

On notera que pour tout $s \in S$, $\pi(s) \neq \emptyset$ et que la sémantique de l'unique macro-transition étiquetée ϵ est l'ensemble $\{(s, f, \epsilon, s) \mid f \in \pi(s)\}$.

Si $f \in \mathbb{F}$, $x_i \in \mathcal{V}$ et $t_i \in \mathbb{T}$ (pour $i = 1 \ldots n$) alors nous notons $f[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$ la formule obtenue en substituant simultanément toutes les occurrences des x_i par t_i (pour $i = 1 \ldots n$) dans f.

Définition 4.1.5 (Composants - Sémantique)
Soit $\mathcal{C} = \langle V_S, V_F, V_L, E, A, M, <\rangle$ un composant. Nous supposons que $V_S = \{s_1, \ldots, s_n\}$, $V_F \cup V_L = \{f_1, \ldots, f_p\}$ et nous introduisons un nouvel ensemble de variables $F' = \{f'_1, \ldots, f'_p\}$.

Posons $M\restriction< $ l'ensemble des macro-transitions construit de la manière suivante :

$$\langle G, e, a\rangle \in M\restriction< \iff \exists t = \langle g, e, a\rangle \in M \wedge G = g \wedge \bigwedge_{e<e'} \neg V_E(e')$$

où pour tout événement $e \in E$, $V_E(e)$ est la formule :

$$V_E(e) = \bigvee_{\langle g,e,a\rangle\in M} V_M(\langle g, e, a\rangle)$$

et pour toute transition $t = \langle g, e, a\rangle \in M$, $V_M(t)$ est la formule :

$$V_M(t) = g \wedge \exists f'_1, \ldots, f'_p, A[f_1 \leftarrow f'_1, \ldots, f_p \leftarrow f'_p][s_1 \leftarrow a(s_1), \ldots, s_n \leftarrow a(s_n)]$$

(On notera que $vlib(G) \subseteq V_{\mathcal{C}}$ puisque toutes les variables f'_i sont quantifiées.)
La sémantique de $\mathcal{C}$ est la sémantique du composant sans priorité $\langle V_S, V_F, V_L, E, A, M\restriction<, \emptyset\rangle$.

□

Intuitivement, $V_M(t)$ exprime la conjonction de la garde et des post-conditions liées aux affectations de la transition t. D'autre part, $V_E(e)$ exprime l'ensemble des formules $V_M(t)$ pour les transitions t étiquetées par l'événement e. Le principe des priorités est qu'une transition n'est possible depuis un état que si, depuis ce même état, il n'existe aucune transition possible étiquetée par un événement plus prioritaire.

4.1.3 SÉMANTIQUE DES NŒUDS

Le langage AltaRica permet de synchroniser des événements définis dans des composants différents : les événements synchronisés sont tirés simultanément dès lors que leurs gardes respectives sont satisfaites, i.e. une transition unique représente les effets cumulés des transitions de chaque événement considéré dans la synchronisation, prises indépendamment les unes des autres depuis l'état initial de la synchronisation. Ces vecteurs peuvent évidemment être ordonnés pour représenter des priorités.

Le concept de diffusion a été ajouté au langage pour permettre de décrire facilement les défaillances de cause commune, très usitées en sûreté de fonctionnement. Un vecteur de synchronisation avec diffusion est un vecteur de synchronisation dans lequel certains événements sont optionnels au sens obligatoire si possible, mais n'empêchent pas la synchronisation s'ils sont impossibles. Les événements optionnels sont suffixés par " ?".

Définition 4.1.6 (Nœuds – Syntaxe abstraite)
Un nœud est un $(n+5)$*-uplet* $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \ldots, \mathcal{N}_n, V\rangle$ *où :*

- V_F *est un ensemble de variables de flux ;*
- $E = E_+ \cup \{\epsilon\}$ *est un ensemble d'événements où* ϵ *est un événement (implicite) distingué de ceux de* E_+ *;*
- $<$ *est un ordre partiel strict sur* E *tel que* ϵ *soit incomparable à tout autre événement de* E *;*
- *Pour* $i = 1 \ldots n$*,* $\mathcal{N}_i$ *est un composant ou un nœud ;* V_{Fi} *est son ensemble de variables de flux,* E_i *est son ensemble d'événements. De plus, nous supposons que leurs ensembles de variables sont disjoints ;*
- $\mathcal{N}_0$ *est un composant particulier appelé* contrôleur*. Son ensemble d'événements est* $E_0 = E$ *ordonné par la relation vide (i.e. tous ses événements sont incomparables). Son ensemble de variables de flux est* $V_{F0} = V_F \cup V_{F1} \cup \cdots \cup V_{Fn}$*.*
- $V = V_d \cup V_{imp}$ *est un ensemble de vecteurs de synchronisation avec diffusion où :*
  - $V_d \subseteq E_0^? \times \cdots \times E_n^? \times 2^{[0,n+1]}$ *est l'ensemble des vecteurs spécifiés. Pour tout* $i \geq 0$*,* $E_i^?$ *est l'ensemble défini par* $E_i^? = E_i \cup \{e? \mid e \in E_i - \{\epsilon\}\}$*.*
  - $V_{imp} \subseteq E_0 \times \cdots \times E_n \times \{0\}$ *est un ensemble de vecteurs implicites construit de la manière suivante :*
    - $\langle \epsilon, \ldots, \epsilon, \{0\}\rangle \in V_{imp}$*.*
    - $\forall i \in [0,n]$*,* $\forall e_i \in E_i \setminus \{\epsilon\}$*,* $[\forall \langle e'_0, \ldots, e'_i, \ldots, e'_n, D\rangle \in V_d,\ e'_i \notin \{e_i, e_i?\}] \Rightarrow \langle \epsilon, \ldots, e_i, \ldots, \epsilon\rangle \in V_{imp}$*. Le vecteur* $\langle \epsilon, \ldots, e_i, \ldots, \epsilon, \{0\}\rangle$ *ne contient que des* ϵ *hormis* e_i *placé à la* i*-ème position. Si un événement du sous-nœud* $\mathcal{N}_i$ *n'est pas impliqué dans une synchronisation alors cet événement doit se produire seul (tous les autres doivent faire* ϵ*).*

□

Les vecteurs sont implicitement complétés par des ϵ (pour les sous-composants qui n'interviennent pas dans la synchronisation par une action différente de ϵ). Ici, afin de simplifier les notations, nous considérons directement des vecteurs à $n+1$ composantes.

Un vecteur de synchronisation avec diffusion définit un ensemble partiellement ordonnés de vecteurs de synchronisation d'Arnold et Nivat [AN82]. Chaque vecteur de cet ensemble est appelé une *instance* du vecteur avec diffusion.

Définition 4.1.7 (Instances d'un vecteur de synchronisation)
Soit $v = \langle e'_0, \ldots, e'_n, D\rangle \in V$ *un vecteur de synchronisation avec diffusion. Une instance de* v *est un vecteur* $u \in E_0 \times \cdots \times E_n$ *tel que* $u = \langle e_0, \ldots, e_n\rangle$ *et*

- *Pour tout* $i \in [0,n]$*,*
  - $e'_i \in E_i \Rightarrow e_i = e'_i$ *(les événements non marqués apparaissent nécessairement dans l'instance).*
  - $e'_i = b? \Rightarrow e_i = b \vee e_i = \epsilon$ *(les événements marqués peuvent être remplacés par* ϵ*).*
- *Le cardinal de l'ensemble* $\{i | 0 \leq i \leq n, e_i \neq \epsilon \wedge e'_i = e_i?\}$ *appartient à* D*.*

L'ensemble $Inst(v)$ *des instances de* v *est ordonné par* $\sqsubset$ *où :*
$\langle e_0, \ldots, e_n\rangle \sqsubset \langle e'_0, \ldots, e'_n\rangle$ *si et seulement si* $\{e_i \mid e_i \neq \epsilon\} \subset \{e'_i \mid e'_i \neq \epsilon\}$*.*

□

La sémantique d'un nœud AltaRica s'obtient en construisant le produit synchronisé des sous-nœuds et du contrôleur de telle manière que les contraintes sur les flux imposées par le contrôleur soient respectées. Après cette construction, deux « filtres » sont appliqués sur l'ensemble des transitions du produit : le premier consiste à conserver les transitions étiquetées par les instances de vecteur les plus prioritaires ; le second consiste à éliminer les transitions les moins prioritaires par rapport à l'ordre sur les actions du contrôleur.

Par analogie avec le produit synchronisé d'Arnold et Nivat, cette opération a été appelée *produit contrôlé* (de systèmes de transitions interfacés) [AGPR99a].

Définition 4.1.8 (Nœuds – Sémantique)
Nous considérons un nœud AltaRica $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \ldots, \mathcal{N}_n, V\rangle$. *Nous notons* $[\![\mathcal{N}_i]\!] = \langle E_i,\ F_i,\ S_i,\ \pi_i,\ T_i\rangle$ *la sémantique de* $\mathcal{N}_i$ *(pour i=1... n).*

La sémantique du nœud $\mathcal{N}$ *est le STI* $[\![\mathcal{N}]\!] = \langle E, F, S, \pi, T\rangle$ *construit de la manière suivante :*

- $F = \mathcal{D}^{V_F}$
- $S = \{s \in S_0 \times \cdots \times S_n \mid \pi(s) \neq \emptyset\}$
- $\pi(\langle s_0, \ldots, s_n\rangle) = \{f \in \mathcal{D}^{V_F} \mid \exists \langle f, f_1, \ldots, f_n\rangle \in \pi_0(s_0), \forall i,\ f_i \in \pi_i(s_i)\}$
- $T \subseteq S \times F \times E \times S$ *est construit de la manière suivante :*
  - *On construit l'ensemble* $W = \bigcup_{v \in V} Inst(v) \times \{v\}$ *et on munit cet ensemble de deux ordres partiels,* $<_?$ *et* $<_0$*, définis par :*
    - $(u, v) <_? (u', v') \iff v = v' \wedge u \sqsubset u'$
    - $(u, v) <_0 (u', v') \iff u = (e_0, \ldots, e_n) \wedge u' = (e'_0, \ldots, e'_n) \wedge e_0 < e'_0$
  - *On construit ensuite l'ensemble des transitions* $T_{\mathcal{N}} \subseteq S \times F \times W \times S$ *défini par* $\langle (s_{1_0}, \ldots, s_{1_n}), f, (u, v),\ (s_{2_0}, \ldots, s_{2_n})\rangle \in T_{\mathcal{N}}$ *si et seulement si il existe* $f_0 = (f,\ f_1,\ \ldots,\ f_n) \in \pi_0\,(s_0)$ *et* $u = (e_0, \ldots, e_n)$ *et pour tout* $i \in [0, n]$, $(s_{1_i}, f_i, e_i, s_{2_i}) \in T_i$ *et* $f_i \in \pi_i(s_{1_i})$.
  - *Enfin la relation* T *est définie par :*
    $\langle s_1, f, e_0,\ s_2\rangle \in T$ *si et seulement si il existe* $(e_0,\ \ldots, e_n, v) \in W$ *tel que* $\langle s_1,\ f,\ (e_0,\ \ldots,\ e_n,\ v),\ s_2\rangle \in (T_{\mathcal{N}} \restriction <_?) \restriction <_0$.

□

D'après la sémantique AltaRica :

- une synchronisation avec diffusion est équivalente à un ensemble de synchronisations implicites : <A.a ?, B.b ?> = {<A.a, B.b>, <A.a, B.ϵ>, <A.ϵ, B.b>, <A.ϵ, B.ϵ>}
- les synchronisations implicites sont ordonnées par rapport au nombre d'événements ϵ :
  <A.a, B.b> est plus prioritaire que <A.a, B.ϵ> et <A.ϵ, B.b>, elles-mêmes plus prioritaires que <A.ϵ, B.ϵ>.

Exemple 4.2

Soit $Main$ *un nœud AltaRica défini par le code ci-dessous :*

```
node Main
  sub A,B: n;
  event E2
  trans true |- E2 -> ;
  sync <E2,A.e2,B.e2>;
edon
```

Désignons par A_{ij}/B_{kl} *les états de* $Main$ *tels que :*

- $s = i$ *et* $f = j$ *pour le composant* A *;*
- $s = k$ *et* $f = l$ *pour le composant* B *;*

. La sémantique du nœud $n\mathcal{A}$ *est représentée par le STI ci-après (par souci de lisibilité, les transitions* ϵ *d'un état sur lui même ne sont pas représentées) :*

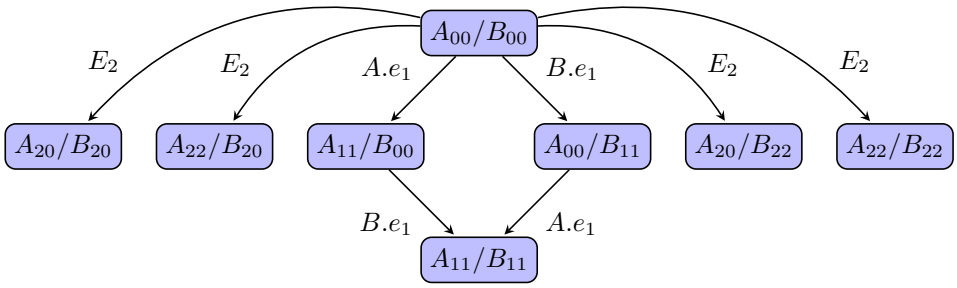


FIG. *4.2: Sémantique du nœud* $n.\mathcal{A}$

4.2 BISIMULATION INTERFACÉE

Il existe de nombreuses relations mathématiques permettant d'établir des liens entre deux sémantiques. Parmi ces relations, les bisimulations jouent un rôle privilégié, la plus fine étant la *bisimulation forte*.. Si deux objets sont bisimilaires, de nombreuses propriétés, dont celles exprimables en μ-calcul, satisfaites par l'un le sont également par l'autre.

Gérald Point s'est intéressé à la bisimulation, dans le cadre du langage AltaRica. Son étude sur les conséquences de cette relation vis-à-vis de la compositionnalité présente dans le langage a abouti à un théorème de compositionnalité.

Définition 4.2.1 (Bisimulation interfacée)
Si $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ *et* $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ *sont des systèmes de transitions interfacés alors* $R \subseteq S \times S'$ *est une relation de* bisimulation interfacée *si :*

1. $\forall s \in S, \exists s' \in S', (s, s') \in R$ *et* $\forall s' \in S', \exists s \in S, (s, s') \in R$
2. $\forall (s, s') \in R, \pi(s) = \pi'(s')$
3. $\forall (s_1, f, e, s_2) \in T, s'_1 \in S', (s_1, s'_1) \in R \Rightarrow \exists s'_2 \in S', (s'_1, f, e, s'_2) \in T'$
4. $\forall (s'_1, f, e, s'_2) \in T', s_1 \in S, (s_1, s'_1) \in R \Rightarrow \exists s_2 \in S, (s_1, f, e, s_2) \in T$

□

L'ensemble F des flux d'un système de transitions interfacé peut être considéré comme un ensemble de propriétés associées aux états du système par l'application π. Une *bisimulation interfacée* est une relation de bisimulation qui préserve ces propriétés.

L'exemple suivant illustre la notion de bisimulation et présente deux systèmes, l'un simple, l'autre plus complexe, pour lesquels on peut établir une relation de bisimulation.

Exemple 4.3

Considérons un interrupteur simple disposant de deux ports (un de chaque coté), dont les valeurs sont égales lorsque l'interrupteur est allumé (position on *vraie) et peuvent différer sinon.*

Le code AltaRica correspondant à ce composant est le suivant :

```
node Switch
  flow   f_left, f_right : bool;
  state  on : bool;
  event  push;
  trans  true |- push -> on := ~on;
  assert on => (f_left=f_right);
  init   on := true;
edon
```

La sémantique de ce composant peut être représentée par la figure suivante (l'état vert symbolise que les ports sont égaux, l'état rouge que les ports peuvent avoir des valeurs différentes) :

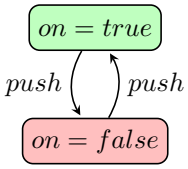


FIG. *4.3: Sémantique du nœud* $SwitchSimple$

Considérons maintenant un type d'interrupteur plus complexe (nœud TwoWaySwitch), disposant de 3 ports : selon la position de l'interrupteur, le flux f1 est égal au flux f2 ou f3.

En branchant deux interrupteurs de type TwoWaySwitch, nous réalisons un système interrupteur (nœud SwitchSystem) dont l'interface (ports et événements) est identique à l'interface d'un interrupteur simple. Nous faisons le choix que l'action $push$ *n'agit que sur l'un des deux interrupteurs complexes à la fois.*

Le code AltaRica correspondant à ce comportement est le suivant :

```
node TwoWaySwitch
  flow  f1, f2, f3 : bool;
  state pos : {1,2};
  init  pos := 1;
  event push;
  trans
    pos=1 |- push -> pos:=2;
    pos=2 |- push -> pos:=1;
  assert
    (pos=1) => (f1 = f2);
    (pos=2) => (f1 = f3);
edon
```

```
node SwitchSystem
  sub   sw1, sw2 : TwoWaySwitch;
  flow  f_left, f_right : bool;
  event push;
  trans true |- push -> ;
  assert
    f_left = sw1.f1;
    sw1.f2 = sw2.f2;
    sw1.f3 = sw2.f3;
    f_right = sw2.f1;
  sync
    <push,sw1.push?,sw2.push?>=1;
edon
```

Par souci de lisibilité, la figure suivante présente la sémantique du nœud SwitchSystem en ne faisant apparaître que les variables d'état et en adoptant le même code couleur que pour la figure 4.3.

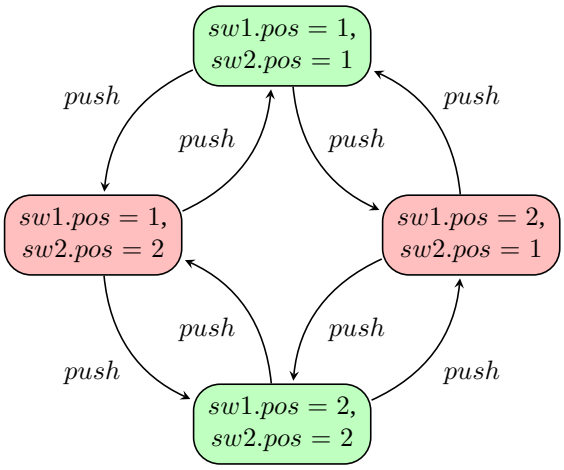


FIG. *4.4: Sémantique simplifiée du nœud* $SwitchSystem$

On observe facilement que toute action push permet de passer d'un état où les ports ont même valeur (vert) à un état où les ports peuvent avoir des valeurs différentes (rouge).

Tout état de chaque nœud est en relation avec un état de l'autre nœud (les états verts sont en relation entre eux, de même pour les états rouges). De plus, depuis deux états en relation, il est possible d'effectuer des transitions de même étiquette conduisant à des états en relation. Il y a donc bisimulation.

Dans [Arn92] il est montré que la bisimulation de deux systèmes de transitions peut être définie à l'aide d'homomorphismes possédant les propriétés suivantes :

Définition 4.2.2 (Homomorphisme de bisimulation interfacée)
Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ deux systèmes de transitions interfacés. Un homomorphisme de bisimulation interfacée $h : \mathcal{A} \to \mathcal{A}'$ *est une application $h : S \to S'$ telle que :*

(h1) *h est surjective*
(h2) $\forall s \in S, \pi(s) = \pi'(h(s))$
(h3) $\forall \langle s_1, f, e, s_2\rangle \in T, \langle h(s_1), f, e, h(s_2)\rangle \in T'$
(h4) $\forall s_1 \in S, s'_2 \in S', \langle h(s_1), f, e, s'_2\rangle \in T' \Rightarrow \exists s_2 \in S, h(s_2) = s'_2 \wedge \langle s_1, f, e, s_2\rangle \in T$

□

Théorème 4.1 ([Arn92])
Deux systèmes de transitions interfacés, $\mathcal{A}$ et $\mathcal{A}'$, sont en bisimulation interfacée si et seulement s'il existe un système de transition interfacé $\mathcal{B}$ et deux homomorphismes de bisimulation interfacés $h : \mathcal{A} \to \mathcal{B}$ et $h' : \mathcal{A}' \to \mathcal{B}$.

Le produit contrôlé est compositionnel pour la bisimulation interfacée. Ce résultat est exprimé par le théorème suivant.

Théorème 4.2
Si $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \ldots, \mathcal{N}_n, V\rangle$ et $\mathcal{N}' = \langle V_F, E, <, \mathcal{N}'_0, \ldots, \mathcal{N}'_n, V\rangle$ sont des nœuds AltaRica tels que pour tout $i = 0 \ldots n$, il existe un homomorphisme de bisimulation h_i de $[\![\mathcal{N}_i]\!]$ vers $[\![\mathcal{N}'_i]\!]$ alors il existe un homomorphisme de bisimulation h de $[\![\mathcal{N}]\!]$ vers $[\![\mathcal{N}']\!]$.

En d'autres termes, le théorème précédent exprime le fait que s'il existe un homomorphisme de bisimulation entre $\mathcal{N}$ et $\mathcal{N}'$ alors $\mathcal{N}$ peut être remplacé par $\mathcal{N}'$ et les systèmes contenant respectivement $\mathcal{N}$ et $\mathcal{N}'$ seront alors équivalents pour la bisimulation.

En pratique, la bisimulation est souvent difficile à obtenir car très contraignante pour des systèmes comme ceux que nous souhaitons traiter.

Nous avons donc souhaité étudier des relations moins fortes que la bisimulation mais permettant d'établir un théorème de compositionnalité en s'inspirant du théorème 4.2. La suite de ce chapitre présente l'étude de deux relations de simulation.

Pour faciliter la lecture de la suite de ce chapitre, nous adopterons pour convention d'écriture que tout élément lié au nœud simulé est identifié par un nom contenant le symbole $'$ (apostrophe). Au contraire, tout élément du nœud qui simule ne contient pas ce symbole. Dans la mesure du possible, nous désignons les états par les lettres s, u et t dans les définitions et des noms de la forme s_i dans les exemples en langage AltaRica.

D'autre part, certaines définitions expriment des relations de la forme "pour tout ... il existe...". Pour faciliter la lecture des représentations graphiques illustrant ces relations, les traits/flèches doubles représentent les contraintes universelles alors que les traits/flèches simples traduisent les contraintes existentielles.

4.3 SIMULATION INTERFACÉE

Nous allons maintenant définir une *simulation interfacée* ayant des propriétés de compositionnalité similaires à celles évoquées dans le théorème 4.2.

4.3.1 DÉFINITIONS

Définition 4.3.1 (Simulation interfacée paramétrée)
Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E', F', S', \pi', T'\rangle$ deux systèmes de transitions interfacés. Soient $RelF \subseteq F \times F'$ une relation entre variables de flux et $RelEvt \subseteq E \times E'$ une relation entre événements. $R(RelF, RelEvt) \subseteq S \times S'$ est une relation de simulation interfacée paramétrée *de $\mathcal{A}$ vers $\mathcal{A}'$ ssi :*

1. $\forall s' \in S', \exists s \in S, (s, s') \in R(RelF, RelEvt)$
2. $\forall (s', f', e', t') \in T', \forall s \in S, (s, s') \in R(RelF, RelEvt) \Rightarrow \exists f \in F, \exists e \in E, \exists t \in S$ *t.q.*
   - $(f, f') \in RelF$,
   - $(e, e') \in RelEvt$,
   - $(s, f, e, t) \in T$,
   - $(t, t') \in R(RelF, RelEvt)$

□

$RelF$ et $RelEvt$ sont introduites afin de mettre en relation les interfaces (parties visibles) des composants AltaRica.

$RelF$ et $RelEvt$ contraignent les relations R mais ne suffisent pas à les définir.

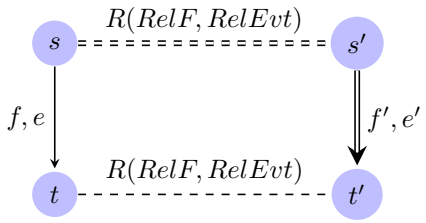


FIG. *4.5: Simulation interfacée paramétrée*

Définition 4.3.2
Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ deux systèmes de transitions interfacés, $initA$ (resp. $initA'$) l'ensemble des états initiaux de $\mathcal{A}$ (resp. $\mathcal{A}'$).

On dira que $\mathcal{A}$ simule $\mathcal{A}'$ ssi il existe une relation de simulation interfacée R, telle que $\forall s' \in InitA'$, $\exists s \in InitA$ et $(s, s') \in R$.

□

Dans le cas où $\mathcal{A}$ et $\mathcal{A}'$ sont les sémantiques respectives des nœuds AltaRica $n\mathcal{A}$ et $n\mathcal{A}'$, on dira que $n\mathcal{A}$ simule $n\mathcal{A}'$.

Remarque

Si $\mathcal{A}$ et $\mathcal{A}'$ sont bisimilaires, alors $n\mathcal{A}$ simule $n\mathcal{A}'$ et $n\mathcal{A}'$ simule $n\mathcal{A}$. La relation inverse n'est pas toujours vraie.

Les trois propriétés ci-dessous traitent des cas particuliers de relation R.

Propriété 4.1

Soient $n\mathcal{A}$ et $n\mathcal{A}'$ deux nœuds AltaRica, $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E', F', S', \pi', T'\rangle$ deux systèmes de transitions interfacés représentant la sémantique respective des nœuds $n\mathcal{A}$ et $n\mathcal{A}'$.

Soient $RelF = F \times F'$, notée true, et $RelEvt = E \times E'$, notée true également.

Pour tout sous-ensemble non vide X de S, la relation $R(true, true) = X \times S'$ est une simulation interfacée paramétrée de $\mathcal{A}$ vers $\mathcal{A}'$.

Preuve : Soient $n\mathcal{A}$ et $n\mathcal{A}'$ deux nœuds AltaRica, $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ les systèmes de transitions interfacés, représentant les sémantiques respectives de $n\mathcal{A}$ et $n\mathcal{A}'$. Montrons que $\forall X \subseteq S$, $R(true, true) = X \times S'$ est une relation de simulation interfacée paramétrée de $\mathcal{A}$ vers $\mathcal{A}'$.

1. Soit $s' \in S'$. On veut montrer qu'il existe $s \in S$ tel que $(s, s') \in R$. Comme l'ensemble X est non vide, il contient au moins un élément s. Par définition de $R = X \times S$, on sait que pour tout $s' \in S'$ on a $(s, s') \in R$.
2. Soit $(s', f', e', t') \in T'$. Soit $s \in S$ tel que $(s, s') \in R$, c'est à dire $s \in X$. Soit $f \in \pi(s)$, considérons la transition (s, f, ϵ, s).
   - $(f, f') \in RelF$ car $RelF = F \times F'$,
   - $(e, e') \in RelEvt$ car $RelEvt = E \times E'$,
   - $(s, f, \epsilon, s) \in T$ du fait de la sémantique d'AltaRica,
   - $(s, t') \in R$ car $s \in X$ et $t' \in S'$.

◇

Propriété 4.2

Soient $n\mathcal{A}$ et $n\mathcal{A}'$ deux nœuds AltaRica, $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ les systèmes de transitions interfacés, sémantiques respectives de $n\mathcal{A}$ et $n\mathcal{A}'$. Soit $RelF = \{(f, f') | f \in F, f' \in F', f = f'\}$ notée Id. Soit $R(Id, RelEvt) \subseteq S \times S'$ une relation de simulation interfacée, alors $\forall (s, s') \in R$, $\pi'(s') \subseteq \pi(s)$.

Preuve : Soient $n\mathcal{A}$ et $n\mathcal{A}'$ deux nœuds AltaRica, $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ les systèmes de transitions interfacés, représentant les sémantiques respectives de $n\mathcal{A}$ et $n\mathcal{A}'$. Soit $R(Id, RelEvt)$ une relation de simulation interfacée paramétrée de $\mathcal{A}$ vers $\mathcal{A}'$.

Soient $(s, s') \in R$ et $f' \in \pi'(s')$.

D'après la sémantique d'AltaRica, on sait que $(s', f', \epsilon, s') \in T'$.

D'après la condition (2) de la définition 4.3.1, une valuation des variables de flux $f \in F$, un événement $e \in E$ et un état $s \in S$ existent tels que $(f, f') \in RelF$. Donc $f' = f$ et $\pi'(s') \subseteq \pi(s)$. ◇

Propriété 4.3

Soient $n\mathcal{A}$ et $n\mathcal{A}'$ deux nœuds AltaRica, $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ les systèmes de transitions interfacés, sémantiques respectives de $n\mathcal{A}$ et $n\mathcal{A}'$.
Soit $RelEvt = \{(e, e') | e \in E,\ e' \in E',\ e = e'\}$ notée Id.
Soit $R(RelF, Id) \subseteq S \times S'$ une relation de simulation interfacée, alors $E' \subseteq E$.

Preuve : Soient $n\mathcal{A}$ et $n\mathcal{A}'$ deux nœuds AltaRica, $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ les systèmes de transitions interfacés, représentant les sémantiques respectives de $n\mathcal{A}$ et $n\mathcal{A}'$. Soit $R(RelF, Id)$ une relation de simulation interfacée paramétrée de $\mathcal{A}$ vers $\mathcal{A}'$. Montrons que $E' \subseteq E$.

Soit $e' \in E'$. D'après la sémantique d'AltaRica, $\exists (s', f', e', t') \in T'$. D'après la condition (1) de la définition 4.3.1, $\exists s \in S$, $(s, s') \in R$. D'après la condition (2) de la définition 4.3.1, $\exists f \in F$, $\exists e \in E$, $\exists t \in S$ tels que $(e, e') \in RelEvt$.

Donc $e' = e$

Donc $E' \subseteq E$. ◇

Dans la suite, sauf mention contraire, pour simplifier les écritures (et les démonstrations) nous ferons l'hypothèse que la relation $RelF$ est l'égalité des flux et que la relation $RelEvt$ est l'identité d'étiquette.

La définition 4.3.1 devient :

Définition 4.3.3 (Simulation interfacée)
Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ *et* $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ *deux systèmes de transitions interfacés.* $R \subseteq S \times S'$ *est une relation de* simulation interfacée *ssi :*

1. $\forall s' \in S', \exists s \in S, (s, s') \in R$
2. $\forall (s', f, e, t') \in T', \forall s \in S, (s, s') \in R \Rightarrow \exists (s, f, e, t) \in T, (t, t') \in R$

□

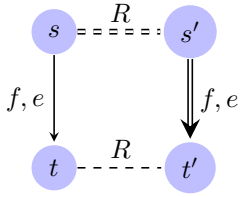


FIG. *4.6: Simulation interfacée*

Remarque

En pratique, en considérant l'égalité des flux et l'identité des événements, si un nœud $n\mathcal{A}$ simule un nœud $n\mathcal{A}'$, alors :
- $n\mathcal{A}$ et $n\mathcal{A}'$ ont les mêmes variables de flux mais le domaine de certaines variables de flux de $n\mathcal{A}'$ est éventuellement restreint par rapport au domaine de ces mêmes variables de flux dans $n\mathcal{A}$,
- les événements présents dans $n\mathcal{A}'$ le sont également dans $n\mathcal{A}$, ce dernier pouvant contenir également d'autres événements.

4.3.2 IMPLÉMENTATION MEC V

Étant donné un nœud AltaRica, l'outil MecV considère les objets suivants :
- les événements
- les configurations (valuations des variables d'états et de flux)
- les transitions
- les configurations initiales.

Dans la définition 4.3.1, la relation $RelF$ s'applique aux variables de flux. Du fait que MecV ne permet pas de manipuler directement les variables de flux (seulement les configurations), la traduction en langage MecV de la relation $RelF$ s'appliquera aux configurations. Ainsi la relation $RelF(s, s')$ précise, pour une configuration s et une configuration s', les variables en relation. (MecV rend envisageable d'établir des relations entre variables d'état)

Afin d'écrire une relation générique de calcul des états en simulation, nous considérons les éléments suivants (à définir au préalable en fonction des nœuds pour lesquels on souhaite vérifier la simulation) :
- $RelF(s, s')$: relation entre les variables de flux des deux nœuds,
- $RelEvt(e, e')$: relation entre les événements des deux nœuds,
- $transA(s, e, t)$ (resp. $transA'(s', e', t')$) : transitions de s à t étiquetées par e du nœud A (resp. les transitions de s' à t' étiquetées par e' du nœud A'),
- $initA(s)$ (resp. $initA'$) : configurations initiales du nœud A (resp. du nœud A').

Selon la définition 4.3.1, le calcul de la plus grande relation de simulation interfacée s'obtient par le plus grand point fixe (syntaxe MecV '-=') suivant :

```
sim(s,s') -=
  RelF(s,s') &
  ([e'][t'](transA'(s',e',t') =>
   <e><t>(RelEvt(e,e') & transA(s,e,t) & sim(t,t'))));
```

La définition 4.3.2 devient :

```
isSim(x) :=
  x = ([s'] (initA'(s') => <s> (initA(s) & sim(s,s'))));
```

En pratique, vérifier qu'un nœud simule un autre nœud se décompose en deux étapes :
– calcul des couples en simulation (relation $sim(s, s')$),
– vérification que les états initiaux sont en simulation (relation $isSim(x)$).

4.3.3 COMPOSITIONNALITÉ ET RESTRICTIONS SUR LE LANGAGE

Comme évoqué précédemment, nous cherchons à établir un théorème de compositionnalité similaire au théorème 4.3 pour la relation de simulation interfacée.

Le langage AltaRica permet à un nœud père d'imposer :
– des priorités entre événements,
– des synchronisations avec diffusion.

Avant d'établir ce théorème, montrons que ces deux aspects ne peuvent pas être conservés dans le langage pour préserver la compositionnalité.

Pour chacune de ces particularités, nous présentons un contre-exemple où le nœud $n\mathcal{A}$ simule $n\mathcal{A}'$ mais le nœud $Main\mathcal{A}$ (contenant $n\mathcal{A}$) ne simule pas $Main\mathcal{A}'$ (obtenu en remplaçant $n\mathcal{A}$ par $n\mathcal{A}'$ dans $Main\mathcal{A}$).

4.3.3.1 Non-préservation de la compositionnalité due à des priorités entre événements

Exemple 4.4

Soient $n\mathcal{A}$ et $n\mathcal{A}'$ deux nœuds AltaRica définis par le code ci-dessous :

```
node nA
  state s:[0,2];
  event e1,e2;
  trans
    s = 0 |- e1 -> s := 1;
    s = 0 |- e2 -> s := 2;
  init  s := 0;
edon
```

```
node nA'
  state s:[0,1];
  event e1,e2;
  trans
    s = 0 |- e1 -> s := 1;
    false |- e2 -> ;
  init  s := 0;
edon
```

Désignons par s_i (resp. s'_i) les états de $n\mathcal{A}$ (resp. $n\mathcal{A}'$) tels que s = i. La sémantique des nœuds $n\mathcal{A}$ et $n\mathcal{A}'$ est représentée par les STI ci-après :

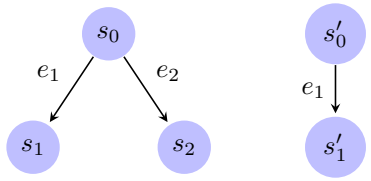


FIG. *4.7: Sémantique des nœuds $n\mathcal{A}$ et $n\mathcal{A}'$*

Les nœuds $n\mathcal{A}$ et $n\mathcal{A}'$, définis ci-dessus, ne contiennent pas de variable de flux, donc tout état de $n\mathcal{A}$ dispose des mêmes flux que tout état de $n\mathcal{A}'$. La relation $RelF$ s'écrit :

```
RelF(s: nA!c, s':nA'!c) := true;
```

D'autre part, considérons la relation $RelEvt$, identité des étiquettes. Cette relation s'écrit formellement $RelEvt = \{(\epsilon, \epsilon), (e_1, e_1), (e_2, e_2)\}$. L'implémentation de cette relation en langage MecV est la suivante :

```
RelEvt(e: nA!ev, e':nA'!ev) := (e.='' '' & e'.='' '')
                             | (e.=''e1'' & e'.=''e1'')
                             | (e.=''e2'' & e'.=''e2'');
```

L'unique transition (s'_0, e_1, s'_1) de $n\mathcal{A}'$ est simulable par la transition (s_0, e_1, s_1) de $n\mathcal{A}$. D'après la définition 4.3.2, on peut donc dire que $n\mathcal{A}$ simule $n\mathcal{A}'$.

Supposons maintenant que $n\mathcal{A}$ et $n\mathcal{A}'$ soient des composants respectifs des nœuds $Main\mathcal{A}$ et $Main\mathcal{A}'$ pour lesquels la priorité suivante est définie : $e_1 < e_2$. Le code AltaRica des nœuds $Main\mathcal{A}$ et $Main\mathcal{A}'$ est le suivant :

```
node MainA
  sub   comp : nA;
  event e1<e2;
  trans true |- e1,e2-> ;
  sync
    <e1,comp.e1>;
    <e2,comp.e2>;
edon
```

```
node MainA'
  sub   comp : nA';
  event e1<e2;
  trans true |- e1,e2-> ;
  sync
    <e1,comp.e1>;
    <e2,comp.e2>;
edon
```

FIG. *4.8: Sémantique des nœuds $Main\mathcal{A}$ et $Main\mathcal{A}'$*

De même que pour $n\mathcal{A}$ et $n\mathcal{A}'$, l'absence de variable de flux conduit à la relation $RelF$ suivante :

```
RelF(s: MainA!c, s':MainA'!c) := true;
```

D'autre part, l'identité sur les événements se traduit par la relation $RelEvt$ suivante :

```
RelEvt(e: MainA!ev, e':MainA'!ev) :=
                              (e.='' '' & e'.='' '')
                            | (e.=''e1'' & e'.=''e1'')
                            | (e.=''e2'' & e'.=''e2'');
```

Du fait de la priorité, la transition (s'_0, e_1, s'_1) de $Main\mathcal{A}'$ n'est plus simulable dans $Main\mathcal{A}$, comme illustré par la figure 4.8, donc $Main\mathcal{A}$ ne simule pas $Main\mathcal{A}'$ alors que $n\mathcal{A}$ simule $n\mathcal{A}'$.

Pour la suite de cette étude sur la simulation interfacée, nous nous restreindrons à des STI sans priorités.

4.3.3.2 Non-préservation de la compositionnalité due à la notion de diffusion dans les synchronisations

La diffusion, en AltaRica, sous-entend une certaine notion de priorité. C'est pourquoi nous avons cherché un contre-exemple illustrant le fait que deux composants $n\mathcal{A}$ et $n\mathcal{A}'$ peuvent être en simulation alors que les nœuds $Main\mathcal{A}$ et $Main\mathcal{A}'$ ($Main\mathcal{A}'$ étant obtenu en remplaçant $n\mathcal{A}$ par $n\mathcal{A}'$ dans $Main\mathcal{A}$) ne le sont pas.

Exemple 4.5

Soient nA et nA' deux nœuds AltaRica définis par le code AltaRica ci-dessous et dont la sémantique est représentée par les STI ci-après (l'étiquette de chaque état représente la valeur de la variable on) :

```
node nA
  state on : bool;
  init  on := true;
  event e1,e2;
  trans
    on |- e1 -> on := ~on;
    on |- e2 -> on := ~on;
edon
```

```
node nA'
  state on : bool;
  init  on := true;
  event e1,e2;
  trans
    on |- e1 -> on := ~on;
    false |- e2 -> ;
edon
```

FIG. 4.9: *Sémantique des nœuds nA et nA'*

De même que dans l'exemple précédent, les nœuds nA et nA' définis ci-dessus ne contiennent pas de variable de flux, donc tout état de nA dispose des mêmes flux que tout état de nA'. La relation $RelF$ est $true$. Soit $RelEvt = \{(\epsilon,\epsilon),(e_1,e_1),(e_2,e_2)\}$, l'unique transition de nA' est simulable pas nA donc nA simule nA'.

Supposons maintenant que nA et nA' soient des composants respectifs des nœuds $MainA$ et $MainA'$ définis par le code AltaRica suivant :

```
node MainA
  sub N : nA;
  state on : bool;
  init  on := true;
  event e1,e2;
  trans
    true |- e1 ->;
    on |- e2 -> on := ~on;
  sync
    <e1, N.e1>;
    <e2, N.e2?>;
edon
```

```
node MainA'
  sub N : nA';
  state on : bool;
  init  on := true;
  event e1,e2;
  trans
    true |- e1 ->;
    on |- e2 -> on := ~on;
  sync
    <e1, N.e1>;
    <e2, N.e2?>;
edon
```

Les nœuds $MainA$ et $MainA'$ synchronisent l'événement $e2$ avec l'événement $N.e2$ (i.e. l'événement $e2$ du nœud fils N) si possible. L'événement $N.e2$ étant possible dans $MainA$ depuis l'état initial, $< e2, N.e2 >$ est plus prioritaire que $< e2, N.\epsilon >$. En revanche, l'événement $N.e2$ n'étant jamais possible dans $MainA'$ de par sa garde, la transition $< e2, N.e2? >$ revient donc à considérer la synchronisation $< e2, N.\epsilon >$.

La figure ci-dessous représente les sémantiques des nœuds $MainA$ et $MainA'$. Chaque état dispose d'une étiquette de la forme x/y où x représente la valeur de la variable on et y représente la valeur de la variable $N.on$.

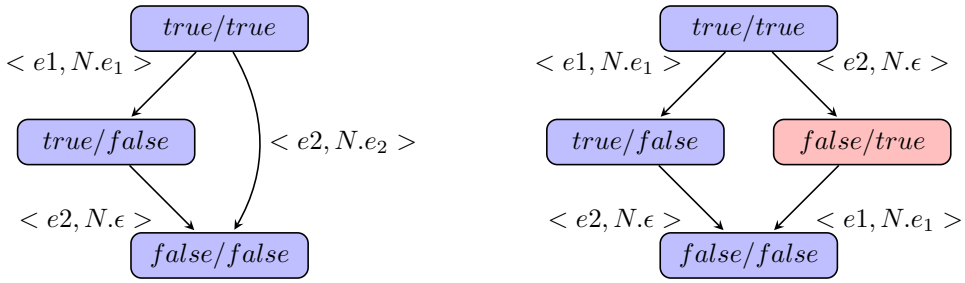


FIG. *4.10: Sémantique des nœuds $MainA$ et $MainA'$*

De même que pour nA et nA', l'absence de variable de flux conduit à la relation $RelF$ suivante :

```
RelF(s: MainA!c, s':MainA'!c) := true;
```

D'autre part, l'identité sur les événements se traduit par la relation $RelEvt$ suivante :

```
RelEvt(e: MainA!ev, e':MainA'!ev) :=
                                    (e.='' ''  & e'.='' '')
                                  | (e.=''e1'' & e'.=''e1'')
                                  | (e.=''e2'' & e'.=''e2'');
```

Du fait de la synchronisation avec diffusion, le nœud $MainA'$ dispose d'un état $false/true$ qui n'est en relation avec aucun état de $MainA$.

Donc le nœud $MainA$ ne simule pas le nœud $MainA'$ alors que le nœud nA simule le nœud nA'.

On en déduit donc qu'un modèle AltaRica contenant des synchronisations avec diffusion ne permet pas de garantir la propriété de compositionnalité établie pour la bisimulation forte et que l'on souhaite établir pour la relation de simulation forte.

Pour la suite de cette étude sur la simulation interfacée, nous nous restreindrons à des STI sans priorités et sans diffusion dans les synchronisations.

4.3.4 SÉMANTIQUE DES COMPOSANTS SANS PRIORITÉ NI DIFFUSION

Compte tenu des différentes restrictions, les définitions de la sémantique des composants et des nœuds AltaRica, établies par Gérald Point, peuvent être réduites. La suite de ce chapitre propose de nouvelles définitions et souligne les différences avec les définitions initiales.

Proposition 4.1
La syntaxe abstraite d'un composant AltaRica sans priorité ni diffusion est similaire à la définition 4.1.3 en considérant, pour la relation d'ordre strict $<$, la relation $\emptyset$ (i.e. tous les événements sont incomparables entre eux).

Remarque

De la définition 4.1.5 sur les priorités syntaxiques, appliquée avec la relation $\emptyset$ pour relation d'ordre strict $<$, on déduit : $[\![M]\!]\restriction \emptyset = [\![M]\!]$.

Proposition 4.2
La sémantique d'un composant AltaRica sans priorité ni diffusion est obtenue en appliquant la définition 4.1.4 avec la relation $\emptyset$ pour relation d'ordre strict $<$ et $[\![M]\!]\restriction \emptyset = [\![M]\!]$.

4.3.5 SÉMANTIQUE DES NŒUDS SANS PRIORITÉ NI DIFFUSION

La syntaxe abstraite d'un nœud sans priorité ni diffusion est un cas particulier de la syntaxe d'un nœud AltaRica, définie dans 4.1.6 :

- l'absence de priorité se traduit par la relation d'ordre strict $\emptyset$,
- la variable D est une contrainte sur le nombre d'événements marqués ?, l'absence de diffusion revient à considérer $D = \{0\}$ (il est donc possible de supprimer les ?, ce qui a pour conséquence $E_i^? = E_i$).

Les conséquences de l'absence de priorité et de diffusion étant importantes, nous établissons ci-dessous de nouvelles définitions de la syntaxe abstraite et de la sémantique d'un nœud propre à ces restrictions.

Définition 4.3.4 (Nœuds sans priorité ni diffusion – Syntaxe abstraite)
Un nœud est un $(n+4)$-uplet $\mathcal{N} = \langle V_F, E, \mathcal{N}_0, \ldots, \mathcal{N}_n, V\rangle$ où :

- *V_F est une ensemble de variables de flux ;*
- *$E = E_+ \cup \{\epsilon\}$ est un ensemble d'événements où ϵ est un événement (implicite) distingué de ceux de E_+ ;*
- *Pour $i = 1 \ldots n$, $\mathcal{N}_i$ est un composant ou un nœud ; V_{Fi} est son ensemble de variables de flux, E_i est son ensemble d'événements. De plus, nous supposons que leurs ensembles de variables sont disjoints ;*
- *$\mathcal{N}_0$ est un composant particulier appelé* contrôleur*. Son ensemble d'événements est $E_0 = E$ ordonné par la relation vide (i.e. tous ses événements sont incomparables). Son ensemble de variables de flux est $V_{F0} = V_F \cup V_{F1} \cup \cdots \cup V_{Fn}$.*
- *$V = V_d \cup V_{imp}$ est un ensemble de vecteurs de synchronisation sans diffusion où :*
  - *$V_d \subseteq E_0 \times \cdots \times E_n$ est l'ensemble des vecteurs spécifiés.*
  - *$V_{imp} \subseteq E_0 \times \cdots \times E_n$ est un ensemble de vecteurs implicite construit de la manière suivante :*
    - *$\langle \epsilon, \ldots, \epsilon\rangle \in V_{imp}$.*
    - *$\forall i \in [0, n]$, $\forall e_i \in E_i \setminus \{\epsilon\}$, $[\forall \langle e'_0, \ldots, e'_i, \ldots, e'_n, \rangle \in V_d,\ e'_i \notin \{e_i\}] \Rightarrow \langle \epsilon, \ldots, e_i, \ldots, \epsilon\rangle \in V_{imp}$. Le vecteur $\langle \epsilon, \ldots, e_i, \ldots, \epsilon, \rangle$ ne contient que des ϵ hormis e_i placé à la i-ème position. Si un événement du sous-nœud $\mathcal{N}_i$ n'est pas impliqué dans une synchronisation alors cet événement doit se produire seul (tous les autres doivent faire ϵ).*

□

De même que pour la syntaxe abstraite, la sémantique d'un nœud AltaRica sans priorité ni diffusion est un cas particulier de la sémantique définie dans 4.1.8.

Du fait de l'absence de diffusion dans les synchronisations, chaque vecteur de synchronisation est sa propre instance et la définition 4.1.7 devient inutile. La définition des transitions d'un nœud sans priorité ni diffusion s'en trouve grandement simplifiée.

Définition 4.3.5 (Nœuds sans priorité ni diffusion – Sémantique)
Soit un nœud AltaRica $\mathcal{N} = \langle V_F, E, \mathcal{N}_0, \ldots, \mathcal{N}_n, V\rangle$. Nous notons $[\![\mathcal{N}_i]\!] = \langle E_i, F_i, S_i, \pi_i, T_i\rangle$ la sémantique de $\mathcal{N}_i$ (pour i=1... n).

La sémantique du nœud $\mathcal{N}$ est le STI $[\![\mathcal{N}]\!] = \langle E, F, S, \pi, T\rangle$ construit de la manière suivante :

- *$F = \mathcal{D}^{V_F}$*
- *$S = \{s \in S_0 \times \cdots \times S_n \mid \pi(s) \neq \emptyset\}$*
- *$\pi(\langle s_0, \ldots, s_n\rangle) = \{f \in \mathcal{D}^{V_F} \mid \exists \langle f, f_1, \ldots, f_n\rangle \in \pi_0(s_0), \forall i,\ f_i \in \pi_i(s_i)\}$*
- *$T_{\mathcal{N}} \subseteq S \times F \times V \times S$ est défini par $\langle (s_0, \ldots, s_n),\ f,\ u,\ (t_0, \ldots, t_n)\rangle \in T_{\mathcal{N}}$ si et seulement si il existe $f_0 = (f, f_1, \ldots, f_n) \in \pi_0(s_0)$ et $u = (e_0, \ldots, e_n)$ et pour tout $i \in [0, n]$, $(s_i, f_i, e_i, t_i) \in T_i$ et $f_i \in \pi_i(s_i)$;*
- *$T \subseteq S \times F \times E \times S$ est obtenu à partir de $T_{\mathcal{N}}$ en ne conservant que la projection sur la première composante (sur le contrôleur $\mathcal{N}_0$) de u.*

□

Le produit contrôlé est compositionnel pour la simulation interfacée. Ce résultat est exprimé par le théorème suivant.

Théorème 4.3
Soient $\mathcal{N} = \langle V_F, E, \mathcal{N}_0, \ldots, \mathcal{N}_n, V\rangle$ et $\mathcal{N}' = \langle V_F, E, \mathcal{N}'_0, \ldots, \mathcal{N}'_n, V\rangle$ deux nœud AltaRica sans priorité et sans vecteur de diffusion. Si, pour tout $i = 0 \ldots n$, $[\![\mathcal{N}_i]\!]$ simule $[\![\mathcal{N}'_i]\!]$ alors $[\![\mathcal{N}]\!]$ simule $[\![\mathcal{N}']\!]$.

Preuve : Considérons la relation de simulation interfacée de la définition 4.3.3 pour laquelle les relations $RelF$ et $RelEvt$ sont respectivement l'égalité des flux et l'identité des événements.

Soit R la relation entre $[\![\mathcal{N}]\!]$ et $[\![\mathcal{N}']\!]$ suivante : $(\vec{s}, \vec{s'}) \in R$ si et seulement si, pour tout $i(s_i, s'_i) \in R_i$, R_i étant une relation de simulation interfacée entre $[\![\mathcal{N}_i]\!]$ et $[\![\mathcal{N}'_i]\!]$.
Montrons que R est une relation de simulation interfacée.

1. Soit $\vec{s'} = \langle s'_0, ..., s'_n\rangle \in S'$. Pour tout i, R_i est une relation de simulation interfacée, on en déduit que :
   pour tout i, pour tout $s'_i \in S'_i$, il existe $s_i \in S_i$ tel que $(s_i, s'_i) \in R_i$.
   Soit $\vec{s} = \langle s_0, ..., s_n\rangle$, on a $\vec{s} \in S$, et $(\vec{s}, \vec{s'}) \in R$, ce que l'on voulait avoir.
2. Montrons que pour tout $(\vec{s'}, f, e, \vec{t'}) \in T'$, pour tout $\vec{s} \in S$, si $(\vec{s}, \vec{s'}) \in R$ alors il existe une transition $(\vec{s}, f, e, \vec{t}) \in T$, telle que $(\vec{t}, \vec{t'}) \in R$.
   Soient $(\vec{s'}, f, e, \vec{t'}) \in T'$ et $\vec{s} \in S$ tel que $(\vec{s}, \vec{s'}) \in R$.
   D'après la définition 4.3.5 appliquée à $\mathcal{N}'$, $(\vec{s'}, f, e, \vec{t'}) \in T'$ si et seulement si il existe $f_0 = (f, f_1, \ldots, f_n) \in \pi'_0(s'_0)$ et $e = (e_0, \ldots, e_n)$ et que pour tout $i \in [0, n]$, $(s'_i, f_i, e_i, t'_i) \in T'_i$ et $f_i \in \pi'_i(s'_i)$
   D'après la propriété 4.2 appliquée à R_0, $\pi'_0(s'_0) \subseteq \pi_0(s_0)$ d'où $f_0 = (f, f_1, \ldots, f_n) \in \pi_0(s_0)$.
   Comme $(\vec{s}, \vec{s'}) \in R$, d'après la propriété 4.2 appliquée à chaque R_i, pour tout $i \in [0; n], \pi'_i(s'_i) \subseteq \pi_i(s_i)$ d'où pour tout $i \in [0; n]$, $f_i \in \pi_i(s_i)$.
   R_i est une relation de simulation interfacée soit, par définition, pour toute transition $(\vec{s_i}, f_i, e_i, \vec{t'_i}) \in T'_i$, pour tout $\vec{s_i} \in S$, si $(\vec{s_i}, \vec{s'_i}) \in R_i$ alors il existe $\vec{t_i} \in S_i$, tel que $(\vec{s_i}, f_i, e_i, \vec{t_i}) \in T_i$.
   Donc pour tout $i \in [0; n]$, $(s_i, f_i, e_i, t_i) \in T_i$.
   Soit $\vec{t} = \langle t_0, ..., t_n\rangle$, $\vec{t} \in S$, $(\vec{s}, f, e, \vec{t}) \in T$ et $(\vec{t}, \vec{t'}) \in R$.

◇

4.3.6 PRÉSERVATION DES TRACES/SÉQUENCES

La sûreté de fonctionnement des systèmes critiques en aéronautique, s'intéresse aux séquences de défaillances. C'est pourquoi, contrairement à de nombreuses études sur la préservation de propriétés d'une logique comportementale (ex : ACTL*), nous souhaitons étudier la préservation des traces/séquences dans le cas de nœuds en relation de simulation. Notre objectif est de profiter des propriétés de la relation de simulation pour calculer les traces/séquences du modèle simulé à partir des traces/séquences générées pour le modèle abstrait, et ainsi éviter de générer les coupes/séquences sur le modèle simulé.

Pour cela, il est tout d'abord nécessaire de préciser certaines notions. Tout d'abord, une trace d'un système désigne un comportement de ce système, une suite de transitions entre deux états. Une définition formelle d'une trace tenant compte de la sémantique d'AltaRica est proposée dans la définition 4.3.6.

Définition 4.3.6 (Trace)
Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ un système de transitions interfacé, $s_0 \in S$ et $f_0 \in \pi(s_0)$.
La suite de transitions $(s_0, f_0, e_0, s_1)...(s_{n-1}, f_{n-1}, e_{n-1}, s_n)$ est une trace de $\mathcal{A}$ issue de (s_0, f_0).

□

Afin de faciliter la lecture de la suite de cette section, nous introduisons quelques notations.

En AltaRica, l'état global d'un nœud, associant une valeur à toutes les variables d'état et de flux, est appelé configuration. D'autre part, pour une valuation donnée des variables d'état d'un nœud, plusieurs valuations des variables de flux sont possibles.

Compte tenu des ces trois particularités d'AltaRica, la définition 4.3.7 introduit les notions d'ensembles de traces depuis une configuration, de traces depuis un état et de traces issues de tous les états initiaux d'un nœud.

Définition 4.3.7 (Ensembles de traces)
Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ un système de transitions interfacé, $s \in S$, $f \in \pi(s)$ et $Init$ le sous-ensemble de S représentant les états initiaux de $\mathcal{A}$.

On note :
- *$tr(s, f)$ l'ensemble des traces issues de la configuration (s, f).*
- *$tr(s)$ l'ensemble des traces issues de l'état s :*

$$tr(s) = \bigcup_{f \in \pi(s)} tr(s, f)$$

- *$tr(\mathcal{A})$ l'ensemble des traces du STI $\mathcal{A}$:*

$$tr(\mathcal{A}) = \bigcup_{s \in Init} tr(s)$$

□

Propriété 4.4

Soient $\mathcal{A}$ et $\mathcal{A}'$ deux systèmes de transitions interfacés et $R(RelF, RelEvt)$ une relation de simulation interfacée paramétrée de $\mathcal{A}$ vers $\mathcal{A}'$.

Soit $tr' = (s'_0, f'_0, e'_0, s'_1)...(s'_{n-1}, f'_{n-1}, e'_{n-1}, s'_n)$ une trace de $\mathcal{A}'$, alors il existe une trace $tr = (s_0, f_0, e_0, s_1) \ldots (s_{n-1}, f_{n-1}, e_{n-1}, s_n)$ de $\mathcal{A}$ telle que $\forall i \in [0, n]$: $(s_i, s'_i) \in R$, $(f_i, f'_i) \in RelF$, $(e_i, e'_i) \in RelEvt$.

Remarque

On dira que tr simule tr'.

Preuve : Notons $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E', F', S', \pi', T'\rangle$ deux systèmes de transitions interfacés tels que $\mathcal{A}$ simule $\mathcal{A}'$.
Soit tr' une trace de $\mathcal{A}'$, $tr' = (s'_0, f'_0, e'_0, s'_1)...(s'_{n-1}, f'_{n-1}, e'_{n-1}, s'_n)$. En utilisant le fait que $\mathcal{A}$ simule $\mathcal{A}'$, on montre par récurrence que $\forall i \in [0, n-1]$, $\exists (s_i, f_i, e_i, s_{i+1}) \in T$ t.q.
- $(s_i, s'_i) \in R(RelF, RelEvt)$,
- $(f_i, f'_i) \in RelF$,
- $(e_i, e'_i) \in RelEvt$,
- $(s_{i+1}, s'_{i+1}) \in R(RelF, RelEvt)$.

Donc il existe une trace tr de $\mathcal{A}$, $tr = (s_0, f_0, e_0, s_1)...(s_{n-1}, f_{n-1}, e_{n-1}, s_n)$ qui simule tr'. ◇

Ayant prouvé que si $\mathcal{A}$ simule $\mathcal{A}'$ alors toute trace de $\mathcal{A}'$ est simulée par une trace de $\mathcal{A}$, considérons maintenant la fonction image des traces de $\mathcal{A}$ par R et la fonction image réciproque des traces de $\mathcal{A}'$ également par R.

Définition 4.3.8 (Images de traces)
Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E', F', S', \pi', T'\rangle$ deux systèmes de transitions interfacés et $R(RelF, RelEvt)$ une relation de simulation interfacée paramétrée de $\mathcal{A}$ vers $\mathcal{A}'$.
Soit tr une trace de $\mathcal{A}$:
- *$Img_R(tr)$ est l'ensemble des traces tr' de $\mathcal{A}'$ telles que tr simule tr',*
- *$Img_R^{-1}(tr')$ est l'ensemble des traces tr de $\mathcal{A}$ telles que tr simule tr'.*

□

Les définitions sont étendues aux ensembles de traces. Ainsi, soit $\mathcal{T}$ un ensemble de traces :

$$Img_R(\mathcal{T}) = \bigcup_{tr \in \mathcal{T}} Img_R(tr)$$

Toute transition de $\mathcal{A}'$ est simulée par une transition de $\mathcal{A}$ cependant $\mathcal{A}$ peut contenir des transitions ne simulant aucune transition de $\mathcal{A}'$. On en déduit que l'ensemble des traces de $\mathcal{A}$ contient les images réciproques des traces de $\mathcal{A}'$ par R et peut être supérieur.

En revanche, toute trace de $\mathcal{A}$ ne simulant aucune trace de $\mathcal{A}'$ n'a pas d'image donc l'ensemble des images des traces de $\mathcal{A}$ par R est égal à l'ensemble des traces de $\mathcal{A}'$. La propriété 4.5 formalise ces deux relations.

Propriété 4.5

Soient $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ et $\mathcal{A}' = \langle E', F', S', \pi', T' \rangle$ deux systèmes de transitions interfacés et $R(RelF, RelEvt)$ une relation de simulation interfacée paramétrée de $\mathcal{A}$ vers $\mathcal{A}'$.
Si $\mathcal{A}$ simule $\mathcal{A}'$, alors :
- $tr(\mathcal{A}) \supseteq Img_R^{-1}(tr(\mathcal{A}'))$,
- $Img_R(tr(\mathcal{A})) = tr(\mathcal{A}')$.

La sûreté de fonctionnement se contente d'observer des suites d'événements, nous introduisons la notion de séquence.

Définition 4.3.9 (Séquence)
Soient $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ un système de transitions interfacé, $s_0 \in S$ et $f_0 \in \pi(s_0)$.
La suite d'événements $e_0 ... e_{n-1}$ est une séquence de $\mathcal{A}$ issue de (s_0, f_0).

□

D'autre part, nous introduisons la notion de séquence obtenue à partir d'une trace.

Définition 4.3.10 (Séquence associée à une trace)
Soient $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ un système de transitions interfacé et $tr = (s_0,\ f_0,\ e_0,\ s_1)\ \ldots\ (s_{n-1},\ f_{n-1},\ e_{n-1},\ s_n)$ une trace de $\mathcal{A}$.
La séquence $seq(tr)$ est la suite d'événements $e_0 ... e_{n-1}$.

□

On étend cette définition aux ensembles de traces. Ainsi, soit T un ensemble de traces :

$$seq(Tr) = \bigcup_{t \in Tr} seq(t)$$

On déduit de la propriété 4.5, relative aux traces, la propriété 4.6 suivante, relative aux séquences.

Propriété 4.6

Soient $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ et $\mathcal{A}' = \langle E', F', S', \pi', T' \rangle$ deux systèmes de transitions interfacés et $R(RelF, RelEvt)$ une relation de simulation interfacée paramétrée de $\mathcal{A}$ vers $\mathcal{A}'$.
Si $\mathcal{A}$ simule $\mathcal{A}'$, alors :
- $seq(tr(\mathcal{A})) \supseteq seq(Img_R^{-1}(tr(\mathcal{A}')))$,
- $seq(Img_R(tr(\mathcal{A}))) = seq(tr(\mathcal{A}'))$.

4.3.7 EXPLOITATION DES PROPRIÉTÉS DE COMPOSITION ET DE PRÉSERVATION DES TRACES/SÉQUENCES

D'après le théorème de composition que nous avons établi, si un nœud `A` simule un nœud `A'`, alors le nœud `M`, contenant `A`, simule le nœud `M'`, obtenu en remplaçant `A` `A'`. Afin de calculer les traces/séquences de `M'` à partir de celles de `M`, il est nécessaire, au préalable, de calculer les traces/séquences de `A'` à partir de celles de `A`.

Nous présentons dans la suite de ce paragraphe, les principes et la mise en œuvre du calcul des traces/séquences au niveau composant puis au niveau nœud.

4.3.7.1 Principes

Étant donnés deux STI $\mathcal{A}$ et $\mathcal{A}'$, il est possible de générer les traces de chaque STI. Néanmoins, s'il existe une relation R telle que $\mathcal{A}$ simule $\mathcal{A}'$, alors, compte tenu des différentes propriétés énoncées précédemment, il est possible de calculer les images des traces de $\mathcal{A}$ par la relation R, ces images étant égales aux traces générées pour $\mathcal{A}'$. Une séquence étant le résultat de la projection d'une trace, les séquences issues des images des traces de $\mathcal{A}$ sont égales aux séquences issues des traces de $\mathcal{A}'$.

Ce processus permettant pour deux STI en simulation d'obtenir les séquences du STI simulé est illustré sur la figure 4.11.

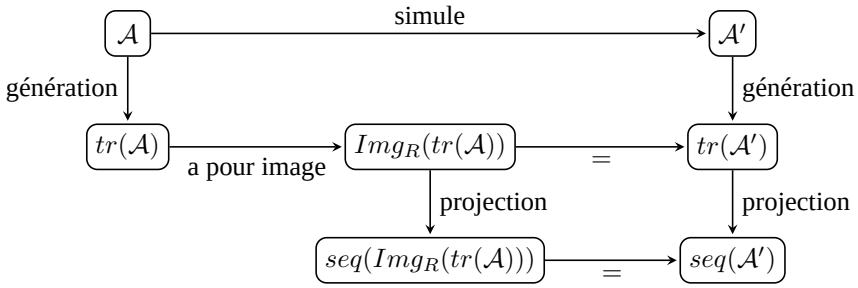


FIG. *4.11: Schéma de principe*

Compte tenu du théorème de compositionnalité 4.3 et des propriétés de conservation des traces énoncées précédemment, les traces de $\mathcal{M}'$, obtenu en remplaçant $\mathcal{A}$ par $\mathcal{A}'$ dans $\mathcal{M}$, peuvent être calculées à partir des traces de $\mathcal{M}$.

Cependant le processus de calcul représenté par la figure 4.11 ne peut être transposé directement au calcul des images des traces de $\mathcal{M}$ car toute trace de $\mathcal{M}$ contient des informations relatives à des composants autres que $\mathcal{A}$, rendant les algorithmes précédents inapplicables. Le calcul des images des traces de $\mathcal{M}$ se décompose en trois étapes :

- chaque trace est tout d'abord projetée sur $\mathcal{A}$ afin de différencier les informations relatives à $\mathcal{A}$, qui est remplacé par $\mathcal{A}'$ dans l'image de $\mathcal{M}$ et donc dans $\mathcal{M}'$, des informations relatives aux composants communs à $\mathcal{M}$ et $\mathcal{M}'$,
- chaque projection est traitée à l'aide de l'algorithme `Images_TraceUnique`, présenté dans la section suivante, pour obtenir les traces simulées contenant $\mathcal{A}'$,
- les traces images des projections sont enfin traitées pour contenir à nouveau les informations relatives aux autres composants de $\mathcal{M}$ (et donc de $\mathcal{M}'$) : la substitution complète les images des traces issues de la projection par les informations que la projection a masquées.

La figure 4.12 est la transposition de la figure 4.11 dans le cas de nœuds composés en simulation. Cette figure illustre la décomposition du calcul (vert) des images des traces de $\mathcal{M}$ en 3 opérations (bleu) distinctes.

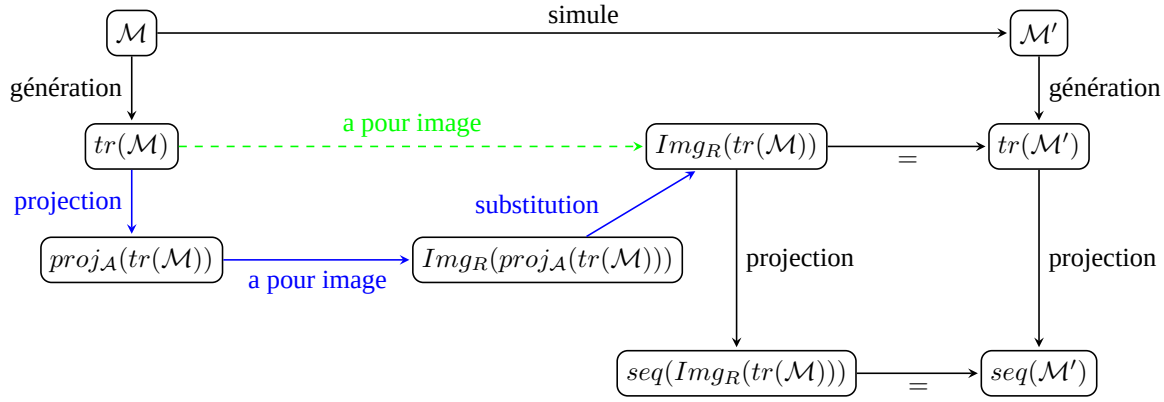


FIG. *4.12: Schéma de principe*

4.3.7.2 Mise en œuvre

Afin de tirer profit des propriétés énoncées et de mettre en œuvre les principes énoncés précédemment, nous avons écrit des algorithmes automatisant le calcul d'$Img_R(tr(\mathcal{A}))$. Nous considérons des traces finies (toute configuration présente n'apparaît qu'un nombre fini de fois).

Avant de pouvoir calculer l'image d'une trace, il est nécessaire de connaître toutes les transitions de $\mathcal{A}'$ en relation de simulation avec une transition de $\mathcal{A}$. La relation MecV suivante effectue ce calcul :

```
SimTrans(s1,e,s2,s1',e',s2') := sim(s1,s1') & sim(s2,s2')
                              & RelEvt(e,e')
                              & transA(s1,e,s2)
                              & transA'(s1',e',s2');
```

Par souci de lisibilité, nous appliquerons les conventions d'écriture suivantes :
- t (resp. t') désigne toute transition de $\mathcal{A}$ (resp. $\mathcal{A}'$) de la forme $(s1, e, s2)$ (resp. $(s1', e', s2')$,
- tr désigne une trace de $\mathcal{A}$,
- Tr désigne un ensemble de traces de $\mathcal{A}$.

D'autre part, nous considérons les fonctions suivantes :
- `sim(t)` calcule l'ensemble des transitions t' de $\mathcal{A}'$ telles que `SimTrans(t,t')`,
- `ori(t)` retourne l'état initial de la transition t,
- `but(t)` retourne l'état final de la transition t,
- `queue(tr)` retourne la trace tr privée de la première transition,
- `tete(tr)` retourne la première transition de la trace tr.

L'algorithme suivant calcule les traces images d'une trace tr étant donnée une relation de simulation $SimTrans$.

```
Images_TraceUnique(tr,Simtrans) {
  si (longueur(tr)=1) alors
     retourner sim(tr)
  sinon
     TMP := Images_TraceUnique(queue(tr), Simtrans);
     res = vide;
     pour tr_tmp in TMP faire
       pour t in sim(tete(tr)) faire
         si (but(t) = ori(tete(tr_tmp))) alors
            res += t.tr_tmp;
     retourner res;
}
```

L'algorithme suivant calcule les traces images d'un ensemble de traces Tr, i.e. l'ensemble $Img_R(tr(\mathcal{A}))$ sur la figure 4.11.

```
Images_Traces(Tr, Simtrans) {
  res = vide;
  pour tr in Tr faire
    res += Images_TraceUnique(tr,Simtrans);
  retourner res;
}
```

L'algorithme suivant effectue le calcul de l'image des traces de $\mathcal{M}$ en considérant la fonction de projection `proj(A,tr)` qui projette la trace `tr` sur le composant `A`, et la fonction de substitution `subst(tr,A,ta)` qui remplace dans `tr proj(A,tr)` par `ta`, où `ta` est une trace donnée de $\mathcal{A}$.

```
Images_TracesComposees(Tr,Simtrans) {
  res = vide;
  pour tr in Tr faire
    projtr = proj(A,tr);
    TMP = Images_TraceUnique(projtr,Simtrans);
    pour ta in TMP faire
      res += subst(tr,A,ta);
  retourner res;
}
```

4.3.7.3 Exemple

Soient $n\mathcal{A}$ et $n\mathcal{A}'$ deux nœuds AltaRica définis par le code ci-dessous :

```
node nA
  state s:[0,3];
  event e1,e2,e3,e4;
  trans
    s = 0 |- e1 -> s := 1;
    s = 0 |- e2 -> s := 2;
    s = 1 |- e3 -> s := 3;
    false |- e4 -> ;
  init  s := 0;
edon
```

```
node nA'
  state s:[0,3];
  event e1,e2,e3;
  trans
    s = 0 |- e1 -> s := 1;
    s = 0 |- e2 -> s := 2;
    s = 1 |- e3 -> s := 3;
    s = 2 |- e4 -> s := 3;
  init  s := 0;
edon
```

Désignons par s_i (resp. s'_i) les états de $n\mathcal{A}$ (resp. $n\mathcal{A}'$) tels que s = i. La sémantique des nœuds $n\mathcal{A}$ et $n\mathcal{A}'$ est représentée par les STI ci-après :

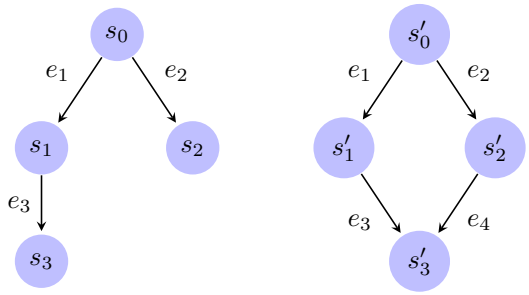


FIG. *4.13: Sémantique des nœuds $n\mathcal{A}$ et $n\mathcal{A}'$*

Les nœuds $n\mathcal{A}$ et $n\mathcal{A}'$ définis ci-dessus ne contiennent pas de variable de flux, donc tout état de $n\mathcal{A}$ dispose des mêmes flux que tout état de $n\mathcal{A}'$. La relation $RelF$ s'écrit :

```
RelF(s: nA!c, s':nA'!c) := true;
```

D'autre part, considérons la relation $RelEvt$ suivante :

```
RelEvt(e: nA!ev, e':nA'!ev) := (e.='' '' & e'.='' '')
                             | (e.=''e1'' & e'.=''e1'')
                             | (e.=''e1'' & e'.=''e2'')
                             | (e.=''e2'' & e'.=''e1'')
                             | (e.=''e2'' & e'.=''e2'')
                             | (e.=''e3'' & e'.=''e3'')
                             | (e.=''e3'' & e'.=''e4'');
```

Les transitions (s'_0, e_1, s'_1) et (s'_0, e_2, s'_2) de $n\mathcal{A}'$ peuvent être simulées par la transition (s_0, e_1, s_1) de $n\mathcal{A}$. D'autre part, les transitions (s'_1, e_3, s'_3) et (s'_2, e_4, s'_3) de $n\mathcal{A}'$ peuvent être simulées par la transition (s_1, e_3, s_3) de $n\mathcal{A}$. D'après la définition 4.3.2, on peut donc dire que $n\mathcal{A}$ simule $n\mathcal{A}'$.

Désignons par t_i (resp. t'_i) la transition de $\mathcal{A}$ (resp. $\mathcal{A}'$) étiquetée par e_i et regardons maintenant les traces de chaque nœud :
- $tr(n\mathcal{A}) = \{t_1t_3, t_2\}$,
- $tr(n\mathcal{A}') = \{t'_1t'_3, t'_2t'_4\}$.

Étant donnée la relation $SimTrans$:
- t_1 est en relation avec t'_1 et t'_2,
- t_3 est en relation avec t'_3 et t'_4,
- t_2 n'est en relation avec aucune transition de $\mathcal{A}'$.

En appliquant les algorithmes `ComputeImagesOf_oneTrace` et `ComputeImagesOf_Traces`, on obtient que $Img_R(tr(n\mathcal{A}))$ est égal aux traces $t'_1t'_3$ et $t'_2t'_4$, donc on a bien $Img_R(tr(n\mathcal{A})) = tr(\mathcal{A}')$.

Soit un nœud $\mathcal{M}$ contenant un nœud $\mathcal{A}$, de type`nA`, et un nœud $\mathcal{B}$, de type`nB`, tels que décrit ci-dessous. L'unique trace de $\mathcal{M}$ est illustrée sur la partie gauche de la figure 4.15.

```
node nB
  state s:[0,2];
  event b1,b2;
  trans
    s = 0 |- b1 -> s := 1;
    s = 0 |- b2 -> s := 2;
  init  s := 0;
edon
```

```
node M
  sub    A:nA;
         B:nB
  state  evtA:bool;
  event  eA,eB;
  trans
     evtA |- eA -> evtA := false;
    ~evtA |- eB -> evtA := true;
  init   evtA := true;
edon
```

La sémantique du nœud $n\mathcal{B}$ est représentée par la figure 4.14.

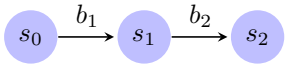


FIG. *4.14: Sémantique du nœud $n\mathcal{B}$*

Soit le nœud $\mathcal{M}'$ obtenu en remplaçant $\mathcal{A}$ par $\mathcal{A}'$ dans $\mathcal{M}$. L'algorithme $ComputeTraces$ fournit, pour $\mathcal{M}'$, les deux traces illustrées sur la partie droite de la figure 4.15.

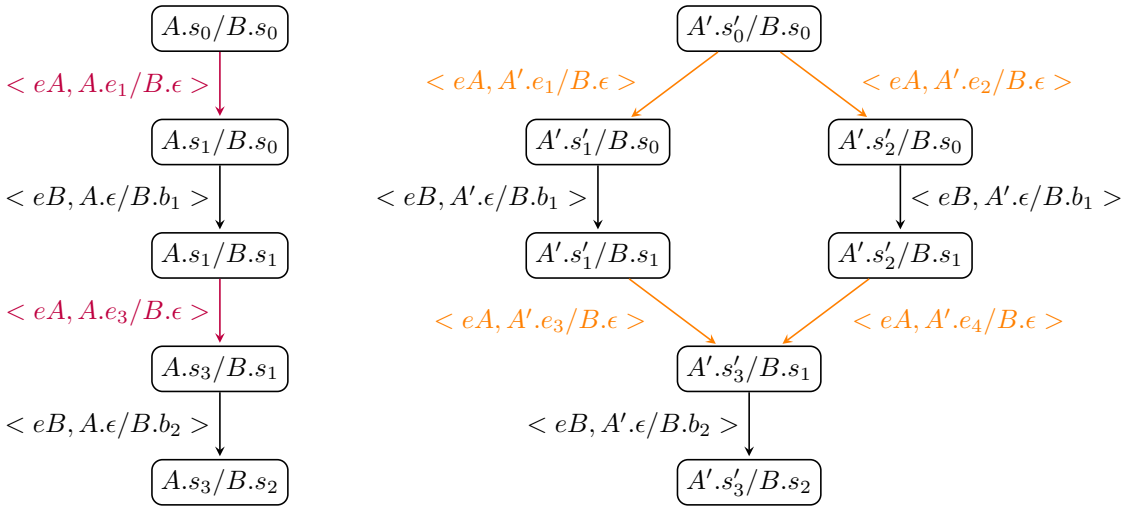


FIG. *4.15: Trace du nœud $\mathcal{M}$ et traces calculées du nœud $\mathcal{M}'$*

4.4 SIMULATION QUASI-BRANCHANTE INTERFACÉE

4.4.1 LIMITES DE LA SIMULATION

Nous avons précédemment défini une relation de simulation telle que tout événement du nœud raffiné est associé à un événement du nœud abstrait.

Cette relation s'applique naturellement à des formes de raffinement à granularité constante, lorsque le modèle raffiné précise certains comportements du modèle abstrait sans introduire de nouveau nœud.

Cependant, la contrainte sur les événements s'avère très forte lorsqu'un nœud abstrait est raffiné par une hiérarchie de nœuds : tout événement d'un sous-nœud, dit *événement détaillé*, doit être associé à un événement du nœud abstrait, dit *événement abstrait*. Détailler un composant en définissant ses sous-composants a pour but de préciser des mécanismes absents du modèle abstrait et donc d'introduire de nouveaux événements qui ne peuvent pas être associés à des événement abstraits.

4.4.2 BESOINS ISSUS DE LA DÉMARCHE MÉTHODOLOGIQUE

La conception de systèmes aéronautiques considère que tout composant peut être soumis à des défaillances et impose donc des objectifs de tolérance aux pannes : chaque système doit remplir sa fonction malgré des défaillances (le nombre de défaillances tolérées dépend de l'importance de la fonction). Des principes d'architectures redondées, des surveillances et des reconfigurations permettent d'atteindre ces objectifs.

Durant les premières phases de conception des systèmes, différentes architectures préliminaires très abstraites sont réalisées, puis comparées suivant différents critères : coût, poids, sécurité, fiabilité... Au fur et à mesure que la conception du système progresse, l'architecture retenue est affinée : les équipements sont décomposés en composants.

Ain de réaliser des analyses de sûreté de fonctionnement des systèmes au plus tôt durant la conception, les modèles AltaRica réalisés doivent évoluer parallèlement aux architectures des systèmes. Ainsi, les premiers modèles très abstraits représentant les architectures préliminaires sont suivis de modèles plus détaillés pour lesquels un nœud abstrait simple est remplacé par une hiérarchie composée d'un nœud ayant les mêmes variables de flux que le nœud abstrait mais contenant plusieurs sous-nœuds. Cette approche correspond à une forme naturelle de raffinement évoquée en introduction de ce chapitre.

Les modèles AltaRica que nous réalisons contiennent deux types d'événements :

– des défaillances (événements stochastiques),

– des événements internes, tels que des surveillances ou des reconfigurations, que l'on peut percevoir comme des événements de contrôle du système.

Nous prenons pour hypothèse que, sauf défaillance particulière d'une surveillance/reconfiguration, les événements internes sont instantanés et permettent, suite à une défaillance, de revenir dans un état sûr unique (plusieurs événements internes peuvent se succéder, mais l'ordre d'occurrence n'influe pas). Cette hypothèse induit une notion de priorité des événements internes sur les défaillances.

Intéressés uniquement par les traces/séquences contenant des défaillances, nous souhaitons masquer les événements internes en transformant le STI de la manière suivante : une transition étiquetée par une défaillance suivie de transitions étiquetées par des événements internes est remplacée par une transition ayant même état initial et même étiquette mais dont l'état final correspond au dernier état accessible par un événement interne.

Cette transformation, détaillée dans le chapitre 5, implique que les STI manipulés dans la suite du chapitre courant considèrent deux types d'événements :

– des défaillances détectées dont l'effet, "contenu" par des moyens de reconfiguration ou de correction, n'est pas visible sur les variables de sortie du composant concerné,
– des défaillances non détectées (ou du moins non "corrigées") dont l'effet est visible sur les variables de sortie des composants concernés.

Compte tenu des particularités de modélisation (décomposition et types d'événements considérés) et des limites de la simulation que nous venons d'évoquer, nous avons souhaité étudier une relation qui permette de prendre en compte le faire qu'un événement soit raffiné par un chemin.

De nombreuses relations sur les systèmes de transitions considèrent deux catégories d'événements : les événements observables et ceux non observables.

Rob J. Van Glabbeek présente, dans [GW96], des relations dites *faibles*, qui considèrent des chemins composés d'événements non observables et d'un événement observable. La relation baptisée bisimulation quasi-branchante, en anglais *quasi-branching bisimulation* (QBB), définie par R.J.Van Glabbeek a attiré notre attention et fait l'objet l'étude que nous décrivons par la suite.

Notre objectif est, en nous inspirant de la QBB de R.J.Van Glabbeek, de définir une relation :

– considérant la notion d'observabilité des transitions ;
– garantissant des propriétés de compositionnalité (c.f. théorèmes de compositionnalité précédents).

4.4.3 RELATIONS FAIBLES

Considérons des séquences composées d'un événement observable, noté e, et d'événements non observables, notés τ. Nous pouvons prévoir trois cas :

– les événements τ à la fois précèdent et succèdent à e : les chemins sont de la forme $\tau^* e \tau^*$,
– les événements τ précédent e : les chemins sont de la forme $\tau^* e$,
– les événements τ succèdent à e : les chemins sont de la forme $e\tau^*$.

Exemple 4.6

Soient les deux STI $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ et $\mathcal{A}' = \langle E, F, S', \pi', T' \rangle$ illustrés sur la figure 4.16. Un état s_i de $\mathcal{A}$ désigne un état du nœud A tel que $s = i$, tandis qu'un état $s'_{i/j}$ de $\mathcal{A}'$ désigne un état du nœud A' tel que $s = i$ et $t = j$.

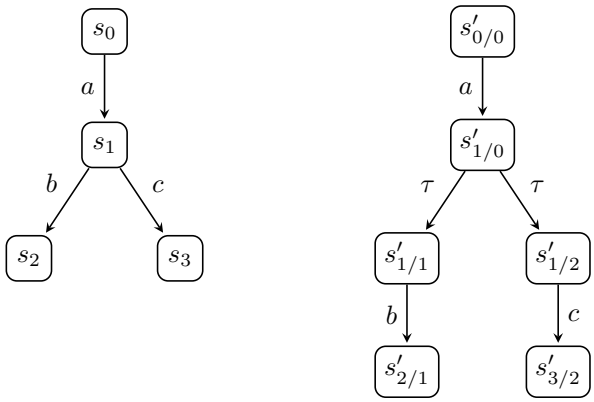


FIG. *4.16: STI $\mathcal{A}$ et $\mathcal{A}'$*

Nous cherchons une relation afin d'établir un lien entre ces deux STI.

La figure 4.17 représente les clôtures transitives de $\mathcal{A}'$, calculées depuis l'état initial, en considérant les trois types de chemins observables : $\tau^* e \tau^*$, $\tau^* e$ et $e\tau^*$.

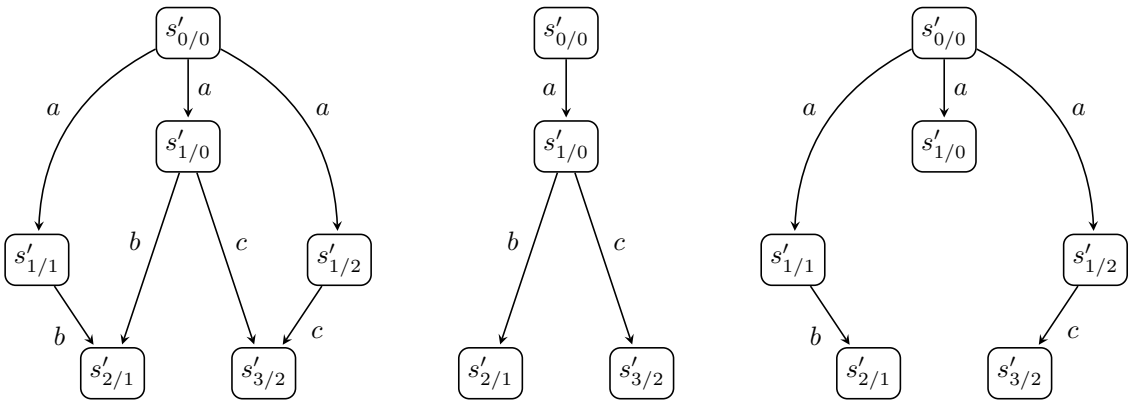


FIG. *4.17: Clotures transitives de $\mathcal{A}'$ selon $\tau^* e \tau^*$,$\tau^* e$ et $e\tau^*$*

Par simple observation des trois STI de la figure 4.17, en considérant la relation sur les flux toujours vraie, on remarque que :

- *il existe une relation de bisimulation interfacée entre $\mathcal{A}$ et la clôture transitive de $\mathcal{A}'$ selon $\tau^* e$,*
- *il existe une relation de simulation interfacée de la clôture transitive de $\mathcal{A}'$ selon $\tau^* e \tau^*$ vers $\mathcal{A}$,*
- *il n'existe pas de relation de simulation interfacée ou de bisimulation interfacée entre $\mathcal{A}$ et la clôture transitive de $\mathcal{A}'$ selon $e\tau^*$.*

Cet exemple oriente nos travaux vers l'étude des relations s'appuyant sur des chemins de type $\tau^* e$. Ce choix est, de surcroît, justifié par le fait que Rob J. Van Glabbeek propose, dans [GW96], une relation d'équivalence sur des chemins de type $\tau^* e$, baptisée bisimulation quasi-branchante, en anglais *quasi-branching bisimulation*. La définition 4.4.1 de cette relation utilise les deux notations suivantes :

- $r \rightarrow^a s$: transition de r vers s étiquetée a
- $r \Rightarrow s$: chemin de r vers s composé de n transitions ($n \geq 0$) non observables τ

Définition 4.4.1 (Bisimulation quasi-branchante)
Deux STI g et g' sont qb-bisimilaires s'il existe une relation symétrique R entre les états de g et g' telle que :

- *les états initiaux sont en relation ;*
- *Si $R(s, s')$ et $s' \rightarrow^a t'$, alors*
  - *soit $a = \tau$ et $\exists s \Rightarrow t$ tel que $R(t, t')$,*
  - *soit il existe un chemin $s \Rightarrow u \rightarrow^a t$ tel que $R(u, s')$ et $R(t, t')$.*

□

Comme nous l'avons mentionné précédemment, les relations de bisimulation sont particulièrement fortes. C'est pourquoi, dans un premier temps, nous souhaitons étudier une relation de simulation inspirée de la définition 4.4.1. En écartant la condition de symétrie sur la relation R dans la définition 4.4.1, nous définissons la relation de *simulation quasi-branchante* suivante :

Définition 4.4.2 (Simulation quasi-branchante)
Un STI g qb-simule un graphe g' s'il existe une relation R entre les états de g et g' telle que :

- *tout état initial de g' est en relation avec un état initial de g ;*
- *Si $R(s, s')$ et $s' \rightarrow^a t'$, alors*
  - *soit $a = \tau$ et $\exists s \Rightarrow t$ tel que $R(t, t')$,*
  - *soit il existe un chemin $s \Rightarrow u \rightarrow^a t$ tel que $R(u, s')$ et $R(t, t')$.*

□

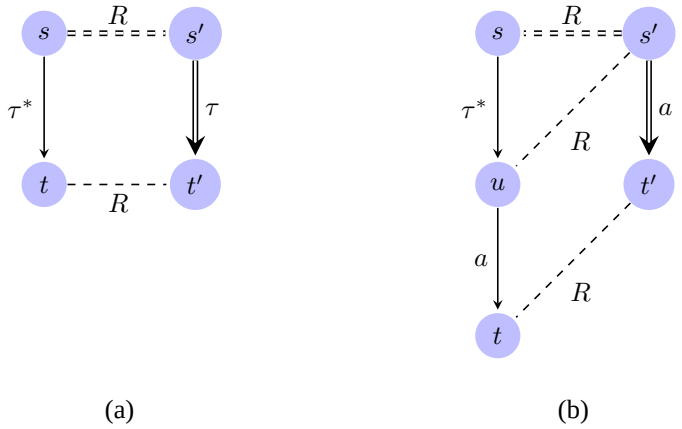


FIG. *4.18: Simulation quasi-branchante*

Remarque

Les relations de simulation et bisimulation, présentées précédemment dans ce chapitre, s'appuient sur la syntaxe initiale du langage AltaRica, tel que défini dans [Poi00]. En revanche, les relations de simulation et bisimulation quasi-branchantes, en introduisant la distinction en événements observables et événements non observables, nécessitent d'étendre la syntaxe du langage et donc de proposer une nouvelle sémantique pour cette extension du langage initial.

4.4.4 SYSTÈME DE TRANSITIONS INTERFACÉ ÉTENDU

En nous inspirant de la définition 4.1.1, nous introduisons la notion de système de transitions interfacé étendu, qui ajoute une notion d'observabilité à chaque événement et chaque transition.

Définition 4.4.3 (Système de transitions interfacé étendu)
Un système de transitions interfacé étendu (STIE) est un système de transitions interfacé $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ où :

- $E = E_+ \cup E_\tau$ *est un ensemble fini d'événements où*
  - E_+ *est un ensemble fini d'événements observables*
  - E_τ *est un ensemble fini d'événements non observables*
  - ϵ *est un événement distingué appartenant à* E_τ.
- $T = T_+ \cup T_\tau$
  - $T_+ \subseteq S \times F \times E_+ \times S$
  - $T_\tau \subseteq S \times F \times E_\tau \times S$
  - $\forall s \in S, (s, f, \epsilon, s) \in T_\tau$

□

La sémantique du langage AltaRica s'appuie sur des STI. Introduire la notion de transitions non observables dans le langage nécessiterait de définir une sémantique basée sur les STIE et donc de proposer une extension du langage.

Pour contourner cette difficulté, nous traitons cette distinction entre transitions observables et non observables directement dans l'outil MecV.

Avant de détailler les résultats de nos travaux et pour faciliter la lecture de la suite de ce chapitre, nous introduisons une notation de chemin.

Définition 4.4.4 (Chemins observables de type $\tau^* e$)
Soient $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ un STI, $e \in E_+$ et T^* la clôture transitive de T, on désigne par $\tau^* e$ les chemins observables de $T_\tau^* T_+$ tels que e étiquette la transition T_+.

□

L'idée évoquée sur l'exemple 4.6 est que lorsqu'un STIE abstrait simule un STIE détaillé, un événement observable du STIE abstrait simule un chemin de type $\tau^* e$. Le STIE détaillé a ainsi la possibilité d'entrelacer les événements observables et les événements non observables pour simuler le STIE détaillé.

Nous observons sur l'illustration b de la figure 4.18 que la relation de simulation quasi-branchante traite le cas contraire à notre idée : un chemin de type $\tau^* e$ du STIE abstrait simule un événement observable du STIE détaillé. Les propriétés suivantes montrent que, du fait que l'événement ϵ est non observable, la simulation quasi-branchante appliquée à des STIE correspond à l'idée évoquée et se traduit par une représentation symétrique de l'illustration b de la figure 4.18.

Propriété 4.7

Les relations de la figure 4.18 sont équivalentes aux relations de la figure 4.19 suivante.

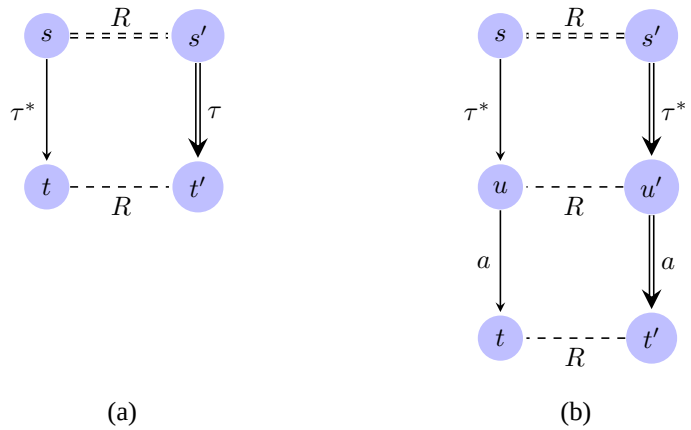


FIG. *4.19: Simulation quasi-branchante*

Soient P_1, P_2 et P_3 les propriétés représentées respectivement par l'automate (a) de la figure 4.18 (et de la figure 4.18 qui est similaire), par l'automate (b) de la figure 4.18 et par l'automate (b) de la figure 4.19.

Preuve :

1. $P_1 + P_3 \Rightarrow P_1 + P_2$ car P_2 est un cas particulier de P_3 ;
2. $P_1 + P_2 \Rightarrow P_1 + P_3$: il suffit de raisonner par induction sur le nombre n de transitions τ à droite dans fig.4.19.(b), en composant des diagrammes.

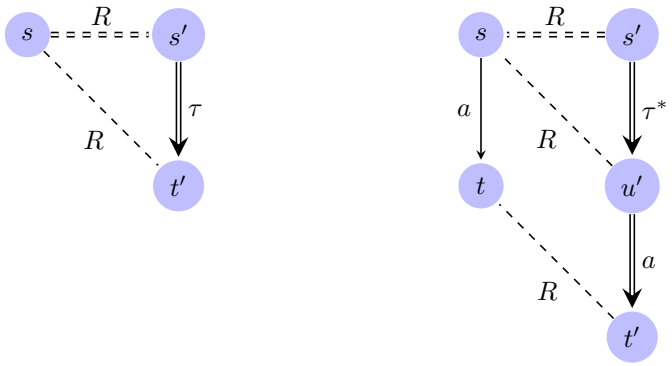
◇

Propriété 4.8

Dans le cas particulier où l'ensemble des événements non observables de g est restreint à ϵ, on constate que :

– l'état s de P_1 est égal à l'état t de cette même propriété ;
– l'état s de P_3 est égal à l'état u de cette même illustration ;

Cela se traduit par les relations illustrées sur la figure 4.20. Les relations de la propriété 4.7 sont alors équivalentes aux relations de la figure 4.20. On note que l'illustration (b) des figures 4.18 et 4.20 sont symétriques.



(a) (b)

FIG. *4.20: Simulation quasi-branchante si ϵ est le seul événement non observable de g*

4.4.5 SIMULATION QUASI-BRANCHANTE INTERFACÉE

Précédemment, nous avons défini la relation de simulation interfacée paramétrée en introduisant les relations $RelF$ et $RelEvt$ pour préciser les liens à considérer entre les interfaces (parties visibles) des composants AltaRica. De façon analogue, ces deux relations sont utilisées pour définir la relation de simulation quasi-branchante interfacée paramétrée suivante :

Définition 4.4.5 (Simulation quasi-branchante interfacée paramétrée)
Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E', F', S', \pi', T'\rangle$ deux systèmes de transitions interfacés étendus. Soient $RelF \subseteq F \times F'$ une relation entre variables de flux et $RelEvt \subseteq E \times E'$ une relation entre événements. $R(RelF, RelEvt) \subseteq S \times S'$ est une relation de simulation quasi-branchante interfacée paramétrée *de $\mathcal{A}$ vers $\mathcal{A}'$ si :*

1. $\forall s' \in S', \exists s \in S, (s, s') \in R(RelF, RelEvt)$;
2. $\forall (s', f', e', t') \in T', \forall s \in S, (s, s') \in R(RelF, RelEvt) \Rightarrow$

- *si* $(s', f', e', t') \in T'_\tau$
  alors $\exists f \in F$, $\exists t \in S$ *t.q.* $(s, f, \tau^*, t) \in T^*_\tau$, $(f, f') \in RelF$, $(t, t') \in R(RelF, RelEvt)$;
- *si* $(s', f', e', t') \in T'_+$
  alors $\exists f, f_i \in F$, $\exists e \in E$, $\exists u, t \in S$ *t.q.*
  - $(f, f') \in RelF$,
  - $(e, e') \in RelEvt$,
  - $(s, f, \tau^*, u) \in T^*_\tau$,
  - $(u, f_i, e, t) \in T_+$,
  - $(u, s') \in R(RelF, RelEvt)$,
  - $(t, t') \in R(RelF, RelEvt)$.

□

$RelF$ et $RelEvt$ contraignent les relations R mais ne suffisent pas à les définir.

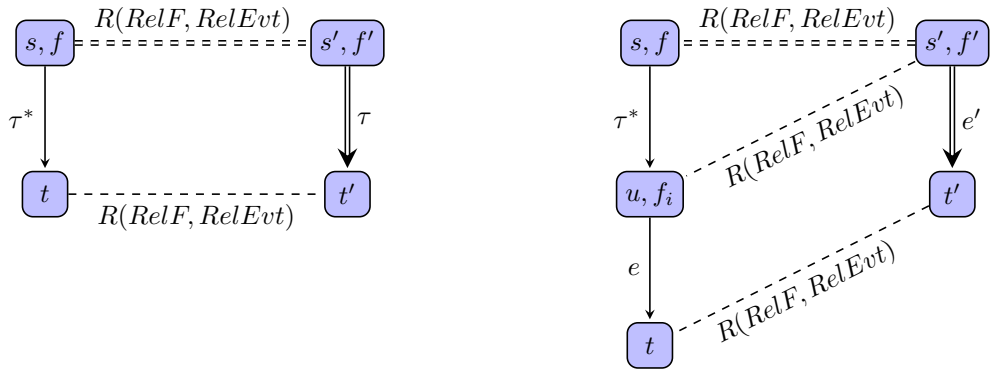


FIG. *4.21: Simulation quasi-branchante interfacée*

Dans la suite, sauf mention contraire, pour simplifier les écritures (et les démonstrations) nous ferons l'hypothèse que la relation $RelF$ est l'égalité des flux et que la relation $RelEvt$ est l'identité d'étiquette.

La définition 4.4.5 devient :

Définition 4.4.6 (Simulation quasi-branchante interfacée)
Si $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ *et* $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ *sont des systèmes de transitions interfacés étendus, alors la relation* $R \subseteq S \times S'$ *est une relation de* simulation quasi-branchante interfacée *de* $\mathcal{A}$ *vers* $\mathcal{A}'$ *si :*

1. $\forall s' \in S', \exists s \in S, (s, s') \in R$;
2. $\forall (s', f', e', t') \in T', \forall s \in S, (s, s') \in R \Rightarrow$
   - *si* $(s', f', e', t') \in T'_\tau$
     alors $\exists t \in S$ *t.q.* $(s, f', \tau^*, t) \in T^*_\tau$ *et* $(t, t') \in R$,
   - *si* $(s', f', e', t') \in T'_+$
     alors $\exists u, t \in S$ *t.q.* $(s, f', \tau^*, u) \in T^*_\tau$, $(u, f', e', t) \in T_+$, $(u, s') \in R$ *et* $(t, t') \in R$.

□

Enfin, comme pour la plupart des relations de simulation ou de bisimulation, la relation de simulation quasi-branchante doit être vérifiée pour les états initiaux.

Définition 4.4.7
Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ *et* $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ *deux systèmes de transitions interfacés,* $initA$ *(resp.* $initA'$*) l'ensemble des états initiaux de* $\mathcal{A}$ *(resp.* $\mathcal{A}'$*).*

On dira que $\mathcal{A}$ *qb-simule* $\mathcal{A}'$ *ssi il existe une relation de simulation quasi-branchante interfacée R, telle que* $\forall s' \in InitA'$, $\exists s \in InitA$ *et* $(s, s') \in R$.

□

Dans le cas où $\mathcal{A}$ et $\mathcal{A}'$ sont les sémantiques respectives des nœuds AltaRica $n\mathcal{A}$ et $n\mathcal{A}'$, on dira que $n\mathcal{A}$ qb-simule $n\mathcal{A}'$.

Remarque

Sur des systèmes où l'unique événement non observable est ϵ, la simulation quasi-branchante coïncide avec la relation standard de simulation, précédemment définie. La simulation quasi-branchante étend donc la simulation.

4.4.6 IMPLÉMENTATION MEC V

La définition 4.4.6 considère des chemins observables de type $\tau^* e$ composés de transitions non observables suivies d'une transition observable (c.f. définition 4.4.4).

Comme nous l'avons indiqué précédemment, pour étudier la relation de simulation quasi-branchante interfacée sans modifier les outils actuels, nous spécifions les événements non observables par une relation MecV, baptisée `isTau_A`. Compte tenu de la définition 4.4.3, cette relation contient au moins l'événement ϵ du nœud abstrait considéré. Soit le nœud abstrait `nA` tel que $E_\tau = \{\epsilon\}$, la relation `isTau_A` s'écrit alors :

```
isTau_A(e : nA!ev) := (e.="");
```

Les transitions observables, baptisées `ObsTrans_A`, et non observables, baptisées `TauTrans_A`, sont calculées à partir de `isTau_A`, par les relations suivantes :

```
TauTrans_A(s,e,t) := (transA(s,e,t) & isTau_A(e));
ObsTrans_A(s,e,t) := (transA(s,e,t) & ~isTau_A(e));
```

Soient `TauStar_A(s,t)` les chemins d'un état s à un état t du nœud abstrait uniquement constitués de transitions non observables, la formule MecV correspondante est :

```
TauStar_A(s,t)+= (s=t)
                 | <e>TauTrans_A(s,e,t)
                 | <v>(TauStar_A(s,v) & TauStar_A(v,t));
```

Soient `TauStarE_A(s,u,e,t)` les chemins d'un état s à un état t du nœud abstrait composés de transitions non observables de l'état s à l'état u puis d'une transition observable étiquetée par l'événement e de l'état u à l'état t. La formule MecV correspondante est :

```
TauStarE_A(s,u,e,t) := (TauStarA(s,u) & ObsTrans_A(u,e,t));
```

Les formules de calcul analogues pour le nœud détaillé sont obtenues en remplaçant A par A'.

La relation `RelEvt` permet de définir l'identité d'étiquette ou, plus généralement, les événements en relation. Dans le cas des chemins de type $\tau^* e$, les transitions non observables sont considérées pour leur observabilité et non leur étiquette. Autrement dit, tout événement non observable du nœud abstrait doit être mis en relation avec tout événement non observable du nœud détaillé. L'événement ϵ est implicite à tout nœud et, d'après la définition 4.4.3, non observable. La relation `RelEvt` doit donc spécifier des équivalences de la forme : ϵ/ϵ, ϵ/τ, τ/ϵ et τ/τ.

La spécification en langage MecV correspondant à la définition 4.4.5 de la simulation quasi-branchante interfacée paramétrée est :

```
A_QBSim_A'(s,s') -=
    ([e'][t'](transA'(s',e',t')=>
                  (TauTrans_A'(s',e',t')
                  & <t>(TauStar_A(s,t)
                        & RelF(s,s')
                        & A_QBSim_A'(t,t')))
                 |(ObsTrans_A'(s',e',t')
                  & <u><e><t>(TauStarE_A(s,u,e,t)
                              & RelF(s,s')
```

```
                                        & RelEvt(e,e')
                                        & A_QBSim_A'(u,s')
                                        & A_QBSim_A'(t,t')))))

A_QBSimule_A'(x) :=
    x = ([s'] (initA'(s) =>
                <s> (initA(s') & A_QBSim_A'(s,s'))));
```

Les relations permettant de vérifier s'il existe une relation de simulation quasi-branchante d'un nœud détaillé vers un nœud abstrait sont obtenues en remplaçant A par A' et inversement.

Exemple 4.7

Considérons les nœuds AltaRica correspondant aux deux STI de la figure 4.16 et vérifions si cette relation permet d'établir le lien entre eux.

```
node nA
  state s: [0,3];
  event a,b,c;
  trans s=0 |- a -> s:=1;
        s=1 |- b -> s:=2;
        s=1 |- c -> s:=3;
  init s:=0;
edon
```

```
node nA'
  state s: [0,3];
        t: [0,2];
  event a,b,c,tau;
  trans s=0 |- a -> s:=1;
        s=1 and t=0 |- tau -> t:=1;
        s=1 and t=0 |- tau -> t:=2;
        s=1 and t=1 |- b -> s:=2;
        s=1 and t=2 |- c -> s:=3;
  init  s:=0;
        t:=0;
edon
```

L'événement `tau` étant le seul événement non observable défini dans le code AltaRica, les événements non observables de ces deux nœuds sont spécifiés par les relations suivantes :

```
isTau_A(e : nA!ev) := (e.="");
isTau_A'(e : nA'!ev) := (e.="")|(e.="tau");
```

Afin de tester la relation `QBSim_A_A'`, considérons les relations `RelF` et `RelEvt` suivantes :

```
RelF(a : nA!c, b : nA'!c) := (a.s = b.s);

RelEvt(e : nA!ev, e' : nA'!ev) := ((e.="a" & e'.="a")
                                  |(e.="b" & e'.="b")
                                  |(e.="c" & e'.="c")
                                  |(e.="" & e'.="tau")
                                  |(e.="" & e'.="""));
```

L'outil MecV nous apprend qu'il existe une relation de simulation quasi-branchante de `nA` vers `nA'` mais pas de `nA'` vers `nA`. Ce second résultat est dû au fait que les états $s'_{1/1}$ et $s'_{1/2}$, ne permettant qu'un événement, ne peuvent être mis en relation avec l'état s_1.

Cet exemple nous montre que notre objectif d'établir un lien entre les deux STI de la figure 4.16 est atteint. Cependant, la nécessité que tous les états comparés soient en relation de simulation quasi-branchante reste une condition trop forte pour établir une simulation dans les deux sens.

4.4.7 COMPOSITIONNALITÉ ET RESTRICTIONS SUR LE LANGAGE

De même que pour la simulation interfacée précédemment étudiée, nous cherchons à établir un théorème de compositionnalité similaire au théorème 4.2 pour la relation de simulation quasi-branchante

interfacée.

Appliquant la même démarche que pour la simulation interfacée, nous étudions deux particularités d'AltaRica :

– les priorités entre événements ;
– les synchronisations avec diffusion.

Il s'avère que ces deux aspects ne permettent pas de préserver la compositionnalité. Pour le démontrer, nous présentons ci-après deux contre-exemples où le nœud $n\mathcal{A}$ simule $n\mathcal{A}'$ mais le nœud $Main\mathcal{A}$ (contenant $n\mathcal{A}$) ne simule pas $Main\mathcal{A}'$ (obtenu en remplaçant $n\mathcal{A}$ par $n\mathcal{A}'$ dans $Main\mathcal{A}$).

4.4.7.1 Non-préservation de la compositionnalité due à la notion de priorité entre événements

Exemple 4.8

Soient $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ et $\mathcal{A}' = \langle E, F, S', \pi', T'\rangle$ deux systèmes de transitions interfacés.

```
node nA
  state s: [0,3];
  event a,tau,b,c;
  trans s=0 |- a -> s:=1;
        s=1 |- b -> s:=2;
        s=1 |- c -> s:=3;
  init s:=0;
edon
```

```
node nA'
  state s: [0,3];
        t: [0,1];
  event a,tau,b,c;
  trans s=0 |- a -> s:=1;
        s=1 and t=0|- tau -> t:=1;
        s=1 and t=1|- b -> s:=2;
        s=1 and t=0|- c -> s:=3;
  init s:=0;
       t:=0;
edon
```

Les systèmes $\mathcal{A}$ et $\mathcal{A}'$ sont définis par les transitions suivantes :

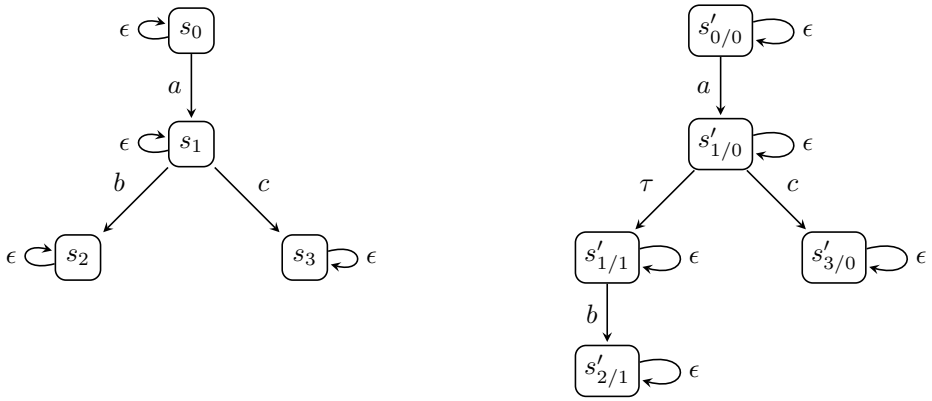


FIG. *4.22: Sémantique des nœuds nA et nA'*

Les nœuds $n\mathcal{A}$ et $n\mathcal{A}'$, définis ci-dessus, ne contiennent pas de variable de flux. Néanmoins, considérons que deux états sont en relation s'ils disposent de la même valeur pour la variable d'état s. La relation $RelF$ s'écrit :

```
RelF(a: nA!c, a': nA'!c) := (a.s = a'.s);
```

D'autre part, considérons la relation $RelEvt$, identité des étiquettes, suivante :

```
RelEvt(e: nA!ev, e': nA'!ev) := (e.=''a'' & e'.=''a'')
```

```
| (e.=''b'' & e'.=''b'')
| (e.=''c'' & e'.=''c'')
| (e.='' '' & e'.='' '')
| (e.='' '' & e'.=''tau'');
```

Déclarons τ et ϵ comme les événements non observables de ces nœuds à l'aide des relations suivantes :

```
isTau_A(e : nA!ev) := (e.="");
isTau_A'(e : nA'!ev) := (e.="")|(e."tau");
```

Les chemins ab et ac de $n\mathcal{A}$ sont respectivement équivalents aux chemin $a\tau b$ et ac de $n\mathcal{A}'$, donc $n\mathcal{A}$ qb-simule $n\mathcal{A}'$.

Supposons maintenant que $n\mathcal{A}$ et $n\mathcal{A}'$ soient des composants des nœuds $MainA$ et $MainA'$ (contenant respectivement $n\mathcal{A}$ et $n\mathcal{A}'$), définis par le code AltaRica suivant :

```
node MainA
  sub    N : nA;
  event a,tau>c>b;
  trans true |- a,tau,c,b -> ;
  sync
    <a,N.a>;
    <b,N.b>;
    <c,N.c>;
    <tau,N.tau>;
edon
```

```
node MainA'
  sub    N : nA';
  event a,tau>c>b;
  trans true |- a,tau,c,b -> ;
  sync
    <a,N.a>;
    <b,N.b>;
    <c,N.c>;
    <tau,N.tau>;
edon
```

Les nœuds $MainA$ et $MainA'$ synchronisent chaque événement de N avec un événement homonyme et définissent les priorités suivantes : $\tau > c > b$.

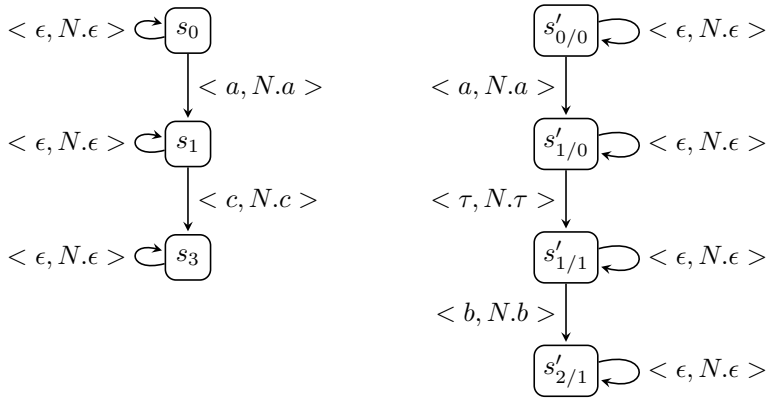


FIG. *4.23: Sémantique des nœuds nA et nA'*

Adaptons les relations `RelF` et `RelEvt` précédentes aux nœuds $MainA$ et $MainA'$.

```
RelF(a: mainA!c, a':nA'!c) := (a.N.s = a'.N.s);
```

```
RelEvt(e: mainA!ev, e': mainA'!ev) := (e.=''a'' & e'.=''a'')
                                    | (e.=''b'' & e'.=''b'')
                                    | (e.=''c'' & e'.=''c'')
                                    | (e.='' '' & e'.='' '')
                                    | (e.='' '' & e'.=''tau'');
```

Supposons que ces deux nœuds soient contenus respectivement dans les nœuds $MainA$ et $MainA'$ qui imposent les priorités suivantes : les priorités imposées par les nœuds $MainA$ et $MainA'$ ($\tau > c > b$) ont pour conséquence qu'un seul chemin est possible dans chacun de ces deux nœuds :

- *ac dans le nœud $MainA$*
- *ab dans le nœud $MainA'$*

Le chemin a ne qb-simule pas le chemin ab, donc le nœud $MainA$ ne qb-simule pas le nœud $MainA'$.

Ce contre-exemple montre que définir des priorités entre des événements observables et des événements non observables ne permet pas de maintenir des propriétés de bisimulation faible dans le cas de la composition.

Étudions la question de la diffusion dans les synchronisations.

4.4.7.2 Non-préservation de la compositionnalité due à la notion de diffusion dans les synchronisations

Exemple 4.9

Soient nA et nA' deux nœuds AltaRica définis par le code ci-dessous :

```
node nA
  state s: [0,2];
  event a,b;
  trans s=0 |- a -> s:=1;
        s=1 |- b -> s:=2;
  init s:=0;
edon
```

```
node nA'
  state s: [0,2];
        t: [0,1];
  event a,tau,b;
  trans s=0 |- a -> s:=1;
        s=1 and t=0 |- tau -> t:=1;
        s=1 and t=1 |- b -> s:=2;
  init s:=0;
       t:=0;
edon
```

Désignons par s_i (resp. s'_i) les états de nA (resp. nA') tels que s = i. La sémantique des nœuds nA et nA' est représentée par les STI ci-après :

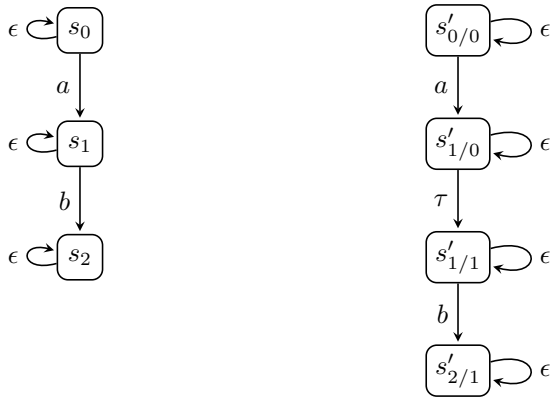


FIG. *4.24: Sémantique des nœuds nA et nA'*

Les nœuds nA et nA', définis ci-dessus, ne contiennent pas de variable de flux. Néanmoins, considérons que deux états sont en relation s'ils disposent de la même valeur pour la variable d'état s. La relation $RelF$ s'écrit :

```
RelF(a: nA!c, a': nA'!c) := (a.s = a'.s);
```

D'autre part, considérons la relation $RelEvt$, identité des étiquettes, suivante :

```
RelEvt(e: nA!ev, e': nA'!ev) := (e.=''a'' & e'.=''a'')
                              | (e.=''b'' & e'.=''b'')
                              | (e.='' '' & e'.='' '')
                              | (e.='' '' & e'.=''tau'');
```

Déclarons τ et ϵ comme les événements non observables de ces nœuds à l'aide des relations suivantes :

```
isTau_A(e : nA!ev) := (e.="");
isTau_A'(e : nA'!ev) := (e.="")|(e.="tau");
```

Le chemin ab de $n\mathcal{A}$ est équivalent au chemin $a\tau b$ de $n\mathcal{A}'$, donc $n\mathcal{A}$ qb-simule $n\mathcal{A}'$.

Supposons maintenant que $n\mathcal{A}$ et $n\mathcal{A}'$ sont des composants des nœuds $MainA$ et $MainA'$ (contenant respectivement $n\mathcal{A}$ et $n\mathcal{A}'$), définis par le code AltaRica suivant :

```
node MainA
  sub   N : nA;
  state s1,s2 : bool;
  event a,b;
  trans
    s1 |- a -> s1:=false;
    not s1 and s2 |- b -> s2:=false;
  sync
    <a,N.a>;
    <b,N.b?>;
  init s1:=true;
       s2:=true;
edon
```

```
node MainA'
  sub   N : nA';
  state s1,s2 : bool;
  event a,b;
  trans
    s1 |- a -> s1:=false;
    not s1 and s2 |- b -> s2:=false;
  sync
    <a,N.a>;
    <b,N.b?>;
  init s1:=true;
       s2:=true;
edon
```

Les nœuds $MainA$ et $MainA'$ synchronisent l'événement b avec l'événement $N.b$ (i.e. l'événement b du nœud fils N) si possible.

La figure ci-dessous représente les sémantiques des nœuds $MainA$ et $MainA'$. Par souci de lisibilité, les boucles d'un état sur lui-même, étiquetées $< \epsilon, N.\epsilon >$, n'apparaissent pas. Chaque état de $MainA$ dispose d'une étiquette de la forme s_x où x représente la valeur de la variable s. Chaque état de $MainA'$ dispose d'une étiquette de la forme $u, v, s_{x/y}$ où u et v sont les valeurs respectives de $s1$ et $s2$, x est la valeur de s et y est la valeur de s.

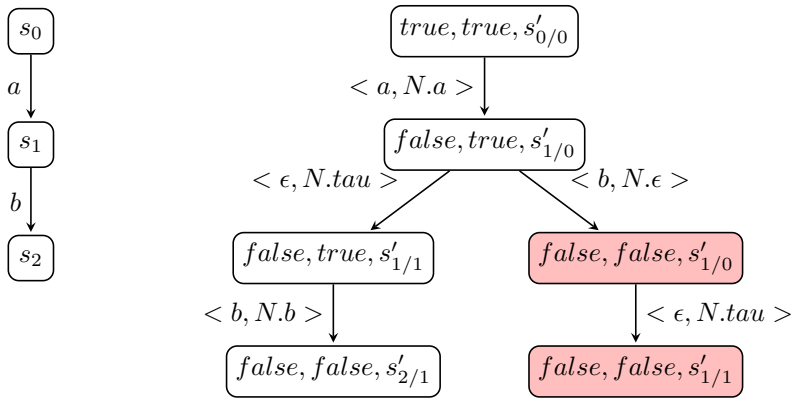


FIG. *4.25: Sémantique des nœuds $MainA$ et $MainA'$*

Définir l'événement τ comme non observable revient à le synchroniser avec ϵ.
Dans le nœud $MainA$, outre les événements ϵ possibles à partir de chaque état, les chemins sont de la

forme ab.
Dans le nœud $MainA'$, une fois l'événement a tiré, deux synchronisations sont tirables : $< \epsilon, N.\tau >$ et $< b, N.\epsilon >$.
Si la première transition est tirée, il est possible de tirer $< b, N.b >$ et le chemin alors joué est équivalent observationnellement au chemin de $MainA$.
Si la seconde transition est tirée, l'événement $N.b$ ne peut plus être tiré (car b a déjà été tiré), donc le chemin n'est pas jouable sur $MainA$.

La transition étiquetée $< b, N.\epsilon >$ du nœud $MainA'$ n'ayant pas d'équivalent dans le nœud $MainA$, il n'y a pas de relation de simulation quasi-branchante entre ces deux nœuds.

Ce contre-exemple montre que la diffusion ne permet pas de préserver la simulation quasi-branchante telle qu'elle est définie.

4.4.8 SÉMANTIQUE INTUITIVE

Pour pouvoir établir un théorème de compositionnalité, il nous est nécessaire de proposer une sémantique à l'extension du langage AltaRica introduisant une notion d'observabilité aux événements et aux transitions.

Les restrictions que nous venons d'identifier sont communes à celles identifiées pour la relation de simulation. Les définitions de la syntaxe abstraite, de la sémantique des composants et des nœuds AltaRica que nous proposons sont donc naturellement proches de celles proposées pour la simulation. Afin de distinguer les définitions suivantes des définitions d'origine, nous parlerons de composants et nœuds AltaRica étendu.

Pour ne pas multiplier les références, la suite de ce chapitre s'appuie sur le langage AltaRica tel que défini par Gérald Point.

4.4.8.1 Sémantique des composants

Un composant AltaRica étendu sans priorité ni diffustion est un composant AltaRica sans priorité ni diffusion pour lequel les événements et les transitions sont chacun classés en deux catégorie : observable ou non observable.

Proposition 4.3
La syntaxe abstraite d'un composant AltaRica étendu sans priorité ni diffusion est similaire à la définition 4.1.3 en considérant :
- *pour la relation d'ordre strict $<$, la relation $\emptyset$ (i.e. tous les événements sont incomparables entre eux) ;*
- *$E = E_+ \cup E_\tau$ est un ensemble fini d'*événements *où*
  - *E_+ est un ensemble fini d'événements observables*
  - *E_τ est un ensemble fini d'événements non observables*
  - *ϵ est un événement distingué appartenant à E_τ.*
- *$M = M_+ \cup M_\tau$ est un ensemble de* macro-transitions *(g, e, a) telles que :*
  - *$g \in \mathbb{F}$ est une formule appelée* garde *; elle doit vérifier $vlib(g) \subseteq V_\mathcal{C}$;*
  - *$M_+ \subseteq \mathbb{F} \times E_+ \times \mathbb{T}^{V_S}$ où $e \in E_+$ est l'événement observable déclenchant la transition ;*
  - *$M_\tau \subseteq \mathbb{F} \times E_\tau \times \mathbb{T}^{V_S}$ où $e \in E_\tau$ est l'événement non observable déclenchant la transition ;*
  - *$a : V_S \rightarrow \mathbb{T}$ est une application qui associe à toute variable d'état un terme $a(s)$ tel que $vlib(a(s)) \subseteq V_\mathcal{C}$.*

L'ensemble M_τ contient implicitement la macro-transition $(g_\epsilon, \epsilon, a_\epsilon)$ où $g_\epsilon = tt$ et a_ϵ est l'identité sur V_S.

La sémantique d'un composant AltaRica étendu est un STIE. Cependant, un STIE ne diffère d'un STI que par la notion d'observabilité des événements et transitions. La proposition suivante décrit la sémantique

d'un composant AltaRica étendu, par rapport à celle d'un composant AltaRica.

Proposition 4.4
La sémantique d'un composant AltaRica étendu sans priorité ni diffusion est obtenue en appliquant la définition 4.1.4 avec la relation $\emptyset$ pour relation d'ordre strict $<$ ($[\![M]\!] \upharpoonright \emptyset = [\![M]\!]$) et en considérant :
- *$T = T_+ \cup T_\tau$ est telle que $T = [\![M_+]\!] \cup [\![M_\tau]\!]$ où :*
  - *$T_+ \subseteq S \times F \times E_+ \times S$ et $[\![M_+]\!] = \bigcup_{t \in M_+} [\![t]\!]$*
  - *$T_\tau \subseteq S \times F \times E_\tau \times S$ et $[\![M_\tau]\!] = \bigcup_{t \in M_\tau} [\![t]\!]$*
  - *$[\![(g, e, a)]\!] = \{(s, f, e, t) \mid \exists l \in \mathcal{D}^{V_L}, (s, f, l) \in [\![A \wedge g]\!] \wedge t = a[s, f, l]\}$ où $a[s, f, l]$ dénote la valuation des variables d'état telle que pour tout $v \in V_S$, $a[s, f, l](v) = [\![a(v)]\!](s, f, l)$.*

4.4.8.2 Sémantique des nœuds

La sémantique d'un nœud AltaRica s'appuie sur la notion de vecteur de synchronisation. Tout vecteur est composé d'un événement par sous-nœud. Si aucun événement n'est spécifié pour un composant dans un vecteur, ce vecteur est implicitement complété par l'événement ϵ pour ce composant. Afin de proposer une définition de la sémantique d'un nœud AltaRica étendu qui ne diffère pas trop de la sémantique d'origine, nous conservons ce principe de complétion des vecteurs de synchronisation. Bien que l'événement ϵ soit non observable, nous le distinguons des autres événements non observables.

L'extension du langage concerne principalement les transitions. Nous proposons le principe suivant :
- une transition d'un nœud AltaRica étendu est non observable si tous les événements du vecteur de synchronisation, qui étiquette cette transition, sont non observables ;
- une transition d'un nœud AltaRica étendu est observable si au moins un événement du vecteur de synchronisation qui étiquette cette transition est observable.

La proposition suivante formalise ces principes.

Proposition 4.5 (Nœuds étendus sans priorité ni diffusion – Sémantique)
La sémantique d'un nœud AltaRica étendu $\mathcal{N}$ est le STIE $[\![\mathcal{N}]\!] = \langle E, F, S, \pi, T \rangle$, comparable au STI d'un nœud AltaRica pour lequel $T_\mathcal{N}$ aurait les propriétés suivantes :
- *$T_\mathcal{N} = T_{\mathcal{N}+} \cup T_{\mathcal{N}\tau}$:*
- *$T_{\mathcal{N}+} \subseteq S \times F \times V \times S$ est défini par $\langle (s_0, \ldots, s_n), f, u, (t_0, \ldots, t_n) \rangle \in T_{\mathcal{N}+}$ si et seulement si il existe $f_0 = (f, f_1, \ldots, f_n) \in \pi_0(s_0)$ et $u = (e_0, \ldots, e_n)$ tels qu'il existe $j \in [O, n]$ pour lequel $e_j \in E_+$ et $(s_j, f_j, e_j, t_j) \in T_{+_i}$ et que, de manière générale, pour tout $i \in [0, n]$:*
  - *$e_i \in E$,*
  - *$(s_i, f_i, e_i, t_i) \in T_i$*
  - *$f_i \in \pi_i(s_i)$,*
- *$T_{\mathcal{N}\tau} \subseteq S \times F \times V \times S$ est défini par $\langle (s_0, \ldots, s_n), f, u, (t_0, \ldots, t_n) \rangle \in T_{\mathcal{N}\tau}$ si et seulement si il existe $f_0 = (f, f_1, \ldots, f_n) \in \pi_0(s_0)$ et $u = (e_0, \ldots, e_n)$ tels que pour tout $i \in [0, n]$:*
  - *$e_i \in E_\tau$,*
  - *$(s_i, f_i, e_i, t_i) \in T_{\tau_i}$*
  - *$f_i \in \pi_i(s_i)$,*

Théorème 4.4
Soient $\mathcal{N} = \langle V_F, E, \mathcal{N}_0, \ldots, \mathcal{N}_n, V \rangle$ et $\mathcal{N}' = \langle V_F, E, \mathcal{N}'_0, \ldots, \mathcal{N}'_n, V \rangle$ deux nœuds AltaRica sans priorité et sans vecteur de diffusion tels que pour tout $i = 0 \ldots n$, $[\![\mathcal{N}_i]\!]$ qb-simule $[\![\mathcal{N}'_i]\!]$ alors $[\![\mathcal{N}]\!]$ qb-simule $[\![\mathcal{N}']\!]$.

Preuve : Considérons la relation de simulation quasi-branchante interfacée de la définition 4.4.6 pour

laquelle les relations $RelF$ et $RelEvt$ sont respectivement l'identité des flux et l'identité des événements.

Soit R la relation entre $[\![\mathcal{N}]\!]$ et $[\![\mathcal{N}']\!]$ suivante : $(\vec{s}, \vec{s'}) \in R$ si et seulement si, pour tout i, il existe R_i, une relation de simulation quasi-branchante interfacée entre $[\![\mathcal{N}_i]\!]$ et $[\![\mathcal{N}'_i]\!]$, telle que $(s_i, s'_i) \in R_i$.
Montrons que R est une relation de simulation quasi-branchante interfacée.

1. $\vec{s'} = \langle s'_0, ..., s'_n \rangle \in S'$
   $\forall i \in [0, n]$, R_i est une relation de simulation quasi-branchante interfacée
   donc, pour tout i, $\forall s'_i \in S'_i, \exists s_i \in S_i$ tel que $(s_i, s'_i) \in R_i$.
   Soit $\vec{s} = \langle s_0, ..., s_n \rangle$, on a bien, par définition de R, $(\vec{s}, \vec{s'}) \in R$.
2. Montrons que :
   (a) pour toute transition $(\vec{s'}, f', \tau, \vec{t'}) \in T'_{\mathcal{N}\tau}$, pour tout $\vec{s} \in S$, si $(\vec{s}, \vec{s'}) \in R$ alors il existe $\vec{t} \in S$ tel que $(\vec{s}, f, \tau^*, \vec{t}) \in T^*_\tau$ et $(\vec{t}, \vec{t'}) \in R$.
   Soient $(\vec{s'}, f', \tau, \vec{t'}) \in T'_{\mathcal{N}\tau}$ et $\vec{s} \in S$ tel que $(\vec{s}, \vec{s'}) \in R$.
   Par définition de la sémantique des nœuds étendus sans priorité ni diffusion, $(\vec{s'}, f', e, \vec{t'}) \in T'_\tau$ si et seulement si il existe $f'_0 = (f', f'_1, \ldots, f'_n) \in \pi'_0(s'_0)$ et $e = (e_0, \ldots, e_n)$ tels que pour tout $i \in [0, n]$, $(s'_i, f'_i, e'_i, t'_i) \in T'_{\tau_i}$ et e'_i est non observable.
   Nous désignons par τ tout événement non observable ainsi que l'étiquette d'un vecteur de synchronisation ne contenant que des événements non observables.
   Par hypothèse, $\mathcal{N}'_i$ est qb-simulé par $\mathcal{N}_i$ pour $i = 0, ..., n$ et, d'après la définition 4.4.6, pour toute transition $(s'_i, f', \tau, t'_i) \in T'_{\tau_i}$ si $(s_i, s'_i) \in R$ alors il existe une transition $(s_i, f', \tau^*, t_i) \in T^*_{\tau_i}$.
   Nous en déduisons que, pour tout i, il existe $(s_i, f'_i, \tau^{n_i}, t_i) \in T^+_{i\tau}$, où n_i est le nombre de transitions étiquetées τ nécessaires depuis s_i pour atteindre t_i, et $(t_i, t'_i) \in R$.
   Donc $(\vec{s}, f', \tau^N, \vec{t}) \in T^+_{\mathcal{N}\tau}$ avec $\vec{t} = \langle t_0, ..., t_n \rangle \in S$ et N désigne le nombre total de transitions τ compris entre $max(n_i)$, lorsque les événements non observables des nœuds $\mathcal{N}_i$ sont synchronisés, et $\sum_{i=0}^{n} n_i$, lorsqu'aucun événement non observable n'est synchronisé.

   (b) pour toute transition $(\vec{s'}, f', e, \vec{t'}) \in T'_{\mathcal{N}+}$, pour tout $\vec{s} \in S$, si $(\vec{s}, \vec{s'}) \in R$ alors les états $\vec{u}, \vec{t} \in S$ existent tels que $(\vec{s}, f', \tau^*, \vec{u}) \in T^*_{\mathcal{N}\tau}$, $(\vec{u}, f', e, \vec{t}) \in T^*_{\mathcal{N}+}$, $(\vec{u}, \vec{s'}) \in R$ et $(\vec{t}, \vec{t'}) \in R$.
   Soient $(\vec{s'}, f', e, \vec{t'}) \in T'_{\mathcal{N}+}$ et $\vec{s} \in S$ tel que $(\vec{s}, \vec{s'}) \in R$.

   La preuve formelle étant lourde en raison des nombreux indices, nous proposons une preuve intuitive plus facile à lire.
   Par hypothèse, pour tout $i \in [0, n]$, $\mathcal{N}'_i$ est qb-simulé par $\mathcal{N}_i$. Une transition observable de $\mathcal{N}'$ est étiquetée par un vecteur de synchronisation composé d'au moins un événement observable :
   – pour tout événement observable du vecteur : d'après la définition 4.4.6 de la simulation quasi-branchante, pour toute transition étiquetée par l'événement observable e'_j du composant $\mathcal{N}'_j$, il existe un chemin de la forme $\tau^* e'_j$ du composant $\mathcal{N}_j$ et les états atteints respectifs sont en relation ;
   – pour tout événement non observable du vecteur : nous avons prouvé ci-dessus que, pour tous les autres composants $\mathcal{N}'_i$, $i \in [0, n]$ et $i \neq j$, dont l'événement au sein du vecteur est non observables, l'état atteint par la transition non observable est en relation avec l'état atteint par un chemin non observable de la forme τ^* dans $\mathcal{N}_i$.
   Les différents chemins peuvent être de longueurs différentes. Or, d'après la sémantique d'AltaRica, l'événement ϵ est possible depuis toute configuration. Nous pouvons donc ajouter des transitions étiquetées par ϵ afin d'obtenir des chemins de même longueur, i.e. dont la longueur est égale au plus long chemin $\tau^* e$. Comme nous considérons cet événement comme non observable, cet ajout respecte la définition de la simulation quasi-branchante. Donc l'état atteint par une transition observable de $\mathcal{N}'$ est en relation avec l'état de $\mathcal{N}$ atteint par un chemin de la forme $\tau^* e$. On en déduit que R est une relation de simulation quasi-branchante interfacée.

◇

4.5 LE RAFFINEMENT ALTARICA COMPARÉ

La notion de raffinement fait généralement référence à la méthode B créée par J.R.Abrial [Abr96]. Cependant, cette notion est jugée trop restrictive par certains. Des travaux tels que le retrenchment s'inspirent du raffinement pour proposer une approche jugée plus adaptée à l'industrie. Enfin, le raffinement n'est que rarement dissocié de son dual : l'abstraction. La vérification de propriété par abstraction assistée est au cœur de l'approche Counter-Example Guided Abstraction Refinement (CEGAR) qui fait l'objet de recherches actuellement.

Ces trois notions liées au raffinement AltaRica sont présentées dans la suite de cette section.

4.5.1 LE RAFFINEMENT ALTARICA ET LA MÉTHODE B

Le raffinement AltaRica, que nous proposons, s'inspire de caractéristiques de la méthode B [Abr96] : les modèles sont développés progressivement, par des transformations successives, et sont construits de façon structurée et modulaire.

Contrairement au langage AltaRica créé pour réaliser des analyses de sûreté de fonctionnement des systèmes, la méthode B a pour but le développement de logiciels. Ainsi, la méthode B considère trois types d'objets :

- la *machine abstraite* définit les ensembles, les constantes, les variables et les opérations qui modélisent les données et les services d'un logiciel, fixant ainsi l'interface, le mode et les conditions d'utilisation de ses constituants ;
- les *raffinements* enrichissent progressivement et en vérifiant certaines conditions, les machines abstraites, pour tendre vers une représentation concrète ;
- l'*implantation* constitue le dernier raffinement d'une machine abstraite et représente un modèle exécutable.

La méthode B repose sur les notions d'*invariant* et d'*obligations de preuves*. Chaque objet contient une rubrique *invariant* qui permet de relier les variables d'un raffinement avec les variables de la machine abstraite ou du raffinement précédent : on parle d'*invariant de collage*. L'invariant définit également des conditions qui doivent être préservées par toutes les opérations. Le raffinement n'est validé qu'une fois que la préservation est prouvée : on parle d'*obligations de preuves*. Les obligations de preuves sont générées pour assister l'utilisateur.

D'une certaine manière, le raffinement AltaRica reprend la notion d'invariant de collage :

- la compositionnalité, que nous souhaitons préserver, implique la conservation des variables de flux ;
- la relation `RelF` spécifie les liens entre variables du nœud abstrait et variables du nœud détaillé.

Le modèle AltaRica le plus abstrait impose donc les variables de flux, tout comme la machine abstraite B. Vis-à-vis des variables de flux considérées dans la relation `RelF`, un nœud AltaRica détaillé raffine un nœud abstrait si ce dernier peut le simuler. Autrement dit le nœud détaillé restreint certains comportements, par exemple en précisant la garde de transitions.

Si les outils associés à la méthode B assistent l'utilisateur dans la vérification du raffinement, l'outil MecV effectue cette opération automatiquement, dès lors que les critères de comparaison sont définis (c.f les relations `RelF` et `RelEvt`). Cependant, MecV ne fournit pas de justification lorsque le raffinement n'est pas vérifié. La vérification des relations de simulation et simulation quasi-branchante est réalisée en deux étapes : recherche des couples d'état en relation puis vérification que tout état initial du nœud détaillé est simulé ou qb-simulé par un état initial du nœud abstrait. La première étape permet, par déduction, d'identifier les états du nœud détaillé qui ne sont pas simulés. Dès lors, sur des modèles de taille raisonnable, pour lesquels l'automate est lisible, l'utilisateur peut chercher un chemin de l'état initial jusqu'à cet état non simulé pour comprendre en quoi le nœud détaillé ne raffine pas le nœud abstrait. Sur des modèles pour lesquels l'automate n'est pas lisible, il serait envisageable d'automatiser cette recherche.

4.5.2 LE RETRENCHMENT

Robert Banach, de l'université de Manchester, estime dans [BP98] que le terme *raffinement* évoque communément deux techniques :

– la première, au sens strict, consiste à conserver les entrées/sorties, affaiblir les pré-conditions des opérations et renforcer leurs post-conditions ;
– la seconde consiste à définir des exigences dans une spécification très abstraite progressivement raffinée en précisant les détails manquants.

R. Banach s'intéresse à la seconde forme de raffinement qu'il juge trop restrictive vis-à-vis des méthodes de travail habituelles en matière d'ingénierie des modèles : selon lui, entre autres constats, les concepteurs ont une idée intuitive précise du modèle détaillé, mais pas du modèle abstrait. En réponse aux restrictions identifiées, R.Banach et M.Poppleton proposent, dans [BP98], une technique alternative baptisée *retrenchment*.

Le retrenchment s'inspire du raffinement de la méthode B pour permettre d'enrichir une description en structurant son évolution. Les concepts de la méthode B, tels que les obligations de preuve, sont conservés et adaptés. Ainsi, le retrenchment autorise, entre autres, les changements de type.

Dans le raffinement AltaRica que nous proposons, la relation `RelF` permet de choisir les flux à comparer. Un raffinement stricte met en relation toutes les variables de flux des nœuds à comparer. Cependant, une relation `RelF` ne portant que sur un sous-ensemble de ces variables de flux constitue un assouplissement des contraintes du raffinement, comparable aux motivations du retrenchment.

4.5.3 COUNTER-EXAMPLE GUIDED ABSTRACTION REFINEMENT

Le raffinement dans les méthodes formelles est souvent associé à la notion complémentaire d'abstraction. Vérifier qu'un modèle M satisfait une propriété φ s'avère difficile lorsque l'espace d'état de M est grand. Considérant l'abstraction de l'espace d'état comme une solution au problème d'explosion combinatoire, l'approche CEGAR [CGJ+03] (Counter-Example Guided Abstraction Refinement) propose de vérifier la propriété de sûreté φ en s'aidant de modèles abstraits construits automatiquement. L'approche se décompose ainsi :

1. un modèle initial M_0, très abstrait, est automatiquement généré ;
2. la propriété φ est vérifiée par model-checking sur le modèle M_0 :
   – si $M_0 \models \varphi$, alors $M \models \varphi$,
   – sinon, le model-checker fournit un contre-exemple C_0,
3. le contre-exemple C_0 est simulé sur le modèle M :
   – s'il s'avère qu'il est correct, alors il permet de conclure que $M \not\models \varphi$,
   – si la simulation ne conduit pas dans un état qui contredit φ, alors le modèle M_0 est trop abstrait et il est donc nécessaire de le raffiner,
4. un modèle M_1 est généré à partir de M_0 de façon à exclure le contre-exemple C_0 ;
5. l'opération 2 est menée de nouveau sur le modèle M_1.

Les états accessibles du modèle abstrait constituent une sur-approximation des états accessibles du modèle concret.

Nous avons fait le choix de ne pas orienter nos travaux vers cette approche mais de nous concentrer uniquement sur le raffinement. Fares Chucri, doctorant au LaBRI, étudie l'approche CEGAR pour des modèles AltaRica en s'appuyant sur le model-checker MecV.

4.6 CONCLUSION

La notion de raffinement est classiquement associée à celle de simulation : un modèle détaillé raffine un modèle abstrait si le modèle abstrait simule le modèle détaillé. Nos travaux se sont donc orientés vers l'étude de relations de type simulation. Le langage AltaRica est compositionnel et pensé pour la sûreté de fonctionnement des systèmes ; or, cette dernière s'intéresse à la préservation des séquences de défaillances. Nous avons donc cherché à définir des relations préservant, à la fois, la compositionnalité et les traces. Notre étude s'est portée sur la relation de simulation classique ou forte, pour laquelle tous les événements sont

comparables, et sur la relation de simulation quasi-branchante, pour laquelle tout événement dit “observable” est comparé à un chemin composé d’un événement observable précédé par une suite d’événements dits “non observables”.

Nous avons prouvé que ces deux relations préservent la compositionnalité sous certaines restrictions communes : les modèles comparés ne doivent contenir ni priorité entre événements, ni diffusion dans les vecteurs de synchronisation. De plus, nous sommes parvenus à démontrer que la première relation de simulation préserve les traces, et donc, les séquences (obtenues en ne conservant que les étiquettes des transitions dans les traces). Forts de ce résultat, nous avons proposé un algorithme permettant de calculer les séquences d’un nœud détaillé à partir des séquences d’un nœud et avons donc pleinement atteint notre objectif concernant cette relation.

En revanche, l’étude de la simulation quasi-branchante, motivée par le souhait qu’un nœud abstrait puisse être raffiné par un nœud détaillé composé de sous-nœuds, nous a confronté au besoin d’enrichir la sémantique du langage AltaRica et donc d’adapter les outils existants. Malgré cela, étant convaincus de l’intérêt de cette relation, nous avons souhaité et sommes parvenus à établir un théorème de compositionnalité, mais n’avons pas poursuivi notre étude au-delà.

Les résultats que nous avons obtenus sont positifs dans l’optique d’une industrialisation de l’approche. L’implémentation de l’algorithme évoqué ci-dessus constituerait une première étape dans ce sens et compléterait le raffinement par simulation. Une seconde étape reviendrait à compléter le raffinement pas simulation quasi-branchante : cela nécessiterait d’établir des relations entre traces/séquences, puis de définir et implémenter un algorithme de calcul de séquences du nœud détaillé à partir des séquences du nœud abstrait.

Les objectifs initiaux nous imposaient le langage AltaRica et l’outil MecV, seul capable à ce moment là de traiter directement du code AltaRica. De plus, nous avons choisi deux relations par rapport aux besoins industriels identifiés dans le domaine aéronautique. Les langages formels, les outils dédiés et les applications industrielles évoluant, le raffinement AltaRica peut donner lieu à de nouvelles recherches pour répondre aux questions suivantes :

- Existe-t-il des langages vers lesquels une traduction préserverait la sémantique AltaRica ?
- Ces langages disposent ils d’outils adaptés à l’approche que nous avons définie ?
- Vis-à-vis des opérations que doit réaliser l’outil support à notre approche, existent-ils des outils plus performants ou capables de justifier un non-raffinement ?
- Certains principes de construction d’architecture de système nécessitent-ils d’étudier de nouvelles relations, comme par exemple la boundary-branching simulation présentée dans l’annnexe B ?

5

UTILISATION DU RAFFINEMENT POUR LE DÉVELOPPEMENT DE SYSTÈMES SÛRS

Nous avons deux motivations pour utiliser le raffinement pour le développement des systèmes sûrs :
– l'analyse progressive des modèles pour suivre le niveau de définition des systèmes ;
– l'analyse de systèmes aéronautiques interfacés (cf. chapitre 1.2.

a) Analyse progressive des systèmes

La conception d'un nouvel avion est un processus de développement industriel long et coûteux. Chaque système embarqué est successivement spécifié, ébauché, dessiné, affiné. Plusieurs architectures sont généralement réalisées avant qu'une soit ne retenue, puis fabriquée et finalement installée à bord. Le choix d'une architecture de système n'est généralement arrêté que quelques années après les premières ébauches. Chaque architecture est analysée pour vérifier si elle satisfait un certain nombre d'exigences réglementaires. Le non-respect d'exigences peut nécessiter des corrections, voire la réalisation d'une nouvelle architecture.

Afin d'optimiser les cycles de conception et de réduire les temps d'analyse et de correction, les industriels cherchent à effectuer les analyses de sûreté de fonctionnement des systèmes au plus tôt. Cet objectif sous-entend que les analyses sont menées de manière itérative, sur des architectures à différents niveaux de maturité : les premières analyses traitent des architectures "grossières" afin d'écarter rapidement celles qui ne sont pas conformes aux principales exigences, puis de nouvelles analyses sont menées au fur et à mesure que les spécifications d'architecture se précisent.

Nous avons montré que les analyses de sûreté de fonctionnement des systèmes peuvent être supportées à l'aide de modèles formels et, plus particulièrement, dans le cas qui nous intéresse, à l'aide de modèles AltaRica de propagation de défaillances. Dans une telle approche, les différents types d'analyse étant partiellement automatisés, la modélisation du système reste l'étape la plus coûteuse : chaque description du système doit être spécifiée puis implémentée et enfin validée avant de pouvoir être analysée d'un point de vue sûreté de fonctionnement des systèmes. Or chaque description d'un même système nécessite la réalisation d'un modèle. Pour répondre à l'objectif industriel de réduction des coûts de conception, nous avons cherché à optimiser cette approche en supportant la validation à l'aide de principes de raffinement : un composant détaillé qui raffine un composant abstrait n'introduit pas de nouveaux modes de défaillance et préserve les exigences qualitatives du composant abstrait (i.e. le nombre minimum de défaillances nécessaires pour atteindre un mode de défaillance ou une situation redoutée).

b) Analyses multi-systèmes

Les analyses de type SSA (cf. paragraphe 1.2.2) traitent en détail un système et peuvent être réalisées à l'aide d'un modèle formel. Cependant, il s'avère que les systèmes autonomes (i.e. remplissant une fonction sans interaction avec un autre système) sont rares, car tout système nécessite généralement au moins une alimentation en énergie. Les analyses de type SSA, bien que mono-système en apparence, nécessitent donc de prendre en compte les défaillances des systèmes en interface dont l'effet influe sur le système au centre de l'étude. Par exemple, le système de contrôle de la gouverne de direction présenté dans le chapitre 3 est

interfacé aux systèmes de génération et distribution électrique et hydraulique, eux-mêmes interfacés avec les moteurs.

Pour répondre à cette problématique, l'approche utilisée aujourd'hui par Airbus considère un type particulier de défaillance appelé *défaillance de système en interface*, en anglais Dependent System Failure (DSF). Une DSF apparaît dans un arbre de défaillances ou un diagramme de dépendance comme un mode de défaillance, sa probabilité d'occurrence étant définie par ailleurs. Généralement, une DSF d'un système A dans l'analyse d'un système B correspond à une situation redoutée (FC) pour le système A qui fait l'objet d'une analyse comme toute FC.

Réaliser des analyses de sûreté de fonctionnement des systèmes à l'aide de modèles permet, en connectant deux modèles, de réaliser une analyse multi-systèmes. Cependant, connecter deux modèles détaillés expose au risque d'explosion combinatoire lié au grand nombre d'états possibles. Pour prévenir ce risque, nous proposons d'adapter la notion de DSF à l'approche FPM en réalisant un modèle abstrait de chaque système en interface. L'analyse multi-système peut alors être conduite sur un modèle composé d'un système détaillé connecté à des systèmes de type DSF. Afin d'assurer la cohérence entre le modèle détaillé et le modèle DSF d'un même système, nous proposons d'utiliser une technique de raffinement en vérifiant certaines relations entre les modèles.

Un modèle formel peut être enrichi de multiples façons. Compte tenu du formalisme AltaRica et des caractéristiques des modèles de type FPM, nous nous intéressons principalement à l'ajout ou la modification de défaillance, à la décomposition d'un nœud en une hiérarchie de sous-nœuds, à la modification des règles de reconfiguration d'un contrôleur.

Dans la suite de ce chapitre, nous illustrons comment utiliser les relations de raffinement définies dans le chapitre 4 pour répondre aux besoins industriels évoqués ci-dessus. Dans un premier temps, nous décrivons, sur l'exemple d'un calculateur embarqué, les nœuds AltaRica et les relations MecV requis pour exploiter les relations du chapitre 4 implémentée dans le langage MecV. Dans un second temps, toujours sur ce même exemple, nous décrivons les étapes successives à réaliser en vue d'établir qu'un nœud détaillé raffine un nœud abstrait. Dans un troisième temps, nous appliquons cette méthode de vérification du raffinement sur un modèle plus important représentant un système de génération et de distribution électrique. Après avoir dressé un bilan de nos expérimentations, nous situons la méthode que nous proposons par rapport à d'autres approches comparables de raffinement.

5.1 MODÉLISATION POUR LE RAFFINEMENT

Le raffinement que nous souhaitons appliquer s'appuie sur le model-checker MecV pour vérifier des relations entre deux modèles AltaRica. Les relations, présentées dans le chapitre précédent et écrites en langage MecV, nécessitent des informations sur les modèles qui ne peuvent pas être définies dans le langage AltaRica, comme par exemple, la notion d'événement non observable ou de flux équivalents. Ajouter ces informations au langage AltaRica reviendrait à proposer une extension du langage qui nécessiterait la mise à jour de la sémantique du langage et conduirait certainement à une modification importante des outils associés. Pour contourner cette difficulté, nous spécifions ces informations complémentaires du modèle AltaRica dans un fichier de spécification MecV.

Avant de nous intéresser à la vérification du raffinement pour la sûreté de fonctionnement des systèmes, nous présentons l'exemple d'un calculateur embarqué que nous utiliserons par la suite : plusieurs descriptions sont modélisées en langage AltaRica tandis que les données requises par les relations permettant de vérifier le raffinement sont spécifiées en MecV.

5.1.1 MODÉLISATION ALTARICA

5.1.1.1 Exemple d'un calculateur embarqué

Afin d'illustrer les idées développées dans la suite de ce chapitre, nous considérons un calculateur, baptisé `Cpu`, pour lequel nous proposons plusieurs descriptions à des niveaux de détail différents.

Chaque description aborde une forme différente d'enrichissement de modèle :
– l'ajout d'un événement et de la transition correspondante ;
– l'ajout d'une variable d'entrée ayant une influence sur une sortie ;
– la modification de la garde d'une transition en ajoutant une condition sur une entrée ;
– la décomposition d'un composant en sous-composants ;
– l'ajout de synchronisations.

Chaque description est accompagnée du code AltaRica correspondant ; les portions de code de couleur orange mettent en avant les ajouts et les modifications par rapport à la description précédente.

La représentation graphique sous forme d'automate (STI) du comportement de certains composants est présentée dans le paragraphe 5.2.2 afin d'illustrer les cas particuliers abordés.

a) Cpu0

Tout d'abord, une description très abstraite du calculateur se focalise sur les états accessibles qui correspondent à différents modes de fonctionnement. Le composant initialement dans un état de bon fonctionnement peut subir une défaillance conduisant à sa perte.

Le comportement du composant `Cpu0` est décrit par le code AltaRica suivant :

```
node Cpu0
  state
    Status:{ok,err,lost};
  flow
    Output:{ok,err,lost}:out;
  event
    loss;
  trans
    Status != lost |- loss -> Status := lost;
  assert
    Output = Status;
  init
    Status := ok;
edon
```

b) Cpu1

Une seconde description très abstraite du calculateur s'inspire de la description précédente en ajoutant une défaillance entraînant un comportement erroné.

Le comportement du composant `Cpu1` est décrit par le code AltaRica suivant :

```
node Cpu1
  state
    Status:{ok,err,lost};
  flow
    Output:{ok,err,lost}:out;
  event
    loss,error;
  trans
    Status != lost |- loss -> Status := lost;
    Status = ok |- error -> Status := err;
  assert
    Output = Status;
  init
    Status := ok;
edon
```

c) *Cpu2*

La première étape de raffinement du calculateur consiste à considérer qu'un calculateur est alimenté par une source électrique qui lui fournit la puissance nécessaire pour remplir son rôle. Cette alimentation est indispensable au bon fonctionnement du calculateur car en cas d'absence, ce dernier ne transmet pas de signal.

Ajouter ces informations au comportement du calculateur décrit précédemment revient à proposer un nouveau modèle que nous appelons `Cpu2` qui étend le modèle `Cpu1` en introduisant une nouvelle variable de flux `Power` représentant l'alimentation électrique. Nous modifions l'assertion pour décrire le fait qu'un calculateur non alimenté ne transmet pas de signal. Le code AltaRica correspondant est le suivant :

```
node Cpu2
  state
    Status:{ok,err,lost};
  flow
    Output:{ok,err,lost}:out;
    Power:bool:in;
  event
    loss,error;
  trans
    Status != lost |- loss -> Status := lost;
    Status = ok |- error -> Status := err;
  assert
    Output = case {Power: Status,
                   else lost};
  init
    Status := ok;
edon
```

d) *Cpu3*

La seconde étape de raffinement du calculateur consiste à considérer qu'un calculateur non alimenté par une source électrique ne subit pas de défaillance.

Ajouter cette précision au comportement du calculateur décrit précédemment revient à proposer un nouveau modèle que nous appelons `Cpu3` qui étend le modèle `Cpu2` en modifiant les gardes des événements `loss` et `error` pour décrire le fait qu'un calculateur non alimenté ne subit pas de défaillance. Le code AltaRica correspondant est le suivant :

```
node Cpu3
  state
    Status:{ok,err,lost};
  flow
    Output:{ok,err,lost}:out;
    Power:bool:in;
  event
    loss,error;
  trans
    Power and Status != lost |- loss -> Status := lost;
    Power and Status = ok |- error -> Status := err;
  assert
    Output = case {Power: Status,
                   else lost};
  init
    Status := ok;
edon
```

e) Cpu4

La troisième étape de raffinement consiste à détailler la structure interne d'un calculateur avionique en considérant l'architecture COMMAND/MONITORING classiquement utilisée par Airbus pour les calculateurs des commandes de vol.

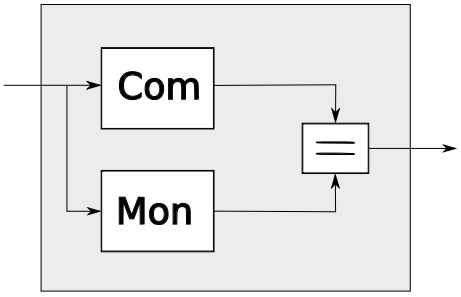


FIG. *5.1: Architecture Com/Mon*

Soit un calculateur, représenté par le nœud `Cpu4`, contenant deux sous-calculateurs similaires, aussi appelés "voies", `com` pour command et `mon` pour monitoring, de type `Cpu3`, reliés à un comparateur `comp` de type `Compator`. Le comparateur transmet l'ordre reçu du calculateur `com` s'il est égal à l'ordre reçu du calculateur `mon`. Un écart entre les deux ordres reçus déclenche un événement instantané (de type Dirac(0)) baptisé `detection`. La sortie du calculateur est alors définitivement perdue.

```
node Comparator
  state
    ErrorDetected:bool;
  flow
    Output:{ok,err,lost}:out;
    Order_Com:{ok,err,lost}:in;
    Order_Mon:{ok,err,lost}:in;
  event
    detection;
  trans
    not ErrorDetected
     and Order_Com!=Order_Mon |- detection -> ErrorDetected:=true;
  assert
    Output = case {not ErrorDetected : Order_Com,
                   else lost};
  init
    ErrorDetected := false;
  extern
    law <detection> = Dirac(0);
edon
```

Dans le composant `Cpu4`, afin de distinguer les défaillances des calculateurs de la détection d'erreur par le comparateur, il est nécessaire de définir des événements dans `Cpu4` et de les synchroniser avec des événements des sous-nœuds. Ne souhaitant pas imposer de contraintes sur ces événements de haut niveau, nous considérons qu'ils sont toujours possibles et sans effet. Le code AltaRica du nœud `Cpu4` est :

```
node Cpu4
  sub
    comp:Comparator ;
    com,mon:Cpu3;
  flow
```

```
    Output:{ok,err,lost}:out;
    Power:bool:in;
  event
    single_failure<detection;
  trans
    true |- single_failure,detection -> ;
  sync
    <single_failure,com.loss>;
    <single_failure,mon.loss>;
    <single_failure,com.error>;
    <single_failure,mon.error>;
    <detection,comp.detection>;
  assert
    Output = comp.Output;
    Power = com.Power;
    Power = mon.Power;
    com.Output = comp.Order_Com;
    mon.Output = comp.Order_Mon;
edon
```

f) Cpu5

La quatrième étape de raffinement consiste à enrichir le modèle précédent en considérant plusieurs défaillances simultanées. Compte tenu de l'architecture du calculateur, il est possible d'envisager deux doubles défaillances : la double perte `total_loss` et la double erreur `error`.

```
node Cpu5
  sub
    comp:Comparator ;
    com,mon:Cpu3;
  flow
    Output:{ok,err,lost}:out;
    Power:bool:in;
  event
    {single_failure,total_loss,error}<detection;
  trans
    true |- single_failure,total_loss,error,detection -> ;
  sync
    <single_failure,com.loss>;
    <single_failure,mon.loss>;
    <single_failure,com.error>;
    <single_failure,mon.error>;
    <double_loss,com.loss,mon.loss>;
    <error,com.error,mon.error>;
    <detection,comp.detection>;
  assert
    Output = comp.Output;
    Power = com.Power;
    Power = mon.Power;
    com.Output = comp.Order_Com;
    mon.Output = comp.Order_Mon;
edon
```

5.1.1.2 Modélisation de l'observateur associé au calculateur

Nous avons montré, dans le chapitre 3, qu'il est possible de représenter en AltaRica les situations redoutées identifiées durant l'analyse préliminaire à l'aide d'un composant appelé *observateur*. A chaque situation redoutée est associée une sortie booléenne de l'observateur.

Pour illustrer cette notion, nous souhaitons observer l'état de l'ordre calculé par le calculateur et plus particulièrement l'absence d'ordre ou la présence d'un ordre erroné. Nous considérons donc deux sorties `CpuLost` et `CpuErroneous`. Le composant AltaRica est le suivant :

```
node Observer
  flow
    OrderFromCpu:{ok,err,lost}:in;
    CpuLost,CpuErroneous:bool:out;
  assert
    CpuLost = (OrderFromCpu = lost);
    CpuErroneous = (OrderFromCpu = err);
edon
```

L'approche classique d'analyse d'un système consiste à vérifier que les séquences d'événements conduisant à une situation redoutée sont acceptables au regard de la criticité de la situation redoutée. Cette approche est partiellement automatisée à l'aide d'outils de génération de séquences. Pour exploiter ces outils et tirer profit de l'observateur modélisé, l'étude du calculateur nécessite la création d'un modèle, identifié `Main_CpuX` (X étant remplacé par le numéro de la version du calculateur étudié), qui permet d'instancier le calculateur et l'observateur puis de définir les liens entre instances.

Afin de conserver la visibilité des événements du calculateur, nous déclarons dans le nœud `Main_CpuX` un événement homonyme à chaque événement du nœud `CpuX`, ces deux événements étant ensuite synchronisés.

Pour étudier le nœud `Cpu2`, le modèle `Main_Cpu2` est décrit comme suit :

```
node Main_Cpu2
  sub
    cpu:Cpu2;
    obs:Observer;
  event
    loss,error;
  trans
    true |- loss,error -> ;
  sync
    <loss,cpu.loss>;
    <error,cpu.error>;
  assert
    cpu.Output = obs.OrderFromCpu;
edon
```

Remarque

Il serait possible d'observer directement la valeur de la variable `Output` du composant `cpu`. Cependant l'observateur permet de formaliser les sorties redoutées qui font l'objet de l'analyse.

5.1.2 SPÉCIFICATION MECV

5.1.2.1 Initialisation des relations MecV requises pour la vérification du raffinement

Comme précisé dans le paragraphe 4.3.2, nous avons souhaité écrire des relations génériques permettant de vérifier les relations de simulation interfacée et de simulation quasi-branchante interfacée. Pour y parvenir, nous considérons les éléments suivants (à définir au préalable en fonction des nœuds à comparer) :

– `RelF(s,s')` : relation entre les variables de flux des 2 nœuds,

- `RelEvt(e,e')` : relation entre les événements des 2 nœuds,
- `transA(s,e,t)` (resp. `transA'(s',e',t')`) : transitions de `s` à `t` étiquetées par `e` du nœud `A` (resp. les transitions de `s'` à `t'` étiquetées par `e'` du nœud `A'`),
- `initA(s)` (resp. `initA'(s)`) : configurations initiales du nœud `A` (resp. du nœud `A'`).

Ces éléments sont définis ou calculés en fonction des nœuds sur lesquels porte l'étude. Il est donc nécessaire de les définir dans un fichier que nous nommerons `init.MecV` dans la suite de ce chapitre.

L'outil MecV ne tient pas compte des configurations initiales pour calculer l'espace d'état d'un nœud. Pour restreindre ce dernier aux seuls états réellement possibles, il est donc nécessaire de calculer les états accessibles à l'aide de la relation `ReachA` (resp. `ReachA'`), présentée dans l'exemple 2.1, pour le nœud abstrait `A` (resp. pour le nœud détaillé `A'`). Les relations `transA` et `transA'`, qui correspondent aux STI à comparer, sont ensuite définies en fonction des états accessibles. Soit `ReachA` les états accessibles d'un nœud `Cpu`, les transitions possibles de ce nœud sont obtenues par la relation MecV suivante :

```
transA(s,e,t) := (Cpu!t(s,e,t) & ReachA(s));
```

5.1.2.2 Choix des flux équivalents

La relation `RelF` permet concrètement de spécifier les variables sur lesquelles effectuer la comparaison entre un état du nœud abstrait et un état du nœud détaillé.

L'état d'un nœud est défini soit par ses propres variables d'état, soit par les variables d'état de ses sous-nœuds. Il apparaît donc plus naturel et plus pratique de baser la comparaison sur les variables de sortie et éventuellement sur les variables d'entrée.

Le raffinement, lorsqu'il est vérifié, assure la cohérence entre deux nœuds en garantissant que les exigences satisfaites par le nœud abstrait sont préservées par le nœud détaillé. Cette préservation peut être :

- globale, auquel cas la relation `RelF` concerne toutes les variables de flux symbolisant les flux physiques, en vue de préserver la compositionnalité (i.e. pouvoir indifféremment connecter le nœud abstrait ou le nœud détaillé avec d'autres nœuds compatibles) ;
- restreinte, auquel cas la relation `RelF` se focalise sur une variable de flux sortant d'un observateur, en vue de préserver les exigences liées à la FC modélisée par cette variable.

Pour garantir la préservation d'exigences liées à plusieurs FC, nous proposons d'ajouter à l'observateur une sortie booléenne traduisant la conjonction des FC.

Pour établir l'existence d'une relation de raffinement entre deux descriptions d'un calculateur, nous comparons les états en vérifiant que l'ordre en sortie du calculateur est le même sur le modèle abstrait et sur le modèle détaillé. Ainsi, en considérant le nœud `Cpu3` comme le nœud abstrait et `Cpu4` comme le nœud détaillé, la relation `RelF` s'écrit :

```
RelF(s : Cpu3!c, s' : Cpu4!c) := (s.Output = s'.Output);
```

Remarque

Nous montrerons par la suite qu'il est nécessaire d'avoir des variables de flux similaires pour pouvoir établir une relation entre deux nœuds.

5.1.2.3 Choix des événements équivalents

Les relations définies au chapitre 4 nécessitent la spécification des événements en relation entre le modèle abstrait et le modèle détaillé, ainsi que les événements non observables. Ces opérations s'avèrent rapidement contraignantes lorsque la taille du modèle augmente. C'est pourquoi nous proposons deux approches (une par relation) permettant d'effectuer une vérification préliminaire intuitive en se focalisant sur les changements de variables de flux plutôt que sur les étiquettes des transitions.

a) Relation de simulation : ignorer les étiquettes des transitions

La relation de simulation interfacée paramétrée définie précédemment nécessite la relation `RelEvt` pouvant être interprétée comme une équivalence d'étiquette de transition (cf. chapitre 4) entre le modèle abstrait et le modèle détaillé. Or spécifier les événements en relation est une opération manuelle lourde à réaliser. De plus, une défaillance du modèle détaillé peut, par exemple, être en relation avec plusieurs défaillances du modèle abstrait. Ainsi, le nombre de relations entre événements abstraits et événements détaillés croit rapidement lorsque le nombre de défaillances augmente dans un des modèles. La spécification des événements en relation s'avère alors encore plus coûteuse.

Nous reprenons l'exemple du calculateur pour illustrer la difficulté de définir la relation `RelEvt`. Le calculateur `Cpu4` décompose le calculateur `Cpu3`. Du fait de la détection d'écart réalisée par le comparateur, toute défaillance de la voie `com` ou de la voie `mon` équivaut à une perte du calculateur, autrement dit l'événement `loss` de `Cpu3` est à mettre en relation avec les événements `loss` et `error` présents dans chaque voie de `Cpu4`, soit 4 événements au total.

Pour faciliter la vérification de l'existence d'une relation de simulation interfacée entre deux modèles, nous proposons d'effectuer dans un premier temps, la vérification sans tenir compte des étiquettes, en ne s'appuyant que sur la relation `RelF`. Soit `Cpu3` un nœud abstrait et `Cpu4` un nœud détaillé à comparer avec `Cpu3`, cette hypothèse équivaut à considérer que tous les événements sont en relation. Cela s'exprime en langage MecV par la relation suivante :

```
RelEvt(e : Cpu3!ev, e' : Cpu4!ev) := true;
```

La relation `RelEvt` définie ci-dessus sert à vérifier s'il existe une relation de simulation paramétrée du modèle abstrait vers le modèle détaillé. La vérification peut établir que les deux modèles ne sont pas en relation, auquel cas l'utilisateur aura économisé le temps de spécification de la relation `RelEvt`. Si la vérification établit qu'une relation de simulation existe, il est alors possible :

- de procéder à une nouvelle vérification en appliquant une démarche rigoureuse de spécification des événement en relation ;
- de calculer à l'aide de l'outil MecV les couples d'événements tels qu'un événement du modèle abstrait est en relation avec un événement du modèle détaillé puis de vérifier que ces couples sont corrects.

Des événements sont en relation s'ils étiquettent deux transitions dont les états de départ et d'arrivée satisfont une relation de simulation interfacée paramétrée. La relation MecV correspondante est :

```
sim_evt(e,e') := <s><s'><t><t'>(transA(s,e,t) & transA'(s',e',t')
                                & sim(s,s') & sim(t,t'));
```

En considérant les relations `RelF` et `RelEvt` décrites précédemment, le calculateur `Cpu3` simule `Cpu4`. La relation `sim_evt` indique alors que des défaillances de `Cpu4` sont en relation avec des défaillances de `Cpu3` mais également avec l'événement ϵ de `Cpu3`. De même, l'événement ϵ de `Cpu4` est en relation avec les défaillances de `Cpu3`. Ces résultats, surprenant au premier abord, se justifient par les deux types de transition que peut étiqueter l'événement ϵ :

- un changement de variable d'entrée ;
- une boucle d'un état sur lui-même ;

Une variable d'entrée est dite *libre* si sa valeur n'est pas contrainte par une assertion en AltaRica LaBRI ou si elle n'est pas reliée à un composant en AltaRica OCAS. Contrairement aux outils industriels, MecV est capable de traiter des nœuds disposant de telles variables. Une variable libre d'un nœud peut changer de valeur :

- indépendamment de tout événement du nœud, auquel cas la transition est étiquetée ϵ ;
- simultanément avec un événement du nœud, auquel cas la transition prend l'étiquette de l'événement.

La variable `Power` est libre dans notre comparaison. Elle peut ainsi prendre pour valeur `false` en étant étiquetée par ϵ dans `Cpu3` et par une défaillance dans `Cpu4` et inversement.

D'autre part, ϵ étiquette la transition sans changement d'état définie dans la sémantique du langage. Dans `Cpu3`, si cette transition ne s'accompagne pas d'un changement de valeur d'une variable d'entrée, les flux sortant restent inchangés. Du fait qu'une défaillance d'une des voies de `Cpu4` suffit pour que le calculateur ne puisse plus transmettre d'ordre (i.e. `Output = lost`) les défaillances suivantes sont sans effet et donc équivalente à une transition ϵ de `Cpu3`.

b) Relation de QBS : Événements observables ou transitions observables ?

La relation de simulation quasi-branchante, présentée dans 4.4.8.2, s'appuie sur la notion d'observabilité des événements.

En toute rigueur, il est nécessaire de différencier les événements observables des événements non observables. Cependant, en pratique, il s'avère que cette distinction n'est pas toujours évidente.

Le fichier `init.mecV`, propre à chaque comparaison de nœuds, permet de spécifier les événements non observables. Deux particularités importantes d'AltaRica concernent l'observabilité des événements :

- tout événement qui n'apparaît pas dans une synchronisation du nœud de plus haut niveau est automatiquement synchronisé avec l'événement ϵ (et donc considéré comme non observable) ;
- la visibilité d'un événement est non contextuelle : un événement non spécifié comme non observable est considéré comme observable, quelque soit la configuration.

Considérons le calculateur `Cpu3`. La détection d'une erreur ayant un effet irréversible (le signal en sortie du calculateur est définitivement perdu), toute défaillance qui se produit après la détection n'aura aucun effet et donc ne sera pas visible.

La visibilité d'un événement AltaRica étant permanente, il est nécessaire de modifier le code du composant `Cpu3` afin de différencier les événements observables des événements non observables. Une solution d'implémentation consiste à :

1. déclarer une variable d'état booléenne `obs` initialisée à `vrai`, nécessaire pour différencier les 2 situations possibles ;
2. remplacer l'événement `single_failure` par deux événements `single_failure_Obs` et `single_failure_notObs` ;
3. définir les vecteurs de synchronisation de `single_failure_Obs` et `single_failure_notObs`, identiques aux vecteurs de synchronisation de `single_failure` ;
4. remplacer la transition précédemment écrite par 3 transitions distinctes, une par événement déclaré de `Cpu3` :

```
     obs |- single_failure_Obs     -> ;
     obs |- detection              -> obs := false;
not  obs |- single_failure_notObs  -> ;
```

Le changement de visibilité de cet exemple est marqué par un événement instantané. Or il n'est pas possible de définir un tel événement dans tous les systèmes. Modéliser le changement de visibilité des défaillances nécessite des modifications du modèle et introduit donc du jugement d'ingénieur, ce qui rend cette technique inutilisable en pratique.

Afin de proposer une signification intuitive de la notion d'observabilité pour des modèles de sûreté de fonctionnement des systèmes, nous considérons la définition suivante : une défaillance est observable si les conséquences de son occurrence sont visibles au niveau le plus haut considéré. Autrement dit, lorsqu'un équipement est constitué de plusieurs composants, l'observabilité des défaillances des composants dépend des effets visibles en sortie de l'équipement.

Pour faciliter la vérification, nous proposons une interprétation des notions de *transition observable* et de *transition non observable* : une transition est observable si un changement d'état est visible (généralement au niveau des flux sortants) lorsqu'elle se produit. Ces deux types de transitions ne sont pas spécifiés par l'utilisateur, mais calculés par comparaison entre l'état de départ et l'état final d'une transition. Soient `s` et `t` deux états d'un nœud AltaRica `nA`, l'utilisateur définit à l'aide de la relation `eqA` les variables de flux à comparer entre les états `s` et `t`. Les transitions non observables, baptisées `TauTrans_A`, et observables, baptisées `ObsTrans_A`, du nœud abstrait sont calculées par les formules MecV suivantes :

```
TauTrans_A(s,e,t) := (transA(s,e,t) & eqA(s,t));
ObsTrans_A(s,e,t) := (transA(s,e,t) & ~eqA(s,t));
```

En considérant des événements non observables, recensés par la relation `isTau`, les deux types de transitions sont alors calculés par les formules MecV suivantes :

```
TauTrans_A(s,e,t) := (transA(s,e,t) & isTau_A(e));
ObsTrans_A(s,e,t) := (transA(s,e,t) & ~isTau_A(e));
```

5.2 SPÉCIALISATION DU RAFFINEMENT POUR LA SÛRETÉ DE FONCTIONNEMENT DES SYSTÈMES

5.2.1 MASQUAGE DES ÉVÉNEMENTS INSTANTANÉS

Comme nous l'avons précisé dans le paragraphe 4.4.2, nous souhaitons manipuler des Système de Transitions Interfacé (STI) composés uniquement de transitions étiquetées par des défaillances. Il est donc nécessaire de procéder à un traitement du STI initial. Par souci de concision, nous baptiserons :

– *transition défaillante* une transition étiquetée par une défaillance ;
– *transition instantanée* une transition étiquetée par un événement instantané.

Toute transition défaillante suivie de transitions instantanées est remplacée par une transition étiquetée par cette même défaillance et ayant la même origine mais dont l'état final correspond à l'état final de la dernière transition instantanée possible (une reconfiguration pouvant provoquer une autre reconfiguration, il est possible que des transitions instantanées se succèdent).

Nous qualifions d'*état observable* un état duquel seules des transitions défaillantes sont possible. Le STI que nous souhaitons manipuler n'est composé que d'états observables. La première étape consiste à différencier, sur le STI initial, les états observables des états non observables.

D'un point de vue pratique pour l'utilisateur, les événements instantanés sont recensés lors de la création du modèle et stockés dans la relation `internalEvt_A'` (pour le nœud détaillé). Un état est observable s'il est accessible et si toute transition depuis cet état n'est pas étiqueté par un événement instantané. Cela se traduit en MecV par les relations suivantes :

```
NoInternalEvt_A'(s) := [e][t](transA'(s,e,t) => ~internalEvt_A'(e));

ObsState_A'(s) := (ReachA'(s) & NoInternalEvt_A'(s));

NonObsState_A'(s) := (ReachA'(s) & ~NoInternalEvt_A'(s));
```

Remarque

La relation `NoInternalEvt_A'` telle que nous la définissons ne permet pas de considérer l'événement ϵ dans l'ensemble d'événements `internalEvt_A'`. En effet, de par la sémantique d'AltaRica, cet événement est possible dans tout état de tout nœud. Le considérer dans `internalEvt_A'` revient à considérer tous les états dans `NonObsState_A'`. Lorsqu'il n'existe aucun événement instantané à déclarer, la relation `internalEvt_A'` a donc pour valeur `false`.

Une fois les états observables identifiés, il est nécessaire de calculer, depuis chaque état observable, quels sont les états observables accessibles en une transition défaillante suivie de transitions instantanées. Le nombre de transitions instantanées consécutives pouvant varier, le calcul des états observables `t` accessibles depuis l'état observable `s` correspond à une fonction récursive exprimée en langage MecV par la recherche du plus petit point fixe suivant :

```
nextObs_A'(s,t) += NonObsState_A'(s) & ObsState_A'(t)
                   & <e>(internalEvt_A'(e)
                         & (transA'(s,e,t)
                            | <u>(NonObsState_A'(u)
                                  & transA'(s,e,u)
                                  & nextObs_A'(u,t))));
```

L'étape suivante consiste à définir les transitions du nouveau STI, reliant l'état `s` à l'état `t` et étiquetées par l'événement `e`, `s` et `t` étant des états observables et `e` n'étant pas un événement instantané. On distingue deux cas :

– soit s et t était initialement reliés par une transition étiquetée par la défaillance e ;
– soit il existe une transition étiquetée par la défaillance e depuis s dont l'état final u est non observable mais duquel il existe une suite de transition instantanée conduisant à l'état observable t.

Ce calcul est implémenté par la formule MecV suivante :

```
transObs_A'(s,e,t) := ObsState_A'(s) & ObsState_A'(t)
                      & ~internalEvt_A'(e)
                      & (transA'(s,e,t)
                        | <u>(NonObsState_A'(u) & transA'(s,e,u)
                             & nextObs_A'(u,t)));
```

Dans les relations de simulation interfacée paramétrée et de simulation quasi-branchante interfacée paramétrée, les transitions du modèle détaillé sont désignées par transA'. Pour ne pas avoir à proposer plusieurs implémentations de ces formules suivant que les modèles contiennent ou non des événements instantanés, nous terminons ces opérations de masquage par le renommage de transObs_A' en transA' et de ObsState_A' en ReachA'.

```
transA'(s,e,t) := transObs_A'(s,e,t);
ReachA'(s) := ObsState_A'(s);
```

Afin de limiter l'utilisation de la mémoire, il est possible d'ajouter les commandes suivantes qui supprime les constantes calculées :

```
:remove-constant NoInternalEvt_A'
:remove-constant ObsState_A'
:remove-constant NonObsState_A'
:remove-constant nextObs_A'
:remove-constant transObs_A'
```

5.2.2 COMPTEURS DE DÉFAILLANCES

5.2.2.1 Motivations

Les analyses de sûreté de fonctionnement des systèmes s'intéressent aux coupes minimales significatives dont la probabilité est la plus importante et la taille raisonnable. Des coupes minimales de taille supérieure à 5 défaillances peuvent exister mais elles sont peu lisibles et difficiles à comprendre. Augmenter la taille maximum des coupes minimales considérées accroît d'autant plus le nombre de coupes minimales à vérifier.

Partant de ce constat sur les pratiques industrielles, il apparaît naturel de ne chercher à considérer, dans une approche FPM, qu'un sous-ensemble des séquences de défaillances possibles.

Borner la taille maximum des traces prises en compte permet de prévenir les contraintes suivantes :

– lisibilité : une augmentation de la taille des séquences prises en compte s'accompagne généralement d'une augmentation du nombre de séquences à vérifier ;
– validation : plus les séquences sont grandes, plus les mécanismes/reconfigurations à considérer pour validation sont nombreux ;
– explosion combinatoire : bien que formalisé de façon modulaire et compacte de par la syntaxe du langage AltaRica, un modèle contenant un grand nombre de composants et de défaillances peut avoir pour sémantique un STI de taille importante ; MecV traitant des STI, un nombre d'états accessibles trop important peut rendre tout calcul impossible.

D'un point de vue sûreté de fonctionnement des systèmes, nous considérons que le raffinement doit garantir l'exigence suivante : un état défaillant n'est pas accessible en un nombre de défaillances inférieur dans le modèle détaillé que dans le modèle abstrait. Il est donc nécessaire de pouvoir associer un compteur de défaillance à chaque état accessible.

Ces arguments nous incitent à enrichir chaque état accessible du nombre de défaillances survenues depuis l'état initial défini.

Un compteur de défaillance peut être défini dans un modèle AltaRica ou implémenté par une relation MecV.

La façon la plus rapide de définir un compteur de défaillance dans un nœud AltaRica est :

- d'ajouter, à chaque sous-nœud, une variable d'état, initialisée à 0, et une variable de flux sortant ;
- d'incrémenter la variable d'état dans chaque transition étiquetée par une défaillance ;
- de définir la variable de flux comme égale à la variable d'état ;
- d'ajouter une variable de flux sortant dans le nœud "père" pour additionner tous les compteurs des sous-nœuds.

Or MecV calcule l'espace d'état d'un nœud AltaRica en fonction de ses variables et de leurs domaines respectifs, sans le restreindre aux seuls états accessibles depuis l'état initial. Compte tenu du nombre de variables à ajouter dans un modèle AltaRica, implémenter un compteur de défaillance alourdit le code, accroît considérablement l'espace d'états et réduit donc les performances de calcul.

MecV considère les changements d'état au niveau du nœud de plus haut niveau. Il apparaît dès lors rapide d'associer à chaque état le nombre de transitions franchies depuis l'état initial.

Pour ces raisons, nous privilégions la définition d'une relation MecV intégrant le comptage du nombre de défaillances.

Nous illustrons les différents cas particuliers pris en compte dans l'élaboration de cette relation à l'aide d'automates (STI) représentant la sémantique des calculateurs présentés précédemment (cf. 5.1.1.1). La couleur d'un état correspond à la valeur en sortie du calculateur : vert pour une donnée fournie correcte, violet pour une donnée fournie erronée, rouge en cas d'absence de donnée fournie.

5.2.2.2 Implémentation

La relation `ReachA(s)`, définie ci-dessous, calcule les états `s` accessibles du nœud `Cpu` :

```
ReachA(s) += initA(s) | (<u>(ReachA(u) & <e>(Cpu!t(u,e,s))));
```

En nous inspirant de la relation `ReachA(s)`, nous souhaitons définir une relation calculant les états accessibles `s` et le nombre de défaillances `n` qui se sont produites depuis l'état initial. MecV ne traite pas les entiers, il est donc nécessaire de définir un intervalle des valeurs possibles du compteur.

En pratique, la borne supérieure de l'intervalle correspond au nombre maximum de défaillances prises en compte. Choisir une valeur importante comme borne supérieure de l'intervalle assure de prendre en compte la totalité de l'automate d'accessibilité alors qu'une valeur faible permet de restreindre le périmètre d'étude.

La relation `Reach_A(s,n :[0,maxA])` (`maxA` étant remplacé par un entier) suivante initialise le compteur à 0 dans tout état initial et l'incrémente d'une unité à chaque transition autre qu'une boucle d'un état sur lui-même :

```
Reach_A(s,n:[0,maxA]) += (initA(s) & n=0)
                        |(<u><e>(u!=s & Reach_A(u,n-1)
                                 & (Cpu!t(u,e,s))));
```

Pour différencier les états suivant la valeur du compteur de défaillances, il est nécessaire de tenir compte de ce dernier dans le calcul des transitions possibles. La relation `trans_A` suivante effectue ce calcul :

```
trans_A(s,n:[0,maxA],e,t,m:[0,maxA]) := (Cpu!t(s,e,t)
                                         & Reach_A(s,n)
                                         & Reach_A(t,m)
                                         & ((s=t & n=m)
                                            |(s!=t & n+1=m)));
```

Afin de conserver des relations de simulation et de simulation quasi-branchante lisibles, nous procédons au renommage des transitions `trans_A`, privées des valeurs de compteur, en `transA`. La formule ainsi obtenue est :

```
transA(s,e,t):= <n:[0,maxA]><m:[0,maxA]>trans_A(s,n,e,t,m);
```

Remarque

De façon analogue, il est possible d'affecter à `ReachA` les états accessibles de `Reach_A` sans conserver les valeurs de compteur. En désignant par `y` l'entier défini par l'utilisateur comme borne supérieure de `n`, la définition de la relation `ReachA` devient la suivante :

```
ReachA(s) := <n:[0,y]> Reach_A(s,n);
```

Du fait que `y` est défini localement dans la relation, il est possible de choisir une valeur de `y` différente de `maxA` :

- si `y < maxA`, alors les états accessibles considérés sont restreints aux états dont le nombre de défaillances maximum est `y` ;
- si `y` $\geq$ `maxA`, alors le nombre maximum d'événements considérés reste égal à la variable `maxA`.

5.2.2.3 Observations

La formule initiale `ReachA` appliquée au composant abstrait `Cpu1` indique que 3 états sont accessibles, similaires aux états de la figure 5.2. La sortie du composant `Cpu1` est égale à son état.

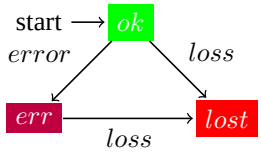


FIG. *5.2: STI du composant abstrait `Cpu1`*

En revanche, le calcul de la formule `Reach_A` par MecV retourne 4 états :

- (Status = ok, 0)
- (Status = err, 1)
- (Status = lost, 1)
- (Status = lost, 2)

L'observation du graphe correspondant permet de mieux comprendre ce résultat :

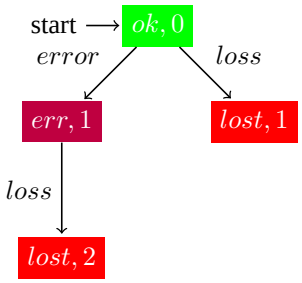


FIG. *5.3: STI avec compteur de défaillances du composant abstrait `Cpu1`*

En effet, la transition `loss` étant possible depuis 2 états différents n'ayant pas la même valeur de compteur, il existe 2 états étiquetés `lost` ayant 2 compteurs différents.

5.2.2.4 Cas particulier : le comportement du composant dépend d'au moins une entrée

Considérons un composant calculateur qui dispose d'une entrée symbolisant l'alimentation électrique, cf. `Cpu3` présenté dans le paragraphe 5.1.1.1.

D'un point de vue AltaRica, le changement de valeur de l'entrée du composant est interprété comme un événement ϵ : une évolution de l'état du composant résultant d'un événement extérieur à celui-ci.

L'automate représenté sur la figure 5.4 illustre ce comportement. La partie haute de chaque état de la figure représente la valeur de la variable d'état `Status` et la présence ou non d'alimentation tandis que la partie basse représente l'état de la sortie (identique à la couleur de l'état).

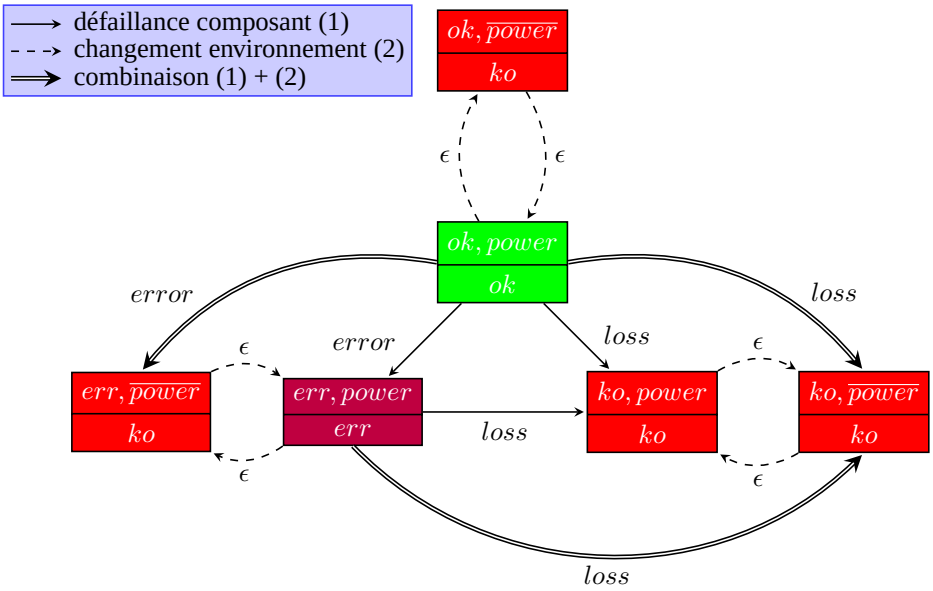


FIG. 5.4: *STI du composant `Cpu3`*

Cette alimentation est totalement indépendante du calculateur et peut donc défaillir à tout moment (entre 2 défaillances ou simultanément avec une défaillance).

Il s'avère que la définition initiale de `Reach_A` comptabilise les transitions, autrement dit les événements epsilon incrémentent le compteur de la même manière qu'une défaillance.

Pour parvenir à définir un compteur de défaillances, il faut donc enrichir la définition précédente pour ne pas incrémenter le compteur lorsqu'une transition symbolise un changement de l'environnement. La relation devient :

```
Reach_A(s,n:[0,maxA]) += (initA(s) & n=0)
                       |(<u><e>(u!=s & Reach_A(u,n-1)
                                & (e.!="" & Cpu!t(u,e,s))));
```

Les transitions possibles sont alors calculées par la relation `trans_A` suivante :

```
trans_A(s,n:[0,max],e,t,m:[0,max]) := (Cpu!t(s,e,t)
                                       & Reach_A(s,n)
                                       & Reach_A(t,m)
                                       & ((e.="" & n=m)
                                          |(e.!="" & n+1=m)));
```

La figure 5.5 représente l'automate du composant `Cpu3` en tenant compte du compteur de défaillance. La légende adoptée pour représenter chaque état reprend la légende de la figure 5.4 en complétant la partie haute du nombre de défaillances survenues depuis l'état initial supposé nominal.

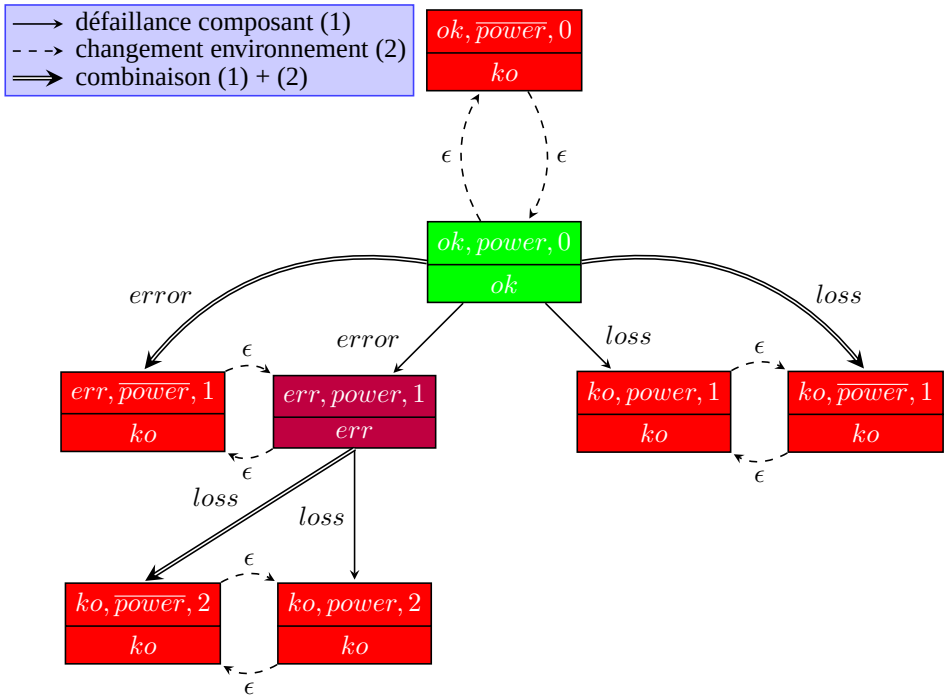


FIG. 5.5: *STI avec compteur de défaillances du composant* `Cpu3`

5.2.2.5 Cas particulier : les événements instantanés

Les systèmes étudiés en sûreté de fonctionnement des systèmes contiennent généralement des mécanismes tels que des reconfigurations ou des détections d'erreur. Leur rôle, au contraire des défaillances, est de ramener le système dans un état sûr suite à une défaillance. Ces mécanismes sont modélisés par des événements instantanés.

Pour que le compteur associé à chaque état indique le nombre de défaillances depuis l'état initial, il est nécessaire de ne pas incrémenter le compteur lorsque la transition franchie pour atteindre un état est une transition instantanée.

Pour différencier les deux types d'événements, il faut définir la relation `internalEvt` contenant tous les événements instantanés dans le fichier mecV, puis adapter la relation `Reach_A`.

```
internalEvt_A(e : Comp!ev) := (e.="Detection");
```

La relation devient :

```
Reach_A(s,n:[0,maxA]) += (initA(s) & n=0)
                        |(<u><e>(u!=s
                                & ((Reach_A(u,n-1)
                                   & e.!=""
                                   & ~internalEvt_A(e))
                                  |(Reach_A(u,n)
                                   & (e.="" | internalEvt_A(e))))
                                & Cpu!t(u,e,s)));
```

Les transitions possibles sont alors calculée par la relation `trans_A` suivante :

```
trans_A(s,n:[0,max],e,t,m:[0,max]) := (Cpu!t(s,e,t)
                                      & Reach_A(s,n)
                                      & Reach_A(t,m)
                                      & ((e.!="" & ~internalEvt_A(e)
```

```
                                    & n+1=m)
                                |((e.="" | internalEvt_A(e))
                                  & n=m)));
```

Remarque

Après analyse des couples `(état, compteur)` obtenus par chacune des relations définies sur un modèle, il ne semble pas nécessaire de préciser que le compteur d'un état atteint par une transition ϵ est identique au compteur de l'état origine de cette transition. En effet, comme l'illustrent les automates précédents, trois types de transitions peuvent être distingués :

- les transitions définies dans le code AltaRica (1) ;
- les transitions ϵ liées aux changements de valeur des variables d'entrée (2) ;
- les transitions représentant la synchronisation d'une transition définie dans le code AltaRica et d'une transition ϵ (3).

Tout état atteignable par une transition ϵ est soit un état initial, soit un état atteint depuis un état non nominal par une transition de type (1) ou (3).

D'autre part, il serait tentant de penser que le dernier cas particulier considéré n'a pas lieu d'être du fait que nous avons défini des relations pour masquer les événements instantanés. Or les relations de masquage nécessitent de connaître les états accessibles. Borner les états accessibles après le masquage revient à :

1. calculer tous les états accessibles sans borner le nombre d'événements considérés ;
2. effectuer le masquage sur ce même ensemble d'états ;
3. réduire cet ensemble d'états au sous-ensemble souhaité par l'utilisateur à l'aide de la borne `max`.

Borner les états accessibles étant la priorité afin de prévenir les problèmes d'explosion combinatoire, il est donc nécessaire de procéder au masquage seulement dans un second temps.

5.2.2.6 Exigences avec comptage des défaillances

Dans le but de valider une architecture, nous avons montré qu'il est possible de modéliser des situations redoutées (FC) dans un modèle à l'aide d'un observateur, baptisé `obs` par la suite. Or une exigence qualitative, identifiée par `MaxFC` dans la suite de ce chapitre, est associée à chaque situation redoutée en fonction de sa criticité, c.f. tableau 1.2. L'outil MecV permet de vérifier automatiquement si cette exigence est satisfaite. Vérifier cette exigence pour une FC équivaut à vérifier s'il n'existe pas un état où :

- la situation redoutée est atteinte ;
- le nombre de défaillances survenues depuis l'état initial est inférieur à `MaxFC`.

La traduction en langage MecV de cette exigence est la relation suivante :

```
FC_Cpt(x) := x = ~(<s><n:[0,x]>(Reach_A(s,n) & s.obs.FC & n<MaxFC));
```

Si la relation, et donc l'exigence, n'est pas satisfaite, il est alors possible d'extraire les états "contre-exemples" à l'aide de la relation suivante :

```
FC_CounterEx(s,n:[0,maxA']) := (Reach_A'(s,n) & s.obs.FC
                                & ~(n<maxFC));
```

Cette approche s'inspire de la démarche expérimentée dans le cadre du projet européen ESACS [BCKS04]. Le model-checker SMV permet de vérifier, sur un modèle AltaRica traduit en SMV, des exigences similaires à celles que nous avons mentionnées mais exprimées en logique LTL, et non en μ-calcul comme le nécessite MecV.

Comme nous l'avons mentionné dans le paragraphe 5.2.2.1, nous considérons qu'un état défaillant ne doit pas être accessible sur un modèle raffiné en moins de défaillances que sur le modèle abstrait.

Cette exigence se traduit par une contrainte supplémentaire dans le calcul des états en relation et nécessite de comparer les transitions prenant en compte le compteur de défaillances associé à chaque état. Pour différencier cette nouvelle relation de la relation de simulation sans compteur, baptisons la `sim_cpt`. Sa formule MecV est la suivante :

```
sim_cpt(s,n:[0,max],s',n':[0,max]) -=
        RelF(s,s') & ~(n>n')
        & ([e'][t'][m':[0,max]](trans_A'(s',n',e',t',m') =>
            <e><t><m:[0,max]>(RelEvt(e,e')
                              & trans_A(s,n,e,t,m)
                              & sim(t,m,t',m'))));
```

La formule permettant de vérifier que tout état initial du nœud détaillé est en relation avec un état initial du nœud abstrait devient :

```
isSim(x) :=  x = ([s'] (initA'(s') =>
                        <s> (initA(s) & sim(s,0,s',0))));
```

Remarque

La relation `sim_cpt` s'appuie sur les relations `RelF`, `RelEvt`, `trans_A` et `trans_A'`. La prise en compte de la contrainte sur le nombre de défaillances n'a pas de conséquence sur ce que nous avons présenté jusqu'à présent sur ces relations. Cependant, l'opération de masquage des événements instantanés, que nous avons définie précédemment, ne traite que des transitions sans compteur. Il est donc nécessaire de la mettre à jour. Les modifications étant mineures, nous faisons le choix d'inclure le détail des relations impactées en annexe (c.f. C).

5.2.3 MÉTHODOLOGIE DE VÉRIFICATION DU RAFFINEMENT

Jusqu'ici,nous avons présenté un certain nombre de relations et d'approches pratiques permettant de vérifier le raffinement. Dans la suite de cette section, nous décomposons la méthodologie de vérification du raffinement en détaillant et justifiant les opérations successives.

Avant d'utiliser les relations prédéfinies, l'utilisateur doit :
- sélectionner un nœud abstrait à comparer avec un nœud détaillé ;
- définir les relations d'initialisation MecV et plus particulièrement :
  - la relation `RelF` en choisissant les variables de flux à observer, en fonction de ses objectifs de préservation,
  - la relation `RelEvt` en spécifiant les événements à considérer comme équivalents pour une approche "vérification" de la relation de simulation, ou en affectant la valeur `true` pour une approche "exploration",
- éventuellement définir suivant les nœuds et les choix d'analyse :
  - le nombre maximum de défaillances à considérer,
  - les événements instantanés, en vue de leur masquage,
  - les événements non observables, pour une vérification rigoureuse de la simulation quasi-branchante,
  - les variables de flux permettant de déterminer un changement d'état observable, pour une vérification préliminaire pratique de la simulation quasi-branchante.

Une fois ces données chargées dans l'outil MecV, l'utilisateur peut vérifier le raffinement par simple appel de relations prédéfinies.

Étant donné que notre approche cherche avant tout à évaluer la sûreté de fonctionnement des systèmes, nous ne considérons que les défaillances et leurs effets. Si des événements instantanés, tels que détection d'erreur ou reconfiguration, sont définis dans un des deux nœuds à traiter, il faut tout d'abord procéder à leur masquage.

Avant de chercher à établir une relation de raffinement, nous proposons de vérifier que, par rapport à la relation sur les flux `RelF`, tout état accessible du nœud détaillé ($s \in FM_{det}$) est en relation avec un état accessible du nœud abstrait ($s \in FM_{abs}$). Cette analyse préliminaire permet d'orienter le choix des relations à tester : si les états accessibles du nœud détaillé sont en relation avec des états du nœud abstrait, alors ce dernier peut simuler ou qb-simuler le nœud détaillé ; dans le cas contraire, aucune des deux relations ne pourra être établie.

Cette vérification s'écrit en langage MecV :

```
AllDetInAbs(x):=
        x = [s'](ReachA'(s') => <s>(ReachA(s) & RelF(s,s')));
```

La relation suivante fournit des contre-exemples, si la relation précédente n'est pas satisfaite :

```
notInDet(s):= (ReachA(s) => ~(<s'>(ReachA'(s') & RelF(s,s'))));
```

Si le résultat du test est positif, la première relation à tester est la relation de simulation.

Une approche "vérification" s'appuie sur une relation `RelEvt` grâce à laquelle l'utilisateur a spécifié les événements du nœud détaillé équivalents à chaque événement du nœud abstrait. Le résultat fourni par MecV permet d'établir directement si le nœud détaillé raffine le nœud abstrait.

Une approche "exploration" s'appuie sur une relation `RelEvt` plus souple qui considère tous les événements équivalents entre eux. Le résultat fourni par MecV permet de conclure rapidement si le raffinement n'est pas établi. Si une relation de simulation existe, la relation `sim_evt` suivante calcule les équivalences entre événements du nœud abstrait et événements du nœud détaillé :

```
sim_evt(e,e') := (<s><s'><t><t'>(transA(s,e,t) & transA'(s',e',t')
                                 & sim(s,s') & sim(t,t')));
```

Les résultats de cette relation sont à analyser et valider par l'utilisateur pour établir si le nœud détaillé raffine le nœud abstrait.

Ces deux approches sont comparables en nombre d'opération, la spécification des équivalences de l'une étant remplacée par la validation des équivalences calculées de l'autre. Néanmoins, si la seconde approche indique que le nœud détaillé ne simule pas le nœud abstrait, le temps nécessaire à spécifier les événements équivalents est économisé. De plus, cette approche peut permettre de comprendre certains comportements en découvrant des équivalences de défaillances que l'utilisateur n'avait pas envisagées.

Si le test a montré que les états accessibles du nœud détaillé sont en relation avec des états du nœud abstrait, mais que la relation de simulation n'a pas pu être établie, la seconde relation à tester est la simulation quasi-branchante.

Une approche "vérification" s'appuie sur la relation `RelEvt` précédemment détaillée et sur la spécification des événements non observables de chaque nœud. De même que pour l'approche rigoureuse de vérification de la simulation, le résultat fourni par MecV permet d'établir directement si le nœud détaillé raffine le nœud abstrait.

Une approche "exploration" s'appuie sur l'observation du changement de valeur de variables de flux pour déterminer si, vis-à-vis des variables de flux observées, le changement d'état est observable ou non. Ainsi, les transitions non observables sont calculées à l'aide de relations sur les variables de flux définies initialement par l'utilisateur. De façon analogue à la relation de simulation, le résultat fourni par MecV permet de conclure rapidement si le raffinement n'est pas établi. Si une relation de quasi-branching simulation existe, la relation `QBS_evt` suivante calcule les équivalences entre événements du nœud abstrait et événements du nœud détaillé :

```
QBS_evt(e,e') := (<s><s'><t><t'>(transA(s,e,t) & transA'(s',e',t')
                                 & A_QBSim_A'(s,s')
                                 & A_QBSim_A'(t,t')));
```

Les résultats de cette relation sont à analyser et valider par l'utilisateur pour établir si le nœud détaillé raffine le nœud abstrait.

Les deux approches présentées pour chaque relation sont comparables en nombre d'opération, la spécification d'informations précises sur les événements de l'une étant remplacée par la validation des équivalences calculées de l'autre. Néanmoins, si la seconde approche indique que le nœud détaillé n'est pas en relation avec le nœud abstrait, le temps de spécification initial est économisé. De plus, cette approche peut permettre de comprendre certains comportements en découvrant des équivalences de défaillances que l'utilisateur n'avait pas envisagées.

Dans le cas de la relation de simulation quasi-branchante, en basant l'observation du changement d'état sur des composants du système et non sur l'observateur, les transitions non observables peuvent symboliser des défaillances non détectées. La relation peut alors comparer une défaillance d'un nœud abstrait avec un chemin constitué de défaillances non détectées suivies d'une défaillance détectée. Le calcul des transitions observables, que nous proposons, se rapproche de la notion de panne cachée ou latente[1].

Une fois le raffinement établi, la génération des séquences des différentes situations redoutées traitées par le nœud détaillé permet d'effectuer, dans un second temps, une quantification précise de l'occurrence de chaque situation redoutée.

La méthodologie que nous venons de décrire est illustrée sur la figure 5.6 suivante.

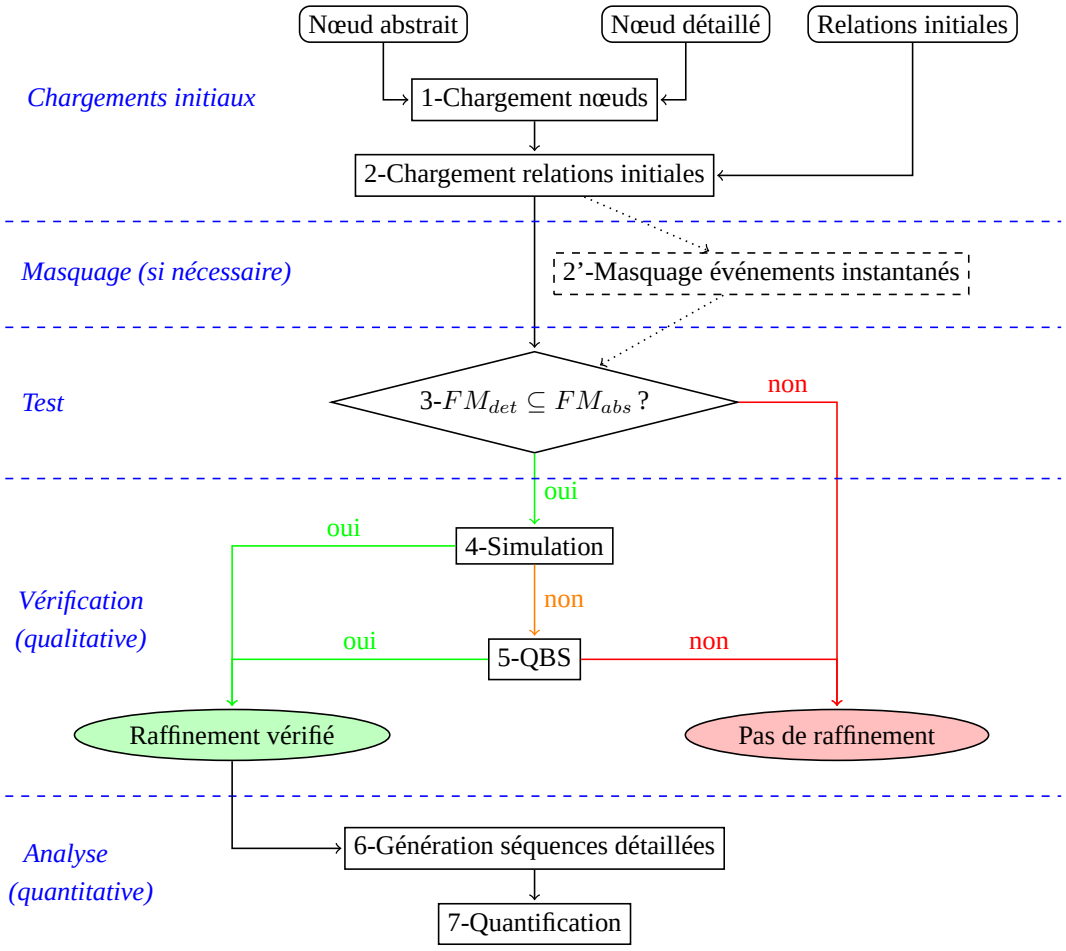


FIG. *5.6: Protocole de vérification du raffinement*

Les deux approches possibles pour vérifier la relation de simulation sont présentées sur la figure 5.7 suivante. Ce schéma se transpose aisément pour la relation de simulation quasi-branchante.

[1]La sûreté de fonctionnement des systèmes aéronautiques considère la notion de panne cachée : un composant de secours peut subir une défaillance qui n'est pas détectée lorsqu'elle se produit, par opposition à la panne active. L'occurrence d'une panne cachée n'est visible que lorsque le composant impacté est sollicité à la suite de défaillances des composants principaux. Les analyses quantitatives considèrent un temps d'exposition supérieur à la durée d'un vol dans le cas d'une panne cachée.

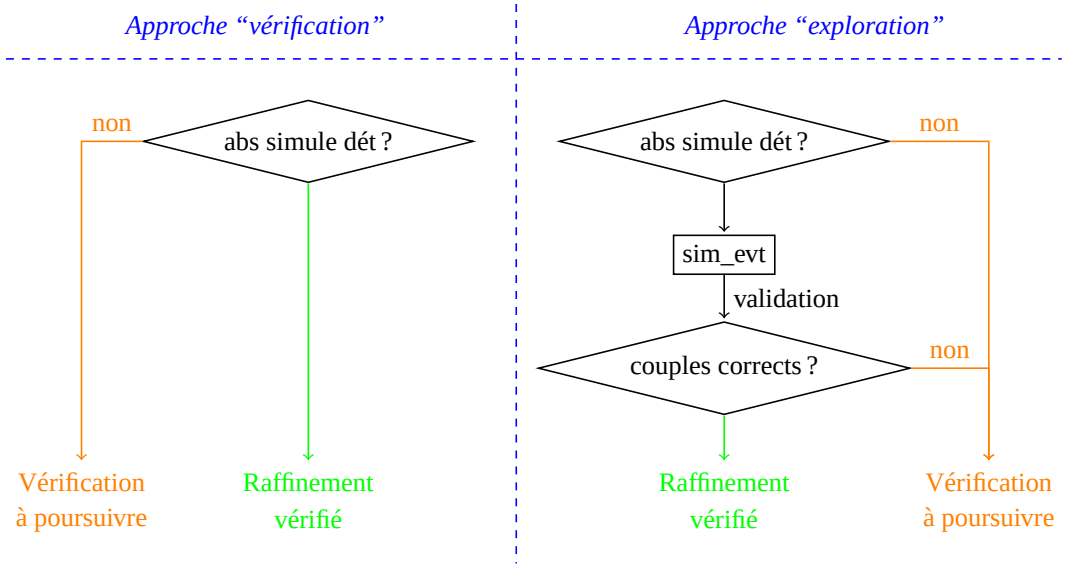


FIG. *5.7: Vérification de la simulation*

Appliquons cette méthodologie pour vérifier les relations existantes entre les différents calculateurs décrits dans ce chapitre. Compte tenu des avantages énoncés, nous faisons le choix de l'approche “exploration” en utilisant l'outil pour générer les événements équivalents. La relation `RelF` et les relations permettant d'établir si une transition est observable sont basées sur les mêmes variables d'état.

Le tableau 5.1 présente les résultats des comparaisons réalisées entre deux descriptions d'un calculateur en considérant la relation `RelF` basée sur la variable de sortie `Output` du calculateur. Chaque case indique si le nœud abstrait, lu sur la gauche, simule ou est en relation de quasi-branching bisimulation avec le nœud détaillé, lu en haut. Le symbole X indique qu'aucune relation n'est vérifiée.

	Cpu0	Cpu1	Cpu2	Cpu3	Cpu4	Cpu5
Cpu0	simule/QBS	X	X	X	X	X
Cpu1	simule/QBS	simule/QBS	X	X	X	X
Cpu2	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS
Cpu3	simule/QBS	simule/QBS	X	simule/QBS	simule/QBS	simule/QBS
Cpu4	simule/QBS	X	X	X	simule/QBS	X
Cpu5	simule/QBS	simule/QBS	X	simule/QBS	simule/QBS	simule/QBS

TAB. *5.1: Relations entre différentes descriptions d'un calculateur, comparaison établie sur la valeur en sortie du calculateur*

Tout nœud est, de manière évidente, en relation de bisimulation avec lui même, il est donc normal que tout nœud se simule et soit en relation de simulation quasi-branchante avec lui même.

D'autre part, la simulation est un cas particulier de la relation de simulation quasi-branchante : le seul événement non observable est ϵ. Il est donc également logique de constater que si la simulation est vérifiée, alors la simulation quasi-branchante l'est également.

Regardons en détail ces résultats en commençant par les comparaisons d'un nœud avec les nœuds plus détaillés. Nous constatons que :

- les calculateurs `Cpu0` et `Cpu1` ne sont en relation avec aucun calculateur plus détaillé ;
- le calculateur `Cpu2` simule les calculateurs plus détaillés ;
- le calculateur `Cpu3` simule les calculateurs `Cpu4` et `Cpu5` ;
- le calculateur `Cpu4` n'est pas en relation avec le calculateur `Cpu5`.

Regardons maintenant les comparaisons d'un nœud avec les nœuds plus abstraits. Nous observons que :

– les calculateurs `Cpu1`, `Cpu2`, `Cpu3`, `Cpu4` et `Cpu5` simulent le calculateur `Cpu0` ;
– les calculateurs `Cpu2`, `Cpu3` et `Cpu5` simulent le calculateur `Cpu1` ;
– le calculateur `Cpu5` simule les calculateurs `Cpu3` et `Cpu4`.

Ces résultats s'expliquent, d'une part, par le fait que les calculateurs `Cpu0` et `Cpu1` ne contiennent pas d'entrée symbolisant l'alimentation électrique. Cette entrée peut varier indépendamment du calculateur et influe sur le comportement de ce dernier. Nous avons réalisé une expérimentation complémentaire en comparant le calculateur `Cpu0` avec le calculateur `Cpu4` pour lequel la variable `Power` était toujours vraie. Cette comparaison a montré que chaque composant simule l'autre.

D'autre part, les modes de défaillance accessibles ne sont pas les mêmes pour tous : les calculateurs `Cpu0` et `Cpu4` n'ont qu'un mode de défaillance, à savoir la perte, contrairement aux autres calculateurs pour lesquels une défaillance ou une combinaison simultanée de défaillances peut conduire à un ordre erroné en sortie. Le calculateur `Cpu4`, bien que composé d'instances du calculateur `Cpu3`, contient un comparateur qui prévient l'état erroné en détectant les écarts entre les deux voies de calcul. Le calculateur `Cpu5`, bien que constitué des mêmes composants que le calculateur `Cpu4`, considère les défaillances simultanées et plus particulièrement l'erreur combinée des deux voies qui ne peut être détectée par le comparateur.

Enfin, la variable de flux entrant `Power` n'a pas d'impact sur les transitions possibles dans le nœud `Cpu2`, une défaillance peut se produire même lorsque le calculateur n'est pas alimenté, contrairement aux calculateurs plus détaillés pour lesquels aucune défaillance n'est possible lorsque `Power=false`.

Cette première expérimentation nous permet de tirer deux enseignements sur l'application du raffinement pour les analyses de sûreté de fonctionnement des systèmes :

– pour qu'un nœud d'un modèle puisse être remplacé dans un modèle par n'importe quel nœud qui le raffine, il est nécessaire que le nœud abstrait et tous ses raffinements soient compatibles entre eux, autrement dit qu'ils disposent des mêmes variables de flux (même nom et même type) ; cela implique que le modèle abstrait doit intégrer tous les modes de défaillances, qui apparaitront dans ses raffinements même s'ils ne sont pas pertinents au niveau abstrait ; comme dans les modèles que nous considérons, les types des variables reflètent les modes de défaillance pris en compte, ceci a pour conséquence que certaines valeurs présentes dans les types peuvent être superflues au niveau abstrait ;
– pour qu'un nœud détaillé raffine un nœud abstrait tout en préservant la compositionalité, il ne doit pas ajouter de nouveau mode de défaillance. Cela se vérifie sur l'exemple du calculateur `Cpu0` qui ne peut être dans un état erroné et n'est raffiné par aucun calculateur plus détaillé car ceux-là peuvent tous atteindre cet état.

Pour compléter ces enseignements, effectuons les mêmes comparaisons en définissant la relation `RelF` par rapport à la sortie booléenne `CpuLost` de l'observateur. Le tableau 5.2 présente les résultats de ces comparaisons. La lecture du tableau s'effectue de façon analogue à la lecture du tableau précédent.

	Cpu0	Cpu1	Cpu2	Cpu3	Cpu4	Cpu5
Cpu0	simule/QBS	simule/QBS	X	X	X	X
Cpu1	simule/QBS	simule/QBS	X	X	X	X
Cpu2	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS
Cpu3	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS
Cpu4	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS
Cpu5	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS	simule/QBS

TAB. *5.2: Relations entre différentes descriptions d'un calculateur, comparaison établie sur l'occurrence de la situation redoutée CpuLost*

Pour les raisons évoquées précédemment, il est normal de constater à nouveau que lorsqu'un nœud simule un autre nœud, il le qb-simule également.

Nous constatons que :

– les calculateurs ayant même interface se simulent mutuellement ;

– les calculateurs munis d'une entrée simulent les calculateurs sans entrée.

Une rapide comparaison avec les résultats du tableau 5.1 montre que, en définissant la relation `RelF` en fonction d'une situation redoutée de l'observateur, certains calculateurs qui n'étaient pas en relation le sont devenus. Les calculateurs `Cpu0` et `Cpu4`, qui précédemment ne simulaient pas les calculateurs de même interface pouvant accéder à l'état erroné, les simulent désormais. En ne regardant que la variable `CpuLost`, l'état erroné est équivalent à l'état nominal car tous deux sont différents de l'état de perte.

Cette seconde expérimentation nous permet de tirer un troisième enseignement sur l'application du raffinement pour les analyses de sûreté de fonctionnement des systèmes : si l'objectif du raffinement est la préservation d'exigences liées à une situation redoutée et non la préservation de la compositionalité, alors il semble possible d'établir une relation de raffinement malgré l'ajout de modes de défaillance, en s'assurant que l'interface reste inchangée.

5.3 EXPÉRIMENTATION

Nous avons souhaité appliquer la méthodologie de vérification du raffinement sur une étude plus conséquente que les différents exemples traités jusqu'ici et qui aborde la problématique multi-système. Pour cela, nous nous sommes inspirés du modèle électrique A320, réalisé dans le projet ESACS, et du système de contrôle de la gouverne de direction, présenté dans le chapitre 3, cf. figure 3.3. Les modèles qui suivent représentent un hypothétique système de génération et distribution électrique d'un avion bi-réacteur.

5.3.1 ÉTUDE DU SYSTÈME DE GÉNÉRATION ET DISTRIBUTION ÉLECTRIQUE

Le système de génération et distribution électrique d'un avion convertit, en vol, les efforts mécaniques des réacteurs en puissance électrique délivrée aux différents systèmes embarqués à bord d'un avion. Lorsque l'avion dispose de deux réacteurs, ce système se compose généralement de trois lignes de distribution : deux lignes nominales respectivement alimentées par les moteurs 1 (side 1) et 2 (side 2) et une ligne de secours (essential), reliée à une éolienne déployée en cas de besoin. Chaque ligne délivre un courant alternatif, via une barre électrique AC, et un courant continu, via une barre électrique DC.

A partir de cette description très abstraite du système, nous avons élaboré trois modèles :

– un modèle abstrait, baptisé Elec1, où chaque ligne est indépendante ;
– un modèle préliminaire, baptisé Elec2, où les lignes sont reliées entre elles pour intégrer des principes de redondance ;
– un modèle détaillé, baptisé Elec3, intégrant des contacteurs pour ébaucher une possible architecture embarquée.

Nous souhaitons vérifier si ces trois vues d'un même système présentent un lien de raffinement.

Par souci d'homogénéité et de lisibilité, nous adoptons pour convention d'écriture que tout port `x` d'un composant électrique dispose de deux variables booléennes `x_o^V` et `x_i^V` représentant respectivement la présence d'une tension en sortie et en entrée du port.

L'ajout des courts-circuits aux modèles décrits se matérialise par l'ajout de deux variables booléennes `x_o^Sc` et `x_i^Sc` représentant respectivement la présence d'un court-circuit en sortie et en entrée du port.

5.3.1.1 Modèle abstrait : Elec1

Le modèle abstrait du système de génération électrique considère que les trois lignes sont totalement ségréguées et ne partagent donc aucun composant. Chaque ligne nominale se compose d'un générateur alimentant une barre AC qui, elle-même, alimente la barre DC. En l'état, les barres DC1 et DC2 n'étant alimentées que par une barre AC, la perte de cette dernière équivaut à la perte simultanée des deux barres de la ligne. En revanche la barre DC_ESS est alimentée soit par la barre AC_ESS, soit par le générateur de secours CSM_G.

Les générateurs `GEN1`, `GEN2` et `CSM_G` sont de type `source`. Leur rôle est de fournir du courant aux barres `AC` lorsque les moteurs ou l'éolienne de secours (en anglais Ram Air Turbine, RAT) fonctionnent.

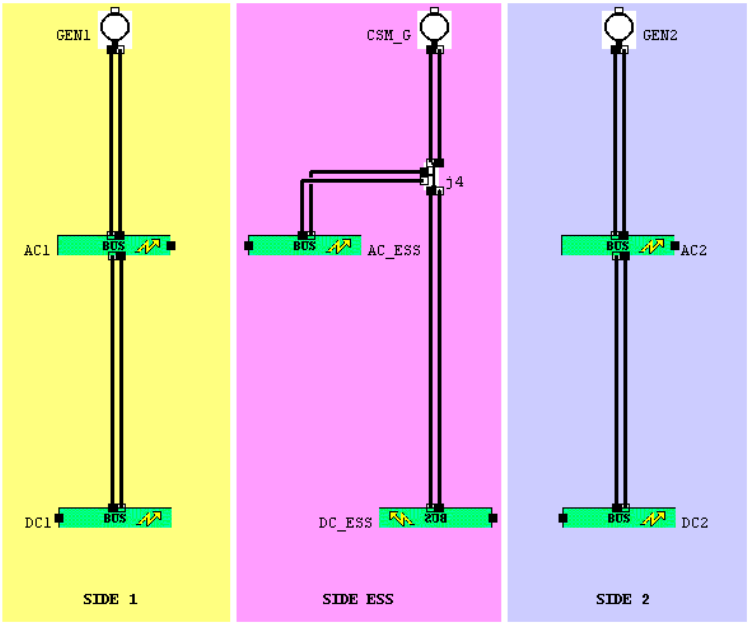


FIG. *5.8: Modèle abstrait du système de génération et distribution électrique*

```
node source
  flow
    a_i^V:bool:in;
    a_o^V:bool:out;
    I:bool:in;
  state
    state_:{lost, ok};
  event
    fail_lost;
  trans
    ((state_ = ok) and I) |- fail_lost -> state_ := lost;
  assert
    a_o^V = (I and (state_ = ok));
  init
    state_ := ok;
edon
```

Les barres `AC1` et `AC2` sont de type `bus_dispatcher`. Elles sont capables de recevoir du courant de deux sources différentes qu'elles peuvent elles-mêmes alimenter.

```
node bus_dispatcher
  flow
    status:bool:out;
    b_o^V:bool:out;
    a_o^V:bool:out;
    b_i^V:bool:in;
    a_i^V:bool:in;
  state
    state_:{lost, ok};
  event
```

```
    fail_lost;
  trans
    ((state_ = ok)
     and (a_i^V or b_i^V)) |- fail_lost -> state_ := lost;
  assert
    a_o^V = false;
    b_o^V = (a_i^V and (state_ = ok));
    status = ((a_i^V or b_i^V) and (state_ = ok));
  init
    state_ := ok;
edon
```

Les barres AC_ESS, DC1, DC2 et DC_ESS sont de type bus_receiver. Elle ne sont alimentées que par une source et ne peuvent fournir du courant qu'à cette même source.

```
node bus_receiver
  flow
    status:bool:out;
    a_o^V:bool:out;
    a_i^V:bool:in;
  state
    state_:{lost, ok};
  event
    fail_lost;
  trans
    ((state_ = ok) and a_i^V) |- fail_lost -> state_ := lost;
  assert
    a_o^V = false;
    status = (a_i^V and (state_ = ok));
  init
    state_ := ok;
edon
```

Le composant j4 est de type barre3. Il dispose de trois entrées et trois sorties pour être connecté à trois composants. Son rôle est de transmettre du courant à chaque composant tant qu'un des deux autres lui en fournit.

```
node barre3
  flow
    a_i^V:bool:in;
    a_o^V:bool:out;
    b_i^V:bool:in;
    b_o^V:bool:out;
    c_i^V:bool:in;
    c_o^V:bool:out;
  assert
    a_o^V = (b_i^V or c_i^V);
    b_o^V = (a_i^V or c_i^V);
    c_o^V = (a_i^V or b_i^V);
edon
```

5.3.1.2 Modèle préliminaire : Elec2

Une description plus détaillée du système introduit des principes de reconfiguration, au détriment de la ségrégation précédemment définie :

– les générateurs GEN1 et GEN2 n'alimentent pas uniquement une barre AC mais chacun peut alimenter les barres AC1 et AC2 ; la barre AC_ESS reste alimentée directement par CSM_G ;
– les trois barres AC sont connectées entre elles, chacune pouvant ainsi alimenter les deux autres ;
– chaque barre DC peut être alimentée par chaque barre AC et chaque autre barre DC.

Le modèle *préliminaire*, illustré sur la figure 5.9, intègre ces considérations.

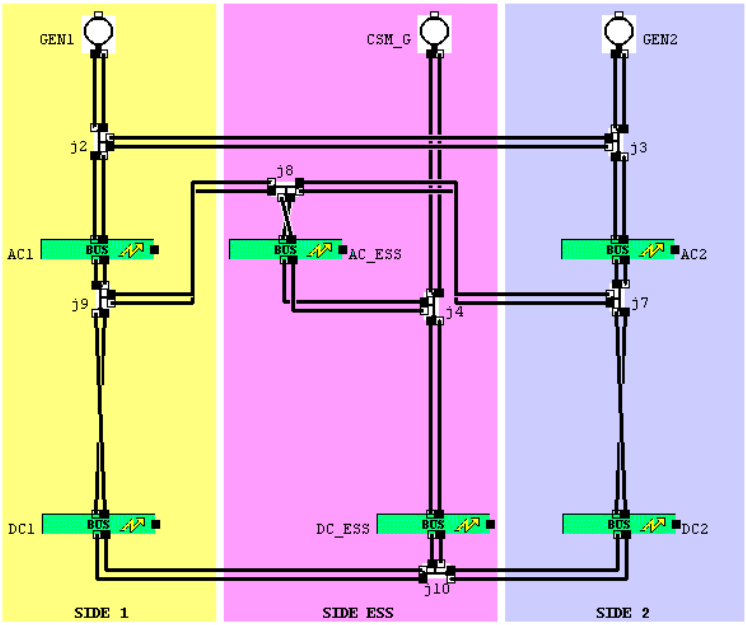


FIG. *5.9: Modèle préliminaire du système de génération et distribution électrique*

Les composants `j2`, `j3`, `j7`, `j8`, `j9` et `j10` sont ajoutés pour prendre en compte les échanges de courant entre les cotés 1, 2 et ESS.

Les barres `AC_ESS`, `DC1`, `DC2` et `DC_ESS` sont désormais de type `bus_dispatcher`, similaires aux barres `AC1` et `AC2`.

Les générateurs `GEN1`, `GEN2` et `CSM_G` ne sont pas modifiés.

5.3.1.3 Modèle détaillé : Elec3

Les barres AC fournissent du courant alternatif. Ce courant doit être traité pour que les barres DC puissent fournir du courant continu. Des transformateurs `TR` doivent donc être ajoutés avant les barres DC.

Le modèle préliminaire décrit des principes de redondance. L'implémentation de ces principes nécessite des contacteurs `ct` pour limiter la circulation de courant et prévenir les risques de courts-circuits.

Le modèle détaillé, illustré sur la figure 5.10 s'inspire du modèle préliminaire et représente une possible architecture contenant des contacteurs et des transformateurs.

Les contacteurs, de type `contactor`, sont ajoutés principalement à la sortie des générateurs, entre un transformateur et la barre DC qu'il alimente ainsi qu'à la jonction entre deux cotés (1/ESS ou ESS/2). S'ils reçoivent un ordre, alors ils laissent passer le courant.

```
node contactor
  flow
    a_i^V:bool:in;
    a_o^V:bool:out;
    b_i^V:bool:in;
    b_o^V:bool:out;
```

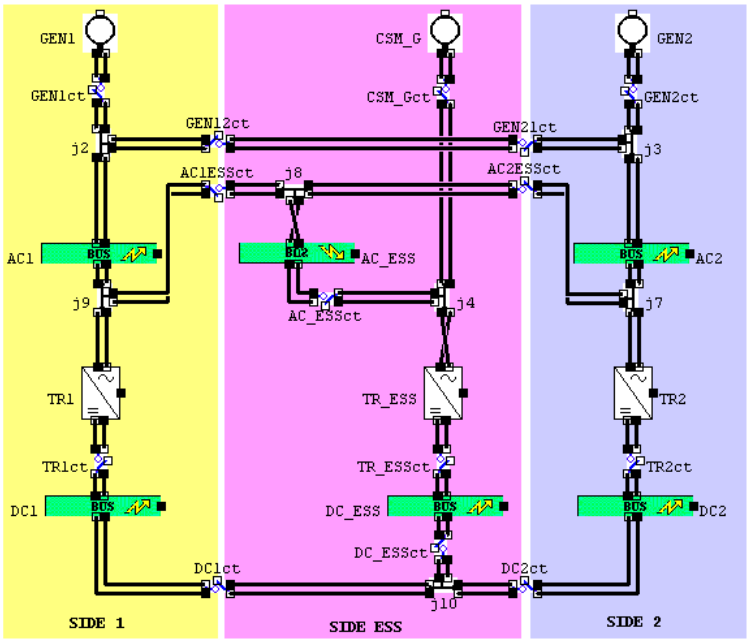


FIG. *5.10: Modèle détaillé du système de génération et distribution électrique*

```
    order:bool:in;
  assert
    a_o^V = (order and b_i^V);
    b_o^V = (order and a_i^V);
edon
```

Les règles d'ouverture/fermeture des contacteurs sont réparties dans quatre composants :

- un contrôleur `GEN_Controller` pour les contacteurs placés entre les générateurs et les barres AC ;
- un contrôleur `AC_Controller` pour les contacteurs placés entre les barres AC ;
- un contrôleur `TR_Controller` pour les contacteurs placés entre les transformateurs et les barres DC ;
- un contrôleur `DC_Controller` pour les contacteurs placés entre les barres DC ;

Les règles que nous avons implémentées[2] sont :

- un générateur (GEN 1 ou GEN 2) suffit à alimenter les deux barres AC nominales ;
- la barre AC de secours (ESS) est alimentée en priorité par AC 1, puis par AC 2 en cas de défaillance de Gen 1 ou AC 1 et enfin par le générateur CSM_G lorsque les autres générateurs sont perdus ;
- un contacteur placé entre un transformateur et une barre DC est fermé tant que le transformateur fonctionne ;
- la barre DC 1 est alimentée par la barre AC 1 ;
- la barre DC 2 est alimentée nominalement par la barre AC 2, puis, en cas de défaillance de Gen 2, TR 2 ou AC 2, par la barre DC 1 (si celle-ci est elle-même alimentée).
- la barre DC de secours (ESS) est alimentée en priorité par DC 1, puis par DC 2 en cas de défaillance de Gen 1, TR1 ou AC 1 et enfin par la barre AC ESS lorsqu'un générateur, un transformateur ou une barre AC de chaque autre ligne est perdu ;

Les transformateurs, de type `tr_rectifier`, sont ajoutés sur le lien d'entrée de chaque barre DC. Une défaillance d'un transformateur interrompt l'alimentation de la barre DC concernée.

```
node tr_rectifier
```

[2]Nous insistons sur le fait que nous avons élaboré ces règles en nous inspirant des principes de reconfiguration appliqués dans l'industrie mais qu'elles n'ont pas été validées par des spécialistes.

```
  flow
    status:bool:out;
    b_i^V:bool:in;
    b_o^V:bool:out;
    a_o^V:bool:out;
    a_i^V:bool:in;
  state
    state_:{lost, ok};
  event
    fail_lost;
  trans
    (state_ = ok) and (a_i^V or b_i^V) |- fail_lost -> state_ := lost;
  assert
    a_o^V = false;
    b_o^V = (a_i^V and (state_ = ok));
    status = ((state_ = ok) and a_i^V);
  init
    state_ := ok;
edon
```

5.3.1.4 Modèle détaillé avec courts-circuits : Elec4

Le court-circuit constitue une défaillance généralement considérée dans les systèmes électriques et particulièrement redoutée car pouvant causer d'importants dégats s'il n'est pas contenu.

Afin d'enrichir le modèle détaillé, nous considérons que les barres électriques et les transformateurs peuvent subir une défaillance ayant pour conséquence un court-circuit qui se propage en direction du composant qui l'alimente :

- un court-circuit sur une barre AC 1 ou 2 impacte le générateur qui l'alimente,i.e. Gen 1 ou 2 ;
- un court-circuit sur la barre AC Ess impacte la barre AC qui l'alimente ou le générateur CSM_G lorsque les barres AC 1 et 2 ne peuvent plus l'alimenter ;
- un court-circuit sur un transformateur impacte la barre AC à laquelle il est connecté ;
- un court-circuit sur la barre DC 1 (resp. DC 2) impacte le transformateur correspondant en situation nominale ou la barre DC 2 (resp. DC 1) si la barre AC 1 ne peut l'alimenter ;
- un court-circuit sur la barre DC Ess impacte la barre DC qui l'alimente tant que les générateurs Gen 1 et 2 sont opérationnels puis impacte la barre AC Ess et le générateur CSM_G.

Nous ajoutons des fusibles qui s'ouvrent lorsqu'ils reçoivent un signal de court-circuit, ce qui a pour effet de stopper la propagation du court-circuit.

Le modèle détaillé avec courts-circuits, illustré sur la figure 5.11, s'inspire du modèle détaillé et diffère graphiquement par l'ajout d'un fusible :

- en amont et en aval de chaque barre AC ;
- en amont du transformateur ;
- en aval de chaque barre DC.

Chaque fusible est identifié par un nom se terminant par le suffixe `bk`.

Comme nous l'avons dit précédemment, modéliser les courts-circuits implique des modifications des composants. Ces modifications étant similaires d'un composant à l'autre, nous les présentons de façon générique.

L'ajout d'une défaillance implique la création d'un nouveau mode de défaillance ce qui se traduit en AltaRica par :

- une nouvelle valeur `sh_circ` pour la variable d'état baptisée `state_` dans les composants de ce modèle ;
- un nouvel événement `fail_sc` représentant l'apparition du court-circuit.

La propagation de la défaillance nécessite l'ajout de deux variables booléennes `x_o^Sc` et `x_i^Sc`, représentant respectivement la présence d'un court-circuit en sortie et en entrée du port `x`, et d'un

événement instantané `update`, introduisant un retard dans la propagation nécessaire pour prévenir les phénomènes de boucle de calcul.

Un composant est soumis à un court-circuit lorsqu'un court-circuit est reçu d'un des composants auquel il est relié ou lorsque du courant est reçu simultanément de ces deux composants. Par souci de lisibilité, nous définissons une variable privée `inst_sc` pour représenter la présence d'un court-circuit dont la définition est la suivante, `a` et `b` étant les ports de ce composant :

```
inst_sc = (b_i^SC or a_i^SC or (a_i^V and b_i^V));
```

Les transitions décrivant l'occurrence d'un court-circuit et sa réception s'écrivent :

```
(state_ = ok) |- fail_sc -> state_ := sh_circ;
((state_ = ok) and inst_sc) |- update -> state_ := sh_circ;
```

Un court-circuit "remonte le courant", autrement dit un court-circuit ne se propage à un composant que si celui-ci est à l'origine du courant. La propagation aux autres composants s'écrit donc à l'aide des assertions suivantes :

```
a_o^SC = (((state_ = sh_circ) or b_i^SC) and a_i^V);
b_o^SC = (((state_ = sh_circ) or a_i^SC) and b_i^V);
```

Ces modifications sont appliquées aux types `bus_dispatcher` et `tr_rectifier` et, à l'exception de l'événement `fail_sc` et de la transition associée, au type `barre3`. Le type `contactor` se contente de propager le court-circuit par les variables de flux sans contenir ni variable d'état, ni événement.

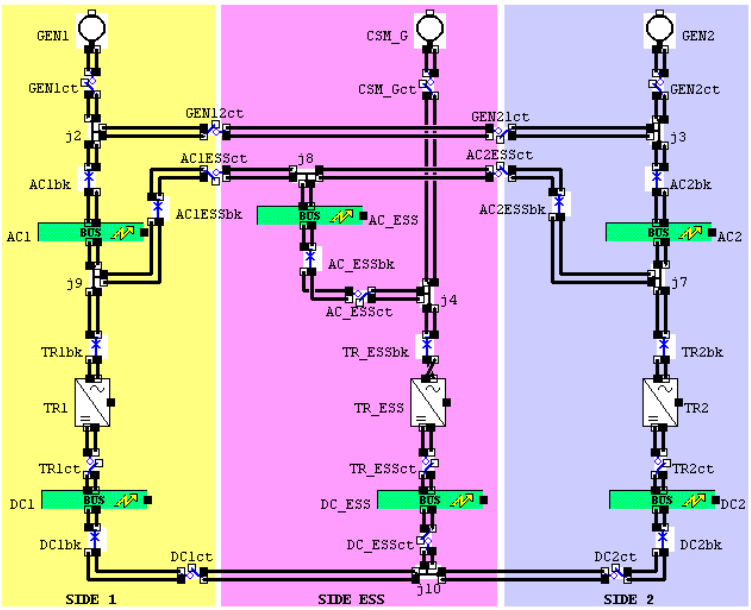


FIG. *5.11: Modèle détaillé du système de génération et distribution électrique, avec courts-circuits*

Les fusibles, de type `circuit_breaker`, contiennent en grande partie les ajouts mentionnés ci-dessus et peuvent être perçus comme un cas particulier, la présence d'un court-circuit n'étant pas propagée mais au contraire stoppée.

Initialement fermé, un événement instantané `update` ouvre le fusible et coupe la circulation de courant lorsqu'un court-circuit est reçu d'un des composants auquel le fusible est relié ou lorsque du courant est reçu simultanément de ces deux composants.

```
node circuit_breaker
  flow
```

```
    inst_sc:bool:private;
    a_i^V:bool:in;
    a_i^SC:bool:in;
    a_o^V:bool:out;
    a_o^SC:bool:out;
    b_i^V:bool:in;
    b_i^SC:bool:in;
    b_o^V:bool:out;
    b_o^SC:bool:out;
  state
    state_:{closed, opened};
  event
    update;
  trans
    ((state_ = closed) and inst_sc) |- update -> state_ := opened;
  assert
    inst_sc = (b_i^SC or a_i^SC or (a_i^V and b_i^V));
    a_o^V = case {(state_ = closed) : b_i^V,
                  else false};
    b_o^V = case {(state_ = closed) : a_i^V,
                  else false};
    a_o^SC = false;
    b_o^SC = false;
  init
    state_ := closed;
  extern
    law <event update> = Dirac(0);
edon
```

5.3.1.5 Observateur

La perte de chaque barre pouvant être observée directement sur chacune de ses sorties, nous définissons un composant observateur afin de prendre en compte les cas où plusieurs barres sont perdues. Nous considérons quatre situations redoutées :

- toutes les barres AC sont perdues, i.e. il n'y a pas une barre AC qui fournisse de la tension ;
- toutes les barres DC sont perdues, i.e. il n'y a pas une barre DC qui fournisse de la tension ;
- deux barres AC sur trois sont perdues, i.e. il n'y a pas deux barres AC qui fournissent de la tension ;
- deux barres DC sur trois sont perdues, i.e. il n'y a pas deux barres DC qui fournissent de la tension ;

```
node observer
  flow
    AC_ESS_status:bool:in;
    AC_side1_status:bool:in;
    AC_side2_status:bool:in;
    DC_ESS_status:bool:in;
    DC_side1_status:bool:in;
    DC_side2_status:bool:in;
    AC_ok:bool:out;
    DC_ok:bool:out;
    AC_Two_ok:bool:out;
    DC_Two_ok:bool:out;
  assert
    AC_ok = (AC_side1_status or AC_side2_status or AC_ESS_status);
    DC_ok = (DC_side1_status or DC_side2_status or DC_ESS_status);
```

```
    AC_Two_ok = ((AC_side1_status and AC_ESS_status)
              or (AC_side1_status and AC_side2_status)
              or (AC_side2_status and AC_ESS_status));
    DC_Two_ok = ((DC_side1_status and DC_ESS_status)
              or (DC_side2_status and DC_ESS_status)
              or (DC_side1_status and DC_side2_status));
edon
```

5.3.2 ANALYSE DU RAFFINEMENT

Ces modèles sont inspirés d'architectures et non des transpositions successives de descriptions industrielles. Les résultats d'analyse des modèles ont valeur d'exemple et ne cherchent pas à évaluer le niveau de sûreté de fonctionnement du système de génération et de distribution électrique.

5.3.2.1 Modèle abstrait (Elec1) comparé au modèle préliminaire (Elec2)

Les modèles Elec1 et Elec2 sont deux représentations abstraites du système de génération et de distribution électrique. Ce niveau abstrait a pour conséquence que les modèles sont de taille suffisamment réduite pour pouvoir procéder à une vérification globale par l'outil MecV.

La difficulté dans une approche de conception par raffinement est de bien choisir le modèle abstrait. Pour illustrer cette problématique du choix de la description initiale abstraite, nous procédons à la comparaison des modèles Elec1 et Elec2 en considérant Elec1 comme le modèle initial abstrait.

Nous concentrons nos travaux sur la vérification de la relation de simulation, à la fois dans une approche "vérification" et dans une approche "exploration". Afin d'illustrer la préservation globale, nous choisissons, pour relation `RelF`, de centrer l'attention sur les sorties physiques observées séparément. Pour illustrer la préservation restreinte, nous procédons de même avec deux sorties d'observateur. Les résultats obtenus pour des états accessibles en maximum 3 défaillances, sont présentés dans le tableau suivant :

	Elec1 simule Elec2	
Sortie comparée	vérification	exploration
AC1	faux	vrai
AC2	faux	vrai
AC_ESS	faux	vrai
DC1	faux	vrai
DC2	faux	vrai
DC_ESS	faux	vrai
AC_ok	faux	vrai
DC_ok	faux	vrai

TAB. *5.3: Résultats de comparaisons entre le modèle abstrait (Elec1) et le modèle préliminaire (Elec2)*

Considérons, tout d'abord, les résultats de l'approche "vérification". Nous constatons que le modèle Elec1 ne simule pas Elec2, quelle que soit la relation `RelF`.

Le modèle Elec2, du fait des reconfigurations, est plus sûr que le modèle Elec1 : la perte d'un générateur n'a pas d'influence sur les barres AC. La contrepartie est qu'une défaillance du second générateur conduit à la perte simultanée des barres AC1 et AC2. En l'état, le modèle Elec1 ne dispose pas de synchronisation de défaillances. La transition entre un état où toutes les barres sont disponibles et un état où deux barres sont perdues n'est pas pris en compte par le modèle Elec1. Ceci justifie le fait qu'Elec1 ne peut simuler Elec2.

=> Elec1 est trop abstrait et Elec2 doit donc servir de référence

5.3.2.2 Modèle préliminaire (Elec2) comparé au modèle détaillé (Elec3)

Contrairement à l'expérimentation précédente sur les modèles Elec1 et Elec2, l'outil MecV ne nous permet pas de procéder à une vérification globale sur les modèles Elec2 et Elec3. Pour contourner cette difficulté, nous procédons à un "découpage horizontal" de chaque modèle :

- la première comparaison restreint le modèle aux générateurs et aux barres AC (cf. figures 5.12 et 5.13) ;
- la seconde comparaison restreint le modèle aux générateurs et aux barres DC (cf. figures 5.14 et 5.15), masquant les barres AC et les liens entres barres AC.

Nous prenons la liberté de relier les barres DC aux générateurs pour que les variables d'entrée de ces barres ne soient pas libres.

Le modèle détaillé diffère principalement du modèle préliminaire par l'ajout des contacteurs. Le modèle préliminaire représente une architecture sûre. La nécessité d'ajouter des contacteurs et d'implémenter des règles de reconfiguration pour prévenir les courts-circuits fait que le comportement décrit par le modèle préliminaire constitue un idéal. Vérifier que le modèle détaillé est simulé par le modèle préliminaire revient à vérifier que les règles de reconfiguration, implémentées dans les contrôleurs, sont cohérentes avec les principes du modèle préliminaire. Lorsque ces contacteurs sont tous fermés, les deux modèles sont similaires.

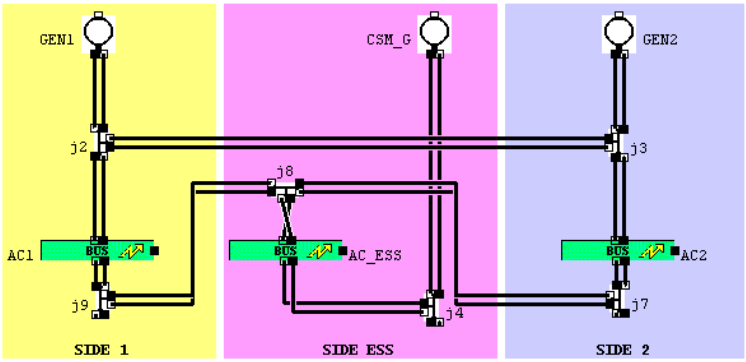


FIG. *5.12: Modèle préliminaire du système de génération et distribution électrique, restreint aux barres AC*

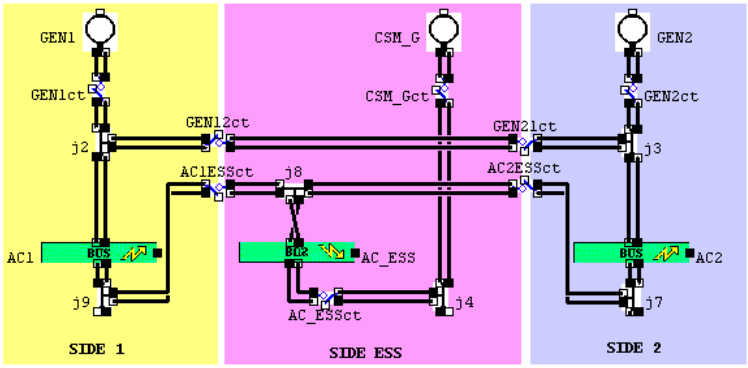


FIG. *5.13: Modèle détaillé du système de génération et distribution électrique, restreint aux barres AC*

Dans un premier temps, nous cherchons à vérifier si le modèle Elec2 simule le modèle Elec3, restreint aux barres AC. Nous étudions les scénarios contenant jusqu'à trois défaillances et vérifions suivant les deux approches et pour des objectifs de compositionnalité et de préservation des exigences liées aux situations redoutées modélisées. Les résultats sont présentés sur la figure 5.4.

	Elec2 simule Elec3	
Sortie comparée	vérification	exploration
AC 1	oui	oui
AC 2	oui	oui
AC ESS	oui	oui
AC 1 & AC 2	oui	oui
AC_ok	oui	oui
AC_Two_ok	oui	oui

TAB. *5.4: Résultats de comparaisons entre le modèle préliminaire (Elec2) et le modèle détaillé (Elec3), restreints aux barres AC*

Ces résultats montrent que le modèle Elec2 simule le modèle Elec3, quelle que soit la relation `RelF`. On peut donc conclure que, concernant le sous-système de génération et distribution électrique AC, le modèle Elec3 raffine le modèle Elec2. Ces résultats prouvent que les règles des contacteurs liés aux générateurs et aux barres AC sont cohérentes avec les principes du modèle préliminaire.

La simulation étant vérifiée par l'approche vérification, il est normal de constater qu'elle est également vérifiée par l'approche exploration moins stricte.

Regardons maintenant le sous-système de génération et distribution électrique DC.

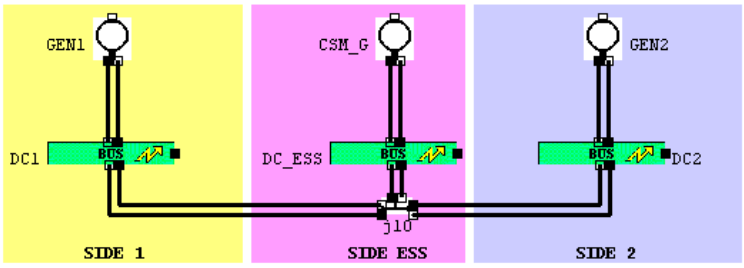


FIG. *5.14: Modèle préliminaire du système de génération et distribution électrique, restreint aux barres DC*

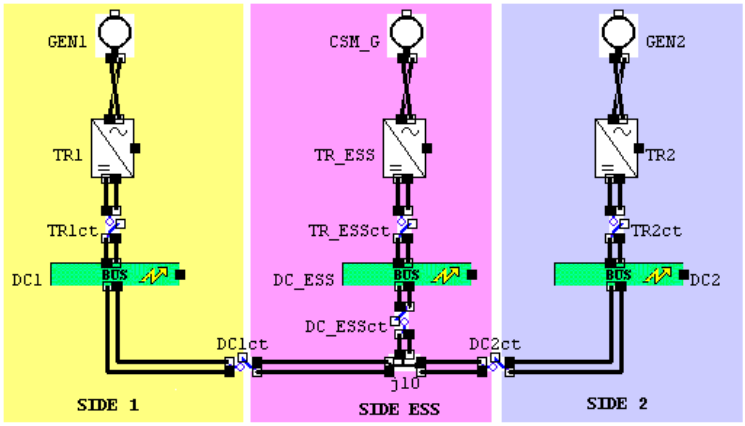


FIG. *5.15: Modèle détaillé du système de génération et distribution électrique, restreint aux barres DC*

Dans un second temps, nous cherchons à vérifier si le modèle Elec2 simule le modèle Elec3, restreint aux barres DC. Nous effectuons une étude analogue à la précédente en choisissant les relations `RelF` adéquates.

Le sous-système de génération et distribution DC du modèle Elec3 contient des contacteurs et des transformateurs. La décomposition en étapes successives étant un fondement du raffinement, nous procédons tout d'abord à la vérification en l'absence de transformateurs, ce qui revient à considérer des transformateurs sans défaillance (nœudtr_rectifier_mono_noSC sans événement ni transition).

Les résultats sont présentés sur la figure 5.5.

	Elec2 simule Elec3	
Sortie comparée	vérification	exploration
DC 1	non	oui
DC 2	non	oui
DC ESS	oui	oui
DC 1 & DC 2	non	non
DC_ok	oui	oui
DC_Two_ok	non	oui

TAB. *5.5: Résultats de comparaisons entre le modèle préliminaire (Elec2) et le modèle détaillé (Elec3), privés du niveau de génération et distribution AC et en présence de transformateurs parfaits*

Nous constatons que la relation de simulation, suivant une approche vérification, est uniquement établie lorsque la relation la relation `RelF` porte sur la barre de secours DC ESS. Ceci montre que, dans le modèle Elec3, le contrôleur des contacteurs liés à cette barre électrique est cohérent avec les principes définis dans le modèle Elec2.

Il n'en est pas de même pour les reconfigurations des contacteurs liés aux barres DC 1 et 2. Une des raisons à cela est que, en l'état dans le modèle Elec3, la barre DC ESS est privilégiée en cas de perte des générateurs GEN 1 et GEN 2. Autrement dit lorsque ces deux générateurs sont perdus, les barres qui leur sont directement associées ne sont plus alimentées et seules les barres AC ESS et DC ESS le sont. Cette règle n'étant pas prise en compte dans le modèle Elec2, il est normal que celui-ci ne simule pas le modèle Elec3.

Ajoutons maintenant une défaillance dont l'effet est qu'un transformateur peut être perdu. Ce simple ajout nous laisse penser que le modèle obtenu devrait raffiner le modèle précédent, i.e. une relation de simulation suivant l'approche vérification devrait être établie.

En nous contentant d'enrichir le code AltaRica du nœud `tr_rectifier_mono_noSC`, les tests montrent que, quelle que soit la relation `RelF`, le raffinement n'est pas vérifié, pas plus que le modèle Elec2 ne simule le modèle Elec3.

En AltaRica, tout événement ajouté dans un sous-nœud est synchronisé avec l'événement ϵ. Or les défaillances des transformateurs ont un effet et ne peuvent naturellement pas être comparées avec un événement ϵ du modèle abstrait. Il faut donc définir, dans le nœud du modèle détaillé, un événement homonyme à chaque nouvelle défaillance d'un transformateur pour pouvoir comparer chaque nouvel événement à une défaillance du nœud abstrait.

Une première idée pour obtenir le raffinement attendu est que, les transformateurs étant situés entre les générateurs et les barres, l'effet de leur perte est comparable à la perte du générateur auquel chacun est connecté. Cela se traduit par l'enrichissement de la relation `relEvt` : chaque défaillance d'un TR sur le modèle détaillé est associée à la perte d'un générateur sur le modèle abstrait. Les tests montrent la simulation n'est malgré cela pas vérifiée.

Une seconde idée est qu'ajouter les transformateurs dans le modèle Elec3, revient à remplacer les générateurs du modèle Elec2, par un couple générateur/transformateur. Cela a pour conséquence que :

- la défaillance d'un transformateur du modèle Elec3 est comparable à une perte de générateur du modèle Elec2 ;
- la perte d'un générateur du modèle Elec3 peut survenir après la défaillance du transformateur auquel le générateur est relié, auquel cas elle n'a pas d'effet.

Le premier cas est évoqué dans le paragraphe précédent. Prendre en compte le second cas par la relation `relEvt` revient à associer chaque défaillance d'un générateur du modèle détaillé à l'événement ϵ du modèle abstrait. Les tests montrent alors que la simulation est vérifiée et donc que le raffinement est établi.

Ce dernier cas illustre bien les difficultés de l'ajout de composants munis de défaillances.

5.3.2.3 Modèle détaillé (Elec3) comparé au modèle détaillé avec courts-circuits (Elec4)

Les modèles Elec3 et Elec4 ne peuvent être traités dans leur globalité par MecV pour les raisons évoquées précédemment. Grâce à la compositionnalité préservée par la relation de simulation, nous pouvons décomposer l'analyse. Nous procédons en deux étapes suivant les principes d'enrichissement appliqués au modèle Elec3 pour créer le modèle Elec4 :

- un fusible est ajouté à la fois en amont et en aval de chaque barre AC, cf. figure 5.16 ;
- un fusible est ajouté en amont de chaque transformateur et en aval de chaque barre DC, cf. figure 5.17.

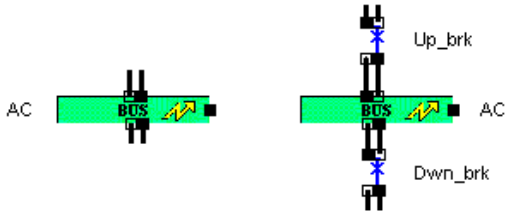


FIG. *5.16: Enrichissement du modèle Elec3 par ajout d'une défaillance entraînant un court circuit de la barre AC contenue par deux fusibles*

Dans un premier temps, nous souhaitons vérifier si une barre AC avec court-circuit reliée à deux fusibles (nœud `AC_cc`) est un raffinement d'une barre AC sans court-circuit ni fusible (nœud `AC`).

La détection d'un court-circuit par un fusible est un événement instantané qui doit être masqué avant vérification de la présence d'une relation de simulation entre les deux nœuds.

Intuitivement, un court-circuit de la barre AC suivi de l'ouverture du fusible positionné entre la barre et la source de tension semble équivalent à une perte de la barre AC. Nous définissons donc la relation `RelEvt` comme suit :

```
RelEvt(e:AC!ev, e' : AC_cc!ev) := (e.="AC_Loss" & e'.="AC_Loss")
                                 |(e.="AC_Loss" & e'.="AC_Sc")
                                 |(e.=""        & e'.="");
```

Nous choisissons d'établir la comparaison sur la présence d'une tension sur la sortie opposée à la source de tension : la sortie de la barre AC dans `AC` est comparée à la sortie du fusible opposé à la source de tension dans `AC_cc`.

Une approche vérification nous indique que le modèle `AC` simule le modèle `AC_cc`, donc `AC_cc` raffine `AC`. Ceci montre que l'ajout d'une défaillance combiné avec l'ajout de fusibles pour contenir les effets de celle-ci préserve le comportement des modèles contenant des barres AC pour lesquelles la perte est l'unique défaillance considérée.

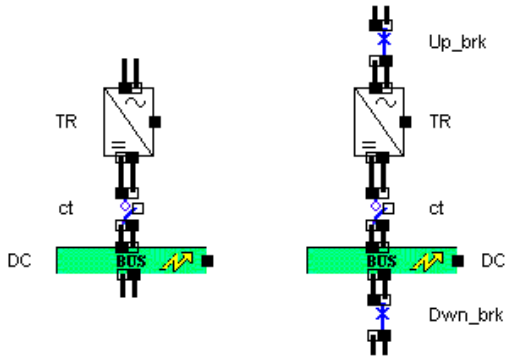


FIG. *5.17: Enrichissement du modèle Elec3 par ajout de défaillances entraînant respectivement un court circuit de la barre DC et du transformateur, contenues par deux fusibles*

Dans un second temps, nous souhaitons vérifier si un transformateur et une barre DC, tous deux avec court-circuit, reliés à deux fusibles (nœud `TR_DC_cc`) est un raffinement d'un transformateur et une barre AC, tous deux sans court-circuit ni fusible (nœud `TR_DC`).

Intuitivement, un court-circuit de la barre DC (resp. du transformateur) suivi de l'ouverture du fusible positionné entre la barre (resp. le transformateur) et la source de tension semble équivalent à une perte de la barre DC (resp. du transformateur). Nous définissons donc la relation `RelEvt` comme suit :

```
RelEvt(e:TR_DC!ev, e' : TR_DC_cc!ev) :=
                                  (e.="DC_Loss" & e'.="DC_Loss")
                                 |(e.="DC_Loss" & e'.="DC_Sc")
                                 |(e.="TR_Loss" & e'.="TR_Loss")
                                 |(e.="TR_Loss" & e'.="TR_Sc")
                                 |(e.=""        & e'.="");
```

Nous choisissons d'établir la comparaison sur la présence d'une tension sur la sortie opposée à la source de tension :

- lorsque la tension provient d'une barre AC en amont : la sortie de la barre `DC` dans `TR_DC` est comparée à la sortie du fusible `Dwn_brk` dans `TR_DC_cc` ;
- lorsque la tension provient d'une barre DC en aval : la sortie du transformateur `TR` dans `TR_DC` est comparée à la sortie du fusible `Up_brk` dans `TR_DC_cc`.

Une approche vérification nous indique que le modèle `TR_DC` simule le modèle `TR_DC_cc`, donc `TR_DC_cc` raffine `TR_DC`. Ceci montre à nouveau que l'ajout de défaillances combiné avec l'ajout de fusibles pour contenir les effets de ces dernières préserve le comportement des modèles contenant des transformateurs et des barres DC pour lesquelles la perte est l'unique défaillance considérée.

Appliquer le principe de compositionnalité pour calculer les séquences du modèle Elec4 à partir du modèle Elec3 réduit le nombre de séquences à tester. L'ajout des courts-circuits sur les barres électriques et les transformateurs augmente le nombre de défaillance considérées de 12 à 21.

La figure 5.18 met en évidence les enrichissements du modèle Elec4 par rapport au modèle Elec3 :
– l'ajout de fusibles à chaque barre AC est représenté par un rectangle orange ;
– l'ajout de fusibles avant chaque transformateur et après chaque barre DC est représenté par un rectangle bleu.

L'étude menée précédemment considérait un fusible de part et d'autre de chaque barre AC. La figure 5.11 montre que chaque barre AC est reliée à trois fusibles (un d'un côté et deux de l'autre). Les fusibles n'ayant pas de défaillance et leur ouverture étant instantanée, ces deux enrichissements (une sortie de barre AC reliée à un ou deux fusibles) sont équivalents.

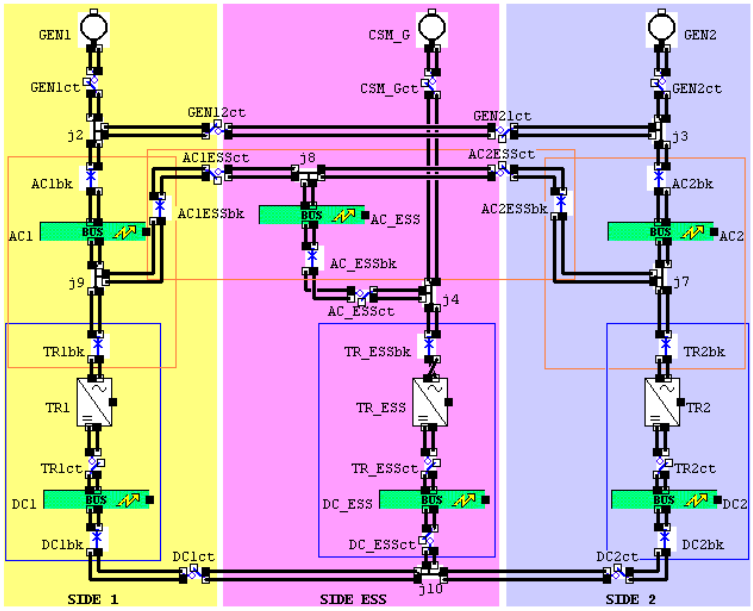


FIG. 5.18: *Motifs d'enrichissement du modèle détaillé du système de génération et distribution électrique*

5.3.3 UTILISATION DU RAFFINEMENT POUR LES ANALYSES MULTI-SYSTÈMES

Le système de contrôle de la gouverne de direction, détaillé dans le chapitre 3 et dont le modèle est présenté sur la figure 5.19, contient des composants nécessitant une alimentation électrique ou hydraulique. La modélisation de ce système, considère les systèmes de génération et distribution électrique et hydraulique de façon très abstraite : chaque sortie initialement vraie peut être perdue.

En réalité, les systèmes de génération et distribution électrique et hydraulique sont plus complexes : chacun dispose de son architecture propre, des pannes multiples peuvent survenir et ces deux systèmes sont liés aux moteurs. Contrairement à ce que la figure 5.19 peut laisser penser, les systèmes de génération et distribution électrique et hydraulique ne sont donc pas indépendants.

Analyser le système de contrôle de la gouverne de direction en détail nécessite de considérer les 4 systèmes précédemment mentionnés pour garantir la prise en compte de toutes les interactions possibles entre ces système. Cela revient donc à procéder à une analyse multi-systèmes : le système de contrôle de la gouverne de direction occupe une place centrale tandis que les trois autres systèmes mentionnés sont en interface.

Réaliser cette analyse détaillée à l'aide de modèles implique de modéliser indépendamment chaque système puis de connecter les modèles entre eux. Le modèle ainsi obtenu, que nous qualifions de *modèle multi-systèmes*, est illustré sur la figure 5.20.

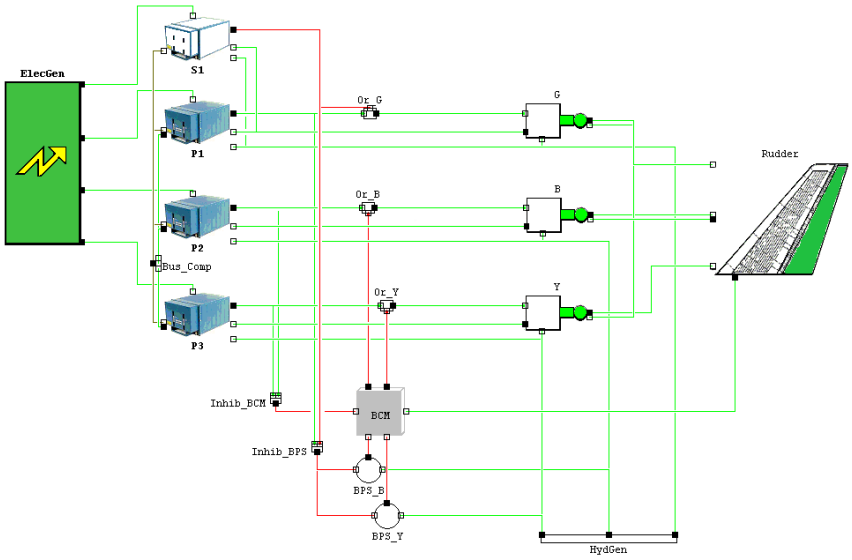


FIG. *5.19: Modèle AltaRica du système de contrôle de la gouverne de direction*

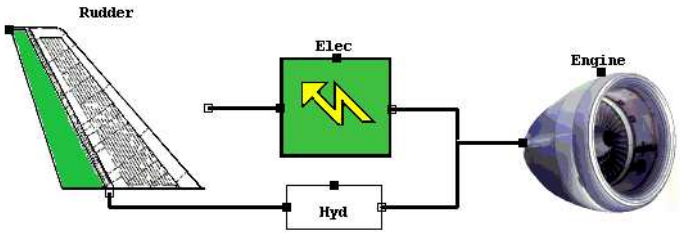


FIG. *5.20: Modèle multi-systèmes du système de contrôle de la gouverne de direction*

Idéalement, une analyse multi-systèmes à l'aide de modèles AltaRica devrait considérer un modèle détaillé pour chaque système. Pour conserver des résultats lisibles et prévenir les risques d'explosion combinatoire, il est préférable de considérer un modèle détaillé pour le système central et des modèles plus abstraits pour les systèmes en interface. Ce choix est possible du fait que le raffinement AltaRica préserve la compositionnalité.

Baptisons M_{Det} le modèle multi-systèmes composé uniquement de modèles détaillés et M_{Abs} le modèle multi-systèmes contenant des modèles abstraits pour les systèmes en interface. Si, pour chaque système en interface, le modèle détaillé raffine le modèle abstrait, alors les séquences de ces deux modèles peuvent être mises en relation. Les séquences de M_{Det} peuvent dès lors être calculées à partir des séquences de M_{Abs} (cf. figure 4.12).

Pour bénéficier pleinement de la compositionnalité et réduire la combinatoire, le modèle à privilégier pour chaque système en interface doit être le modèle le plus abstrait pour lequel chaque modèle détaillé représente un raffinement. Dans le cas du système de génération et distribution électrique, nous avons montré précédemment que :

– le modèle Elec4 raffine totalement le modèle Elec3 ;

– le modèle Elec3 raffine partiellement le modèle Elec2 ;
– le modèle Elec2 ne raffine pas le modèle Elec1.

Le modèle Elec1 bien que très abstrait n'est pas raffiné par les modèles plus détaillés et ne peut donc pas être utilisé pour le modèle multi-systèmes centré sur le système de contrôle de la gouverne de direction. Le modèle Elec3 raffine totalement le modèle Elec2 pour la partie AC et partiellement pour la partie DC. Corriger les règles de reconfiguration des contacteurs liés aux barres DC 1 et 2 devrait permettre d'obtenir un raffinement total. Le modèle Elec2 constitue donc le modèle à choisir pour le modèle multi-systèmes centré sur le système de contrôle de la gouverne de direction.

Une fois les modèles à connecter choisis, ces derniers doivent disposer de ports compatibles pour pouvoir être connectés. Ceci implique de définir a priori les ports nécessaires aux connexions et tout particulièrement de préciser leur type.

Le raffinement AltaRica est établi pour une relation `RelF` donnée et donc pour les variables de flux prises en compte dans cette relation. La compositionnalité n'est donc garantie que vis-à-vis de ces variables. Pour pouvoir tirer profit de la compositionnalité dans l'analyse multi-système, ces variables représentent les ports à connecter.

Un événement redouté d'un système peut être modélisé par une variable de flux booléenne sortant d'un observateur. Lorsque ce principe est appliqué à un système en interface, la variable de l'observateur peut servir à connecter le modèle du système en interface au modèle du système central. Cette variable correspond alors à la notion de Dependent System Failure (DSF) (cf. chapitre 3), présente dans les analyses classiques par arbre de défaillance ou diagramme de dépendance.

Travailler sur les interfaces d'un système peut offrir plus de souplesse que considérer tous les détails d'une architecture. Cette alternative a déjà fait l'objet de travaux qui ont conduit à la définition de la notion de *safety interfaces*.

Dans [ENTM05], J.Elmqvist, S.Nadjm-Tehrani et M.Minea présentent une approche visant à compléter le développement de systèmes critiques basé sur les composants, par l'identification des contraintes nécessaires pour garantir la tolérance à une ou deux défaillances. Le comportement fonctionnel d'un composant est décrit dans un *module* à l'aide d'un formalisme proche d'AltaRica et basé sur la notion de modules réactifs. Comparable à l'extension de modèle présentée dans le chapitre 3, cette approche ajoute au comportement fonctionnel des défaillances sur les entrées, baptisées *Input Fault Mode*, un *environnement* et introduit le concept de *safety interface*. Étant donnée une propriété φ, la safety interface d'un module M se compose de :

– l'environnement dans lequel la propriété est satisfaite ;
– l'ensemble des contraintes sur l'environnement associées à chaque défaillance simple ou combinaison de deux défaillances pour que la propriété soit satisfaite.

L'article [ENTM05] présente l'algorithme de calcul de l'environnement initial, ainsi que les moyens d'identification des environnements en cas de défaillance(s).

Bien que cette approche diffère de la notre qui se base sur des modèles de type FPM, toutes deux partagent plusieurs principes et objectifs.

Théoriquement, toutes les combinaisons de défaillances sans limite de taille pourraient être considérées dans les safety interfaces. Néanmoins, seules les défaillances simples et les combinaisons doubles sont traitées pour parer aux problèmes d'explosion combinatoire, précisant que les combinaisons de taille supérieure sont considérées dans l'industrie comme rares. Notre choix de borner les états accessibles est motivé par le même constat et tend à prévenir les mêmes risques.

D'autre part, la notion de raffinement, définie pour cette approche, s'appuie sur l'inclusion de traces : un module M raffine un module N si toutes les traces de M sont aussi des traces de N. Le raffinement basé sur la relation de simulation, que nous définissons dans le chapitre 4, préserve également les traces.

Enfin, cette approche s'appuie sur la notion de composition pour pleinement tirer profit du raffinement. Soit une exigence de sûreté de fonctionnement φ :

– chaque module est analysé individuellement pour déterminer sa safety interface (l'environnement d'un module est une abstraction des autres modules) ;
– on sait qu'un module composé avec son environnement vérifie φ ;

– chaque module, composé avec son environnement, est ensuite comparé aux environnement des autres modules pour vérifier qu'il raffine ces derniers ;

Si chacun de ces raffinements est vérifié alors, par inférence, le système, i.e. la composition de tous les modules, vérifie φ.

Il serait intéressant d'étudier s'il est possible, avec la notion de raffinement que nous proposons pour AltaRica, de définir des safety interface pour AltaRica.

5.4 PRÉSERVATION D'EXIGENCES QUANTITATIVES

La société Airbus, pour laquelle nous avons réalisé nos recherches, dispose de son propre outil de calcul de probabilité d'une situation qui permet d'effectuer des analyses quantitatives à partir de séquences. Cet outil étant reconnu par les autorités de certification, nous avons souhaité travailler en amont de celui-ci : sur les séquences. Nous nous sommes donc focalisés sur la préservation des exigences qualitatives. Néanmoins, il existe des approches de raffinement visant à préserver des exigences quantitatives.

En 1991, K.G.Larsen et A.Skou définissent la notion de bisimulation probabiliste [LS91]. Cette relation impose sur les transitions à la fois l'identité des étiquettes et l'égalité des probabilités d'occurrence.

Dans [Yam03], Satoshi Yamane définit les relations de simulation temporisée et de simulation faible temporisée probabiliste entre deux automates temporisés probabilistes (ATP). La simulation temporisée se distingue de la relation de simulation que nous avons considérée par l'ajout de contraintes sur les horloges et probabilités associées à chaque état. La simulation faible temporisée probabiliste permet de considérer qu'une transition d'un ATP abstrait simule une succession de transitions d'un ATP concret si les états initiaux (resp. terminaux) de la transition et du chemin sont en relation et que la dernière transition présente la même étiquette. Contrairement à la simulation quasi-branchante que nous avons définies, la simulation faible temporisée probabiliste n'impose aucune contrainte sur les états intermédiaires du chemin de l'ATP concret.

Dans [MM05], Annabelle McIver et Caroll Morgan définissent une méthode de vérification de préservation d'exigences qualitatives basée à la fois sur les techniques d'abstraction et de raffinement. Les règles de substitution de la méthode B sont transposées en *expectation transformers*. La méthode présentée garantit la préservation des *expectations*. De plus, du fait que le raffinement est compositionnel, la vérification suit une démarche hiérarchique en décomposant le système :

– le sous-système de plus bas niveau est analysé ;
– les propriétés alors établies ont valeur d'abstraction et sont utilisées pour analyser le niveau supérieur.

Les différents niveaux d'une analyse sont ainsi liés par des principes d'abstraction et de raffinement.

Dans [Tro99], Elena Troubitsyna présente une extension du formalisme des systèmes d'action et s'intéresse aux problèmes de sûreté de fonctionnement des systèmes. Le langage des commandes guardées est enrichi d'un opérateur de choix probabiliste binaire. Le raffinement de données dans un contexte probabiliste est utilisé pour établir une relation entre des exigences globales au système et des taux de défaillance des composants du systèmes. L'exemple support est un programme informatique, dont l'objectif est que son exécution se termine. De même que nous proposons, dans le raffinement AltaRica, que le nombre de défaillances nécessaire pour atteindre chaque état défaillant ne soit pas plus petit dans un nœud concret que dans un nœud abstrait, E.Troubitsyna impose que la probabilité qu'un programme se termine ne doit pas être inférieure dans le programme concret.

De même J.Elmqvist et S.Nadjm-Tehrani se sont intéressés aux analyses quantitatives (cf. [ENT08]), compléter le raffinement AltaRica afin de garantir la préservation d'exigences quantitatives constitue une perspective de poursuite de nos travaux. Cela s'avère d'autant plus envisageable que les idées mentionnées précédemment semblent transposables et qu'une probabilité d'occurrence peut être associée à chaque événement AltaRica à l'aide d'un attribut. Ce travail nécessiterait de modifier MecV pour pouvoir exploiter ces attributs, tout comme nous avons mentionné la prise en compte d'un attribut de visibilité pour traiter la relation de simulation quasi-branchante.

5.5 CONCLUSION

Nous avons montré comment adapter le raffinement AltaRica aux besoins des analyses de sûreté de fonctionnement des systèmes :

- chaque état est enrichi d'un compteur de défaillances permettant de compléter les contraintes sur les transitions par des exigences qualitatives ;
- les événements instantanés, qui n'apparaissent pas dans les coupes minimales, sont masqués avant de vérifier l'existence d'une relation entre deux nœuds ;
- la relation `RelF` permet de choisir entre une préservation globale, si la relation porte sur toutes les variables d'interface, et une préservation restreinte, si la relation porte seulement sur certaines variables d'interface ou sur une variable d'un observateur ;
- la relation `RelEvt` permet de choisir entre une approche vérification, nécessitant de spécifier toutes les équivalences d'étiquette de transition mais indiquant directement l'existence d'une relation de raffinement, et une approche exploration, calculant les équivalences entre étiquettes des transitions mais nécessitant de vérifier que ces équivalences ne sont pas contraires au comportement souhaité.

Le raffinement que nous proposons a l'avantage de préserver les séquences et d'être facilement vérifiable grâce aux différentes relations que nous avons implémentées en langage MecV et à la méthodologie que nous avons élaborée (cf. figure 5.6). La liberté dont dispose l'utilisateur pour définir les relations `RelF` et `RelEvt`, offre une grande souplesse quant aux contraintes à satisfaire.

La méthodologie que nous avons définie, basée sur le raffinement, permet de réaliser des analyses multi-systèmes en prévenant les risques d'explosion combinatoire.

Cependant, les expérimentations ont montré que l'utilisation du raffinement influe sur la méthode de modélisation :

- les variables de flux, qui ont valeur d'interface, doivent être identifiées et leurs types définis dès la spécification du système et avant toute modélisation :
  - ces variables doivent être conservées dans tous les raffinements, pour que chaque modèle puisse être remplacé par un raffinement ou un modèle plus abstrait du même système,
  - dans l'optique de réaliser un modèle multi-systèmes, ces ports permettent de connecter les modèles mono-systèmes qui doivent naturellement disposer de ports compatibles,
  - la compositionnalité et la cohérence sont préservées vis-à-vis des variables de flux spécifiées dans la relation `RelF`, et seulement vis-à-vis de celles-là : l'exemple du calculateur, dont la sortie peut être correcte, erronée ou perdue mais pour lequel le raffinement s'appuie sur la sortie d'un observateur ne détectant que si la sortie du calculateur est perdue, montre que le raffinement ne garantit pas la préservations de tous les comportements défaillants,
- chaque nouveau modèle doit être réalisé en pensant au précédent : l'exemple des contacteurs du système de génération et distribution électrique montre qu'il n'est pas trivial de rester cohérent avec le comportement décrit précédemment ;

Une même spécification peut mener à plusieurs implémentations dont seulement certaines raffineront un modèle abstrait. Une approche de modélisation par raffinement nécessite de procéder à une vérification de chaque modèle une fois réalisé et non de créer plusieurs modèles vérifiés a posteriori.

D'autre part, les expérimentations nous ont montré que la gamme de modèles que peut traiter l'outil MecV est plus large que pour la plupart des outils industriels mais les performances en pâtissent. MecV est en effet capable de traiter des modèles AltaRica LaBRI ou AltaRica OCAS, des modèles ouverts (i.e. une variable d'entrée d'un composant n'est connectée à aucune variable de sortie d'un autre composant) ou des modèles non déterministes (i.e. deux transitions ayant la même étiquette et le même état initial n'ont pas le même état final). Les modèles AltaRica réalisés dans l'industrie n'exploitent pas ces particularités. Nous pensons qu'une optimisation de l'outil MecV pour les modèles AltaRica OCAS, ou au moins Data-Flow, permettrait un gain de performances.

La méthode de vérification du raffinement n'est, aujourd'hui, pas supportée par des outils industriels. Néanmoins, nous avons écrit les formules de vérification MecV de manière générique de façon à pouvoir automatiser leur appel. Une implémentation dans un outil industriel de modélisation AltaRica permettrait

à l'utilisateur, une fois les modèles à comparer identifiés, de ne renseigner que certaines relations initiales (nombre de défaillances à considérer, `RelF` et `RelEvt`) avant d'appeler successivement les relations adéquates comme indiquer sur la figure 5.6.

Les industriels cherchent à capitaliser tout travail réalisé. Dans cette optique, les outils industriels de modélisation AltaRica disposent d'une bibliothèque d'objets AltaRica (modèles, composants simples, composants hiérarchiques, types de données). Une autre perspective pour aider à l'industrialisation des analyses de sûreté de fonctionnement des systèmes à l'aide de modèles, serait d'exploiter ces bibliothèques en proposant aux utilisateurs des familles de composants AltaRica : un composant physique donné disposerait d'un modèle abstrait et de raffinements successifs.

CONCLUSION

Les travaux décrits dans cette thèse se sont déroulés dans le cadre d'une Convention Industrielle de Formation par la REcherche (CIFRE). Les travaux réalisés se situent au croisement de deux domaines :

- la sûreté de fonctionnement des systèmes complexes aéronautiques ;
- l'informatique fondamentale et les méthodes formelles.

Pour bien comprendre la contribution de cette thèse à l'intégration de travaux fondamentaux dans un cadre industriel, nous présentons les liens entre les entités qui ont initié cette thèse puis l'ont coencadré. Nous résumons les enseignements des premières collaborations qui ont permis de fixer les objectifs de cette thèse. Nous poursuivons par un bilan et proposons quelques perspectives de poursuite des travaux réalisés, à chaque fois d'un point de vue académique et industriel.

Une thèse entre recherche fondamentale et recherche appliquée pour répondre à des besoins industriels

Historiquement, la société Airbus s'est intéressée à l'utilisation des méthodes formelles pour supporter ses analyses de sûreté de fonctionnement des systèmes. Airbus a ainsi participé à des projets de recherche en collaboration avec l'ONERA. De son coté, l'ONERA expérimentait le langage AltaRica, développé par le LaBRI.

Les expérimentations menées par Airbus et l'ONERA ont montré :

- les apports majeurs des modèles AltaRica pour les analyses de sûreté de fonctionnement des systèmes :
  - lisibilité de la représentation graphique par rapport à une description textuelle,
  - représentation synthétique facilitant la communication entre concepteurs et ingénieurs en charge des analyses de sûreté de fonctionnement des systèmes,
  - support commun à l'analyse de plusieurs situations redoutées,
  - automatisation de la génération de séquences,
- les inconvénients en vue d'une industrialisation de l'approche :
  - utiliser un nouveau langage implique d'apprendre une nouvelle syntaxe ; or, les ingénieurs n'ont pas tous le même rapport à l'informatique et l'apprentissage d'un langage s'avère un facteur bloquant pour certains,
  - une nouvelle méthode de travail nécessite de former les ingénieurs,
  - la génération de séquences partiellement automatisée modifie les méthodes de travail entre concepteurs et ingénieurs en sûreté de fonctionnement des systèmes : les concepteurs doivent apprendre à valider un modèle (i.e. s'assurer que le comportement formalisé est fidèle au comportement pensé par les concepteurs) et non vérifier au travers de la lecture des différents arbres de défaillances que les séquences sont correctes et complètes (i.e. les combinaisons de défaillances conduisent bien à la situation redoutée et aucune combinaison ne manque)

Afin de proposer une méthode de travail accessible à toutes les populations d'ingénieurs, quels que soient leur domaine et leur rapport à l'informatique, Airbus a fait développer son outil de modélisation disposant d'une interface de saisie pensée pour que les connaissances syntaxiques du langage nécessaires à la réalisation de modèles soient minimales.

L'introduction des modèles dans les processus d'analyse a suscité de nouveaux objectifs : modéliser la totalité des systèmes pour obtenir un modèle de l'avion dans son intégralité. Une première étape dans ce

sens fût la réalisation de modèles multi-systèmes. Les expérimentations ont soulevé des besoins propres aux modélisations multi-systèmes et conduit à l'élaboration de cette thèse dont :

- l'objectif industriel était de développer des techniques pour réaliser des modèles multi-systèmes permettant de supporter des analyses de sûreté de fonctionnement multi-systèmes en utilisant des modèles de granularité hétérogène ;
- l'objectif académique était de définir une notion de raffinement pour le langage AltaRica qui préserve les traces.

Le raffinement fut choisi comme méthode pour assurer la cohérence entre différents modèles d'un même système.

Bilan

D'un point de vue théorique, nous avons défini un raffinement AltaRica en nous appuyant sur la thèse de Gérald Point [Poi00] qui définit le langage AltaRica. La vérification du raffinement est basée sur l'utilisation de l'outil MecV.

Dans un premier temps, nous nous sommes inspirés de ses travaux sur la relation de bisimulation entre nœuds AltaRica pour conduire l'étude de la relation de simulation. Cette étude fût menée à son terme pour disposer :

- d'un théorème de compositionnalité similaire à celui défini pour la relation de bisimulation ;
- d'un algorithme permettant, une fois le raffinement vérifié, de calculer les traces d'un composant (resp. nœud) détaillé à partir des traces d'un composant (resp. nœud) abstrait puis d'en déduire les séquences.

Dans un second temps, nous avons étudié la relation de simulation quasi-branchante dérivée de la bisimulation quasi-branchante [GW96]. Cette relation distingue les événements en deux catégories, observables et non observables, ce qui nécessite d'étendre le langage AltaRica et d'adapter les outils. Comme pour la relation de simulation, nous sommes parvenus à définir un théorème de compositionnalité. La démonstration de ce théorème a nécessité de proposer une sémantique de l'extension du langage AltaRica, ce qui, vis-à-vis de l'étude du langage AltaRica, dépasse les objectifs de cette thèse. Nous avons privilégié la définition de cette extension du langage AltaRica à l'élaboration d'un algorithme comparable à celui mentionné pour la relation de simulation.

Concernant les attentes industrielles, nous avons spécifié le raffinement AltaRica pour la sûreté de fonctionnement des systèmes en intégrant la notion de compteur de défaillances et en ajoutant la contrainte qu'un état défaillant ne doit pas être accessible sur un modèle détaillé en moins de défaillances que sur un modèle abstrait.

La relation de simulation que nous utilisons fait le lien entre événements du nœud abstrait et événements du nœud détaillé. Nous proposons deux approches permettant à l'utilisateur soit de spécifier les équivalences entre événements, soit de vérifier les équivalences calculées par MecV. L'outil vérifie directement le raffinement dans la première approche alors que la seconde nécessite que l'utilisateur valide les équivalences générées par l'outil.

Nous avons élaboré une méthodologie de vérification du raffinement, qui définit la chronologie d'utilisation des différentes relations MecV.

Nous avons appliqué cette méthodologie sur plusieurs cas d'études en focalisant nos efforts sur la relation de simulation. Nous avons tout d'abord validé les différentes spécificités du raffinement pour les analyses de sûreté de fonctionnement des systèmes sur l'exemple d'un calculateur de commande de vol. Puis nous avons cherché à appliquer le raffinement à la modélisation d'un système de génération et de distribution électrique.

Perspectives

Les perspectives de poursuite des travaux initiés dans cette thèse sont nombreuses. Comme précédemment, nous détaillons tout d'abord les perspectives qui concernent la recherche fondamentale puis celles liées à l'industrie.

D'un point de vue théorique, nous avons atteint les objectifs que nous nous étions fixés concernant la relation de simulation. Néanmoins, nous pourrions adapter le théorème de préservation suivant : si M simule M' alors toute formule φ de la logique ACTL* satisfaite par M est satisfaite par M'. Ce théorème est notamment mentionné dans [CGJ+03] pour les structures de Kripke.

Dans un second temps, l'étude de la relation de simulation quasi-branchante peut être poursuivie afin d'élaborer un algorithme de calcul de séquences comparables à celui que nous proposons pour la relation de simulation.

Plus généralement, de nombreuses relations de simulation ou de bisimulation existent ou peuvent être définies, comme par exemple la boundary-branching simulation. Chaque relation peut donner lieu à une étude comparable à celle menée pour la simulation. Cela peut s'avérer d'autant plus pertinent si elles reflètent des pratiques industrielles.

D'un point de vue industriel, une étape importante pour appliquer le raffinement basé sur la relation de simulation dans le processus Airbus consisterait à intégrer un outil de vérification, comparable à MecV, dans l'outil interne de modélisation AltaRica. Une première solution serait d'intégrer MecV, ce qui permettrait d'utiliser en l'état les relations que nous avons définies. Une alternative serait d'optimiser MecV pour traiter des modèles industriels. Quel que soit l'outil intégré, l'idée serait de tirer profit de la généricité des relations MecV que nous avons définies et de rendre leur utilisation accessible grâce à une interface graphique ergonomique et intuitive.

Le raffinement permet de disposer de plusieurs modèles d'un même système. Or les outils industriels de modélisation AltaRica intègrent le concept de bibliothèque de composants. Il serait envisageable de créer des familles de modèles dans la bibliothèque de composants : une famille serait constituée de différents modèles plus ou moins détaillés d'un même système, tous les modèles vérifiant une relation de raffinement et pouvant donc être indifféremment connecté. Cela permettrait de capitaliser les modèles construits et les vérifications réalisées.

Annexes

A

ABRÉVIATIONS

AADL	Architecture Analysis and Design Language
ABD	Airbus Directive
A/C	Aircraft
AESA	Agence Européenne de la Sécurité Aérienne
AMC	Acceptable Means of Compliance
AMDE	Analyse des Modes de Défaillance et de leurs Effets
ATP	Automate Temporisé Probabiliste
ARC	AltaRica Checker
ARP	Aerospace Recommended Practice
BCM	Back-Up Control Module
BPS	Back-Up Power Supply
CAT	Catastrophic
CCA	Common Cause Analysis
CEGAR	Counter-Example Guided Abstraction Refinement
CIFRE	Convention Industrielle de Formation par la REcherche
CMA	Common Mode Analysis
CS	Certification Specification
CSMG	Constant Speed Motor Generator
DAL	Design Assurance Level
DCC	Défaillance de Cause Commune
DD	Dependence Diagram
DSF	Dependent System Failure
EASA	European Aviation Safety Agency
ESACS	EnHance Safety Assessment for Complex Systems
EUROCAE	European Organisation of Civil Aviation Equipment
FAA	Federal Aviation Agency
FAR	Federal Aviation Regulation
FC	Failure Condition
FHA	Functional Hazard Assessment
FM	Failure Mode
FMEA	Failure Mode and Effect Analysis
FMES	Failure Mode and Effect Summary

FPM	Failure Propagation Model
FT	Fault Tree
HAZ	Hazardous
Hip-HOPS	Hierarchically Performed Hazard Origin and Propagation Studies
ISAAC	Improvement of Safety Activities on Aeronautical Complex systems
JAA	Joint Aviation Authorities
LaBRI	Laboratoire Bordelais de Recherche en Informatique
LADS	Laboratoire d'Analyse des Dysfonctionnements de Système
LAMIH	Laboratoire d'Automatique, de Mécanique et d'Informatique industrielles et Humaines
MA	Markov Analysis
MAJ	Major
MBDSA	Model Based Design Safety Assessment
MCDL	Modèle du Comportement Défaillant Local
MIN	Minor
MISSA	More Integrated and cost efficient System Safety Assessment
OACI	Organisation de l'Aviation Civile Internationale
OCAS	Outil de Conception et d'Analyse Système
ONERA	Office National d'Études et de Recherches Aérospatiales
PRA	Particular Risk Analysis
PSSA	Preliminary System Safety Assessment
RAT	Ram Air Turbine
QBB	Quasi-Branching Bisimulation
QBS	Quasi-Branching Simulation
SADT	Structure Analysis and Design Technique
SAE	Society of Automotive Engineers
SC	Servocommande
SSA	System Safety Assessment
ST	Système de Transitions
STI	Système de Transitions Interfacé
STIE	Système de Transitions Interfacé Étendu
SysML	System Modeling Language
UML	Unified Modeling Language
WG	Working Group
ZSA	Zonal Safety Analysis

B

BOUNDARY BRANCHING SIMULATION INTERFACÉE

Nous estimons que, dans la relation de simulation quasi-branchante, la contrainte sur l'état initial de la transition observable du nœud abstrait est trop forte. Nous souhaitons nous en affranchir tout en conservant un lien entre cet état du modèle abstrait et l'état initial de la transition de même étiquette du nœud détaillé.

En nous inspirant de la définition 4.4.5 de la simulation quasi-branchante interfacée paramétrée, nous définissons la relation de boundary branching simulation interfacée paramétrée. Cette relation est obtenue en remplaçant la condition $(s_i, s'_1) \in R(RelF, RelEvt)$ par $(f_i, f'_1) \in RelF$.

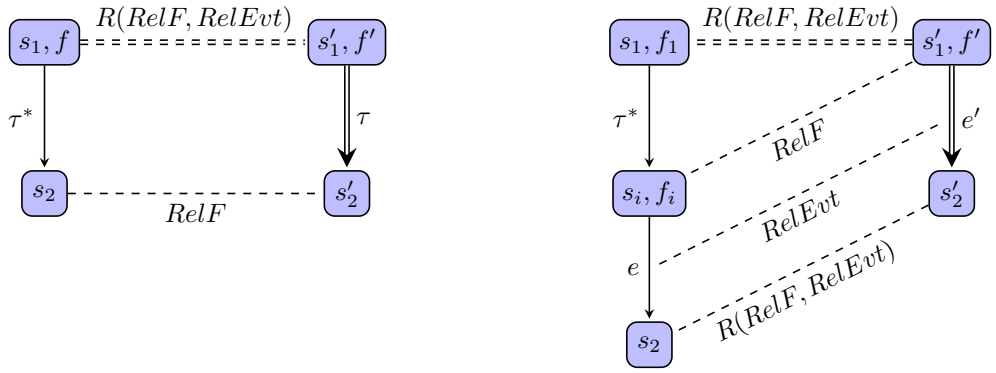


FIG. *B.1: Boundary-Branching simulation interfacée*

La définition, ainsi obtenue, de la boundary branching simulation interfacée paramétrée, est :

Définition B.0.1 (Boundary branching simulation interfacée paramétrée)
Soient $\mathcal{A} = \langle E, F, S, \pi, T \rangle$ et $\mathcal{A}' = \langle E', F', S', \pi', T' \rangle$ deux systèmes de transitions interfacés étendus. Soient $RelF \subseteq F \times F'$ une relation entre variables de flux et $RelEvt \subseteq E \times E'$ une relation entre événements. $R(RelF, RelEvt) \subseteq S \times S'$ est une relation de boundary branching simulation interfacée paramétrée *de $\mathcal{A}$ vers $\mathcal{A}'$ si :*

1. $\forall s' \in S', \exists s \in S, (s, s') \in R(RelF, RelEvt)$;
2. $\forall (s'_1, f', e', s'_2) \in T', \forall s_1 \in S, (s_1, s'_1) \in R(RelF, RelEvt) \Rightarrow$
   - *si* $(s'_1, f', e', s'_2) \in T'_\tau$
     alors $\exists f \in F$, $\exists s_2 \in S$ *t.q.* $(s_1, f, \tau^*, s_2) \in T^*_\tau$, $(f, f') \in RelF$, $(s_2, s'_2) \in R(RelF, RelEvt)$;
   - *si* $(s'_1, f', e', s'_2) \in T'_+$
     alors $\exists f_1, f_i \in F$, $\exists e \in E$, $\exists s_i, s_2 \in S$ *t.q.*
     - $(f_1, f') \in RelF$ *et* $(f_i, f') \in RelF$,
     - $(e, e') \in RelEvt$,
     - $(s_1, f_1, \tau^*, s_i) \in T^*_\tau$,
     - $(s_i, f_i, e, s_2) \in T_+$,
     - $(s_2, s'_2) \in R(RelF, RelEvt)$.

□

Remarque

Sur des systèmes où l'unique événement non observable est ϵ, la relation de boundary branching simulation (BBS), tout comme la simulation quasi-branchante, coïncide avec la relation standard de simulation, précédemment définie. La BBS étend donc la simulation.

C

RELATIONS MECV

C.1 MASQUAGE DES ÉVÉNEMENTS INSTANTANÉS

C.1.1 SANS COMPTEUR DE DÉFAILLANCE

```
NoInternalEvt_A(s) :=
      [e][t](transA(s,e,t) => ~internalEvt_A(e));

ObsState_A(s) := ReachA(s) & NoInternalEvt_A(s);

NonObsState_A(s) := ReachA(s) & ~NoInternalEvt_A(s);

nextObs_A(s,t) += NonObsState_A(s) & ObsState_A(t)
                  & <e>(internalEvt_A(e)
                        & (transA(s,e,t)
                           | <u>(NonObsState_A(u)
                                 & transA(s,e,u)
                                 & nextObs_A(u,t))));

transObs_A(s,e,t) := ObsState_A(s) & ObsState_A(t)
                     & ~internalEvt_A(e)
                     & (transA(s,e,t)
                        | <u>(NonObsState_A(u)
                              & transA(s,e,u)
                              & nextObs_A(u,t)));

transA(s,e,t) := transObs_A(s,e,t);
ReachA(s) := ObsState_A(s);
```

C.1.2 AVEC COMPTEUR DE DÉFAILLANCE

```
NoInternalEvt_A(s,n:[0,max]) :=
      [e][t][m:[0,max]](trans_A(s,n,e,t,m) => ~internalEvt_A(e));

ObsState_A(s,n:[0,max]) := Reach_A(s,n) & NoInternalEvt_A(s,n);

NonObsState_A(s,n:[0,max]) := Reach_A(s,n) & ~NoInternalEvt_A(s,n);

nextObs_A(s,n:[0,max],t,m:[0,max]) +=
      NonObsState_A(s,n) & ObsState_A(t,m)
      & <e>(internalEvt_A(e) & (trans_A(s,n,e,t,m)
```

```
                                  | (<u><uc:[0,max]>
                                     (NonObsState_A(u,uc)
                                      & trans_A(s,n,e,u,uc)
                                      & nextObs_A(u,uc,t,m)))));

transObs_A(s,n:[0,max],e,t,m:[0,max]) :=
      ObsState_A(s,n) & ObsState_A(t,m) & ~internalEvt_A(e)
      & (trans_A(s,n,e,t,m)
         | <u><p:[0,max]>(NonObsState_A(u,p)
                          & trans_A(s,n,e,u,p)
                          & nextObs_A(u,p,t,m)));

trans_A(s,n:[0,max],e,t,m:[0,max]) := transObs_A(s,n,e,t,m);
Reach_A(s,n:[0,max]) := ObsState_A(s,n);
```

C.2 TEST DES ÉTATS ACCESSIBLES

```
AllDetInAbs(x):= x =
      [s'](ReachA'(s') => <s>(ReachA(s) & RelF(s,s')));

notInAbs(s'):= (ReachA'(s') => ~(<s>(ReachA(s) & RelF(s,s'))));

AllAbsInDet(x):=
      x = [s](ReachA(s) => <s'>(ReachA'(s') & RelF(s,s')));

notInDet(s):= (ReachA(s) => ~(<s'>(ReachA'(s') & RelF(s,s'))));
```

C.3 SIMULATION

C.3.1 SANS COMPTEUR

```
sim(s,s') -= RelF(s,s')
                & ([e'][t'](transA'(s',e',t') =>
                   <e><t>(RelEvt(e,e') & transA(s,e,t)
                          & sim(t,t'))));


isSim(x) :=  x = ([s'] (initA'(s') =>
                                <s> (initA(s) & sim(s,s'))));
```

C.3.2 AVEC COMPTEUR

```
sim_cpt(s,n:[0,max],s',n':[0,max]) -=
      RelF(s,s') & ~(n>n')
      & ([e'][t'][m':[0,max]](trans_A'(s',n',e',t',m') =>
                              <e><t><m:[0,max]>
                              (RelEvt(e,e') & trans_A(s,n,e,t,m)
                               & sim_cpt(t,m,t',m'))));


isSim(x) :=
      x = ([s'](initA'(s') => <s>(initA(s) & sim_cpt(s,0,s',0))));
```

C.4 SIMULATION QUASI-BRANCHANTE

C.4.1 CALCUL DES TRANSITIONS T_τ ET T_+

C.4.1.1 Événements non observables

```
TauTrans_A(s,e,t) := (transA(s,e,t) & isTau_A(e));
ObsTrans_A(s,e,t) := (transA(s,e,t) & ~isTau_A(e));
```

C.4.1.2 Transitions non observables

```
TauTrans_A(s,e,t) := (transA(s,e,t) & eqA(s,t));
ObsTrans_A(s,e,t) := (transA(s,e,t) & ~eqA(s,t));
```

C.4.2 VÉRIFICATION DE LA SIMULATION QUASI-BRANCHANTE

```
TauStar_A(s,t)+=    s=t
                    | <e>TauTrans_A(s,e,t)
                    | <v>(TauStar_A(s,v) & TauStar_A(v,t));

TauStar_A'(s,t)+=   s=t
                    | <e>TauTrans_A'(s,e,t)
                    | <v>(TauStar_A'(s,v) & TauStar_A'(v,t));

TauStarE_A(s,u,e,t) := (TauStar_A(s,u) & ObsTrans_A(u,e,t));

TauStarE_A'(s,u,e,t) := (TauStar_A'(s,u) & ObsTrans_A'(u,e,t));

A_QBSim_A'(s,s') -=
      ([e'][t'](transA'(s',e',t')=>
                (TauTrans_A'(s',e',t') & <t>(TauStar_A(s,t)
                                             & RelF(s,s')
                                             & A_QBSim_A'(t,t')))
                |(ObsTrans_A'(s',e',t')
                  & <u><e><t>(TauStarE_A(s,u,e,t)
                              & RelF(s,s')
                              & RelEvt(e,e')
                              & A_QBSim_A'(u,s')
                              & A_QBSim_A'(t,t')))));

A_QBSimule_A'(x) :=
      x = ([s'](initA'(s') => <s>(initA(s) & A_QBSim_A'(s,s'))));
```

D

BIBLIOGRAPHIE

[ABB+03] O. Akerlund, P. Bieber, E. Böde, C. Bougnol, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, M. Cimatti, A. Griffault, C. Kehren, A. Lawrence, A. Lüdtke, S. Metge, C. Papadopoulos, R. Passarello, T. Peikenkamp, P. Persson, C. Seguin, L. Trotta, L. Valacca, A. Villafiorita, and G. Zacco. ESACS : an integrated methodology for design and safety analysis of complex systems. In *ESREL*, 2003.

[ABB+06] O. Akerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, A. Griffault, O. Lisagor, A. Lüdtke, S. Metge, C. Papadopoulos, T. Peikenkamp, L. Sagaspe, C. Seguin, H. Trivedi, and L. Valacca. ISAAC, a framework for integrated safety analysis of functionale, geometrical and human aspects. In *ERTS*, Janvier 2006.

[Abr96] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AGPR99a] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 1999. A tribute to Peter Lauer (Soumission).

[AGPR99b] A. Arnold, A. Griffault, G. Point, and A. Rauzy. Manuel méthodologique. Technical report, Groupe AltaRica, 1999. version 1.0.

[AN82] A. Arnold and M. Nivat. Comportements de processus. *Coloque AFCET "Les Mathématiques de l'Informatique"*, pages 35–68, 1982.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communiquants*. MASSON, Paris, 1992.

[BCC+03] M. Bozzano, A. Cavallo, M. Cifaldi, L. Valacca, and A. Villafiorita. Improving safety of complex systems : An industrial case study. In *FM*, September 2003.

[BCKS04] P. Bieber, C. Castel, C. Kehren, and C. Seguin. Analyse des exigences de sûreté d'un système électrique par model-checking. In *Lambda Mu 14*, 2004.

[BDRS06] M. Boiteau, Y. Dutuit, A. Rauzy, and J.P. Signoret. The altarica data-flow language in use : Assessment of production availability of a multistates system. In *Reliability Engineering and System Safety*, volume 91, pages 747–755, 2006.

[BES94] J. Van Benthem, J. Van Eijck, and V. Stebletsova. Modal logic, transition systems and processes. *Journal of Logic and Computation*, 4(5) :811–855, 1994.

[BFFG04] J.Y. Brunel, A. Ferrari, E. Fourgeau, and P. Giusto. How aerospace and transportation design challenges can be addressed from simulation-based virtual prototyping for distributed safety critical automotive applications. In *2nd European Congress ERTS*, 2004.

[Bou90] M. Bouissou. Figaro : un outil de modélisation en fiabilité. In *Revue "Faits marquants" d'EDF/DER*, 1990.

[BP98] R. Banach and M. Poppleton. Retrenchment : an engineering variation on refinement. In *B'98 : Proceedings of the Second Internation B Conference on Recent Advances in the Development and Use of the B Method*, pages 129–147, London, UK, 1998. Springer-Verlag.

[BS06] M. Bouissou and C. Seguin. Comparaison des langages de modélisation altarica et figaro. In *Procs. of Lambda-Mu 15*, octobre 2006.

[BV03] M. Bozzano and A. Villafiorita. Improving system reliability via model checking : the fsap/nusmv-sa safety analysis platform. In *Procs. of the 22nd International Conference on Computer Safety, Reliability and Security*, September 2003.

[CBR06] L. Cauffriez, V. Benard, and D. Renaux. A new formalism for designing and specifying rams parameters for complex distributed control systems : The safe-sadt formalism. In *IEEE Transations on Reliability*, volume 55/3, pages 397–410, 2006.

[CCJ+] R. Cavada, A. Cimatti, C.A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. Nusmv 2.4 user manual.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specification. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. *Journal of the ACM*, 50(5) :752–794, september 2003.

[CGP00] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 2000.

[Cru89] P. Crubillé. *Réalisation de l'outil Mec, spécification fonctionnelle et architecture*. PhD thesis, Université Bordeaux 1, 1989.

[DA04a] J. Deneux and O. Akerlund. A common framework for design and safety analyses using formal methods. In *ESREL*, 2004.

[DA04b] J. Deneux and O. Akerlund. Designing safe, reliable systems using scade. In *ISOLA*, October 2004.

[DJL+08] C. DeJiu, R. Johansson, H. Lönn, Y. Papadopoulos, A. Sandberg, F. Törner, and M. Törngren. Modelling support for design of safety-critical automotive embedded systems. In *Procs. of the 27th International Conference on Computer Safety, Reliability and Security*, 2008.

[DR97] Y. Dutuit and A. Rauzy. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia. *Reliability Engineering and System Safety*, 58 :127–144, 1997.

[EAS08] EASA. *CS-25 Certification Specifications for Airworthiness of Large Aeroplanes, Amendment 5*. European Aviation Safety Agency, 5 September 2008.

[ENT08] J. Elmqvist and S. Nadjm-Tehrani. Formal support for quantitative analysis of residual risks in safety-critical systems. In *HASE '08 : Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 154–164, Washington, DC, USA, 2008. IEEE Computer Society.

[ENTM05] J. Elmqvist, S. Nadjm-Tehrani, and M. Minea. Safety interfaces for component-based systems. In 24^{th} *International Conference on Computer Safety, Reliability and Security(SAFECOMP'05)*, 2005.

[Eri99] C. Ericson. Fault tree analysis - a history. In 17^{th} *International System Safety Conference*, 1999.

[FAA03] FAA. *FAR Part 25 - Airworthiness Standards : Transport Category Airplanes, Amendment 16*. Federal Aviation Agency, 1 May 2003.

[FMNP94] P. Fenelon, J.A. McDermid, M. Nicholson, and D.J. Pumfrey. Towards integrated safety analysis and design. In *ACM Applied Computing Review*, 1994.

[FR07] P. Feiler and A. Rugina. Dependability modeling with the architecture analysis and design language(aadl). Technical report, Software Enginering Institute, Carnegie Mellon, July 2007.

[GFL05] F. Gervais, M. Frappier, and R. Laleau. Vous avez dit raffinement ? Technical report, CEDRIC 829, 2005.

[GH02] H. Garavel and H. Hermanns. On combining functional verification and performance evaluation using cadp. In *Proceedings of the 11th International Symposium of Formal Methods Europe FME'2002*, 2002.

[GLP+98] A. Griffault, S. Lajeunesse, G. Point, A. Rauzy, J.-P. Signoret, and P. Thomas. The AltaRica language. In *European Safety and Reliability International Conference, ESREL'98*, 1998.

[GLP+99] A. Griffault, S. Lajeunesse, G. Point, A. Rauzy, J.-P. Signoret, and P. Thomas. Le langage AltaRica. In *Actes du 11ème congrès $\lambda\mu$*. Hermès, Octobre 1999.

[GV03] A. Griffault and A. Vincent. Vérification de modèles AltaRica. In *MAJESTIC*, 2003.

[GV04] A. Griffault and A. Vincent. The Mec 5 model-checker. In *Computer Aided Verification, CAV*, 2004.

[GW96] Rob J. Van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3) :555–600, 1996.

[HBB+04] H.-J. Holberg, E. Böde, M. Bretschneider, I. Brückner, T. Peikenkamp, and H. Spenke. Model-based safety analysis of a flap control system. In *INCOSE*, 2004.

[Hum08] S. Humbert. *Déclinaison d'exigences de sécurité du système vers le logiciel, assistée par des modèles formels*. PhD thesis, Université Bordeaux 1, 2008.

[JH05] A. Joshi and M.P.E. Heimdal. Model-based safety analysis of simulink models using scade design verifier. In *SAFECOMP*, 2005.

[JWH05] A. Joshi, , M. Whalen, and M.P.E. Heimdal. Model-based safety analysis final report. Technical report, NASA, 2005.

[Keh05] C. Kehren. *Motifs formels d'architecture de systèmes pour la sûreté de fonctionnement*. PhD thesis, École Nationale Supérieure de l'Aéronautique et de l'Espace, 2005.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27 :333–354, 1983.

[LS91] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inf. Comput.*, 94(1) :1–28, 1991.

[MEP+05] M. McKelvin, G. Eirea, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. A formal approach to fault tree synthesis for the analysis of distributed fault tolerant systems. In *Procs. of the 5th ACM International Conference on Embedded Software*, pages 237–246, September 2005.

[Mil80] R. Milner. Calculus of communicating systems. In *LNCS 92*, pages 167–183, London, UK, 1980. Springer-Verlag.

[MM05] A. McIver and C. Morgan. Abstraction and refinement in probabilistic systems. *SIGMETRICS Perform. Eval. Rev.*, 32(4) :41–47, 2005.

[Pag04] C. Pagetti. *Extension temps réel d'AltaRica*. PhD thesis, École Centrale de Nantes - Université de Nantes, 2004.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5^{th} GI-Conference on Theoretical Computer Science, LNCS 104*, pages 167–183, London, UK, 1981. Springer-Verlag.

[PM01] Y. Papadopoulos and M. Maruhn. Model-based synthesis of fault trees from matlab-simulink models. *Dependable Systems and Networks, International Conference on*, 2001.

[PMSH01] Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering and System Safety*, 71 :229–247, 2001.

[Poi00] G. Point. *AltaRica : Contribution á l'unification des méthodes formelles et de la Sûreté de fonctionnement*. PhD thesis, Université Bordeaux 1, 2000.

[PR99a] G. Point and A. Rauzy. AltaRica - Constraint automata as a description language. *European Journal on Automation*, 33, 1999. Special issue on the *Modelling of Reactive Systems*.

[PR99b] G. Point and A. Rauzy. AltaRica - Langage de modélisation par automates à contraintes. In *Actes du Congrés Modélisation des Systèmes Réactifs, MSR'99*, ENS, Cachan, 1999. Hermès.

[QS83] J-P. Queille and J. Sifakis. Fairness and related properties in transition systems - a temporal logic to deal with fairness. *Acta Inf.*, 19 :195–220, 1983.

[Rau93] A. Rauzy. New algorithms for fault trees analysis. *Reliability Engineering and System Safety*, 40 :203–211, 1993.

[Rau01] A. Rauzy. Mathematical foundation of minimal cutsets language. *IEEE Transactions on Reliability*, 50(4) :389–396, December 2001.

[Rau02] A. Rauzy. Modes automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78 :1–12, 2002.

[RMR+07] C. Raksch, R. Van Maanen, D. Rehage, F. Thielecke, and U. B. Carl. Performance degradation analysis of fault-tolerant aircraft systems. In *DGLR/CEAS Congress*, septembre 2007.

[SAE96a] SAE. *ARP 4761 Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment*. SAE, December 1996.

[SAE96b] SAE. *ARP 4754 Certification Considerations for Highly-Integrated or Complex Aircraft Systems*. SAE Systems Intergration Requirements Task Group, June 1996.

[Sag08] L. Sagaspe. *Allocation sûre dans les systèmes aéronautiques : Modélisation, Vérification et Génération*. PhD thesis, Université Bordeaux 1, 2008.

[SBB+99] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit. *Vérification de logiciels : Techniques et outils du model-checking*. Vuibert, 1999.

[SSN07] H. Soueidan, D. J. Sherman, and M. Nikolski. BioRica : a multi model description and simulation system. In *Proceedings of Foundations of Systems Biology in Engineering (FOSBE)*. F Allgöwer and M Reuss, 2007.

[SSS00] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD '00 : Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125. Springer-Verlag, 2000.

[Tho02] P. Thomas. *Contribution à l'approche booléenne de la sûreté de fonctionnement : l'atelier logiciel Aralia Workshop*. PhD thesis, Université Bordeaux 1, 2002.

[Tro99] Elena Troubitsyna. Refining for safety. Technical report, 1999.

[Vil88] A. Villemeur. *Sûreté de fonctionnement des systèmes industriels*. Collection de la Direction des Études et Recherches d'Électricité de France (EDF). Eyrolles, 1988.

[Vin03] A. Vincent. *Conception et réalisation d'un vérificateur de modèles AltaRica*. PhD thesis, Université Bordeaux 1, 2003.

[Yam03] S. Yamane. Formal probabilistic refinement verification of embedded real-time systems. In *Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES'03)*, 2003.