Motifs formels d'architectures de systèmes pour la sûreté de fonctionnement

Christophe Kehren

N° d'ordre : 447 Année 2005

THÈSE

présentée en vue de

l'obtention du titre de

DOCTEUR

de

L'ÉCOLE NATIONALE SUPÉRIEURE

DE L'AÉRONAUTIQUE ET DE L'ESPACE

ÉCOLE DOCTORALE : Informatique et télécommunications

SPÉCIALITÉ : Programmation et systèmes

Christophe KEHREN

Motifs formels d'architectures de systèmes pour la sûreté de fonctionnement

Soutenue le 20 décembre 2005 devant le jury :

Mme	**K.**	**KANOUN**	**Présidente**
MM.	**A.**	**ARNOLD**	
	Y.	**LEDRU**	**Rapporteur**
	A.	**RAUZY**	
	O.	**ROUX**	**Rapporteur**
Mlle	**C.**	**SEGUIN**	**Directrice de thèse**
M.	**J.P.**	**HECKMANN**	**Membre invité**

Remerciements

Cette thèse et son bon déroulement ne seraient rien sans Christel Seguin, ma directrice de thèse. Ses compétences techniques et son pragmatisme tout d'abord, puis sa conception de la recherche et son esprit d'ouverture m'ont permis de réaliser cette thèse dans les meilleures conditions possibles. Merci de m'avoir tant appris, d'avoir toujours été disponible. Merci de m'avoir fait confiance. Merci de m'avoir soutenu. Etre à ses côtés au cours de ces trois années a été pour moi une chance extraordinaire. Mille mercis !

Merci à André Arnold et Antoine Rauzy d'avoir accepté d'encadrer cette thèse. Leurs conseils et leurs commentaires ont été très importants pour la réalisation de ce travail.

Je remercie très chaleureusement Karama Kanoun d'avoir accepté de présider le jury de cette thèse, ainsi que Olivier Roux et Yves Ledru d'en avoir été les rapporteurs. Je remercie également tout particulièrement Jean-Pierre Heckmann, à l'origine du sujet de cette thèse, d'avoir examiné ce travail.

Merci à Pierre "Magic" Bieber pour ses "idées du jour" géniales, ses conseils, sa sagesse, sa gentillesse et sa pédagogie. Son enthousiasme et sa gaieté ont éclairé les jours gris. J'ai été très fier de le côtoyer pendant ces trois années. Je ne sais comment lui exprimer ma gratitude autrement qu'en lui promettant d'agir comme lui avec les étudiants, si un jour l'occasion m'en est donnée.

Je remercie Charles Castel pour son aide précieuse tout au long de cette thèse. Je n'oublierai jamais sa nomenclature exotique des patterns ainsi que notre périple à Bourges.

Merci à Laurent Sagaspe (alias Mapping Manager Designer) pour son soutien, son aide en LaTeX(et oui, c'est grâce à lui que je sais faire ça !) et nos pauses inoubliables. Sa rencontre constitue un des aspects positifs de cette thèse car, outre un thésard doué et un snowboarder confirmé, c'est un véritable pote que j'ai trouvé.

Merci à Claire Pagetti (alias Spaghet) pour le temps passé à lire ce manuscrit et pour avoir redonné un peu de rigueur à ma plume qui a quelques fois tendance à déraper. Merci pour son soutien, ses soirées et son amitié.

Je remercie Virginie Wiels pour sa participation active à la préparation de ma soutenance, ses conseils et sa bonne humeur quotidienne.

Un grand merci à Guy Zanon pour nos discussions sur les réseaux, ses lignes de commande magiques et son humour subtil.

Je tiens également à remercier mes amis thésards (attention, aucune relation d'ordre n'existe dans cet ensemble) : Stéphanie Gaudan et sa cuisine originale, Alexandre Cortier et sa 405 de collection de prestige, Emmanuel Julliard et son short porte-bonheur, Eric Divouron et ses parents Monique et François, Florian Longueteau et son directeur de thèse, Maud Lavieille et sa Bretagne, Hélène Roinard et ses trop rares "ah Christophe !", Romain Bernard et son t-shirt "Mickey plage", Matthieu Mardi et ses chaussettes de foot, François Revest et tout ce qui va avec, Domingo Llacer et ses ragots, et Gilbert Chowi pour tous les bons

moments, pauses, matches de squash, matches de foot et autres défaites partagées.

J'adresse un grand merci à Christiane Payrau pour sa gentillesse et son efficacité. Je remercie Josette Brial et Nöelle Desblanc pour leur aide et leur accueil.

Merci également aux autres membres et amis du DTIM pour leur accueil et en particulier à son directeur Jacques Cazin de m'avoir accepté au sein du département. Je tiens aussi à remercier la Délégation Générale pour l'Armement d'avoir financé ces trois années de thèse.

Merci à mon frère, Philippe, de m'avoir montré la voie et d'avoir toujours été un modèle (formel) pour moi.

Merci à mes grands-parents Suzanne, Yvonne et Roger pour leur soutien sans faille. Je pense également à mon grand-père Gilbert, j'espère qu'il aurait été fier de moi.

Merci à Alexandre (alias Alex Mex), Antoine (alias Nanuge), Jeanne, Michel (alias Le bouineur), Marie-Anne (alias Nanou), Annabelle (alias La nine), Allan (alias Paupiette), Martine, Jean-Claude, Kelly (alias Nicole), Alf, Vortex, Pixie (alias Le pic) et Ulma (alias Grand you). Je salue également mon "Grégounet" Buatois et mon "Sébounet" Giboult, mes amis de longue date.

Merci à Scums, Kelly Buchanan, Blink182, Simple Plan, Something Corporate, Jack's Mannequin, Our Lady Peace, Nada Surf, The Damnwells, Green Day, Bush, Linkin Park, Coldplay et Foo Fighters pour leur soutien intensif durant les derniers mois de cette thèse.

Merci de tout mon cœur à mon Amour, Gabrielle, et à ma petite Virgule de me supporter quotidiennement. Sans elles rien aurait été possible.

Je termine par un grand remerciement à mes parents, Claudine et Michel, pour leur amour, leur support indéfectible et pour avoir toujours trouvé que ce que je faisais était bien. Je leur dédie ce mémoire de thèse.

"Tout le monde savait que c'était impossible. Il est venu un imbécile qui ne le savait pas et qui l'a fait."

Marcel Pagnol, spécialiste en sûreté de fonctionnement à son insu.

Table des matières

Table des figures

Première partie

Introduction générale

Survol de la thèse

L'objectif de cette thèse, initiée par Airbus France, a été de proposer des méthodes et outils assistant la modélisation et l'évaluation de l'architecture de sûreté de fonctionnement de systèmes complexes lors des phases amont de conception des systèmes.

Les sources de complexité dans les systèmes embarqués critiques tels que ceux d'Airbus sont nombreuses et variées. En effet, leur taille et les interactions entre composants sont deux exemples de caractéristiques rendant leurs analyses problématiques. De plus, la dynamique de ces systèmes augmente considérablement la combinatoire des cas à étudier.

Pour vaincre la complexité, deux pistes complémentaires ont été étudiées conjointement.

- Assister la structuration de la modélisation et de l'évaluation. Les architectures sont généralement construites à partir de mécanismes de sûreté (duplication, voteur, ...) couramment utilisés et bien connus des experts de ce domaine. Ces motifs d'architecture peuvent servir de guide dans la structuration de l'architecture d'un système.
- Automatiser le raisonnement sur la dynamique du système en utilisant des moyens de modélisation et de calcul rigoureux.

Le choix de cette démarche nous a conduit à étudier plus particulièrement les points suivants.

Les motifs d'architectures de systèmes pour la sûreté de fonctionnement

Pour capitaliser le savoir faire des experts en matière de structuration des analyses de sûreté de fonctionnement, nous avons choisi d'appliquer l'approche "*pattern*" développée à la fin des années 1970 par l'architecte Christopher Alexander qui souhaitait cataloguer et réutiliser les formes d'architecture qu'il trouvait intéressantes. Cette approche a ensuite été utilisée en génie logiciel avec les "*design patterns*" qui proposent des solutions à des problèmes récurrents, renseignent sur les conséquences et le contexte d'utilisation et proposent des noms génériques aux patterns permettant de définir un vocabulaire précis. Afin d'expliquer plus concrètement le rôle du *pattern* on peut le comparer aux patrons utilisés en couture, qui sont réutilisables et permettent de réaliser des vêtements (dans notre cas on parlera de fonctions) indépendamment du tissu (des composants) et des outils (des langages) utilisés. A notre tour, nous avons souhaité appliquer cette approche au domaine de la sûreté de fonctionnement ce qui, à notre connaissance, n'a pas encore été réalisé.

Notre premier objectif a donc été de caractériser des modèles généraux d'architectures (i.e. une structure globale ainsi que les règles qui gouvernent les interactions entre les composants) qui correspondent à des éléments de sûreté réutilisables. Nous avons appelé ces modèles "motifs d'architectures de systèmes pour la sûreté de fonctionnement" ou "Safety Architecture Patterns" (SAP). Nous avons défini le contenu des SAP de la manière suivante. Ils renferment quatre types d'attributs : tout d'abord un nom générique qui permet de distinguer les SAP ; puis un ensemble d'hypothèses d'utilisation (sur leur environnement, leurs entrées, les modes de défaillances, ...) ; une spécification correspondant à l'architecture de l'élément de sûreté qu'est censé représenter le pattern ; et enfin des exigences garanties qui vont dépendre à la fois des hypothèses et de l'architecture.

Ces attributs doivent permettre de capitaliser le savoir d'experts des différents domaines d'utilisation ainsi qu'une meilleure utilisation/choix des patterns lors de la phase amont de conception d'architectures. Ainsi, classiquement, un pattern correspond à une abstraction d'une architecture réelle. En effet, il doit être général et donc faire abstraction de certaines informations présentes dans l'architecture de départ tout en gardant celles nécessaires à son bon comportement. Il en va de même en ce qui concerne les hypothèses faites sur l'environnement d'utilisation du pattern. Ces hypothèses abstraient les conditions d'utilisation des architectures réelles. Elles doivent être à la fois suffisamment fortes pour garantir que le pattern a le jeu de propriétés attendu, mais elles doivent également être assez générales pour pouvoir remplir la fonction de réutilisation que l'on souhaite attribuer au pattern.

De plus, d'un système à l'autre, il peut exister de subtiles variations sur l'architecture de sûreté de fonctionnement et ainsi un très grand nombre de motifs concrets peuvent apparaître. Cependant, ces motifs spécialisés semblent tous construits à partir des mêmes mécanismes élémentaires. Un catalogue de patterns d'architectures de sûreté élémentaires a également été réalisé pour assister la conception d'architectures plus complexes.

Formalisation des motifs

Après avoir défini les besoins et les solutions appropriées de manière informelle, nous nous sommes intéressés à la manipulation de ces patterns (i.e. instanciation, reconnaissance, validation, etc.) et donc à leur formalisation. En effet, un des objectifs est de pouvoir remplacer une architecture "vraie" par le pattern associé. Nous avons donc étudié dans quelles conditions cette substitution est réalisable (mise en correspondance de l'architecture vraie et du SAP). Un second objectif est l'étude de leur combinaison. Les systèmes complexes seront vraisemblablement constitués de plusieurs SAP, il est donc nécessaire d'étudier si la combinaison de plusieurs patterns est possible et à quoi elle correspond afin de savoir si les exigences finales seront bien tenues et si les propriétés initiales seront bien préservées.

Le comportement des systèmes en présence de pannes peut être formalisé à l'aide de langages formels comme AltaRica, développé au Laboratoire Bordelais de Recherche en Informatique. En effet, le langage AltaRica est dédié à la modélisation orientée sûreté de fonctionnement et les modèles AltaRica peuvent être compilés vers des formalismes connus (arbres de défaillance, automates, réseaux de Petri stochastiques, etc.) et analysés à l'aide d'outils existants (traitement d'arbres, preuve de propriétés, etc.).

AltaRica nous permet de réaliser un modèle opérationnel (i.e. simulable) d'un système. Dans notre cas, les motifs correspondent à des abstractions d'architectures concrètes et donc requièrent une modélisation plus déclarative, i.e. à l'aide de propriétés. Ces propriétés étant en général dynamiques, nous avons choisi la Logique Temporelle Linéaire pour les modéliser. La description des patterns est donc constituée d'une partie AltaRica et d'une partie LTL. Ce type de modélisation mixte n'existe pas à notre connaissance et possède plusieurs intérêts, notamment lors de la conception en phase amont d'architectures de systèmes où il est possible de considérer qu'une partie du système est clairement définie et qu'une autre partie ne l'est pas complètement ou en cours d'étude. Cette dernière correspond à un pattern de sûreté. Nous avons donc défini une nouvelle notation mixant de l'opérationnel (AltaRica pour les composants bien définis) et du dénotationnel (une variante de LTL pour les composants spécifiés à partir de propriétés). Cette extension de la syntaxe AltaRica permet de spécifier et contraindre le comportement des composants par des propriétés temporelles à la fois sur les états et sur les événements.

Le problème théorique suivant se pose alors : il faut caractériser le comportement d'un système défini de manière mixte (opérationnel/dénotationnel). Pour cela nous avons plongé la sémantique d'AltaRica dans les Structures de Kripke Etiquetée. En effet, les SKE définissent la sémantique de State-Event LTL, notre logique de spécification de propriétés et d'exigences. De plus, afin obtenir des structures manipulables informatiquement, nous avons traduit les propriétés de SE-LTL en automates de Büchi. Le produit de ces deux structures nous permet d'obtenir un automate de Büchi caractérisant le comportement satisfaisant les contraintes fixées à la fois par la partie AltaRica et par la partie SE-LTL.

Application de la démarche à des systèmes embarqués

Les liens étroits avec Airbus France durant cette thèse nous ont permis de valider globalement notre approche sur des modèles de systèmes avion. Ainsi, les systèmes de génération hydraulique (permettant d'alimenter les différentes surfaces mobiles, freins etc.) et électrique (qui alimente aussi bien des consommateurs critiques comme les équipements de navigation que les galleys) d'un avion de type A320 ont été modélisés en AltaRica. Ces systèmes sont relativement complexes. Le modèle du système hydraulique contient plus de vingt composants et une trentaine d'événements de défaillance ont été modélisés. Le modèle du système électrique contient lui environ 70 composants pour une cinquantaine de défaillances. Leur complexité ainsi que leur rôle capital au sein de l'avion (leur perte totale est classée catastrophique) font d'eux des systèmes critiques, donc très intéressants pour nos analyses.

Structure du document

Cette thèse est organisée de la manière suivante :

– Le premier chapitre présente le contexte de l'étude avec notamment les spécificités des systèmes étudiés, une présentation des diverses techniques utilisées dans le domaine de la sûreté de fonctionnement ainsi que la problématique soulevée.

- Le second chapitre s'intéresse plus particulièrement à la réutilisation dans le domaine du génie logiciel qui fut l'inspiration des motifs d'achitectures de sûreté que nous avons développés.
- Le troisième chapitre aborde les outils de modélisation formelle nous permettant de réaliser des modèles d'architectures ainsi que des logiques nous servant à spécifier formellement les propriétés de ces architectures. Il introduit également le langage de modélisation que nous avons manipulé, le langage AltaRica.
- Le quatrième chapitre traite de la mise en œuvre de notre approche, introduit la notion de motif d'architecture de sûreté et sa formalisation.
- Le cinquième chapitre développe les opérations réalisables sur les motifs et propose des illustrations de ces concepts à l'aide de différents outils informatiques.
- Enfin, le sixième chapitre présente un cas d'étude concret basé sur l'architecture d'un système électrique de type Airbus.

Deuxième partie

Introduction au domaine d'étude

Chapitre 1

Éléments de sûreté de fonctionnement

1.1 Introduction

Le 11 octobre 2004, au dessus de l'Australie, un de Havilland DHC-8 effectuant un vol commercial entre Horn Island et Cairns a subi un incident sérieux lié à un défaut de conception de l'architecture du système électrique. En vol, les pilotes furent confrontés à un départ de feux qui enfuma le poste de pilotage suivi d'un bruit sourd provenant de l'arrière du tableau de bord d'un des pilotes. Presque simultanément, les voyants d'alarme des convertisseurs de courant continu en alternatif primaire et de secours s'allumèrent indiquant leur perte et donc la perte des instruments électroniques de navigation de ce pilote. Après avoir suivi les procédures standards, le second pilote put poser l'appareil sans encombre. L'enquête de l'ATSB, équivalent du Bureau Enquête Accidents français, conclut que la défaillance de l'inverter primaire fit sauter bon nombre de disjoncteurs dont celui de l'inverter de secours qui empêcha la reconfiguration nécessaire pour récupérer l'alimentation électrique des instruments du pilote. Ce point de panne unique, lié a la non ségrégation des deux inverters, montre indéniablement une mauvaise analyse des modes de défaillances, de leur effet et de leur criticité (AMDEC) réalisée par l'avionneur.

Cette histoire récente montre bien l'importance des analyses de sûreté lors de la phase de conception d'architectures de systèmes critiques et embarqués, et donne une bonne vision du contexte des études que nous réalisons. Bien que la majorité des méthodes d'analyse de sûreté soient désormais informatisées, certaines, commes les AMDEC [JM72, 92667], sont encore aujourd'hui réalisées sans l'aide d'outils informatiques. Or, l'évolution technologique constante de notre société implique une complexité croissante de la conception et donc de l'analyse des systèmes de manière générale. Simultanément, l'évolution sociétale pardonne de moins en moins les incidents, et encore moins les accidents, qui se produisent sur ces systèmes complexes utilisés dans des domaines de plus en plus vastes et avec des fréquences de plus en plus importantes. Les systèmes embarqués critiques que nous étudions ont des exigences fonctionnelles comme les autres systèmes, ce qui les différencie sont leurs exigences plus sévères de qualité de service comme la prédictabilité (prévisibilité), la sûreté et la fiabilité dans des environnements parfois hostiles. Ces exigences doivent être tenues car la vie de millions de personnes en dépend, directement ou indirectement. Nous verrons dans ce chapitre quelles techniques sont ou peuvent être utilisées pour mener à bien des analyses de sûreté sur de tels systèmes. Mais au préalable, présentons les spécificités des systèmes analysés.

1.2 Spécificités des systèmes étudiés

Dans la suite de cette thèse, nous parlerons souvent de *système*, mais qu'entendons nous par *système* ? Nous considérerons la définition de [Vil88] pour qui un *système* est un ensemble déterminé d'éléments discrets (ou composants) interconnectés ou en interaction.

1.2.1 Des systèmes embarqués

On qualifiera d'*embarqué* un système complètement encapsulé dans le dispositif qu'il contrôle. En effet, les systèmes embarqués sont intégrés dans un système plus large pour lequel ils réalisent des fonctions de calcul, de surveillance ou de contrôle. Ils se distinguent également des autres systèmes par leurs exigences sévères en matière de qualité de service (QoS). Les domaines d'utilisation de ce type de systèmes sont de plus en plus vastes et envahissent de plus en plus notre vie quotidienne. Les domaines les plus connus sont bien sûr l'aéronautique et l'espace mais également l'automobile ou encore la téléphonie. Ces systèmes sont généralement des systèmes temps réel, i.e. ils contrôlent ou pilotent un procédé physique à une vitesse adaptée à l'évolution du procédé.

1.2.2 Des systèmes critiques

Les systèmes embarqués sont parfois qualifiés de *critiques* dans le sens où leur dysfonctionnement a un impact important sur leur environnement matériel, humain ou même économique. Les modes de défaillances liés à la perte de tels systèmes doivent avoir des taux d'occurrence si faibles qu'ils ne sont pas supposés se produire au cours du cycle de vie de l'avion. Ces modes de défaillance sont donc considérés et qualifiés de "catastrophiques" pour les autorités aéronautiques qui leur imposent des probabilités d'occurrence inférieures à 10^{-9} par heure de vol.

1.3 Notions générales de SdF

La sûreté de fonctionnement (SdF) des systèmes est un vaste domaine. On la définit souvent la comme la science des défaillances. Son objectif est d'étudier leurs mécanismes d'occurrence et leur propagation afin d'évaluer leur risque et de les maîtriser. Dans cette section nous introduisons les principaux concepts de la SdF et présentons les méthodes d'évaluation traditionnellement utilisées.

1.3.1 Attribut de SdF étudié

Dans cette thèse, nous nous intéresserons à la sûreté des systèmes. Il est important de ne pas confondre *sûreté* et *fiabilité*. La fiabilité d'un composant nous informe sur son aptitude à accomplir une fonction dans des conditions données, pendant un intervalle de temps donné, c'est tout. Aucune précision sur d'éventuels états de fonctionnement dégradés n'est apportée : on ne sait pas si le système défaillira de manière sûre, c'est à dire qu'il n'aboutira pas à un incident ou à un accident. C'est ainsi qu'un composant sûr défaillira plus fréquemment qu'un autre plus fiable mais il garantira un état de faute non dangereux du système. On peut même opposer ces deux notions puisque la fiabilité d'un système peut parfois être obtenue au détriment de sa sûreté. La sûreté est un problème systémique alors que la fiablilité est

une considération de plus bas niveau. Bien sûr, on peut améliorer la sûreté des composants d'un système afin d'améliorer la sûreté du système global, mais en fin de compte c'est sur les interactions des composants au sein du système qu'il faut s'attarder pour réduire les risques.

1.3.2 Entraves à la SdF

Les définitions qui suivent sont extraites de l'ARP4754[1] [SAE96] standard pour la certification dans le domaine aéronautique. Nous parlerons de *défaillance* pour indiquer le fait que qu'un élément ou un ensemble d'éléments ne sont plus aptes à accomplir leur fonction. Nous parlerons également de *panne* ou de *faute* pour caractériser un état non satisfaisant d'un élément, qui résulte toujours d'une défaillance, le rendant inapte à réaliser sa fonction. Parmi ces circonstances indésirables qui sont des entraves à la SdF, on distingue les défaillances qui mènent à un état de panne d'un composant, des erreurs qui peuvent se produire tout au long du cycle de vie et qui ne font que perturber le système. De plus, on fait également la distinction entre les fautes dites systématiques dues à des erreurs de conception ou à des exigences inappropriées, des fautes stochastiques liées à l'usure des composants par exemple. Enfin, nous parlerons de *mode de défaillance* pour faire référence à l'effet par lequel une défaillance est observée et de *condition de défaillance* pour préciser le contexte pouvant causer ou contribuer à une défaillance. Une bonne distinction de ces fautes, et une bonne connaissance du vocabulaire de ce domaine en général, est importante car les moyens de contrôle qu'on leur associera vont en dépendre.

1.3.3 Moyens pour la SdF

Le développement de systèmes sûrs nécessite l'utilisation de méthodes d'analyse des dangers. D'un point de vue purement informel, il existe un certain nombre de documents, issus de brainstorming, dont le but est d'identifier les dangers, leurs risques associés, le temps de tolérance de ces dangers avant qu'ils ne deviennent des incidents, leurs causes d'apparition ainsi que la manière de les traiter. Sur ce dernier point, de nombreuses possibilités s'offrent au concepteur :

- Rendre la réalisation du danger impossible
- Apprendre à l'utilisateur comment éviter l'apparition de ce danger
- Alerter l'utilisateur du danger afin qu'il réagisse
- Automatiser la résolution du danger
- Faire de la vérification interne de manière à prévoir un futur état dangereux
- Ajouter des équipements de sûreté
- Isoler les systèmes potentiellement dangereux ou ne les rendre accessibles qu'aux personnes conscientes du danger
- Bien informer l'utilisateur du danger en l'indiquant clairement

Pour être considéré comme "sûr", aucune défaillance d'un composant ni aucune condition de panne ne doit mener à un état de panne du système. Voici deux exemples élémentaires d'architectures sûres fournissant les moyens de contrôler les risques éventuels : les canaux simples (CS) et les canaux doubles (CD).

[1] Une Aerospace Recommended Pratice est une recommandation requise par la réglementation.

Dans le CS représenté sur la figure 1.1, toute défaillance d'un composant entraîne la perte du système entier car il ne possède qu'un unique canal d'information (i.e. les composants sont en série). Cependant, cette architecture peut être rendue sûre en appliquant certains contrôles, aux bons endroits (e.g. watchdog, ...).

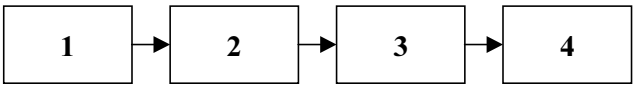


FIG. 1.1 – Canal simple

Dans un CD représenté sur la figure 1.2, le contrôle du système et les mesures effectuées dessus sont dissociés, chacun empruntant un canal indépendant. Ainsi, une défaillance du canal de contrôle sera immédiatement détectée par celui de surveillance qui alertera l'utilisateur. Une défaillance du canal de surveillance n'aura pas (immédiatement) de conséquence donc le contrôle continuera à jouer son rôle puisqu'ils sont totalement indépendants. Reste alors à vérifier qu'il n'existe pas de panne commune aux deux canaux.

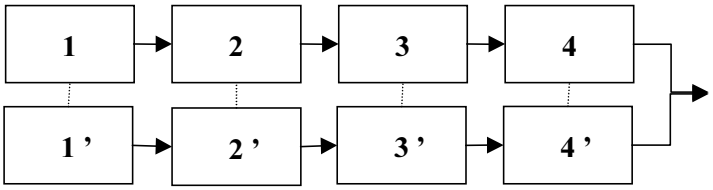


FIG. 1.2 – Canal double

Voici quelques méthodes classiques d'analyse de sûreté de fonctionnement, qualitatives et quantitatives, sur de telles architectures de systèmes.

1.4 Analyses de sûreté des systèmes dynamiques

L'analyse de sûreté d'un système peut être réalisée à l'aide de nombreux outils. La simulation et la méthode d'analyse par arbres de défaillance (AdD) sont les plus répandues pour les études de sûreté et de fiabilité car simples à mettre en œuvre. Voici un peu plus de détails sur les méthodes d'analyses de systèmes dynamiques les plus utilisées.

1.4.1 Simulation de Monte Carlo

La simulation de Monte Carlo [DS64, Bat86] est une technique utilisée pour estimer la probabilité de résultats en répétant un grand nombre de fois une expérience à l'aide de la simulation et en utilisant des variables aléatoires. On entend par simulation toute méthode qui a pour but d'imiter un système réel. On utilise généralement la simulation de Monte Carlo lorsque d'autres analyses sont mathématiquement trop complexes ou trop difficiles à reproduire.

Exemple Voici un exemple standard de l'utilisation de la méthode de Monte Carlo. Au 18eme siècle, le Comte de Buffon proposa le problème suivant qui porte désormais le nom de "problème de l'aiguille de Buffon". Supposons que nous disposions d'une table où serait déssiné un certain nombre de lignes parallèles séparées une à une de d cm. Supposons également que nous disposions d'une aiguille de d cm de long. Le fait de lancer cet aiguille sur la table nous permet d'atteindre deux états possibles : soit l'aiguille chevauche ou touche une ligne, soit l'aiguille n'en chevauche ou touche aucune. Le principe est maintenant de répéter l'opération N fois et de compter le nombre de fois où l'aiguille chevauche une ligne, que l'on notera C. Si N est assez grand, alors $2N/C$ est une bonne approximation de π. En effet, un calcul assez simple montre que la probabilité que l'aiguille chevauche une ligne est de $2/\pi$ qui est aussi égal, après un grand nombre d'expériences, à C/N. C'est cette méthode (avec 3408 lancés d'aiguilles) qui fut utilisée en 1901 pour la fameuse approximation à 355/113 de π .

Cette technique présente plusieurs avantages. D'une part, elle accepte tous les types de distribution. D'autre part, elle est plus efficace que les autres méthodes analytiques lorsque le système possède un grand nombre de redondances. Son principal inconvénient est la durée de traitement qui est très importante pour des systèmes fiables.

1.4.2 Approche Markovienne

Les modèles de Markov, utilisés en sûreté de fonctionnement depuis les années cinquante, sont des automates probabilistes à états finis. Ils sont largement utilisés pour la modélisation à des fins de sûreté de fonctionnement ou de fiabilité. Ils se présentent sous la forme d'un ensemble d'états représentés par des cercles, et d'arcs orientés et valués (cf. figure 1.3). L'orientation de ces arcs indique les possibilités d'évolution des états du système et leur valeur précise les probabilités de transition ($\rho_{i,j}$) d'un état à un autre (de l'état i à l'état j). Ces probabilités de transition rendent compte des comportements physiques du système et par conséquent correspondent soit à un taux de défaillance (λ) d'un composant, soit à un taux de réparation (μ). Ces modèles se basent sur l'hypothèse de Markov qui peut se résumer par "l'évolution du système ne dépend que de l'état courant". On dit que le processus est sans mémoire. Cette hypothèse est très forte puisqu'elle impose que chaque état contienne suffisamment d'informations pour que le système évolue correctement.

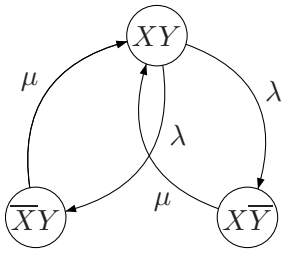


FIG. 1.3 – Cas de deux composants identiques ($\overline{x}$ correspond à la défaillance de x)

Exemple Considérons le modèle d'un avion en vol et intéressons nous à sa trajectoire. Le nombre de paramètres à prendre en compte pour réaliser ce modèle est trop important pour

pouvoir satisfaire l'hypothèse de *Markov*. On peut néanmoins simplifier ce modèle en considérant que la trajectoire de l'avion ne dépendra que de la position de ses surfaces mobiles et de sa vitesse. Alors, s'il n'y a pas trop de vent, un tel modèle peut être considéré comme Markovien.

Pour la classe des systèmes Markoviens, le graphe est traduit en un système d'équations différentielles, dit de Chapman-Kolmogorov, du premier ordre à coefficients constants. Il est possible d'extraire de ce système d'équations différentielles les trois grandeurs instantanées fondamentales de la SdF (fiabilité, disponibilité et maintenabilité) ainsi que les grandeurs moyennes (Mean operating Time Between Failures, Mean operating Time To Repair, etc.). L'approche Markovienne permet de comparer des architectures et des modes opératoires. Un graphe de Markov peut être construit de manière simple et précise en suivant une procédure prédéfinie et invariante. Sa simplicité et son efficacité font de la modélisation Markovienne une des méthodes les plus appréciées pour les évaluations de sûreté et de disponibilité.

1.4.3 Arbres de défaillance

La méthode d'analyse par arbre de défaillances (AdD), également appelé arbre des causes, créée au début des années soixante par Watson de Bell Telephone puis améliorée par Haasl de Boeing, est utilisée dans de nombreux domaines de l'industrie pour les études de sûreté et de fiabilité. Comme son nom l'indique, un AdD est une structure arborescente dont le sommet représente un événement redouté du système, aussi appelé événement non souhaité (ENS). Il se construit de manière ascendante par la combinaison de portes logiques et d'événements élémentaires qui mènent au sommet. Du point de vue de la théorie des graphes, un AdD peut être défini comme un 1-graphe, quasi fortement connexe, sans boucle et sans circuit. Cette représentation sous forme de graphe permet l'utilisation des nombreux algorithmes de traitement ou de construction qui existent. Le but de cette construction est d'en extraire une formule booléenne, dite fonction de structure, permettant d'expliciter l'ensemble des combinaisons minimales d'événements qui mènent au sommet de l'arbre (appellées implicants premiers) et ainsi d'identifier les composants ou les parties du système qui sont sensibles ou susceptibles de causer la perte du système. L'AdD permet aussi de calculer les valeurs statistiques classiques de sûreté de fonctionnement comme la fiabilité, l'indisponibilité et tous les temps moyens caractéristiques du modèle (Mean Time Between Failures, etc.). Les calculs de disponibilité d'un système peuvent se faire par l'évaluation des coupes minimales puis par l'application d'une technique de disjonction de ces coupes ou bien en une seule étape si l'on traduit la fonction de structure de l'arbre de défaillance sous forme de diagrammes de décision binaire (BDD). Les BDD sont des structures de données qui permettent un codage optimal de fonctions booléennes et peuvent se construire à partir de la décomposition de Shannon. On distingue deux types d'AdD : les arbres classiques statiques [Cha00, Vil88] et, afin de combler les lacunes de ces derniers, notamment en ce qui concerne l'absence de prise en compte de l'ordre d'apparition des événements et des dépendances fonctionnelles, les arbres dynamiques [CM02, MDCS98].

Comme nous venons de la voir, l'AdD est une représentation statique du système. Plusieures variantes existent, comme les arbres dits cohérents ou avec restriction. L'arbre est constitué d'événements de base (B), d'événements dits "maison" (H) qui ne sont pas des événements dysfonctionnels et qui permettent de représenter différentes configurations du système étudié, et de portes logiques (G). Ces portes logiques permettent de modéliser, entre autre, des

conjonctions ou des disjonctions d'événements, des votes k/n mais également des délais ou des portes matricielles. L'ensemble de ces constituants sont des blocs de formes normalisées. Mathématiquement, on peut représenter un arbre de défaillances statique par un ensemble d'équations booléennes du type :

$$G_i = f(G_p, H_s, B_j) \text{ avec } i, p \in \{1, ..., P\},\ s \in \{1, ..., S\},\ j \in \{1, ..., J\} \text{ et}$$
P le nombre de portes, S le nombre d'événements "maison", J le nombre d'événements de base. Notons qu'il n'existe pas de relation entre i et p.

L'approche par AdD a l'avantage d'être facilement compréhensible par des personnes autres que le créateur même de l'arbre mais de par sa construction, l'arbre classique possède un certain nombre de limites trop fortes dans le cadre des analyses que nous souhaitons mener. Ainsi, il est la représentation d'une formule booléenne sous forme de graphe et ne tient donc pas compte de l'ordre d'apparition des événements et des dépendances fonctionnelles, caractéristiques pourtant très importantes dans les systèmes physiques. Afin de combler ces lacunes, de nouvelles approches intégrant l'aspect dynamique des systèmes ont été étudiées. Celle proposée par Dugan [MDCS98] de l'Université de Virginie (Etats-Unis) propose de résoudre ce problème en décomposant l'arbre initial en une partie statique et une partie dynamique. La partie statique de l'arbre est alors traitée et encodée de manière classique à l'aide de structures de données efficaces, appelées diagrammes de décision binaire (BDD) [Bry86], la partie dynamique étant quand à elle traitée à l'aide d'une autre méthode appelée méthode de Markov [How60]. Malheureusement nous perdons toute possibilité d'analyse qualitative puisque la notion de coupe dynamique n'a pas encore été définie. D'autres outils, appelés "générateurs de séquences" permettent de trouver des coupes munies d'un ordre total. Cependant, les algorithmes de résolution des outils que nous avons eus à disposition font apparaître des séquences d'événements ne menant pas à l'événement redouté de manière permanente. En effet, certaines configurations d'un système peuvent autoriser l'occurrence de cet événement de manière temporaire, sans causer pour autant la perte définitive du système.
Plusieures approches permettent de prendre en compte l'aspect dynamique des modèles étudiés [CM02, MDCS98, Bou02]. Dugan propose elle-même plusieures techniques. L'une d'elles consiste à décomposer l'aspect dynamique de l'AdD en contraintes logiques, i.e. des contraintes qui traduisent comment les événements se combinent en utilisant les opérateurs $\wedge$ et $\vee$, et contraintes temporelles traduisant l'ordre d'occurrence des événements. Elle obtient alors l'algorithme de génération de séquences suivant :

Étape 1 - Remplacement des portes dynamiques par les portes statiques qui correspondent à leurs contraintes logiques,

Étape 2 - Génération des coupes minimales sur l'AdD statique obtenu,

Étape 3 - Raffinement des coupes minimales en prenant compte les contraintes temporelles (on peut se contenter de raffiner le sous-ensemble des coupes initialement issues de portes dynamiques).

Cepin et Mavko ont proposé une approche d'arbre dynamique basée sur le principe de la "composition" de sous arbres correspondant à chacunes des configurations du système (cf. figure 1.4). L'évolution de manière discrète dans le temps des événements maison est encodée dans une matrice. Cette matrice permet, lors du traitement de l'arbre, d'activer ou de désactiver certaines branches de l'arbre et donc de se placer dans une configuration précise à un instant t donné. Ils donnent la définition suivante à leurs arbres dynamiques :

$G_i = f(G_p, H_{st}, B_j)$ avec H_{st} la valeur de l'événement maison H_s au temps t.

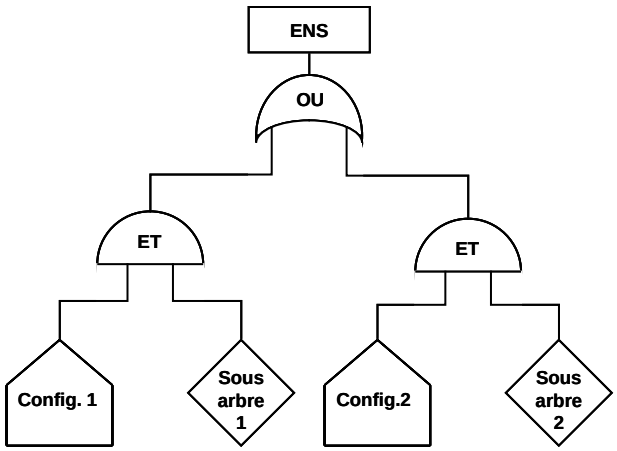


FIG. 1.4 – Arbre de défaillance dynamique

1.5 Conclusion

Ce chapitre nous a permis de faire un rapide tour d'horizon des méthodes classiques utilisées pour les analyses de sûreté de fonctionnement. Comme nous venons de le voir, ces techniques permettent de réaliser des analyses qualitatives et quantitatives sur des modèles statiques et dynamiques. Malheureusement, chaque type d'analyse nécessite souvent une modélisation différente du système étudié. Le langage ALTARICA, que nous présentons au chapitre 3, est un langage formel de modélisation dont le but est de pouvoir servir de base à divers outils performants de l'atelier ALTARICA. Ces outils permettent de réaliser toutes les analyses de sûreté que nous avons présentées précédemment à partir du seul modèle ALTARICA d'un système. Ces capacités intéressantes nous ont poussé à utiliser ce langage pour la modélisation de nos systèmes.

Chapitre 2

Réutilisation en génie logiciel

2.1 Introduction

Les différentes techniques que nous venons de voir s'appliquent sur des modèles de systèmes dont l'architecture reflète le niveau de sûreté que l'on souhaite imposer au système. Les années 90 ont vu le développement de techniques orientées sur la description de telles architectures dans le domaine du logiciel [Gro00]. Ces techniques ont pour but d'améliorer la compréhension ainsi que la conception de systèmes complexes. Elles sont basées sur l'utilisation de langages appelés *Architecture Description Language* que nous allons maintenant détailler. Nous verrons également une autre approche permettant la réutilisation de solutions, toujours dans le domaine du logiciel, apparue bien avant les ADL est basée sur la notion de *pattern*.

2.2 Les langages d'architectures

Les langages de description formelle d'architectures [MT97] de systèmes, ou ADL, représentent un support pour la définition et la conception de structures logicielles de haut niveau. Il existe deux types d'ADL : les ADL orientés vers le logiciel et ceux orientés vers le matériel (aussi appelés *Machine Description Language*). Leur but est de permettre une meilleure compréhension et la réutilisation d'architectures en vue de faciliter les analyses des systèmes (et également de réduire les coûts de développement). Le domaine d'application des ADL est vaste. Ils permettent, entre autre, la spécification du comportement de composants (e.g. unité de traitement ou de stockage de données) ou de protocoles. Parmi les ADL les plus utilisés on trouve Wright [All97, AG97], ACME [GMW00, GMW97, Uni98], Rapide [LKA$^+$95, LV95] ou encore MetaH [BEJV96, Ves96] développé par Honeywell et dont le domaine d'application est l'avionique temps réel. Ce dernier mélange composants logiciels (e.g. programmes, processus) et matériels (e.g. mémoires, processeurs). Les outils disponibles autour de MetaH autorisent des analyses d'ordonnancement, de fiabilité et de sûreté de systèmes. Comme cela a été précisé au chapitre 1.2, un système est un ensemble de composants en interaction, donc connectés les uns aux autres. Nous allons maintenant définir ces différents objets au sens des ADL.

2.2.1 Composant

Le niveau de détail attribué à un composant est variable. Il va de la simple fonction à l'application complète. Un composant est caractérisé par cinq attributs, définis par Medvidovic et Taylor dans [MT97] : *interface*, *type*, *sémantique*, *contrainte* et *évolution* dont voici une brève description.

- Une *interface* permet de définir les interactions entre un composant et son environnement. Elle décrit les propriétés requises par le composant et celles qu'il fournit lui-même.
- Les *types* permettent de définir un ensemble d'instances de composants ayant des propriétés communes.
- Un modèle définissant la *sémantique* du composant permet de faire du test, de la simulation ou de vérifier des *contraintes* imposées sur les composants.
- Les *contraintes* sont des propriétés ou des hypothèses d'un composant. Ces contraintes peuvent être déclarées à l'aide de la notation de l'ADL ou bien en utilisant un langage approprié.
- Enfin, l'*évolution* des composants doit être prise en compte dans l'ADL.

2.2.2 Connecteur

Un connecteur est un élément de l'architecture permettant de faire communiquer les composants entre eux. Tout comme les composants, leur complexité est variable et peut aller du simple appel de fonction à un protocole de communication complexe. La définition des connecteurs est sensiblement similaire à celle des composants. En effet, l'interface du connecteur définit le rôle des différents participants à l'interaction. Elle ne décrit pas de services fonctionnels, mais les mécanismes de connexion entre composants. Ensuite, la description de l'implantation du connecteur définit le protocole associé à l'interaction qu'il est nécessaire de mettre en œuvre. Toujours dans le but de supporter l'analyse des interactions entre composants et la vérification des contraintes imposées sur les composants et les connecteurs, la sémantique associée à ces derniers doit être spécifiée. En règle générale, les ADL utilisent le même modèle sémantique pour les composants et les connecteurs. Rapide, au travers des posets (Partially Ordered Set of Events), et Wright, au travers du langage CSP (Communicating Sequential Processes) [Hoa03], permettent d'exprimer la sémantique des connecteurs.

2.2.3 Configuration

Une configuration traduit la structure de l'architecture d'une application en un graphe de composants et de connecteurs. Elle permet de valider l'assemblage des composants et connecteurs en terme d'interfaces fournies et requises, ainsi que de valider le comportement global d'une architecture en en évaluant les performances ou les risques d'interblocage. La définition de l'architecture doit permettre de comprendre au mieux une application ou un système, d'en fournir une abstraction. L'ADL doit en fournir une vue simple et compréhensible.

2.2.4 Rapide : un ADL formel

Rapide est langage de description d'architectures qui a été développé par l'Université de Stanford pour le prototypage d'architectures de systèmes distribués. Rappelons que le but des

ADL formels comme Rapide est de capturer le comportement d'applications afin d'automatiser leur vérification. Rapide s'appuie sur la simulation pour vérifier les architectures logicielles.

La communication entre composants au sein d'une architecture se produit par échange de messages, par l'intermédiaire d'événements. La notion de patron d'événements introduite dans Rapide a pour but de spécifier les contraintes d'un composant en caractérisant les événements à l'origine des interactions entre les composants. Trois opérateurs permettent d'exprimer la dépendance au sein d'un patron : la dépendance causale ($\rightarrow$), l'indépendance ($\|$) et la simultanéité (*and*). La déclaration des composants se fait à l'aide d'interfaces qui renseignent sur leur comportement ainsi que sur les services requis et fournis par les composants.

Comme précisé précédemment, la configuration définit les composants utilisés et leurs règles d'interconnexion. Une règle est définie par deux patrons : tout d'abord un patron à vérifier puis un second à déclencher si le premier est vérifié. Notons qu'une architecture peut également contenir des contraintes sur les connexions, toujours en utilisant des patrons d'événements. La spécification d'une connexion se fait en mettant en relation deux patrons d'événements : le patron relatif aux services requis et celui relatif aux services fournis. Trois types d'opérateurs de connexion existent. L'opérateur *To* met en relation deux composants, un emetteur et un récepteur. L'opérateur $\|>$ connecte deux patrons quelconques. Enfin l'opérateur $=>$ ajoute la notion d'ordre d'évaluation à l'opérateur précédent.

Exemple Voici une règle d'interconnexion entre un composant `c` (e.g. une pompe hydraulique) et sa ressource `r` (e.g. de l'électricité).

```
with Component, Ressource;

 ?c : Component ;
 !r : Ressource ;
 ?m : Message ;

 ?c.Send (?m) => !r.Receive (?m) ;
```

Cette règle précise qu'un composant peut demander l'obtention de ressource par l'envoi du message `m`. L'utilisation de l'opérateur $=>$ signifie que sa requête ne pourra aboutir que lorsque les requêtes émises antérieurement sur cette ressource par d'autres composants auront été évaluées.

Toutes ces caractéristiques font des ADL des langages très intéressants pour le prototypage et l'évaluation d'architectures logicielles. Leur capacité à décrire les comportements est comparable à celle d'AltaRica. On peut néanmoins noter la pauvreté des outils supports de certains ADL, les limitant ainsi parfois au prototypage d'architectures. Ils constituent par conséquent une bonne source d'inspiration pour les motifs que nous souhaitons développer. Une autre possibilité permettant de garantir un niveau de sûreté satisfaisant lors de la conception d'architectures consiste à pratiquer la réutilisation d'éléments sûrs. C'est de cette approche que nous allons traiter dans le suite de ce chapitre.

2.3 La réutilisation dans d'autres domaines

2.3.1 La notion de patron

Définition 1 *Un patron (ou pattern) est une solution récurrente qui décrit et résout un problème général dans un contexte particulier.*

C'est l'architecte Christopher Alexander [Ale77] qui, dans les années 70, proposa pour la première fois une solution permettant de formaliser les formes d'architectures qu'il trouvait intéressantes. Cette idée fut ensuite appliquée à de nombreux domaines.

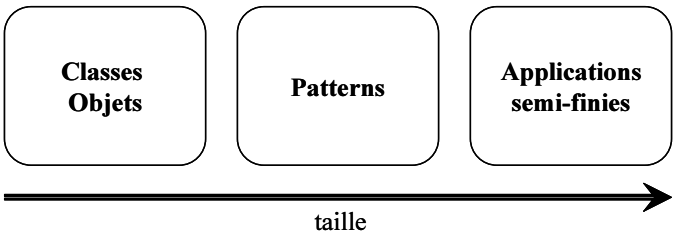


FIG. 2.1 – Taille des différentes classes

La réutilisation est une technique simple qui permet de gagner un temps précieux dans la phase de développement d'une architecture [Lut99, Lut97]. Elle possède néanmoins quelques inconvénients. Une réutilisation inappropriée, comme par exemple l'application d'un pattern dans un contexte différent de celui pour lequel il a été créé, ou encore le fait que cette réutilisation ne dépende que de la capacité de l'utilisateur à reconnaître une réutilisation possible et à se souvenir de l'information appropriée peut entraîner de graves erreurs. Par conséquent, des opportunités de réutilisations peuvent être perdues [MK97]. Enfin on peut craindre une perte du savoir-faire car on se trouve en totale dépendance envers des personnes pour pratiquer la réutilisation lorsque la documentation est insuffisante pour la faire de manière automatique. Si ces personnes partent, la réutilisation devient alors difficile. L'utilisation de patrons permet de solutionner ces problèmes. En effet, une documentation sur l'applicabilité et une description de l'environnement d'utilisation du patron permettront d'éviter les réutilisations inappropriées. En ce qui concerne la capacité de reconnaissance, elle est facilitée par la création de catalogues de patterns ou d'outils d'aide à la reconnaissance. Enfin, le patron contient le savoir-faire des experts et facilite donc sa transmission. Aucune perte du savoir-faire n'est donc à craindre, au contraire le but de cette réutilisation est de le capitaliser.

2.3.2 Les patrons d'architectures en génie logiciel

En génie logiciel, un pattern permettra de proposer une ou plusieurs solutions éprouvées et matures, à des problèmes récurrents en tenant compte du contexte d'utilisation [Bec94, BSSR99, San02]. Il permet de transmettre du savoir-faire car chaque utilisateur peut enrichir le pattern, l'adapter à la situation, le but étant qu'il soit réutilisable et indépendant des outils et du langage.

Il existe différentes familles de patterns témoignant de l'évolution de l'intérêt porté aux patterns par ses utilisateurs. On trouve ainsi un certain nombre de types de patterns qui

appartiennent à la classe informatique avec, notamment, les patterns de conception (aussi appelés design patterns), qui sont les plus populaires.

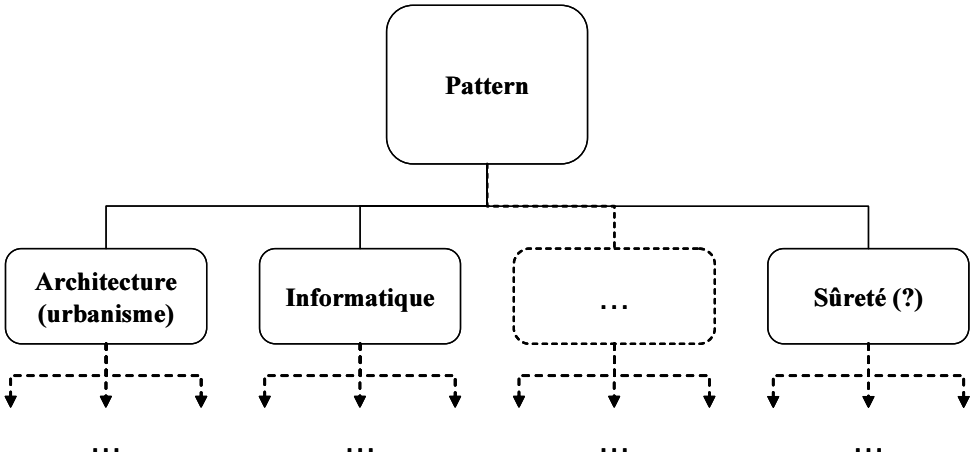


FIG. 2.2 – Différents types de patrons

Ils permettent d'améliorer la qualité des modèles de conception en réutilisant des connaissances et du savoir-faire.

- Les patterns d'analyse font de même dans l'activité d'analyse.
- Les patterns métier (ou de domaine) dépendent d'un secteur d'activité particulier et sont proches des patterns d'analyse.
- Les patterns d'architecture ne considèrent que la description structurelle des architectures logicielles.
- Les patterns organisationnels résolvent des problèmes d'organisation.
- Les patterns pédagogiques résolvent des problèmes du domaine de l'apprentissage et de la formation.
- Les patterns d'implémentation décrivent la manière de coder certains composants ou certaines fonctions dans un langage de programmation donné.

Tous ces patterns ont les mêmes caractéristiques exceptés les patterns d'implémentation qui perdent l'indépendance vis à vis du langage utilisé. Le principal intérêt des patterns en informatique est qu'ils permettent d'augmenter la puissance d'expression des langages de modélisation en ne s'occupant que des solutions de haut niveau. Afin d'être compréhensible par une personne autre que celle l'ayant réalisé, le pattern doit posséder une documentation qui renferme l'expérience et le savoir-faire des experts qui l'ont défini, son but, l'environnement dans lequel il s'applique, les raisons du choix de cette solution particulière ainsi qu'une aide pour son utilisation.

En génie logiciel, un pattern se caractérise par les attributs indiqués dans la figure 2.3.

Il n'existe aucun standard pour la description des patterns cependant un format de description existe et recommande d'utiliser, au minimum, les caractéristiques de la figure 2.3 lors de la création d'un pattern. En revanche, pour leur représentation, UML propose le concept de collaboration (regroupe un ensemble d'éléments de modélisation et décrit l'agencement

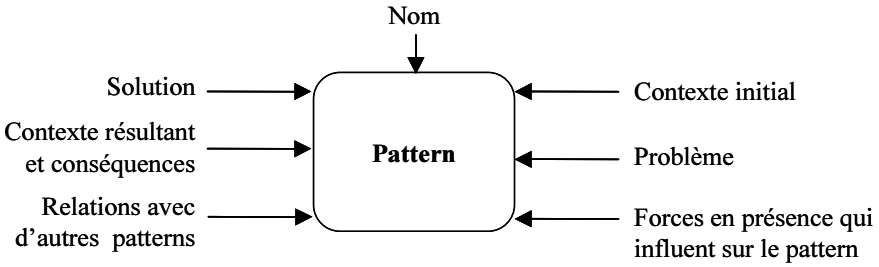


FIG. 2.3 – Description d'un patron

de ces éléments et leurs liens) paramétrable pour représenter les patterns de structure et de comportement.

Assemblages de patrons : les *frameworks*

Les patterns sont de petites architectures qui permettent de résoudre des problèmes ponctuels, ils ne permettent pas de représenter une application de manière globale. Pour cela on construit une architecture composée de plusieurs patterns. On crée alors un pattern de plus haut niveau appelé framework . Ces frameworks imposent une architecture générale, définissent les mécanismes de contrôle ainsi que les liens entre les objets. Se pose le problème de la compréhension (et donc de l'utilisation) du framework par une personne autre que son créateur. En effet, lorsqu'un pattern nécessite une documentation simple, celle du framework doit être plus détaillée car il utilise de nombreux composants. Contrairement aux patterns, ces architectures semi-finies ont pour but d'implémenter rapidement de nouvelles applications et donc perdent cette indépendance vis à vis du langage de programmation utilisé d'où une marge de manœuvre beaucoup plus réduite.

On distingue deux classes de frameworks : une première selon leurs domaines d'application et une seconde selon leurs structures (boite blanche/noire). Leur utilisation implique les mêmes avantages que celle des patterns mais cette fois au niveau application, auxquels s'ajoute une maintenance plus simple car l'architecture d'origine est sûre ainsi qu'un gain de productivité dû à la possibilité de réutilisation.

Catalogue et classification de patrons

La conception d'abstractions d'une application nécessite la combinaison de plusieurs design patterns. La consultation d'un catalogue regroupant divers patterns permet de simplifier cette conception. Le catalogue de patterns de Gamma, Helm, Johnson et Vlissides [GHJV99] contient environ une vingtaine de micro-architectures et leurs relations. Afin de faciliter la compréhension de la structure d'un tel catalogue de patterns il est nécessaire d'organiser leurs relations en catégories.

Dans la classification de Zimmer [Zim95] ni la direction des relations (uni ou bidirectionnelle) ni leur force (forte ou faible), qui est une mesure très subjective, ne sont prises en compte. Cette classification propose trois catégories de relations entre un pattern A et un

pattern B :

- A utilise B dans sa solution
- A est semblable à B
- A peut être associé à B

Elle permet, par exemple, lors de la recherche d'un pattern permettant la résolution d'un problème précis, de trouver la famille de pattern proposant des solutions adaptées pour ensuite choisir le pattern qui satisfait les exigences. A partir d'un catalogue, comme celui de Gamma, et d'une classification, comme celle de Zimmer, nous allons voir comment résoudre le problème le plus important lors de l'utilisation des designs patterns, à savoir le manque d'outils permettant l'automatisation de leur instanciation et de leur détection.

Instanciation et détection de design patterns

Il existe un ensemble d'outils récents, créés entre autre à l'Ecole des Mines de Nantes, permettant :

- Le choix d'un pattern en fonction de l'environnement d'utilisation
- L'adaptation d'un pattern à des exigences particulières
- La détection de versions dites complètes (i.e. identiques) ou approchées d'un design pattern
- La transformation des versions approchées pour se rapprocher du pattern d'origine

Ces outils utilisent un formalisme de résolution de contraintes appelé CSP (Constraint Satisfaction Problem) [FD93, Fre78] ainsi qu'un solver de contraintes nommé PALM (Propagation And Learning with Move) [JB00]. La solution proposée pour permettre l'automatisation de l'instanciation et de la détection de pattern est basée sur la création de méta-modèles permettant de formaliser les patterns de manière à pouvoir les manipuler.

Méta-modèles de *design patterns*

Le but est de décrire les design patterns de manière à pouvoir outiller leur manipulation. La technique de méta-modélisation permet de formaliser les patterns. PDL (Pattern Description Language) [AA03] permet la création de méta-modèles. D'autres langages de méta-modélisation existent mais se cantonnent à l'instanciation et à la validation sans prendre en compte l'aspect détection. La description d'un pattern avec PDL est réalisée par instanciation et composition d'un ensemble défini de méta-entités. Cette composition suit des règles sémantiques définies par la relation entre ces entités, ceci permet de formaliser le pattern. Le résultat obtenu est alors un modèle abstrait qui décrit à la fois la structure et le comportement du pattern modélisé mais qui n'est pas dépendant de son environnement (le modèle enrichi de son contexte est alors appelé modèle concret). Les méta-modèles, les modèles abstraits et concrets peuvent être représentés en UML.

L'utilisation d'un design pattern peut être décomposée en trois étapes :
- Le choix du pattern correspondant aux exigences
- L'instanciation du pattern

– La production du code nécessaire à son implémentation

L'outils PatternBox [AA03] permet de réaliser ces trois tâches en permettant un accès au catalogue de patterns, en fournissant un mécanisme pour adapter un modèle abstrait à un contexte (donc en créant un modèle concret) et en générant le code associé. Actuellement, la navigation dans le catalogue de patterns se fait à partir d'une version HTML du catalogue de Gamma. Puis, une fois le modèle abstrait choisi, l'outil utilise Java pour manipuler et adapter les patterns.

Identification automatique de patrons

L'outil PTIDEJ (Pattern Trace Identification, Detection and Enhancement for Java) [Gué05] permet d'identifier automatiquement des patterns (qu'ils soient dans leurs versions complètes ou approchées). Comme dit précédemment, un design pattern est représenté par un modèle abstrait. Au niveau du code, ce modèle abstrait est traduit en ce que les auteurs de [AACGJ01] appellent "micro-architectures" et que j'appellerai "code d'architecture" pour éviter toute confusion. On dit qu'un code d'architecture est une version complète du pattern s'il correspond strictement au code d'architecture du modèle abstrait du design pattern, sinon, si des relations manques, on dit que le code d'architecture est une version partielle. Si l'outil détecte des versions partielles d'un pattern, il calcule leurs différences avec les codes d'architecture des modèles abstraits et propose des corrections pour les réduire et permet de les référencer. L'outil génère le CSP à partir du modèle abstrait du pattern qu'on lui fournit. Il permet également :

– d'afficher l'architecture d'une application,
– de produire un modèle simplifié de cette architecture,
– de résoudre le CSP à partir du modèle simplifié (i.e. identifie les versions complètes et partielles et les règles de transformation),
– de sélectionner une version partielle spécifique,
– de transformer les parties du code de l'application correspondant aux versions partielles pour qu'elles correspondent le plus possible au code d'architecture du design pattern correspondant,
– d'afficher la nouvelle architecture de l'application avec une notation proche d'UML.

L'implémentation de PTIDEJ est réalisée en Java et le code modélisé en PDL. PTIDEJ génère également un modèle dans le langage de programmation Claire [CL96], aussi utilisé pour exprimer les CSP. L'objectif des CSP est de fournir un formalisme permettant d'identifier dans le code source les codes d'architecture identiques ou proches du code d'architecture définie par le design pattern en fonction de leurs variables (i.e. les entités qui constituent le modèle abstrait), des contraintes qui existent entre elles (i.e. les relations entre entités) et de leurs domaines. Un CSP est obtenu automatiquement à partir du modèle abstrait du design pattern.

L'outil qui utilise ce formalisme est appelé PALM. C'est un solver de contraintes capable de fournir des explications et informations sur ce qu'il fait ou ne peut pas faire. En effet, ce ne sont pas les versions complètes qui posent problème, mais les versions partielles qui, elles, ne satisfont pas toutes les contraintes et ce sont ces contraintes non-satisfaites qui peuvent fournir des informations intéressantes. Or les solvers de contraintes classiques ne permettent pas de trouver les versions partielles mais seulement les complètes. S'ils se retrouvent face à

une version partielle, ils se contenteront de dire qu'il n'y a pas de solution. D'où l'utilité de définir un nouveau solver capable de traiter ces versions partielles. Ce nouveau type de solver est basé sur des explications sur les actions qu'il effectue et qui permettent :

- de savoir quelles contraintes sont responsables de la solution éventuelle,
- de savoir quelles contraintes sont responsables d'une contradiction éventuelle.

Les raisons pour lesquelles aucune instance d'un pattern donné n'est trouvée sont clairement expliquées. A partir de ces explications, des contraintes peuvent être supprimées pour définir des versions partielles. PALM est construit sur la base du solver de contraintes open-source Choco [Lab00] et est donc implémenté en Claire.
L'utilisation de patrons est de plus en plus courant et ne se limite pas au domaine de l'architecture ou de la programmation de logiciel. En effet, le domaine de la logique, dans l'optique de spécifier des propriétés, utilise également ce genre de structures.

2.3.3 Des patrons de spécification de propriétés

Introduction

La vérification permet de prouver un certain nombre de propriétés d'un modèle à états finis. Ces propriétés sont généralement exprimées en logique temporelle [Pnu81, MP92], que nous présentons au chapitre suivant, ou à l'aide d'expressions régulières. La logique temporelle utilisée par ces patrons pour spécifier les propriétés est la logique temporelle linéaire (LTL) propositionnelle. La partie non temporelle de cette logique n'est autre que la logique propositionnelle classique. Par conséquent, les formules sont construites à partir de propositions atomiques et des connecteurs classiques $\vee$ et $\neg$, ainsi qu'à l'aide des opérateurs temporels suivants dont la sémantique est présentée dans la définition 10 :

Opérateur	Signification
X	Prochain (*neXt*)
F	Futur (*Finally*)
G	Toujours (*Globally*)
U	Jusqu'à (*Until*)

TAB. 2.1 – Opérateurs temporels de LTL

Quant aux systèmes, ils sont en général traduits en systèmes de transitions à états finis. Des outils, ne nécessitant pas l'aide d'un expert, existent pour tous les types de techniques de vérification : model-checking, bisimulation, ... Malgré ces aides, l'utilisateur doit tout de même être capable de spécifier les exigences du système dans le langage adéquat. En effet, même si toutes les capacités et le pouvoir d'expression associés aux techniques de vérification formelles ne sont pas exploités pleinement, les utilisateurs doivent être assez compétents pour exprimer exactement les exigences qu'ils souhaitent vérifier dans le formalisme de spécification approprié. Bien que les propriétés soient relativement simples, les écrire en logique temporelle nécessite un certain savoir-faire. Cet obstacle pourrait être contourné en trouvant un moyen de capter l'expérience et le savoir-faire des experts afin d'effectuer un "transfert de compétence". Les patterns de spécification [ACD98b, ACD98a] semblent être la solution.

Notion de patron de spécification

Un pattern de spécification de propriété est une description généralisée d'une exigence récurrente. Il décrit la structure essentielle du comportement d'un système et fournit des expressions de ce comportement dans le formalisme voulu. Chaque pattern possède une portée bien définie sur l'exécution du programme. On distingue cinq portées de base : tout d'abord la portée globale (*global*) qui s'étend sur l'ensemble du programme, la portée avant (*before*) qui ne tient compte de l'exécution que jusqu'à un certain événement ou état, la portée après (*after*) qui ne prend en compte l'exécution qu'à partir d'un certain événement/état, la portée entre (*between*) qui ne prend en compte l'exécution qu'à partir d'un événement/état donné et jusqu'à un autre événement/état donné. Enfin, la portée après-jusqu'à (after-until) ressemble au *between* à la seule exception qu'il peut ne pas s'arrêter si le second événement ne se produit pas (cf figure 2.4).

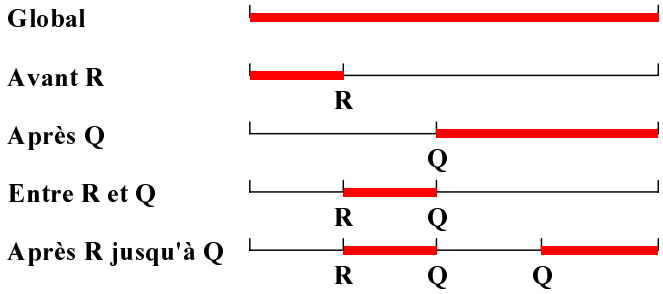


FIG. 2.4 – Différentes portées

La portée est donc définie par un état/événement de départ, un autre d'arrivée. Elle contient l'ensemble des états/événements compris entre ce point de départ et ce point d'arrivée. La portée est un paramètre optionnel.

Un système de patrons de spécification

Ce système de patterns de spécification est un ensemble de patterns classés et organisés hiérarchiquement selon leur sémantique, de manière à faciliter la recherche de patterns (i.e. trouver celui qui correspond aux exigences voulues). En effet, certains patterns nécessitent l'apparition ou la non-apparition d'états/événements (e.g. pattern d'absence), d'autres sont basés sur l'ordre d'apparition de ces états/événements (e.g. pattern de réponse) ou encore sont la composition de patterns de base. Le système doit donc bien distinguer ces différents types de propriétés. Cependant, les différents points de vue possibles autorisent l'apparition d'un pattern dans plusieurs catégories.

Trois sous-groupes de patterns de propriétés seront présentés : les patterns d'occurrence, les patterns d'ordre et les patterns composés.

Les **patterns d'occurrence** sont composés des patterns suivants :
- Pattern d'absence : un événement/état P donné ne se réalise pas (avec une certaine portée)

But : Décrire une partie de l'exécution d'un système qui ne contient pas un état/événement donné.

Structure :
Propriété : P est faux

Portée	Traduction en LTL
Globale	$G(\neg P)$
Avant R	$FR \rightarrow \neg P\,UR$
Après Q	$G(Q \rightarrow G(\neg P))$
Entre Q et R	$G(Q \wedge FR) \rightarrow \neg P\,UR$
Après Q jusqu'à R	$G(Q \rightarrow \neg P\,U(R \vee G\neg P)$

Relation avec d'autres patterns : Ce pattern est le dual du pattern d'existence.

FIG. 2.5 – Patron d'absence

– Pattern d'existence : un événement/état donné P se réalise (avec une certaine portée)
– Pattern d'existence bornée : un événement/état donné P doit se réaliser k fois. Des variantes de ce pattern précisent que l'événement/état doit se réaliser au moins ou au plus k fois

But : Décrire les relations de cause à effet entre une paire d'états/événements. L'occurrence du premier (la cause) doit être suivie par l'occurrence du second (l'effet).

Structure :
Propriété : S répond à P

Portée	Traduction en LTL
Globale	$G(P \rightarrow FS)$
Avant R	$(P \rightarrow (\neg RUS))\,U(R \vee G\neg R)$
Après Q	$G(Q \rightarrow G(P \rightarrow FS)$
Entre Q et R	$G((Q \wedge FR) \rightarrow (P \rightarrow (\neg RUS))\,UR)$
Après Q jusqu'à R	$G(Q \rightarrow ((P \rightarrow (\neg RUS))\,UR) \vee G(P \rightarrow (\neg RUS)))$

Utilisation : Description d'une exigence : une ressource doit être accordée lorsqu'elle à été demandée.
Relation avec d'autres patterns : Ce patron est lié au pattern de précédence.

FIG. 2.6 – Patron de réponse

– Pattern d'universalité : un événement/état P se réalise tout au long de la portée.

Les **patterns d'ordre** comprennent :
– Pattern de précédence : un événement/état doit toujours être précédé d'un autre événe-

ment/état donné (sur une portée),
- Pattern de réponse : un événement/état P doit toujours être suivi d'un événement/état donné Q sur une portée. C'est un mélange des patterns d'existence et de précédence.

Enfin, les **patterns composés** sont les suivants :
- Pattern de précédence en chaîne : une séquence d'événements/états $P_1, ..., P_n$ doit toujours être précédée par une séquence d'événements/états $Q_1, ..., Q_n$. Ce pattern est une généralisation du pattern de précédence,
- Pattern de réponse en chaîne : une séquence d'événements/états $P_1, ..., P_n$ doit toujours être suivie d'une séquence d'événements/états $Q_1, ..., Q_n$. Ce pattern est une généralisation du pattern de réponse.

Les patrons permettent également la spécification de systèmes ou de fonctions critiques comme le montre l'exemple qui suit.

2.3.4 Systèmes critiques et patrons

Un système critique se doit d'être robuste (faible probabilité d'erreur de l'opérateur et facilité des actions correctives) en terme d' "utilisabilité". Le designer de tels systèmes interactifs doit donc s'efforcer de réduire les erreurs possibles de l'utilisateur. Une solution proposée dans [HM99] consiste à utiliser le principe de langage de patterns pour traiter ce problème. Lors de la conception d'un système critique, plusieurs points doivent être étudiés :

- la prévention des erreurs : interdire à l'utilisateur de mettre le système dans un état dangereux,
- la réduction des erreurs : donner à l'utilisateur une représentation claire de l'état du système,
- la correction des erreurs : fournir un feedback, supervision, surveillance automatique.

Un langage de patterns à donc été créé dans ce but par Mahemoff et Hussey. Il contient quatre groupes de patterns :

- management des tâches (TM),
- exécution de tâches (TE) : mécanisme d'exécution des tâches,
- information (INF) : informations présentées aux utilisateurs,
- contrôle machine (MC) : automatisation des systèmes.

Afin de trouver des patterns d'"utilisabilité" sûrs, les auteurs ont analysé plusieurs cas d'études dont le système *Druide*, prototype du CENA (Centre d'Etude de la Navigation Aérienne). Ce système de contrôle du trafic aérien permet d'envoyer des ordres aux appareils (changement de cap, d'altitude etc.) se trouvant dans le secteur couvert par le contrôleur aérien au moyen d'un menu interactif qui s'affiche sur l'écran du radar. Ce système permet l'affichage d'énormément d'informations sur l'appareil (identifiant, vitesse, etc.). Toutes ces informations sont impossibles à afficher simultanément sous peine de rendre l'affichage totalement incompréhensible. Par conséquent, c'est au contrôleur de demander à l'appareil, quand cela est nécessaire, les informations supplémentaires. Voici donc le pattern "interrogation" qui fait partie du groupe de patterns "Information" du langage :

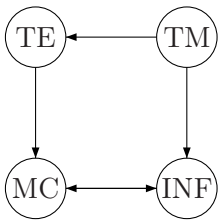


FIG. 2.7 – Structure d'un langage de patrons

Nom : Interrogation

Contexte :
- Système complexe fournissant de nombreuses informations, toutes ne pouvant être affichées simultanément.
- Certaines informations sont utilisées plus souvent que d'autres.
- Certaines informations sont plus facilement affichées que d'autres.

Problème : La plupart des systèmes critiques interactifs proposent à l'utilisateur une représentation de l'état du système. Dans beaucoup de cas, cet état est complexe et ne peut pas être représenté de manière compréhensible. Pour de tels systèmes, afficher l'état global du système peut occulter certains paramètres importants et induire des erreurs. Comment l'utilisateur peut-il avoir accès à l'état global du système sans être écrasé par la quantité d'information ?

Contraintes :
- Les utilisateurs ont une capacité d'attention limitée (la limité supérieure est généralement fixée à quatre informations en parallèle).
- Les écrans ont une résolution et une capacité limitées.

Solution : Permettre à l'utilisateur de demander des informations supplémentaires. De cette façon, toutes les informations n'ont pas à être affichées simultanément. L'interrogation doit être simple et intuitive.

Il est également possible de définir des patrons de plus bas niveau permettant d'assurer un niveau de fiabilité et de sûreté donné pour une architecture.

2.3.5 Designs patterns pour la fiabilité et la sûreté

Les design patterns peuvent également servir à guider la conception d'architectures sûres de systèmes embarqués [Dou98]. Ces patrons, touchant à la fois la fiabilité et la sûreté d'un système, sont principalement architecturaux car ils affectent de nombreuses parties d'un système. En voici quelques-uns représentés en UML :

Patron 1 : Homogeneous Redundancy Pattern (HRP) Le HRP permet d'améliorer la fiabilité en redondant les canaux et en ajoutant un contrôleur qui va prendre les décisions (e.g. par

vote). L'avantage d'une telle architecture est qu'elle est relativement simple à mettre en œuvre car il s'agit de systèmes strictement identiques qui sont redondés. De plus, elle autorise les pannes aléatoires comme la panne d'un composant d'un sous-système. Par contre, elle possède un inconvénient majeur car elle est sensible aux erreurs systématiques : si dans certaines conditions d'utilisation (e.g. phase de vol d'un avion) un composant d'un système redondé d'une architecture HRP tombe en panne, on peut s'attendre à ce que toutes les redondances vont tomber en panne simultanément ce qui réduit grandement son intérêt. Ajoutons à cela que les HRP ne sont pas efficaces pour des erreurs de logiciels puisque les systèmes redondés sont identiques.

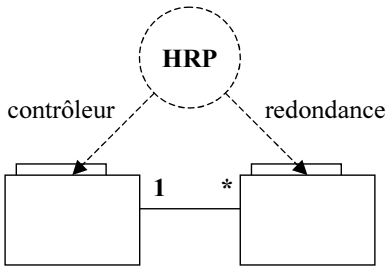


FIG. 2.8 – Patron HRP

La question du choix du type de redondance à utiliser est importante : actif ou passif ? En effet, bien que la redondance passive possède l'inconvénient d'un éventuel refus de démarrage, la redondance homogène (ou semblable) active ne fait que multiplier des systèmes identiques, venant du même fournisseur, possédant les même caractéristiques (MTTF, MTBF, etc.). Par conséquent, ces systèmes en redondance active vont réagir de la même manière face à l'environnement, vieillir de la même façon. On peut donc imaginer que lorsqu'un système tombe en panne (pour des raisons d'usure, d'environnement, etc.) l'ensemble des systèmes en redondance active peut également tomber en panne (si on allume deux ampoules identiques en permanence, lorsque l'une d'entre elle grille, on peut s'attendre à ce que l'autre grille rapidement après ; de même, si on applique les mêmes conditions d'utilisation hostiles aux deux ampoules (e.g. atmosphère saturée en humidité) on peut s'attendre aux mêmes conséquences).

Patron 2 : Diverse Redundancy Pattern (DRP) Le DRP fait de la redondance dite dissemblable car les systèmes redondés seront différents tout en continuant à assurer la même fonction (on peut donc maintenant envisager des erreurs de logiciels) . En génie logiciel cela se traduira par une même sémantique et interface mais une implémentation différente. Le travail de conception à fournir est donc nettement plus important que pour les HRP.

Patron 3 : Monitor-Actuator Pattern (MAP) Le MAP est un type de DRP basé sur un canal double, c'est à dire que l'on y retrouve un système de contrôle et un système de surveillance.

Patron 4 : Safety Executive Pattern (SEP) Le SEP est constitué d'un contrôleur appelé safety controler, d'un ou de plusieurs chiens de garde qui vérifient l'état du système, un ou plusieurs actionneurs et enfin un ou plusieurs système de rétablissement en cas de faute sur le système.

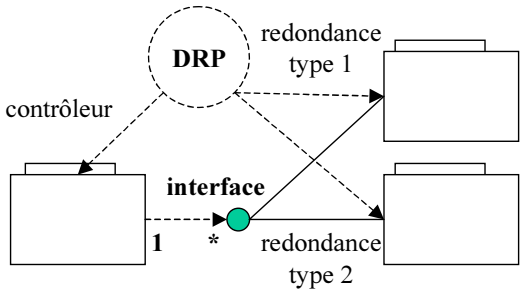


FIG. 2.9 – Patron DRP

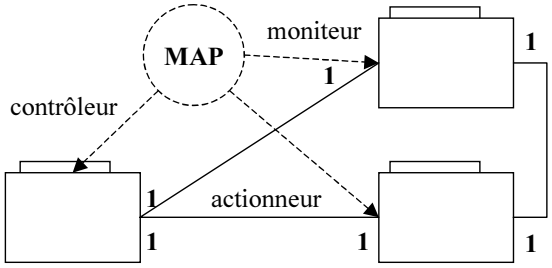


FIG. 2.10 – Patron MAP

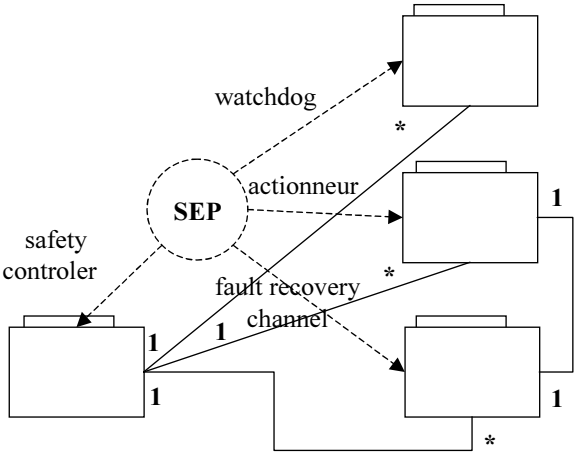


FIG. 2.11 – Patron SEP

2.4 Conclusion

D'après Dwyer [ACD98b], les patterns et les gens qui les créent se caractérisent par un manque total d'originalité. En effet, le but du pattern est d'être le plus générique possible afin de pouvoir être réutilisé le plus souvent possible. Les différentes utilisations qui ont été faites et que nous avons partiellement montrées au cours de ce chapitre montrent que les patterns semblent être la clé pour un gain de temps ainsi qu'un très bon moyen de transmission du savoir-faire. L'exemple des patterns de spécification de propriétés, par exemple, nous a per-

mis de constater que la vérification de systèmes à états finis peut être rendue plus facilement accessible aux développeurs de systèmes concurrents et réactifs grâce à leur utilisation car ils permettent d'écrire et de lire rapidement des spécifications formelles sans faire appel à des experts du domaine. Ils permettent également d'améliorer la communication en fournissant un langage commun pour discuter des problèmes ou proposer des solutions, la documentation en capitalisant le savoir-faire des experts et en fournissant une base commune pour comprendre un système, enfin la réutilisation en proposant des catalogues de patterns. Cependant, ils ne possèdent pas que des avantages. En effet, leur compréhension (surtout en ce qui concerne les frameworks) peut poser des problèmes. De plus, ces architectures peuvent contenir des composants inutiles pour une application donnée et risque donc de la compliquer. Enfin, l'incompatibilité des architectures des différents frameworks produit des problèmes d'intégration. Bref, ces outils semblent être une solution simple et efficace, encore faut-il les documenter, les comprendre et les utiliser correctement.

Chapitre 3

Modèles formels et spécification logique

3.1 Introduction

L'analyse ou la vérification de systèmes lors des phases préliminaires de conception passe nécessairement par une phase de modélisation. Cette modélisation nous permet de mieux comprendre, de décrire et de prédire le fonctionnement et l'évolution de systèmes réels. Un modèle traduit la perception que le créateur du modèle a de la réalité. Il correspond à une abstraction d'un système réel, c'est à dire au fragment utile du point de vue des analyses à mener. Cette opération est délicate car elle doit permettre de simplifier un système complexe tout en conservant son comportement et ses propriétés. L'opération d'abstraction peut, au premier abord, paraître complexe. C'est pourtant une opération que l'on réalise tous les jours sans s'en rendre forcément compte, un peu comme Monsieur Jourdain. Qui n'a jamais raconté un film, un livre ou ses vacances à ses collègues de bureau ? Synthétiser deux heures de films, les actions principales, l'histoire et les personnages relève de l'abstraction. On réalise alors une sorte de modéle (informel) du film. Il doit être suffisamment clair pour en donner une image fidèle, sans pour autant entrer dans les détails qui rendraient le discours beaucoup trop complexe et long. La modélisation visant à la compréhension et à l'analyse de systèmes industriels nécessite un peu plus de rigueur. On parle alors de modèle formel, représentation mathématique qui sert de base à toute recherche scientifique rigoureuse en général. Nous présentons dans la suite de ce chapitre quelques techniques de modélisation formelle ainsi que quelques logiques permettant de spécifier leurs propriétés. Notons toutefois que dans le domaine de la sûreté de fonctionnement qui nous intéresse, nous nous focalisons sur des modèles de propagation de pannes, des modèles d'évolution des systèmes avec injection de défaillances.

3.2 Modélisation formelle par automates

3.2.1 Les systèmes de transitions

Un *système de transition* [Arn94] consiste en un ensemble de configurations et de transitions permettant de décrire des processus dynamiques. Les configurations correspondent aux états du processus et les transitions renseignent sur son évolution. Il existe plusieurs types

de systèmes de transition comme les systèmes étiquetés ou paramétrés. En utilisant un bon niveau d'abstraction, de nombreux processus, e.g. réseaux, systèmes de communication etc., peuvent être vus comme des séquences de configurations et donc décrits à l'aide de ce formalisme. Les graphes de Markov, présentés précédemment, correspondent à de tels systèmes. L'objectif principal de cette description formelle des processus est de pouvoir les analyser et analyser leurs propriétés. Traditionnellement, deux types de propriétés sont intéressantes : les propriétés dites de *sûreté* et celles dites de *vivacité*. Une propriété de *sûreté* est utilisée pour exprimer que quelque chose de non souhaité ne doit jamais se produire, comme, par exemple, vérifier qu'en loi normale les commandes de vol d'un avion ne permettent jamais d'atteindre un angle de roulis supérieur à 67°. Les propriétés de *vivacité* expriment le fait que quelque chose de souhaité finira toujours par se réaliser au moins une fois, e.g. la perte totale du système électrique d'un A320 doit aboutir au deploiement de l'éolienne de secours située sous l'appareil. Nous définissons maintenant les systèmes de transitions sur lesquels il est possible d'analyser ce genre de propriétés.

Définition 2 (Système de transitions) *Un système de transitions est un quadruplet* $A = \langle S, S_0, T, F \rangle$ *avec :*

- S *un ensemble fini ou infini d'états*
- $S_0 \subseteq S$ *un ensemble fini ou infini d'états initiaux*
- $T \subseteq S \times S$ *un ensemble fini ou infini de transitions*
- $F \subseteq S$ *un ensemble fini ou infini d'états finaux*

Notons que des transitions particulières appelées ϵ-transitions initiées par un événement ϵ, interne au processus et inobservable, permettent de garantir qu'un processus ne bloque pas. Si S et T sont finis on parle de systèmes de transitions finis. Si S est fini on parle alors aussi d'automate.

Définition 3 (Chemin d'un système de transitions) *On appelle chemin de longueur* n $(n > 0)$ *d'un système de transitions fini* A *une suite de transitions* $t_1, ..., t_n$ *telle que* $\forall i, 1 \leqslant i < n, t_i = (s_i, s_{i+1})$. *Alors* s_1 *appartient à l'ensemble* $S_0 \subseteq S$ *des états initiaux de* A *et* s_n *appartient à l'ensemble* F *des états finaux de* A.

De même, un chemin infini est une suite infinie de transitions $t_1, ..., t_n, ...$ *telle que* $\forall i \geqslant 1, t_i = (s_i, s_{i+1})$. *Comme dans le cas des chemins finis,* $s_1 \in S_0$. *Etant donné qu'il n'est pas possible d'imposer à l'état terminal d'un chemin infini d'appartenir à* F, *une autre contrainte doit être définie et satisfaite pour ce type de chemins. Dans le cas des systèmes de transitions infinis comme ceux de Richard Büchi, que nous étudierons plus tard, cette contrainte correspond au fait de devoir passer une infinité de fois par l'ensemble des états de* F.

Exemple Considérons deux composants réalisant une même fonction en parallèle. Un de ces composants n'est pas activé initialement (appelé secondaire). Il ne s'active qu'après détection de la défaillance (e.g. absence de sortie) du composant initialement activé (appelé primaire). Ce type de mécanisme est communément appelé une redondance passive. Chaque composant peut défaillir à tout moment. Le système de transitions correspondant à ce processus est représenté sur la figure 3.1. L'état 1 correspond à l'état initial où les deux composants sont opérationnels, le composant 1 est actif alors que le 2 ne l'est pas. Aprés défaillance du composant 1 le système passe dans l'état 2 où il est possible d'activer le composant 2 pour passer

dans l'état 5. Si l'on observe une panne cachée du composant 2 le système passe dans l'état 3. La défaillance du composant encore fonctionnel dans les états 2 ou 3 mène le système dans l'état 4. On passe finalement de l'état 5 à l'état 6 sur défaillance du composant 2.

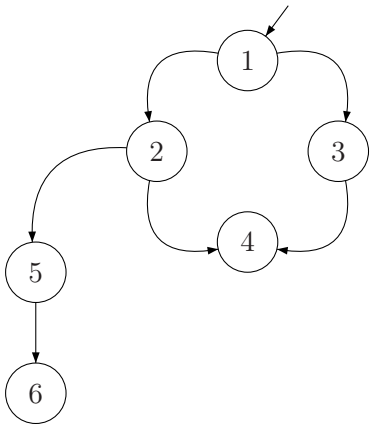


FIG. 3.1 – Système de transitions

L'avantage de ce formalisme réside dans sa simplicité de conception et de compréhension pour des systèmes de taille raisonnable. En effet, il est nécessaire d'expliciter l'ensemble des états du processus ce qui devient rapidement problématique pour des cas pratiques réalistes. On est alors face au problème très connu de l'explosion combinatoire du nombre des états.

Cependant, à ce stade, les systèmes de transitions ne permettent pas une représentation détaillée des processus. Une transition, par exemple, correspond à une action précise, il serait intéressant de représenter le nom de cette action sur le graphe. De la même manière il serait également intéressant de représenter l'ensemble des propriétés satisfaites en un état du graphe. On passe alors dans le domaine des systèmes de transitions étiquetés.

Définition 4 (Système de transitions étiqueté) *Un système de transitions étiqueté est un quintuplet* $A = \langle S, S_0, \Sigma, T, F \rangle$ *avec :*

- S *un ensemble fini ou infini d'états*
- $S_0 \subseteq S$ *un ensemble fini ou infini d'états initiaux*
- Σ *un ensemble fini ou infini d'événements*
- $T \subseteq S \times \Sigma \times S$ *un ensemble fini ou infini de transitions*
- $F \subseteq S$ *un ensemble fini ou infini d'états finaux*

Exemple Le système de transitions étiqueté de l'exemple précédent est représenté sur la figure 3.2.

Replaçons nous dans le cadre qui nous intéresse. Les systèmes réels que nous étudions sont certes complexes mais ils sont composés d'un nombre fini d'états. Afin d'augmenter l'expressivité des systèmes de transitions pour faciliter la modélisation de systèmes réels, il est possible d'étendre leur modèle. Ainsi, les *automates à propriétés* permettent d'associer

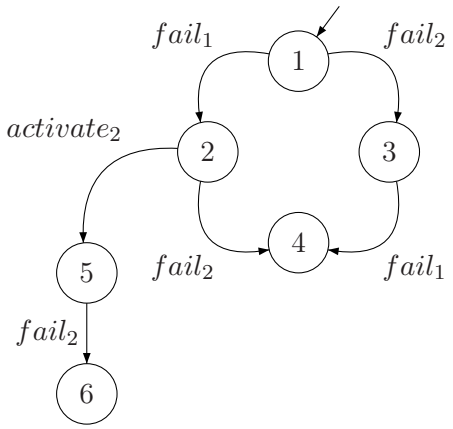


FIG. 3.2 – Système de transitions étiqueté

une propriété logique à chaque état du graphe. Si l'on reprend l'exemple de la redondance, il serait possible, par exemple, d'étiqueter les états par les propriétés *p : "un seul composant est défaillant"*, ou *q : "tous les composants sont défaillants"*. Une autre extension possible consiste à introduire un ensemble fini de variables discrètes dans le modèle des systèmes de transitions. On peut ainsi définir des gardes aux transitions qui imposent certaines conditions sur les variables pour autoriser leur franchissement. Ce franchissement peut lui même modifier la valeur des variables. Ce type d'automate à variables permettrait par exemple d'introduire un compteur qui s'incrémenterait à chaque occurrence d'une défaillance. Dans cette quête croissante d'expressivité, il est possible de généraliser le concept d'automates à variables en introduisant des paramètres entiers en quantité éventuellement infinie. Un jeu de valuation de ces paramètres nous permet de caractériser un état et à chaque transition est toujours associée une contrainte sur ces paramètres. C'est le principe des automates à contraintes que nous allons maintenant présenter.

3.2.2 Automates à contraintes

Un automate à contraintes [APGR00] est un automate à états finis usuel dont les états et transitions ne sont pas définis explicitement mais par des contraintes sur les variables. Ce type d'automate est assez général pour décrire la plupart des modèles formels dédiés à la vérification de systèmes. Intuitivement, un automate à contraintes se décrit à l'aide d'un ensemble de transitions de la forme $G(V) \xrightarrow{e} V = \sigma(V)$ avec V un ensemble de variables. Cette transition est franchie si la garde $G(V)$ définissant une contrainte sur les variables de V est satisfaite et l'événement e se produit. On passe alors dans un nouvel état défini par la valuation de ses variables $V = \sigma(V)$.

Définition 5 (Automate à contraintes) *Un automate à contraintes A est un 7-uplet $A = \langle D, S, F, \Sigma, T, A, I \rangle$ avec :*

- *D est un domaine fini ou infini*
- *S et F sont deux ensembles disjoints de variables appelés respectivement variables d'état et de flux. Les variables d'état signalent, comme leur nom l'indique, l'état interne du*

composant. Les variables de flux correspondent à des valeurs auxiliaires, qui ne font pas partie de l'état du composant, et qui permettent au composant de communiquer avec son environnement. Comme indiqué précédemment, ces deux types de variables interviennent pour contraindre le comportement du composant au sein des transitions *et* assertions.
- Σ *est un ensemble d'événements*
- T *est un ensemble de transitions. Une transition est un triplet* (g, e, a) *avec* $g \subseteq D^{|S \cup F|}$ *une contrainte sur* $S \cup F$ *appelée garde,* $e \in \Sigma$ *et* a *est une application qui définie les états sucesseurs :* $a : D^{|S \cup F|} \rightarrow D^{|S|}$.
- $A \subseteq D^{|S \cup F|}$ *est un ensemble d'assertions, i.e. de contraintes sur les valeurs des variables*
- $I \subseteq D^{|S \cup F|}$ *est l'ensemble des états initiaux*

Une restriction intéressante de ce modèle des automates à contraintes permet de travailler sur un nombre fini de variables et d'orienter les variables de flux qui ne sont plus alors bidirectionnelles. Cette restriction correspond au modèle des automates de mode [Rau02, MR98] que nous allons maintenant présenter.

Définition 6 (Automate de mode) *Formellement, un automate de mode* A *est un 9-uplet* $A = \langle D, S, F^{in}, F^{out}, dom, \Sigma, \delta, \sigma, I \rangle$ *avec :*

- D *est un domaine fini*
- *Notons* V *l'ensemble fini de variables scindées en trois catégories* S, F^{in} *et* F^{out}. *Ces variables sont des sous-ensembles de* V *deux à deux disjoints appelées respectivement variables d'état, de flux d'entrée et de flux de sortie.*
- $dom : V \rightarrow 2^D$ *telle que* $\forall v \in V, dom(v) \neq \emptyset$ *associe à une variable son domaine*
- Σ *est un ensemble fini d'événements*
- δ *est une fonction partielle appelée* transition *:* $dom(S) \times dom(F^{in}) \times \Sigma \rightarrow dom(S)$, $dom(S) \times dom(F^{in})$ *est la garde de la transition*
- σ *est une fonction totale appelée* assertion *:* $dom(S) \times dom(F^{in}) \rightarrow dom(F^{out})$
- $I \subset \sigma$ *est une fonction partielle qui décrit les conditions initiales*

Les états et les transitions de cet automate sont définis en intention par δ et σ. On notera qu'il en existe un nombre fini puisque V et D sont finis. Le lecteur intéressé pourra se tourner vers [PR99] pour plus de détails.

Exemple Reprenons l'exemple de la redondance passive de deux composants identiques. Dans cet exemple, les composants C_1 et C_2 sont des sources électriques (type `source` défini par la suite) qui se redondent passivement. Chaque composant C_i possède un état noté s_i ainsi qu'une défaillance $fail_i$ et C_2 possède un événement d'activation (`activate2`) permettant de changer son état a_2 d'activation. Les sorties o_1 et o_2 des composants permettent de construire une sortie unique notée $o = o_1 \vee o_2$. Toutes ces variables sont booléennes. Initialement les composants sont non-défaillants (i.e. $s_i = 1$) et le composant C_2 n'est pas activé. Dans cet état o est vraie. Si C_1 défaille alors C_2 peut être activé et prend le relai. Dans le cas où les deux composants sont défaillants (i.e. $fail_1$ et $fail_2$ ont été franchies) o devient faux. L'automate équivalent est présenté par la figure 3.3. On notera que pour des raisons de simplification nous n'avons pas modélisé les ϵ-transitions.

Les caractéristiques de cet automate sont les suivantes :

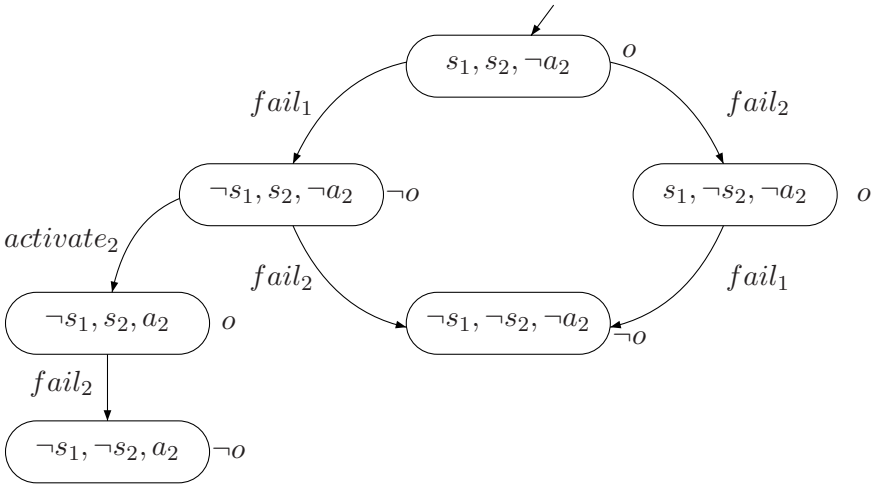


FIG. 3.3 – Automate de mode

- $V = S \cup F^{in} \cup F^{out}$ avec : $S = \{s_1, s_2, a_2\}$, $F^{in} = \{\emptyset\}$, $F^{out} = \{o\}$
- Toutes ces variables sont booléennes : $dom(s_i) = dom(a_2) = dom(o) = \{true, false\}$
- Les événements sont : $\Sigma = \{fail_1, fail_2, activate_2\}$
- Les transitions sont :
  $\delta(s_1, s_2, \neg a_2, fail_1) = \langle \neg s_1, s_2, \neg a_2 \rangle$,
  $\delta(s_1, s_2, \neg a_2, fail_2) = \langle s_1, \neg s_2, \neg a_2 \rangle$,
  $\delta(s_1, \neg s_2, \neg a_2, fail_1) = \langle \neg s_1, \neg s_2, \neg a_2 \rangle$,
  $\delta(\neg s_1, s_2, \neg a_2, fail_2) = \langle \neg s_1, \neg s_2, \neg a_2 \rangle$,
  $\delta(\neg s_1, s_2, \neg a_2, activate_2) = \langle \neg s_1, s_2, a_2 \rangle$ et
  $\delta(\neg s_1, s_2, a_2, fail_2) = \langle \neg s_1, \neg s_2, a_2 \rangle$
- L'assertion est : $o = (s_1 \vee (s_2 \wedge a_2)$. Donc σ est telle que :
  $\sigma(s_1, s_2, \neg a_2) = \langle o \rangle$
  $\sigma(s_1, \neg s_2, \neg a_2) = \langle o \rangle$
  $\sigma(\neg s_1, s_2, a_2) = \langle o \rangle$
  $\sigma(\neg s_1, s_2, \neg a_2) = \langle \neg o \rangle$
  $\sigma(\neg s_1, \neg s_2, \neg a_2) = \langle \neg o \rangle$
  $\sigma(\neg s_1, \neg s_2, a_2) = \langle \neg o \rangle$
- L'état initial est : $(s_1, s_2, \neg a_2)$

Après avoir présenté succinctement les automates de mots finis de la théorie classique des automates, nous allons présenter ceux que l'on utilise dans le domaine de la vérification : on cherche à caractériser des comportements sur des traces d'exécutions infinies et on est donc amené à travailler sur des automates de mots infinis.

3.2.3 Automates de mots infinis

Définition 7 (Automate multi-Büchi [Var96, VW86]) *Un automate multi-Büchi est un 6-uplet $B = \langle S, S_0, P, \mathcal{L}, T, \mathcal{F} \rangle$ avec S un ensemble fini d'états, $S_0 \subseteq S$ un ensemble d'états initiaux, P un ensemble fini de propositions atomiques sur les états, $\mathcal{L} : S \to 2^{2^P}$ une fonction*

d'étiquetage des états, $T \subseteq S \times S$ une relation de transition et $\mathcal{F} = \{F_1, ..., F_k\}$ un ensemble d'ensembles d'états acceptants. Notons que pour $k = 1$ un tel automate est un automate de Büchi.

Soit π une suite infinie d'états d'un automate multi-Büchi B, nous noterons $inf(\pi) \subseteq S$ l'ensemble des états qui se produisent une infinité de fois dans π. Par conséquent, π est un chemin accepté par B s'il contient infiniment souvent un état de chaque F_i c'est à dire, formellement, si $\forall i = 1, ..., k,\ inf(\pi) \cap F_i \neq \emptyset$. Nous noterons L_ω l'ensemble de ces chemins acceptés par l'automate. Notons que tout multi-Büchi à N états est équivalent à un automate de Büchi à $(k+1)N$ états.

La vérification de modèle telle qu'introduite sommairement au chapitre 5.2 peut élégamment être traitée en calculant un produit d'automates de Büchi. En effet, si B_M représente l'automate de Büchi du système, c'est à dire la structure de Kripke modélisant le système avec $F = S$, et $B_{\neg\phi}$ celui de la négation de la propriété à vérifier, il suffit de calculer leur produit $B = B_M \times B_{\neg\phi}$ et de montrer que le langage $L_\omega(B)$ des mots reconnus par B est vide pour conclure que $M \models \phi$. De la même façon, il est également possible de décrire un système réactif comme un produit d'automates de Büchi. Nous allons donc maintenant introduire ce produit d'automates de mots infinis.

Définition 8 (Produit d'automates multi-Büchi) *Soient $B_1 = \langle S_1, S_{0_1}, P_1, \mathcal{L}_1, T_1, \mathcal{F}_1 \rangle$ avec $\mathcal{F}_1 = \{F_{1_1}, ..., F_{1_k}\}$ et $B_2 = \langle S_2, S_{0_2}, P_2, \mathcal{L}_2, T_2, \mathcal{F}_2 \rangle$ avec $\mathcal{F}_2 = \{F_{2_1}, ..., F_{2_l}\}$ deux automates multi-Büchi, le produit $B_1 \times B_2$ est un automate multi-Büchi $B = \langle S, S_0, P, \mathcal{L}, T, \mathcal{F} \rangle$ avec :*

- $S = S_1 \times S_2$
- $P = P_1 \cup P_2$
- $\mathcal{L} : (S_1 \times S_2) \rightarrow 2^P$
- $(s, b) \times (s', b') \in T \iff (s, s') \in T_1$ *et* $(b, b') \in T_2$
- $S_0 = S_{0_1} \times S_{0_2}$
- $\mathcal{F} = \{F_{1_1} \times S_2, ..., F_{1_k} \times S_2, S_1 \times F_{2_1}, ..., S_1 \times F_{2_l}\}$

Exemple Soient deux automates de Büchi B_{ϕ_1} et B_{ϕ_2} dont les caractéristiques sont contenues dans la table 3.1. La définition 8 nous permet de calculer aisément l'automate de Büchi correspondant au produit $B_{\phi_1} \times B_{\phi_2}$.

Les états initiaux S_0 de $B_{\phi_1} \times B_{\phi_2}$ sont $\{(1,1),(1,2)\}$ et tous les états sont acceptants puisque $F_2 = S_2$. Partons d'un état initial et essayons de trouver des transitions menant à un état acceptant. On remarque que la transition $((1,1),(2,1))$ est possible car $(1,2) \in T_1$ et $(1,1) \in T_2$. De même, $((2,1),(3,1)) \in T$ car $(2,3) \in T_1$ et $(1,1) \in T_2$. Par conséquent, $\pi = (1,1),(2,1),(3,1)$ est un chemin de $B_{\phi_1} \times B_{\phi_2}$.

Une fois le modèle du système réalisé, il est intéressant de pouvoir spécifier ses propriétés afin de décrire son comportement.

B_{ϕ_1}	B_{ϕ_2}
$S_1 = \{1, 2, 3, 4\}$	$S_2 = \{1, 2, 3, 4\}$
$S_{0_1} = \{1\}$	$S_{0_2} = \{1, 2\}$
$P_1 = \{o, fail_1\}$	$P_2 = \{s_1, a_2\}$
$\mathcal{L}_1 = \{1 \mapsto T,$ $2 \mapsto T,$ $3 \mapsto \{fail_1\},$ $4 \mapsto \{o\}\}$	$\mathcal{L}_2 = \{1 \mapsto \{s_1\},$ $2 \mapsto T,$ $3 \mapsto \{a_2\},$ $4 \mapsto \{s_1, a_2\}\}$
$T_1 = \{(1,2), (1,3), (1,4), (2,2),$ $(2,3), (3,3), (4,4)\}$	$T_2 = \{(1,1), (1,2), (2,3), (2,4),$ $(3,3), (4,1), (4,2)\}$
$F_1 = \{3, 4\}$	$F_2 = \{1, 2, 3, 4\}$

TAB. 3.1 – Eléments de calcul de $B_{\phi_1} \times B_{\phi_2}$

3.3 Spécification des propriétés et exigences

3.3.1 De nombreuses logiques

La spécification des propriétés permet d'exprimer formellement les caractéristiques réelles ou souhaitées du modèle. Les logiques modales [Eme95, Kri63, CCD98] qui nous intéressent ont été développées par les philosophes pour étudier les différents types (modes) de vérité afin de formaliser leur raisonnement. Alors que la logique classique cherche uniquement à déterminer la valeur de vérité des propositions, la logique modale cherche à déterminer leur manière d'être vraies ou fausses. Considérons par exemple la propostion "*le système tombe en panne*". Il est possible de modifier cette proposition à l'aide de modes particuliers comme "*possible*" ou "*impossible*". Cette proposition devient alors :

- "Il est *possible* que le système tombe en panne"
- "Il est *impossible* que le système tombe en panne"

On peut considérer ces opérateurs comme des opérateurs unaires qui, appliqués à des propositions, en donnent de nouvelles. Sur cet exemple on voit clairement la faiblesse de la logique propositionnelle. Dans cette classe de logiques dites modales, on trouve également logique déontique qui comme son nom l'indique porte sur le devoir (obligation, interdiction, permission, facultatif), la logique doxastique portant sur la croyance et les logiques temporelles [Pnu81, MP92] qui nous intéressent tout particulièrement pour capturer l'ordre d'occurrence des événements dans les propriétés et nous permet de parler de l'évolution d'un système en qualifiant ou en questionnant son futur. Contrairement aux logiques propositionnelles qui ne permettent d'exprimer que des propriétés locales, totalement indépendantes de la structure du système de transition, les logiques temporelles permettent d'exprimer des propriétés sur des chemins d'exécution ce qui nous intéresse pour étudier l'évolution de nos modèles.

Une logique temporelle est donc une sorte de logique modale permettant de décrire le changement des valeurs de vérité de propositions au cours du temps sur des séquences d'états. Dans ce type de logiques, plusieurs modalités aussi appelées opérateurs temporels, permettent cette description. L'opérateur *toujours*, par exemple, signifie que la proposition est vraie à l'instant

présent et le sera également dans tous les états futurs. Ces opérateurs nous permettront d'encoder les propriétés dynamiques des systèmes que nous étudions. En effet, lors de nos études de sûreté de fonctionnement, il est important de pouvoir distinguer les états transitoires des états stables. Suite à la perte d'un système, qualifier de défaillants les états qui précèdent une reconfiguration ne paraît pas des plus judicieux. En utilisant le mode *toujours*, une perte permanente, plaçant le système dans un état stable, peut alors s'encoder simplement : *toujours*(perte du système). Tout comme dans le cas des logiques modales, il existe plusieurs logiques temporelles : propositionnelle, du premier ordre, linéaire ou arborescente, du passé ou du futur ...

Propositionnelle ou du premier ordre ? La logique temporelle propositionnelle est construite en étendant la logique propositionnelle classique par des opérateurs temporels. La logique temporelle du premier ordre étend les propositions atomiques de la logique temporelle propositionnelle en construisant des expressions à partir de variables, constantes, fonctions, prédicats et quantificateurs. Bien que l'expressivité de la logique temporelle du premier ordre soit supérieure, nous avons choisi de nous intéresser à la logique temporelle propositionnelle car elle est suffisante pour la spécification de nos propriétés. Considèrons un système construit à partir de plusieurs composants. La perte d'un seul composant ne doit pas entraîner la perte du système. Voici comment il est possible de spécifier cette propriété.

- Logique temporelle propositionnelle : *toujours*("perte d'un composant" $\rightarrow \neg$"perte du système")
- Logique temporelle du premier ordre : $[\forall x, \forall y, (composant(x) \wedge perte(x) \wedge composant(y) \wedge perte(y)) \rightarrow x = y] \rightarrow \neg$ perte(système)

Linéaire ou arborescente ? Faisons une analogie un peu originale et étrange en assimilant le temps à une fourmi. Celle-ci ne peut donc qu'avancer. Si cette fourmi évolue sur un arbre de taille infini, il arrivera un moment où plusieurs futurs seront possibles : prendre la branche de droite, prendre celle de gauche, etc. Cette structure du temps est arborescente, on parle alors logiquement de temps arborescent. Si le monde de cette même fourmie se limite à un fil infini, un seul futur est possible. On parle alors de temps linéaire. La logique du temps arborescent possède des modalités qui lui sont propres car il est nécessaire de pouvoir quantifier existentiellement et universellement sur plusieurs futurs.

3.3.2 La logique temporelle linéaire propositionnelle à états et événements

Les propriétés et exigences que nous manipulons traitent de l'aspect dynamique des systèmes que nous avons modélisés. Nous avons donc cherché la logique temporelle la plus simple permettant de répondre à nos besoins. L'expressivité de LTL [Pnu81, MP92], fragment de CTL*, nous permet de manipuler aisément ce type de propriétés dynamiques. De plus son aspect linéaire nous convient car nous raisonnons sur des chemins d'exécutions. Enfin, LTL permet de décrire aisément les états d'un système que l'on souhaite analyser. On peut noter l'existence de sucres syntaxiques, comme Sugar [BBDE$^+$01], qui facilitent la spécification de propriétés en LTL. Le pricipal inconvénient de cette logique est qu'elle ne permet pas parler des événements. Or il est intéressant dans nos études d'identifier les événements qui conduisent à des états redoutés et ceci est moins naturel en LTL.

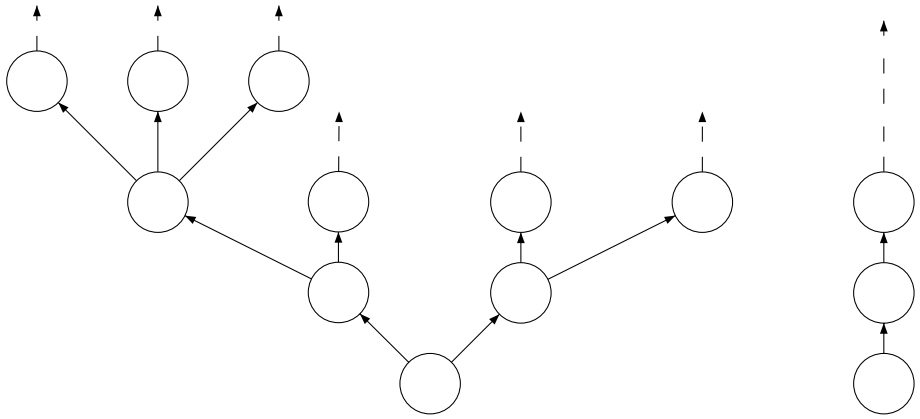


FIG. 3.4 – Structures du temps : arborescent (a) et linéaire (b)

Voici un exemple de spécification que nous avons rencontrée lors de nos études : "*Lorsqu'une pompe hydraulique reçoit un ordre d'activation, qu'elle possède les ressources nécessaires, du fluide en entrée et qu'elle ne se trouve pas dans un état défaillant alors elle fournit de la puissance hydraulique en sortie*". Cette spécification implique à la fois des états et des événements [KV98].

Partant de l'observation que les techniques usuelles de modélisation représentent les machines à états finis par des graphes d'états ou d'événements dirigés et étiquetés, et que l'encodage des événements dans les états (ou inversement) entraîne une augmentation importante de l'espace d'état, Chaki, Clarke et al. ont défini dans [CCO+04] un cadre dans lequel l'expression, la composition et la vérification de propriétés portant à la fois sur les états et sur les événements sont possibles. Ce cadre correspond à une structure de Kripke étiquetée (LKS) dont la logique de spécification est une variante de la logique temporelle linéaire qui prend en compte les événements comme des propositions atomiques. Nous utiliserons ce cadre pour spécifier ainsi que pour intégrer des propriétés temporelles aux automates à contraintes.

Syntaxe

Voici la syntaxe d'une logique permettant de se référer facilement aux états et aux événements, appelée SE-LTL, présentée dans [CCO+04] :

$$\phi ::= p \mid a \mid \neg\phi \mid \phi \wedge \phi \mid X\phi \mid G\phi \mid F\phi \mid \phi U \phi$$

a correspond aux événements, également considérés comme propositions atomiques dans SE-LTL. X est l'opérateur *Prochain* tel que Xp est vraie si p est vraie à l'instant suivant. F est

l'opérateur *Futur* tel que la formule Fp est vraie s'il existe un instant dans le futur (présent inclus) où la formule p est vraie et G est l'opérateur *Toujours* (ou *Globalement*) tel que la formule Gp est vraie si la formule p est vraie dans tous les instants futurs (présent inclus). Enfin, U est l'opérateur *Jusqu'à* tel que pUq est vraie si p est vraie jusqu'à ce que q le soit.

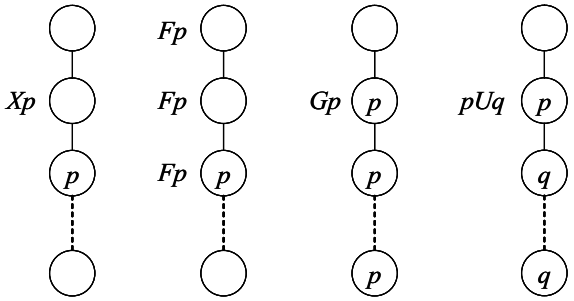


FIG. 3.5 – Opérateurs temporels

Sémantique

Définition 9 (Structure de Kripke étiquetée [CCO+04]) *Une structure de Kripke étiquetée est un 7-uplet $\langle S, S_0, P, \mathcal{L}, T, \Sigma, \varepsilon\rangle$ avec S un ensemble fini d'états, $S_0 \subseteq S$ un ensemble d'états initiaux, P un ensemble fini de proposition atomique sur les états, $\mathcal{L} : S \to 2^P$ une fonction d'étiquetage des états, $T \subseteq S \times S$ une relation de transition, Σ un ensemble fini d'événements et $\varepsilon : T \to (2^\Sigma \setminus \{\emptyset\})$. Les états sont donc étiquetés par un ensemble de propositions atomiques et les transitions par un ensemble non vide d'événements.*

Définition 10 (Chemin) *Un* chemin $\pi = s_1, a_1, s_2, a_2, \ldots$ *d'un LKS est une séquence infinie alternante d'états et d'événements telle que $\forall i \geqslant 1, s_i \in S, a_i \in \Sigma$ et $s_i \xrightarrow{a_i} s_{i+1}$. On note π^i le suffixe de π à partir de à l'état s_i. La relation de satisfaction de formules SE-LTL est définie ainsi :*

- $\pi \models p$ *ssi* s_1 *est le premier état de* π *et* $p \in \mathcal{L}(s_1)$
- $\pi \models a$ *ssi* a *est le premier événement de* π
- $\pi \models \neg\phi$ *ssi* $\pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2$ *ssi* $\pi \models \phi_1$ *et* $\pi \models \phi_2$
- $\pi \models X\phi$ *ssi* $\pi^2 \models \phi$
- $\pi \models G\phi$ *ssi* $\forall i \geqslant 1, \pi^i \models \phi$
- $\pi \models F\phi$ *ssi* $\exists i \geqslant 1, \pi^i \models \phi$
- $\pi \models \phi_1 U \phi_2$ *ssi* $\exists i \geqslant 1$ *tel que* $\pi^i \models \phi_2$ *et* $\forall j, 1 \leqslant j \leqslant i-1, \pi^j \models \phi_1$

Définition 11 (Langage) *Le* langage *d'un LKS M (noté $L(M)$) est l'ensemble des chemins maximaux de M dont l'état initial appartient à l'ensemble des états initiaux de M.*

Définition 12 (Satisfaction) *Soit M un LKS et ϕ une formule de SE-LTL. Alors, $M \models \phi$ ssi $\forall \pi \in L(M)$, $\pi \models \phi$.*

Définition 13 (Validité) *On dit qu'une formule ϕ est valide ssi pour tout LKS M, $M \models \phi$. On note alors $\models \phi$.*

Définition 14 (Composition parallèle [CCO+04]) *La composition parallèle de deux LKS est définie de la manière suivante. Soient M_1 et M_2 deux LKS tels que $P_{M_1} \cap P_{M_2} = \emptyset$ alors leur composition parallèle, notée $M_1 \parallel M_2$ est un LKS tel que :*

- $S_{M_1 \| M_2} = S_{M_1} \times S_{M_2}$
- $S_{0_{M_1 \| M_2}} = S_{0_{M_1}} \times S_{0_{M_2}}$
- $P_{M_1 \| M_2} = P_{M_1} \cup P_{M_2}$
- $\Sigma_{M_1 \| M_2} = \Sigma_{M_1} \cup \Sigma_{M_2}$

De plus, $\forall s_1, s_1' \in S_{M_1}, \forall s_2, s_2' \in S_{M_2}$ et $a \in \Sigma_{M_1 \| M_2}$ la fonction d'étiquetage $\mathcal{L}_{M_1 \| M_2}$ et la relation de transition $T_{M_1 \| M_2}$ sont définis de la manière suivante :

- $\mathcal{L}_{M_1 \| M_2} = \mathcal{L}_{M_1}(s_1) \cup \mathcal{L}_{M_2}(s_2)$
- *si* $s_1 \xrightarrow{a} s_1'$ *et* $s_2 \xrightarrow{a} s_2'$ *alors* $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$
- *si* $s_1 \xrightarrow{a} s_1'$ *et* $a \notin \Sigma_{M_2}$ *alors* $(s_1, s_2) \xrightarrow{a} (s_1', s_2)$
- *si* $a \notin \Sigma_{M_1}$ *et* $s_2 \xrightarrow{a} s_2'$ *alors* $(s_1, s_2) \xrightarrow{a} (s_1, s_2')$

Exemple Reprenons l'exemple de la source présenté précédemment. La figure 3.6 représente un LKS équivalent à cette source. Considérons maintenant deux sources identiques à celle-ci et composons les LKS.

Notons M_1 et M_2 les LKS des deux sources. Leurs caractéristiques sont les suivantes :

- $S_i = \{a_i, b_i\}$
- $S_{0_i} = \{a_i\}$
- $P_i = \{o_i\}$
- La fonction d'étiquetage des états est :

$$\mathcal{L}_i = \begin{cases} a_i \mapsto \{o_i\} \\ b_i \mapsto \emptyset \end{cases}$$

- $\Sigma_i = \{fail_i\}$
- $T_i = \{(a_i, b_i)\}$
- $\varepsilon_i : (a_i, b_i) \mapsto fail_i$

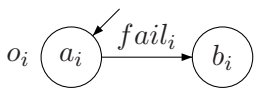


FIG. 3.6 – LKS d'une source M_i

Nous obtenons alors $M_3 = M_1 \parallel M_2 = (S_3, S_{0_3}, P_3, \mathcal{L}_3, T_3, \Sigma_3, \varepsilon_3)$ (figure 3.7) avec :

- $S_3 = \{(a_1, a_2), (a_1, b_2), (b_1, a_2), (b_1, b_2)\}$
- $S_{0_3} = \{(a_1, a_2)\}$
- $P_3 = \{o_1, o_2\}$
- La fonction d'étiquetage des états est :

$$\mathcal{L}_3 = \begin{cases} (a_1, a_2) \mapsto \{o_1, o_2\} \\ (a_1, b_2) \mapsto \{o_1\} \\ (b_1, a_2) \mapsto \{o_2\} \\ (b_1, b_2) \mapsto \emptyset \end{cases}$$

- $\Sigma_3 = \{fail_1, fail_2\}$
- $T_3 = \{((a_1, a_2), (a_1, b_2)), ((a_1, a_2), (b_1, a_2)), ((a_1, b_2), (b_1, b_2)), ((b_1, a_2), (b_1, b_2))\}$
- La fonction d'étiquetage des transitions est :

$$\varepsilon_3 = \begin{cases} ((a_1, a_2), (a_1, b_2)) \mapsto \{fail_2\} \\ ((a_1, a_2), (b_1, a_2)) \mapsto \{fail_1\} \\ ((a_1, b_2), (b_1, b_2)) \mapsto \{fail_1\} \\ ((b_1, a_2), (b_1, b_2)) \mapsto \{fail_2\} \end{cases}$$

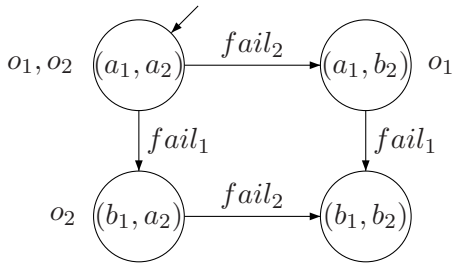


FIG. 3.7 – Composition parallèle de deux LKS

Une fois nos hypothèses, propriétés et exigences traduites en SE-LTL, il est possible de les représenter sous formes d'automates de mots infinis. Pour cela une passerelle entre LTL et les automates existe. En effet, Wolper, Vardi et Sisla ont montré qu'une formule de LTL φ peut être traduite en un automate sur mots infinis A, dit de Büchi, tel que $L_\omega(A_\varphi)$ est exactement l'ensemble des executions satisfaisant la formule φ. Par conséquent, une formule de SE-LTL peut également être traduite en automate de Büchi puisque une formule ϕ de SE-LTL sur P (ensemble des propositions atomiques d'états) et Σ (ensemble fini d'événements) peut être interprétée comme une formule de LTL sur $P \cup \Sigma$, "nouvel" ensemble de propositions atomiques sur les états. Syntaxiquement, les formules SE-LTL et LTL sont identiques, elles diffèrent uniquement dans leur interprétation sémantique.

Exemple L'automate de la figure 3.3 montre que si le composant 1 ne défaille pas alors le système fournit toujours sa sortie. Cette propriété garantie (assez faible) peut se traduire en SE-LTL par $\phi_1 : \neg(Ffail_1) \Rightarrow Go$. L'automate de Büchi équivalent à cette propriété est présenté sur la figure 3.8.

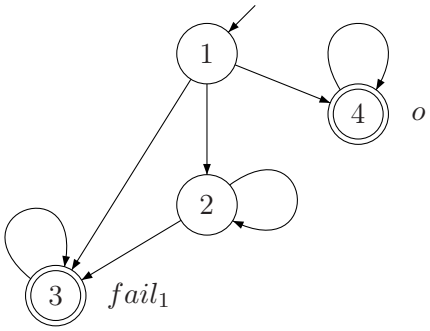


FIG. 3.8 – Automate de Büchi de ϕ_1

Nous allons maintenant présenter le langage formel de modélisation que nous avons utilisé pour la réalisation de nos cas d'étude. La sémantique de ce langage est celui des automates à contraintes que nous avons présentés précédemment.

3.4 Le langage AltaRica

3.4.1 Présentation générale

ALTARICA est un langage créé par le Laboratoire Bordelais de Recherche en Informatique (LaBRI) au milieu des années 90 [APGR00, Poi00]. Ce langage formel nous intéresse pour la modélisation de systèmes en vue d'une analyse de sûreté car il permet une modélisation à la fois fonctionnelle et dysfonctionnelle. Cette vue dysfonctionnelle est permise grâce à l'introduction d'événements modélisant la défaillance des composants. De plus, sa capacité à réaliser des modèles compositionnels et hiérarchiques lui permet de modéliser des systèmes complexes, classe à laquelle appartiennent les cas d'étude que nous avons modélisés. Enfin, ALTARICA est un langage à la fois formel et graphique grâce aux ateliers OCAS (Outil de Conception et d'Analyse Système) développé par *Dassault* et SIMFIA développé par EADS Apsys qui facilitent la programmation, la simulation ainsi que l'analyse des modèles.
Un des principaux avantage de ce langage est sa polyvalence. En effet, les modèles ALTARICA peuvent se traduire en divers autres formalismes, comme le montre la figure 3.9. De plus, des passerelles vers d'autres langages ont également été réalisées et autorisent l'utilisation des outils de vérification et de validation disponibles pour ces langages. A titre d'exemple, notons la réalisation d'une passerelle vers Lustre développée au LaBRI, ainsi qu'une passerelle vers le langage d'entrée du vérificateur de modèles SMV qui a été réalisée à l'ONERA-Cert.

Différents dialectes pour différents outils Le langage ALTARICA est un langage ouvert. Il a ainsi, entre autre, été adopté par Dassault et EADS qui l'utilisent comme langage de modélisation dans leurs ateliers. Cette dissémination du langage a mené à des évolutions propres à chaque utilisateur. A l'heure actuelle, on dénombre trois dialectes du langage principal, que l'on nommera ALTARICA standard et qui est basé sur les automates à contraintes comme nous venons de le voir. De cette base sont nées des restrictions et des extensions. L'IRCCyN a ainsi développé une extension temps réel d'ALTARICA [Pag04] basée sur les automates à contraintes

temporisés qui se compilent en automates temporisés. L'outil Tarc permet de réaliser cette compilation. Une version à flot de données, appelée AltaRica Data Flow [Rau02, MR98] propose une restriction du langage d'origine en vue de faciliter la compilation. Le nombre de variables autorisé est fini, les variables d'état appartiennent à des intervalles finis d'entiers et les flux ne sont plus bidirectionnels. On a également une simplification de la notion de priorité entre événements. On passe alors du côté des automates de mode. Cette version allégée est utilisée par la ToolBox d'Arboost Technologies d'Antoine Rauzy. Enfin, la dernière version est celle utilisée dans Cecilia OCAS. C'est également une restriction du langage de base dans le sens où elle correspond à la version Data Flow étendue par des macros. Des rapprochements au niveau de la syntaxe des différents dialectes sont en cours. Dans cette thèse, nous nous intéresserons à la version Data Flow utilisée au sein de l'outil OCAS que nous avons utilisé pour réaliser nos modèles et certaines analyses.

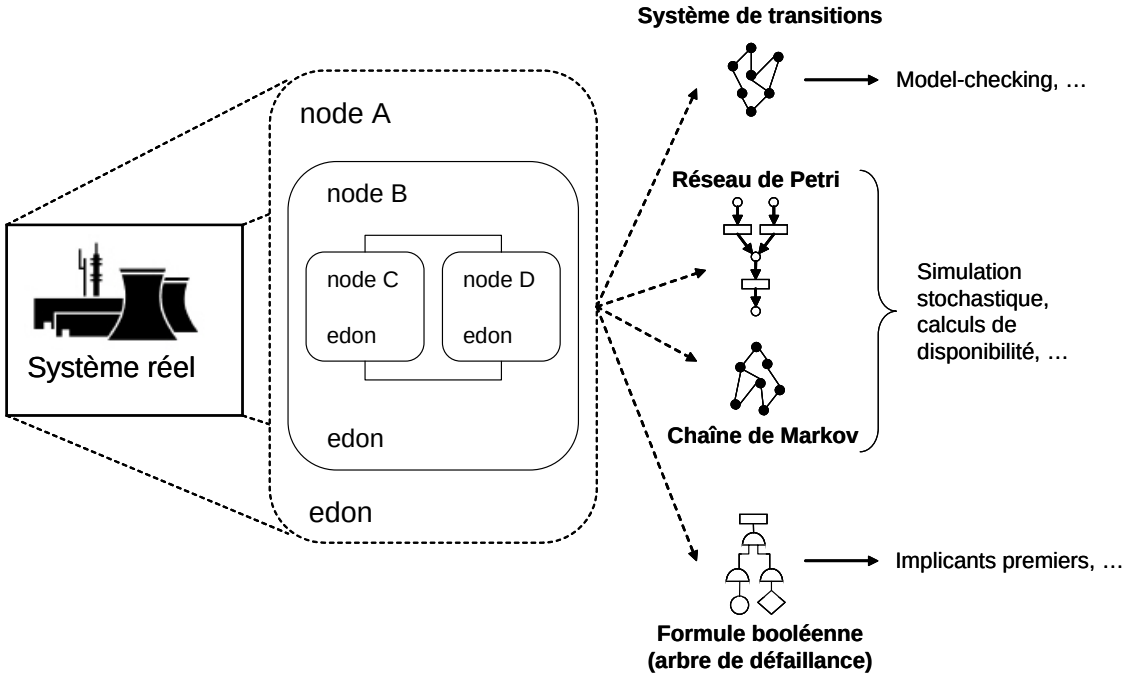


Fig. 3.9 – Atelier AltaRica

3.4.2 Concepts

AltaRica hiérarchise les composants qui sont considérés comme des nœuds (`node`). On peut ainsi utiliser un nœud à l'intérieur d'un autre nœud (`sub`) et les faire communiquer. Chaque nœud possède un nombre fini de variables internes dites d'état (`state`), dont les valeurs sont locales et isolées de l'environnement dans lequel elles se trouvent, et également un nombre fini de variables de flux (`flow`) qui font le lien entre le composant et son environnement. Très souvent ces deux types de variables sont liés. Un point important d'Altarica est que les valeurs des variables sont toujours discrètes. De plus, chaque nœud peut être décrit comme un automate à contraintes. En effet, le comportement du composant est décrit par des contraintes sur les changements d'états, appelés `transitions`, où la satisfaction d'une

garde composée de variables d'entrée et d'état, ainsi que l'occurrence d'un événement (`event`) permet de changer d'état. Il peut également changer d'état interne par des transitions causées par des événements spéciaux (notés ϵ) qui ne sont pas spécifiés dans le modèle lui même mais définis au niveau sémantique et qui permettent de garantir qu'un nœud ne bloque pas. On peut également utiliser des contraintes globales sur les variables, appelées `assertions` qui permettent de valuer les sorties du composant en fonction de variables d'entrée et d'état. Notons que trois types de constantes peuvent être utilisées : les constantes entières, énumérées et booléennes. L'état intial du composant est déclaré après le mot clé `init`. Il est également possible de fournir, dans le code ALTARICA, des informations utiles à des outils de traitement externe. Ces informations pourront correspondre à des probabilités d'occurrence associées à des événements. Elles seront ensuite traitées par des outils de traitement de coupes minimales comme SimTree, par exemple. Ces données supplémentaires sont entrées après le mot clé `extern` dans le code ALTARICA.

Exemple On souhaite modéliser une source électrique. Son état binaire (fonctionnement ou dysfonctionnement) sera représenté par une variable d'état notée `s`. Une sortie de ce système, notée `o`, fournit une valeur booléenne en fonction de l'état de la source. Une défaillance `fail` peut entraîner la perte du système. Dans ce cas le système ne fournit plus de sortie. Voici le code correspondant à ce composant.

```
node source
 state s1:bool;
 flow o:out:bool;
 event fail;
 trans s1 |- fail -> s1:=false;
 assert o=s1;
 init s1:=true;
edon
```

La syntaxe ALTARICA associée à l'exemple de la redondance dont la sémantique est représentée sur la figure 3.3 et qui utilise le composant `source` est la suivante :

```
node cold_redundancy
 state a2, s2:bool;
 flow o:out:bool;
 event fail2, activate2;
 sub c1:source;
 trans s2 |- fail2 -> s2:=false;
  not(c1.s1) |- activate2 -> a2:=true;
 assert o=(c1.s1 or (s2 and a2));
 init a2:=false, s2:=true;
edon
```

Un système, tout comme son modèle, est un assemblage de composants élémentaires. Nous allons maintenant présenter comment assembler ces composants, représentés sous forme d'automates de modes, pour obtenir le modèle global du système. Cette assemblage est réalisé à l'aide de trois opérations que nous allons maintenant présenter.

3.4.3 Connection de composants

La connection de composants, i.e. d'automates de modes, consiste à contraindre certaines variables d'entrée d'un composant à être égales à une sortie d'un autre composant. Pour que cette connection soit valide, il est nécessaire que ces variables soient indépendantes, c'est à dire que les variables d'entrée n'interviennent pas dans les calculs de la valeur de la variable de sortie. D'un point de vue pratique, une connection peut être assimilée à un renommage de certaines variables.

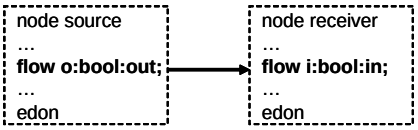


FIG. 3.10 – Connection

Soit $\mathcal{V}$ un jeu de valuation des variables, nous noterons dans la suite $\mathcal{V}[u]$ la valeur de la variable u dans cette valuation $\mathcal{V}$. Soit $A_i = \langle D, S_i, F_i^{in}, F_i^{out}, dom, \Sigma_i, \delta_i, \sigma_i, I_i \rangle$ un automate de mode, $o \in F^{out}$ et $i_1, ..., i_k \in F^{in}$ telles que o et $i_1, ..., i_k$ soient indépendants et $dom(o) \subseteq dom(i_1), ..., dom(o) \subseteq dom(i_k)$. Soient $F_\star^{in} = F^{in} \setminus \{i_1, ..., i_k\}$, $s \in dom(S)$, et $I \in dom(F_\star^{in})$. Notons $I_{s,o/i_1,...,o/i_k}$ la valuation de F^{in} telle que :

$$I_{s,o/i_1,...,o/i_k}[u] \stackrel{def}{=} \begin{cases} I[u] & \text{si } u \in F_\star^{in} \\ \sigma(s, I')[o] & \text{sinon} \end{cases}$$

Avec I' n'importe quelle extension de I dans la valuation de F^{in}. L'automate de mode A où $i_1, ..., i_k$ sont connectés à o est l'automate de mode $A_{o/i_1,...,o/i_n} = \langle D, S, F_\star^{in}, F^{out}, dom, \Sigma, \delta', \sigma', I \rangle$ avec δ' et σ' tels que :

- $\forall s \in dom(S), I \in dom(F_\star^{in})$ et $e \in \Sigma$ si $\delta(s, I_{s,o/i_1,...,o/i_n}, e)$ est défini, alors $\delta'(s, I, e) = \delta(s, I_{s,o/i_1,...,o/i_n}, e)$
- $\forall s \in dom(S), I \in dom(F_\star^{in}), \sigma'(s, I) = \sigma(S, I_{s,o/i_1,...,o/i_n})$

3.4.4 Composition parallèle

La composition consiste à placer plusieurs nœuds ALTARICA dans un même cadre, une même vue afin de pouvoir les faire communiquer et interagir.
Soient $A_1, ..., A_n$ n automates de mode, avec $A_i = \langle D, S_i, F_i^{in}, F_i^{out}, dom, \Sigma_i, \delta_i, \sigma_i, I_i \rangle$ et tels que leur vocabulaires soient distincts, i.e. $\forall i, j, 1 \leqslant i \leqslant j \leqslant n, (S_i \cup F_i^{in} \cup F_i^{out}) \cap (S_j \cup F_j^{in} \cup F_j^{out}) = \emptyset$ et $\Sigma_i \cap \Sigma_j = \emptyset$. Alors la composition parallèle (parfois appelée aussi produit libre) de $A_1, ..., A_n$ est l'automate de mode $A = \langle D, S, F^{in}, F^{out}, dom, \Sigma, \delta, \sigma, I \rangle$ tel que :

- $S = \bigcup_{i=1}^n S_i$, $F^{in} = \bigcup_{i=1}^n F_i^{in}$, $F^{out} = \bigcup_{i=1}^n F_i^{out}$, $\Sigma = \bigcup_{i=1}^n \Sigma_i$
- δ est obtenu en remontant les δ_i dans A, i.e. soient $s_1 \in dom(S_1), ..., s_n \in dom(S_n)$, soient $i_1 \in dom(F_1^{in}), ..., i_n \in dom(F_n^{in})$, soit $t_i \in dom(S_i)$ et $e \in \Sigma_i$ tel que $t_i = \delta_i(S_i, I_i, e)$ alors :

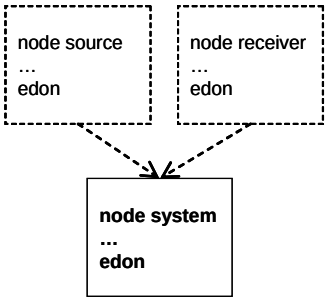


FIG. 3.11 – Composition

$$\delta(s_1, ..., s_n, i_1, ..., i_n, e) = \langle s_1, ..., s_{i-1}, t_i, s_{i+1}, ..., s_n \rangle$$

– De la même manière, σ est obtenu en remontant les σ_i dans A,

$$\sigma(s_1, ..., s_n, i_1, ..., i_n) = \langle \sigma_1(s_1, i_1), ..., \sigma_n(s_n, i_n) \rangle$$

– $I = \langle I_1, ..., I_n \rangle$

3.4.5 Synchronisation

Les transitions dans les automates de mode sont asynchrones. L'opération de synchronisation [BR94] permet de tirer plusieurs transitions simultanément. L'ensemble des événements ainsi synchronisés sont regroupés au sein d'un vecteur de synchronisation. Nous allons tout d'abord présenter la notion de composition de valuations puis l'opération de synchronisation.

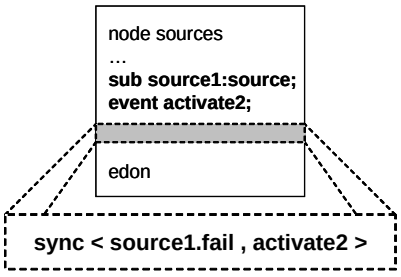


FIG. 3.12 – Synchronisation

Soit $S \subseteq V$ (ensemble fini de variables de l'automate), $\mathcal{V}$, $\mathcal{V}'$ et $\mathcal{V}''$ des jeux de valuation des variables. On dit que $\mathcal{V}'$ et $\mathcal{V}''$ sont incompatibles par rapport à $\mathcal{V}$ si il existe une variable $v \in S$ telle que $\mathcal{V}[v]$, $\mathcal{V}'[v]$ et $\mathcal{V}''[v]$ sont distinctes. La composition des deux valuations $\mathcal{V}'$ et $\mathcal{V}''$ par rapport à $\mathcal{V}$, notée $\mathcal{V}' \circ_{\mathcal{V}} \mathcal{V}''$, est la valuation définie par :

$$\forall v \in S,\ \mathcal{V}' \circ_{\mathcal{V}} \mathcal{V}'' \stackrel{def}{=} \begin{cases} \mathcal{V}'[v] & \text{si } \mathcal{V}'[v] \neq \mathcal{V}[v] \\ \mathcal{V}''[v] & \text{sinon} \end{cases}$$

Soit $A = \langle D, S, F^{in}, F^{out}, dom, \Sigma, \delta, \sigma, I\rangle$ un automate de mode et $\vec{e_1}, ..., \vec{e_r}$ r vecteurs de synchronisation. Σ' est le sous-ensemble de Σ qui contient les événements présents dans les vecteurs de synchronisation. la synchronisation de A par les $\vec{e_i}$ est un automate de mode $A\,|\vec{e_1}, ..., \vec{e_r} = \langle D, S, F^{in}, F^{out}, dom, (\Sigma \setminus \Sigma') \cup \{\vec{e_1}, ..., \vec{e_r}\}, \delta', \sigma, I\rangle$ avec :

- $\forall e \in \Sigma \setminus \Sigma'$, $s \in dom(S)$ et $I \in dom(F^{in})$ si $\delta(s, I, e)$ est définie alors $\delta'(s, I, e)$ est aussi définie et $\delta'(s, I, e) \stackrel{def}{=} \delta(s, I, e)$
- $\forall\ \vec{e_i} = \langle e_1, ..., e_k\rangle$, $s \in dom(S)$, et $I \in dom(F^{in})$, si $\delta(s, I, e_1), ..., \delta(s, I, e_k)$ sont définies et compatibles deux à deux, alors $\delta'(s, I, \vec{e_i}) \stackrel{def}{=} \delta(s, I, e_1) \circ_s ... \circ_s \delta(s, I, e_k)$

En AltaRica la notion de temps s'exprime par un ordonnancement des événements. Ainsi, il est possible d'exprimer des priorités en introduisant un ordre partiel sur les événements. On obtient alors un automate de mode avec priorités qui est un couple $(A, \prec)$ avec A un automate de mode et $\prec$ un ordre partiel. Bien entendu ces priorités ne priment pas sur les gardes des transitions.

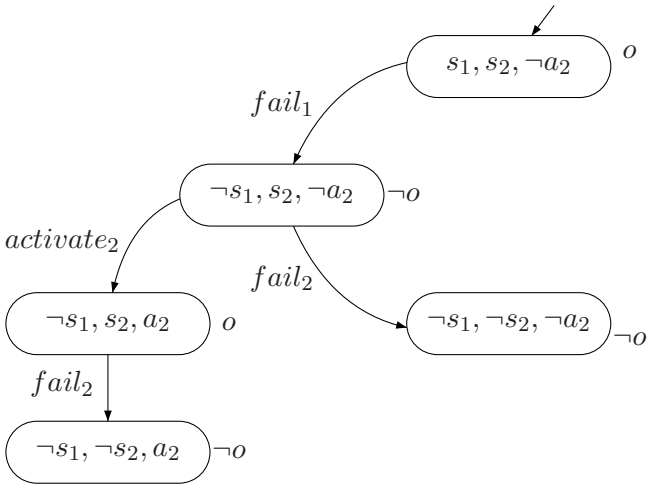


FIG. 3.13 – Automate à contraintes avec priorité $fail_1 \succ fail_2$

Troisième partie

Mise en œuvre

Chapitre 4

Motifs d'architectures de sûreté

4.1 Introduction

Dans ce chapitre, nous définissons formellement les motifs d'architectures de sûreté, que l'on nommera aussi Safety Architecture Patterns ou encore SAP. Les motifs que nous développons correspondent à des abstractions d'architectures réelles répondant à un problème précis, et ne pouvant être utilisées que dans un contexte précis. Ils correspondent généralement à des méchanismes de sûreté. Cette approche permet donc de réduire la complexité des modèles manipulés. Leurs objectifs sont d'assister la modélisation et l'évaluation de l'architecture de sûreté de fonctionnement des systèmes complexes, ainsi que d'exhiber les propriétés satisfaites par un système afin de les comparer aux exigences fixées par les documents de sûreté. La volonté de concevoir ce type de motifs a pour origine la simple constatation que, dans un monde idéal, si les systèmes pouvaient être conçus par les spécialistes des domaines à partir de "bons" composants, validés et réutilisables, alors les considérations sur la sûreté de fonctionnement de ces systèmes et architectures pourraient se ramener à l'utilisation correcte de ces composants génériques.

En formalisant les caractéristiques de ces architectures types, les problèmes qu'elles permettent de résoudre et leurs conditions d'utilisation, nous souhaitons pratiquer la réutilisation. En effet, la réutilisation de solutions sûres, tôt dans le cycle de développement, permet de simplifier les analyses de sûreté à mener. Pour atteindre ces objectifs, les motifs sont basés sur l'intégration de deux méthodes formelles bien connues. L'une permet de prendre en compte l'aspect modélisation comportementale ; l'autre permet de formaliser les propriétés qui posent des contraintes supplémentaires sur le comportement de l'architecture, explicite des contraintes d'utilisation ou les services rendus par l'architecture. Nous allons maintenant présenter les briques élémentaires utiles à la conception des modèles comportementaux des motifs.

4.2 Formes génériques réutilisables d'architectures

Un motif d'architecture de sûreté est une abstraction d'un mécanisme de sûreté mais reste néanmoins un composant du système qui doit interagir avec le reste de l'architecture. Par conséquent il possède, comme n'importe quel composant, des entrées, une partie fonctionnelle et des sorties. Ces motifs sont obtenus par assemblage de briques élémentaires (figure 4.1) répartis en quatre classes : entrée/sortie, fonctionnelle, test et contrôle. Les combinaisons de

ces briques de base nous permettent d'obtenir l'ensemble des modèles comportementaux que nous avons rencontrés lors de nos études. Nous allons maintenenant présenter ces classes.

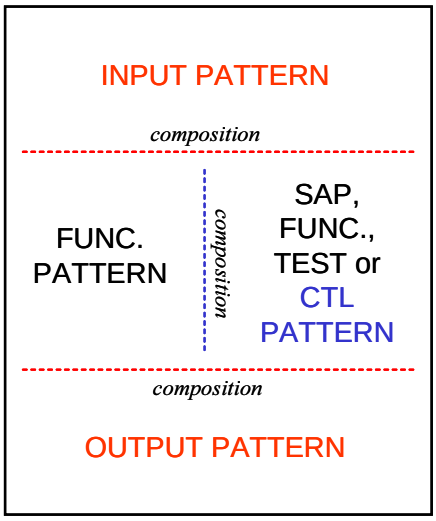


FIG. 4.1 – Conception d'un motif

4.2.1 Entrées/sorties

Les motifs élémentaires correspondant aux entrées et sorties sont au nombre de cinq. Le motif élémentaire qui vient tout de suite à l'esprit est celui constitué de deux voies indépendantes. Le *splitter* correspond à une conjonction, i.e. si l'entrée est vraie alors les deux sorties le sont également. Le motif *merger* est son dual : si au moins une des deux entrées est vraie alors la sortie est aussi vraie. Viennent ensuite deux derniers motifs plus complexes que nous avons rencontré lors de la phase de modélisation de plusieurs systèmes. Ainsi le *switch* autorise les reconfigurations en transmettant la valeur de l'entrée à une sortie ou à l'autre. Enfin, le motif à dépendance variable permet la transmission d'une entrée quelconque sur une sortie quelconque.
Prenons l'exemple du *splitter* d'entrée et de la sortie à dépendance variable. En voici un encodage possible :

```
node input_splitter
 flow i:in:bool;
  o1,o2:out:bool;
 assert o1 = i;
  o2 = i;
edon
```

```
node var_dependency_output
 flow i1,i2:in:bool;
  o1,o2:out:bool;
 assert o1 = (i1 or i2);
  o2 = (i1 or i2);
edon
```

4.2.2 Fonctions

Le motif élémentaire fonctionnel de base que nous avons construit comporte, outre une variable d'entrée (définie par un motif d'entrée), une variable de sortie (définie par un motif de sortie) et

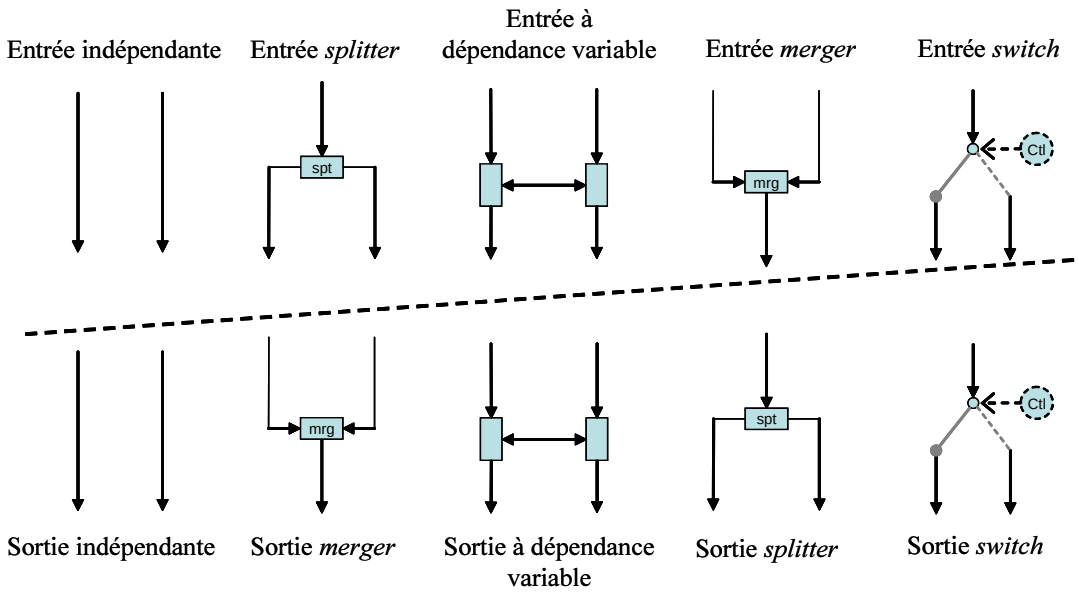


FIG. 4.2 – Motifs élémentaires d'entrée/sortie

Nom	Type	Signification
a	in	activation
f	in	défaillance
i	in	entrée
o	out	sortie
r	in	ressource
s	out	état

TAB. 4.1 – Variables d'un motif élémentaire fonctionnel

une variable d'état comme cela est précisé dans le tableau 4.1, il possède également une entrée concernant les ressources nécessaires à son fonctionnement (celles-ci peuvent être toujours disponibles ou bien fournies sur demande), une entrée d'activation du motif (celui-ci peut soit être toujours activé, soit contrôlé) et enfin une entrée correspondant à la défaillance du motif et l'on précisera son type (e.g. perte totale, dégradée, retard). Voici un exemple de code pour ce type de motif élémentaire :

```
node elem_function
 flow a, f, i, r:in:bool;
  o, s:out:bool;
 assert s = not(f),
  o = a and i and r and not(f);
edon
```

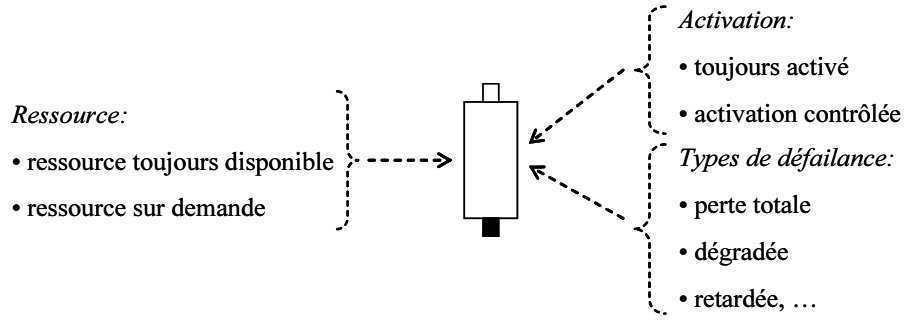


FIG. 4.3 – Motif élémentaire fonctionnel

4.2.3 Test

Nous avons identifié trois principales classes de patrons élémentaires de test : la détection de fonction perdue ou dégradée que l'on retrouve dans les mécanismes du type n-plication, les comparateurs que l'on retrouve dans les mécanismes de vote et le *watchdog* dans les systèmes temps réel, par exemple (figure 4.4).

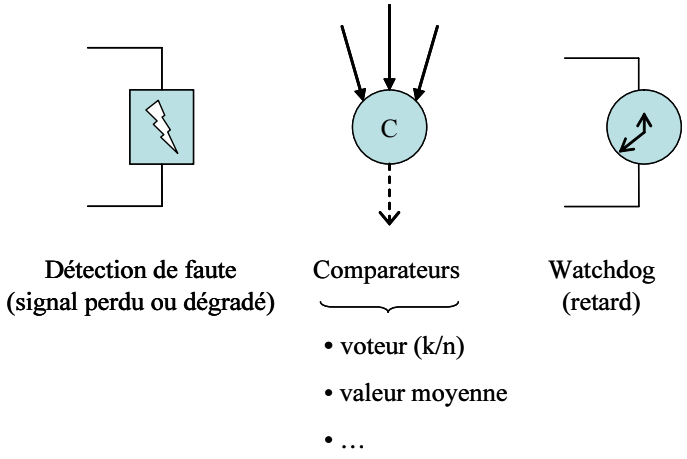


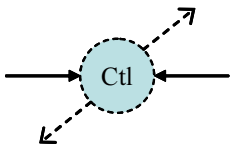
FIG. 4.4 – Motifs élémentaires de test

Ces motifs nous permettent de vérifier le mode de fonctionnement des motifs fonctionnels qui leur sont connectés. Prenons l'exemple du motif élémentaire de test *voteur 2/3*. Ce motif possède une entrée par motif fonctionnel testé et une sortie fournissant le résultat du vote. Voici un exemple de code simple correspondant à un motif de ce type.

```
node 2outof3_voter
 flow i1,i2,i3:in:bool;
  o:out:bool;
 assert o = (i1 and i2) or (i2 and i3) or (i1 and i3);
edon
```

4.2.4 Contrôle

L'activation de parties fonctionnelles (e.g. redondances) peut être réalisée à l'aide de patrons élémentaires de contrôle de la figure 4.5.



Contrôleur

FIG. 4.5 – Motif élémentaire de contrôle

Considérons un contrôleur permettant l'activation d'un composant (en état de marche) sur perte d'un autre composant.

```
node controller
 flow status1, status2:in:bool;
  activation:out:bool;
 assert activation = not(status1) and status2;
edon
```

L'expérience nous a montré que la composition de ces quelques briques élémentaires, dont la liste ci-dessus n'est bien entendu pas exhaustive, nous permet de modéliser la plupart des mécanismes de sûreté que nous avons rencontré lors de nos études. En effet, il est très aisé d'obtenir toutes sortes de n-plications ou encore de développer des structures plus complexes comme des architectures COM-MON.

4.2.5 Exemple

Dans cet exemple nous allons utiliser les motifs élementaires que nous venons de développer pour construire le modèle comportemental d'une redondance active basique, i.e. sans contrôle, ni test. Elle est composée d'une entrée de type `splitter`, de deux composants de type `elem_function` et d'une sortie de type `var_dependency_output`. L'assemblage de ces briques par composition parallèle donne le composant ALTARICA suivant :

```
node redundancy
 state C1_down, C2_down:bool;
 flow C1_failed, C2_failed:out:bool;
 event fail_C1, fail_C2;
 sub input:input_splitter;
  C1, C2:elem_function;
  output:var_dependency_output;
 trans not(C1.down) |- fail_C1 -> C1.down := true;
  not(C2.down) |- fail_C2 -> C2.down := true;
 assert C1.i = input.o1;
```

```
  C2.i = input.o2;
  output.i1 = C1.o1;
  output.i2 = C2.o2;
  C1.a = true;
  C1.r = true;
  C1.f = C1_down;
  C2.a = true;
  C2.r = true;
  C2.f = C2_down;
 init C1_down := false, C2_down := false;
edon
```

Ci-dessus, le mot clé `sub` permet de déclarer des instances de composants déjà définis. La partie assertion du code permet ici de faire le lien entre les différents composants de la hiérarchie.

4.2.6 Représentation graphique

Afin de faciliter la manipulation des motifs nous avons développer une signalétique permettant de les identifier au sein d'une architecture. Ainsi nous définissons le symbole $\diamond$ pour représenter une redondance active et les symboles $\triangleleft$ et $\triangleright$ pour les redondances passives. En cas de redondances multiples, ces symboles sont numérotés pour spécifier l'ordre de déclenchement des activations.

Sur la figure 4.6, le motif (a) est une redondance active entre les motifs fonctionnels A et B. Dans (b), B redonde passivement A et inversement dans (c).

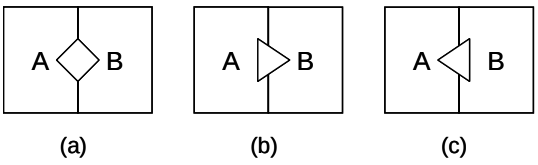


FIG. 4.6 – Représentation graphique des redondances

Quelques exemples de triplications sont présentés sur la figure 4.7. Sur la triplication (a), A et B se redondent activement et C redonde passivement B. L'ordre des redondances signifie que suite à une défaillance de B, A prend le relai. Si A défaille alors une défaillance de B entraîne l'activation de C. La seconde triplication consiste en une redondance active entre A et C et B les redonde tous les deux passivement.

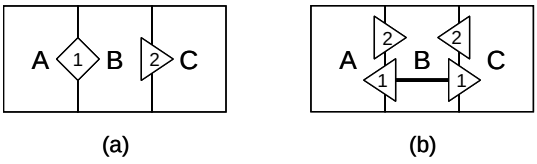


FIG. 4.7 – Exemple de redondances

Les briques élémentaires que nous venons de définir nous servent de base pour la construction de nos motifs de sûreté. La section suivante propose une définition des motifs.

4.3 Définition des motifs

Le principal objectif des motifs d'architectures est de capitaliser les solutions des experts dans le domaine de la sûreté de fonctionnement. Ces motifs représentent des abstractions d'architectures réelles et doivent donc se comporter comme celles-ci. On trouve au moins deux catégories de motifs. Tout d'abord des motifs de sûreté d'architectures logiciels, comme par exemple les n-plications dissimilaires, qui peuvent inclure des mécanismes de vote ou d'autotest et qui supportent les défaillances énumératives (e.g. valeur correcte, absente ou erronée). On rencontre également des motifs matériels, comme les redondances actives ou passives, qui peuvent inclure des mécanismes de détection ou de ségrégation et qui tolèrent les défaillances dites *fail silent*. Prenons deux exemples concrets : l'architecture du système électrique de l'A320 possède plusieurs redondances passives et les commandes de vol de ce même avion possède une 5-plication dissimilaire des calculateurs. De plus, afin de garantir un gain de temps en phase de conception et d'analyse d'architectures de systèmes, les motifs possèdent des propriétés pré-prouvées, sous certaines hypothèses, qui vont permettre de faciliter les analyses. L'utilisation de ces motifs permet également d'obtenir une vue synthétique du système et d'exhiber les propriétés satisfaites par le modèle pour une comparaison plus rapide avec les exigences contenues dans les documents de sûreté. Enfin, afin de transmettre certaines informations informelles importantes, des attributs nous permettent de documenter les motifs sur leur contenu et leur utilisation. Globalement, les motifs d'architectures de sûreté doivent supporter des spécifications précises ainsi que des analyses formelles, par conséquent leur fondation doit elle même être formelle.

La modélisation formelle d'architectures est aujourd'hui principalement étudiée dans le domaine des ADL. Alors que la plupart de ces ADL sont basés sur une unique méthode formelle, nous avons décidé de baser les motifs sur un formalisme dual combinant automates de mode et logique temporelle linéaire à états et événements. La partie automate définit le comportement du motif alors que les formules de logique temporelle permettent de spécifier leurs hypothèses et propriétés dynamiques. Cette dualité nous semble offrir de nombreux avantages notamment celui de pouvoir décrire et analyser différents aspects des systèmes, accéder à des techniques complémentaires pour une meilleure compréhension des systèmes et une réduction de leur complexité. Par conséquent, nous définissons un motif d'architectures de sûreté par un comportement, un ensemble d'hypothèses et de propriétés et un ensemble d'attributs informels permettant de le documenter.

Ces attributs, formels et informels, permettent de caractériser les motifs, de capitaliser l'information pour assurer un meilleur cataloguage. En effet, la conception de l'architecture d'un système nécessite la combinaison de plusieurs motifs. La consultation d'un catalogue regroupant divers types de motifs permet de simplifier cette conception. Afin de faciliter la compréhension de la structure d'un tel catalogue de motifs il est nécessaire de bien les déclarer, à l'aide d'attributs. Les motifs d'architectures, au sens de la construction d'édifices cette fois, de Christopher Alexander ont une structure bien définie par un ensemble d'attributs. La forme de ses motifs est la suivante :

```
SI vous vous trouvez dans CONTEXTE
 par exemple EXEMPLES
 avec PROBLEMES
 nécessitant FORCES
ALORS pour certaines RAISONS
 appliquez FORME ARCHITECTURE ET/OU REGLE
 pour construire SOLUTION
 menant à AUTRE CONTEXTE et AUTRES MOTIFS
```

Les mots clés importants de cette déclaration concernent le contexte qui réfère à un ensemble de situations récurrentes pour lesquels le motif peut être appliqué, le problème qui fait référence un ensemble de buts et de contraintes qui ont lieu dans ce contexte et enfin la solution qui correspond à une forme d'architecture canonique ou une règle de conception que quelqu'un peut appliquer pour résoudre ce problème. Il y a énormément de variantes concernant la déclaration des motifs, aucun standard n'existe pour l'instant. En ce qui concerne les motifs d'architectures de sûreté nous avons défini les quatre catégories d'attributs suivantes. Dans la suite, les lignes de code correspondent à un langage d'expressions régulières. Celles commençant par `%` sont des commentaires.

4.3.1 Références du motif

```
% nom : string
% autres noms : string*
% liens : (nom relation nom)*
```

Le premier sous-attribut spécifie le nom générique du motif permettant de l'identifier. Le sous-attribut *autres noms* permet de donner quelques exemples d'autres appellations pour un même motif dans le cas où différents noms d'un même mécanisme de sûreté existent dans la littérature. En effet, il est courant d'attribuer plusieurs noms à même composant (e.g. duplication, redondance double, ...) en fonction du domaine d'utilisation et il est utile de savoir que l'on se réfère à un même motif. Le sous-attribut *liens* permet de mettre en relation les différents motifs à l'aide de catégories. La classification que nous avons choisie s'inspire de celle de Zimmer [Zim95]. Elle propose trois catégories de relations entre un pattern A et un pattern B :

- A utilise B dans sa solution (*utilise*)
- A est semblable à B (*semblable à*)
- A peut être associé à B (*associable à*)

Elle permet lors de la recherche d'un motif permettant la résolution d'un problème précis, de trouver la famille de motifs proposant des solutions adaptées pour ensuite choisir le motif qui satisfait les exigences. A partir du catalogue et de la classification nous pensons pouvoir automatiser l'instanciation de motifs dans une architecture concrète et leur détection. Les caractéristiques permettant la classification des motifs sont contenues dans les attributs actuellement présentés.

Exemple Dans le cas de la redondance passive :

```
% nom : CoSSAP (Cold Spare SAP)
```

```
% autres noms : redondance passive, redondance froide, duplication passive
```

Les différents liens avec ce motif sont :

```
% liens : CoSSAP uses motif fonctionnel élémentaire, CoSSAP compatible with
%  tous les motifs de sa catégorie
```

4.3.2 Propriétés intrinsèques du motif

Cette partie formalise le comportement du motif. Elle correspond à ce que l'on connaît de la dynamique interne d'un motif (transitions, assertions), ses états, les événements lui permettant de changer d'état comme ses défaillances, les différentes informations qu'il partage avec son environnement comme ses variables d'entrée et de sortie. Elle comporte également la déclaration des sous-composants présents dans le motif. Ce modèle comportemental est réalisé en AltaRica dans sa version *data flow*, dont l'expressivité est celle des automates de modes. Cette partie comportementale peut également être vide dans le cas du développement d'un motif uniquement à partir d'une spécification et donc de formules de SE-LTL caractérisant le comportement attendu. Dans ce cas, seuls les propriétés, hypothèses et attributs seront définis. Dernier cas possible, le motif est défini à la fois par son modèle comportemental et par des formules qui se contraignent mutuellement. Les attributs informels de cette partie sont les suivants :

Défaillances prises en compte

```
% défaillances : (nom_evt : explication)*
```

Le sous-attribut *défaillance* permet de déclarer les différentes conditions de défaillances prises en compte par le motif.

Exemple

```
% défaillances : faute : défaillance simple d'un composant, perte de sa
%  fonction
```

Hypothèses internes Cet attribut permet de décrire les contraintes internes au motif, comme des reconfigurations internes par exemple. Ces contraintes peuvent être liées à la technologie utilisée ou à la topologie de l'architecture.

```
% assertion : (label : explication)*
```

Exemple

```
% assertion : indépendance : Les deux composants en redondance doivent être
%  indépendants
```

4.3.3 Hypothèses externes

Cet attribut permet de spécifier le cadre d'application du motif c'est à dire les contraintes, également éventuellement liées à la technologie utilisée, sur l'implantation du motif dans une architecture hôte. Ces contraintes peuvent être des contraintes de ségrégation ou d'indépendance entre plusieurs composants par exemple.

```
% environnement : (label : explication)*
```

Exemple

```
% environnement : ressources : Les ressources sont toujours disponibles et les
%  entrées sont toujours vraies
```

Les hypothèses externes définissent les contraintes que l'on souhaite voir satisfaites par l'environnement pour une utilisation correcte du motif. Pour simplifier les preuves de la validité d'un motif, que nous verrons plus tard, nous avons choisi de distinguer deux types de contraintes : les contraintes d'implantation liées à l'environnement dans lequel évolue le motif et que l'on nommera *hypothèses d'environnement*, des contraintes locales propres au motif et à son fonctionnement que l'on appelle *hypothèses internes*. Cette distinction est importante car chaque type d'hypothèse joue un rôle particulier au sein du motif à un moment donné dans l'utilisation de ce motif. Ces différents types de contraintes doivent prendre en compte l'aspect dynamique des motifs sur lesquels ils s'appliquent, par conséquent on les spécifiera en SE-LTL, logique du futur distinguant les états des événements, tout comme cela est fait dans le langage AltaRica. Les hypothèses d'environnement ne sont pas traitées directement par les outils AltaRica par conséquent on les déclare dans une clause externe. Les hypothèses internes sont des contraintes globales sur les variables du composant et seront, elles, déclarées comme des assertions classiques.

4.3.4 Propriétés garanties

Non comportementales

```
% propriétés : (maturité : explication)
% propriétés : (complexité : explication)
% propriétés : (support : explication)
```

Les propriétés non comportementales permettent de donner des informations sur la conception du motif. Elles renseignent sur la maturité de la technologie utilisée afin d'évaluer le degré de confiance que l'on peut placer dans le motif, la complexité de cette technologie en terme de nombre de composants utilisés dans le motif, dissimilarité etc. Enfin, un sous- attribut permet également de donner des détails sur le support disponible pour la technologie : militaire, sur étagère, commercial ou *ad-hoc* par exemple.

Exemple Toujours dans le cas de la redondance passive :

```
% propriétés : maturité : forte car solution classique très répandue
% propriétés : complexité : faible, deux composants éventuellement
%  dissimilaires et un contrôleur
% propriétés : support : militaire, commercial, sur étagère, ad-hoc
```

Comportementales

```
% propriétés : (label : explication)*
```

La solution apportée par le motif est présentée sous la forme de propriétés garanties. Ces propriétés sont déduites du modèle formel comportemental et des hypothèses prises sur le motif. Elles sont garanties car elles ont été validées à l'aide d'un vérificateur de modèles, montrant ainsi que le modèle du motif satisfait bien ce jeux de propriétés. Bien que le "coût" de la vérification de telles propriétés puisse être important, il reste raisonnable puisque cette vérification n'est réalisée qu'une seule et unique fois. En effet, si les contraintes d'implantation sont vérifiées alors les propriétés garanties sont vérifiées sans avoir à tout redémontrer. Tout comme les hypothèses présentées précédemment, les propriétés sont spécifiées en SE-LTL. Un sous-attribut informel permet de les déclarer de manière textuelle.

Exemple

```
% propriétés : redondance : fonction tolérante à une faute
```

4.3.5 Exemple

Reprenons l'exemple de la redondance créée à la section 4.2.6 pour en faire un motif d'architecture de sûreté comme nous venons de les définir.

```
node Cossap

% nom : CoSSAP (Cold Spare SAP)
% autres noms : redondance passive, redondance froide, duplication passive

 state C1_down, C2_down:bool;
 flow C1_failed, C2_failed:out:bool;

  event fail_C1, fail_C2;

 % défaillances : fail_C1 : défaillance simple d'un composant, perte de sa fonction,
 % fail_C2 : défaillance simple d'un composant, perte de sa fonction

 sub input:input_splitter;
  C1, C2:elem_function;
  output:var_dependency_output;

  % liens : CoSSAP utilise motif elem_function, CoSSAP utilise input_splitter,
  % CoSSAP utilise var_dependency_output

 trans not(C1.down) |- fail_C1 -> C1.down := true;
  not(C2.down) |- fail_C2 -> C2.down := true;
 assert C1.i = input.o1;
  C2.i = input.o2;
  output.i1 = C1.o1;
```

```
  output.i2 = C2.o2;
  C1.a = true;
  C1.f = C1_down;
  C2.f = C2_down;

  % assertion : activation : activation de C2 suite à perte de C1

  G(not(C1.s) -> XC2.a);
 extern initial_state = C1_down := false, C2_down := false;

  % environnement : ressources : Les ressources sont toujours disponibles et les
  %  entrées sont toujours vraies

  environment resources : G(C1.r = true and C2.r = true);

  % propriétés : redondance : fonction tolérante à une faute

  guaranteed redondance : not(Ffail_C1 and Ffail_C2) -> G(output.o or Xoutput.o);

  % propriétés : maturité : forte car solution classique très répandue
  % propriétés : complexité : faible, deux composants éventuellement
  %  dissimilaires et un contrôleur
  % propriétés : support : militaire, commercial, sur étagère, ad-hoc

edon
```

Comme nous l'avons vu au chapitre précédent, le comportement des systèmes en présence de pannes peut être formalisé à l'aide de langages formels comme AltaRica qui est dédié à la modélisation orientée sûreté de fonctionnement et nous permet de réaliser un modèle opérationnel (i.e. simulable) d'un système. Dans notre cas, les motifs correspondent à des abstractions d'architectures concrètes et donc requièrent une modélisation davantage déclarative, i.e. à l'aide de propriétés. Ces propriétés sont en général dynamiques et nous avons choisi la Logique Temporelle Linéaire pour les modéliser. La description des patterns est donc une vue mixant une partie AltaRica et d'une partie LTL. Ce type de modélisation mixte possède plusieurs intérêts, notamment lors de la conception en phase amont d'architectures de systèmes où il est possible de considérer qu'une partie du système est clairement définie, dans sa version stable et donc modélisée en AltaRica, et qu'une autre partie ne l'est pas complètement ou en cours d'étude et sera spécifiée sous forme mixte AltaRica/formules de logique temporelle. Cette dernière correspond à un pattern de sûreté. Nous avons donc défini une notation mixant de l'opérationnel (AltaRica pour les composants bien définis) et du dénotationnel (une variante de LTL pour les composants spécifiés à partir de propriétés). Nous présentons donc dans la suite cette extension d'AltaRica permettant d'obtenir au sein d'un même cadre à la fois le modèle d'un système et ses exigences et propriétés. Dans cette partie nous montrons comment traduire un automate de mode en une structure de Kripke étiquetée.

4.4 Formalisation des motifs

Comme nous venons de le voir dans la partie précédente, un motif d'architecture de sûreté est composé d'un modèle comportemental et de propriétés spécifiées par des formules de logique temporelle. L'objectif de ce qui suit est de caractériser le comportement d'un système défini à la fois de manière opérationnel, caractérisé par notre modèle comportemental et dénotationnel qui nous permet de spécifier les propriétés et les exigences à satisfaires. Pour cela nous avons choisi de plonger la sémantique d'AltaRica dans les Structures de Kripke Etiquetée qui définissent la sémantique de State-Event LTL, notre logique de spécification de propriétés et d'exigences. De plus, afin obtenir des structures manipulables informatiquement, nous avons décidé de traduire les propriétés de SE-LTL en automates de Büchi. Le produit de ces deux structures nous permet d'obtenir un automate de Büchi caractérisant le comportement satisfaisant les contraintes fixées à la fois par la partie AltaRica et par la partie SE-LTL. Nous avons donc du également étendre la syntaxe AltaRica pour permettre de spécifier et contraindre le comportement des composants par des propriétés temporelles à la fois sur les états et sur les événements.

4.4.1 Syntaxe

L'extension syntaxique correspond à l'ajout de propriétés en SE-LTL dans le corps d'un composant AltaRica. Ces propriétés correspondent aux attributs formels introduits dans [KSB+04]. Ces attributs traduisent les hypothèses prises lors de la conception du motif ainsi que les propriétés garanties. Les hypothèses prises sur l'environnement seront déclarées après le mot clé `environment`, celles concernant les hypothèses propres au patron seront déclarées directement dans les assertions, sans mot clé spécifique. Avec le comportement décrit par le code AltaRica standard et sous les hypothèses précédentes le patron garantit certaines propriétés (`guaranteed`). Ces hypothèses et propriétés concernent aussi bien le patron lui même que son envionnement et contraignent globalement le comportement des composants c'est pourquoi nous avons choisi de toutes les déclarer au niveau des assertions.

```
node example
 state s:type;
 flow f:type;
 event e;
 trans formule(s,f) |- e -> s:=expression(s,f);
 assert formule(s,f);
  formule_temporelle(s,f,e);
 init formule(s,f);
 extern
  environment formule_temporelle(s,f,e);
  guaranteed formule_temporelle(s,f,e);
edon
```

Fig. 4.8 – Syntaxe d'un composant en AltaRica étendu

4.4.2 Sémantique

Nous allons maintenant décrire comment un automate de mode $M = \langle D, S, F^{in}, F^{out}, dom, \Sigma, \delta, \sigma, S_0 \rangle$, correspondant à une spécification AltaRica, peut être représenté par une structure de Kripke étiquetée $M' = \langle S', S'_0, P', \mathcal{L}', T', \Sigma', \varepsilon' \rangle$. Nous pouvons définir le LKS précédent, qui existe toujours pour tout automate à contraintes donné, uniquement par les définitions suivantes :

- $S' = \{\text{états atteignables de } M\}, \mid S' \mid < 2^{|P'|}$
- Les propositions atomiques de M' correspondent à $P' = \{a = b / a \in V, b \in dom(a)\}$.
- L'étiquetage des états est réalisé par $\mathcal{L}' : S' \rightarrow dom(S) \times dom(F^{in}) \times dom(F^{out}) / \mathcal{L}'(s') \in \sigma$. On notera $\mathcal{L}' = \mathcal{L}'^{in} \cup \mathcal{L}'^{out} \cup \mathcal{L}'^{S}$ avec $\mathcal{L}'^{in} : S' \rightarrow dom(F^{in})$, $\mathcal{L}'^{out} : S' \rightarrow dom(F^{out})$ et $\mathcal{L}'^{S} : S' \rightarrow dom(S)$.
- $\Sigma' = \Sigma$
- L'ensemble T' des transitions est défini par : $T' \subseteq S' \times S'$
- L'étiquetage des transitions est réalisé avec des éléments de Σ : $\varepsilon' : T' \rightarrow \Sigma$ est une fonction totale. Si $s_1, s_2 \in S'$ et $t \in \Sigma$ alors $(s_1, s_2) \mapsto t$ ssi $\delta(\mathcal{L}'^{S}(s_1), \mathcal{L}'^{in}(s_1), t) = \mathcal{L}'^{S}(s_2)$
- $S'_0 \subseteq S'$ et $\forall i \in S'_0, \mathcal{L}'(i) \in S_0$

Soit M' défini comme précédemment. Notons B_φ l'automate de Büchi d'une propriété φ. Le calcul de $M' \otimes B_\varphi$ nous permet de déterminer les contraintes qu'impose B_φ sur M' (si $L_\omega(M' \otimes B_\varphi) \neq \emptyset$) et donc de conclure sur la consistance de φ avec M' (i.e. $\exists \pi / M', \pi \models \varphi$). La vérification de la consistance des propriétés A_b avec le modèle est capitale pour montrer la cohérence du motif. En pratique cette cohérence sera montrée à l'aide d'un vérificateur de modèles comme nous le verrons au chapitre suivant. Nous pouvons maintenant définir le produit d'un automate de Büchi avec un LKS [CCO+04] permettant, entre autre, de calculer ces contraintes.

Définition 15 (Produit Büchi/LKS) *Soit $M = \langle S, S_0, P, \mathcal{L}, T, \Sigma, \varepsilon \rangle$ un LKS et $B = \langle S_B, S_{0_B}, P \cup \Sigma, \mathcal{L}_B, T_B, F \rangle$ un automate de Büchi sur l'ensemble des propositions atomiques d'états $P \cup \Sigma$. Le produit $M \otimes B = (S', S'_0, -, -, T', F')$ est un automate de Büchi qui satisfait :*

- $S' = \{(s, b) \in S \times S_B \mid \widetilde{\mathcal{L}(s)} \Rightarrow \exists \Sigma . \mathcal{L}_B(b)\}$ *avec* $\widetilde{\mathcal{L}(s)} = \bigwedge \mathcal{L}(s) \wedge \bigwedge \{\neg p \mid p \in P \setminus \mathcal{L}(s)\}$ *et* $\exists \Sigma . \mathcal{L}_B(b)$ *représente la formule* $\mathcal{L}_B(b)$ *dans laquelle on a supprimé les propositions atomiques sur les événements*
- $(s, b) \longrightarrow (s', b')$ *ssi* $\exists x \in \Sigma \mid s \xrightarrow{x} s'$ *et* $b \longrightarrow b'$ *et* $(\widetilde{\mathcal{L}(s)} \wedge \tilde{x}) \Rightarrow \mathcal{L}_B(b)$ *avec* $\tilde{x} = x \wedge \bigwedge \{\neg y \mid y \in \Sigma \setminus \{x\}\}$
- $(s, b) \in S'_0$ *ssi* $s \in S_0$ *et* $b \in S_{0_B}$
- $(s, b) \in F'$ *ssi* $b \in F$

Théorème 1 (Satisfaction d'une propriété) *Soit M un LKS et ϕ une formule de la logique temporelle linéaire à états et événements. Alors :*

$$M \models \phi \text{ ssi } L_\omega(M \otimes B_{\neg\phi}) = \emptyset,$$

avec $B_{\neg\phi}$ l'automate de Büchi de la négation de la formule ϕ.

Montrons que la projection par rapport à M du langage infini du produit des automates correspond au langage initial de M contraint par l'automate de Büchi B. Pour cela, nous commençons par définir une projection.

Définition 16 (Projection) *Soient $V_1, ..., V_n$ n ensembles finis. Soit π un chemin d'états de $V_1 \times V_2 \times ... \times V_n$. Alors on écrit $\pi = (v_{1_0}, v_{2_0}, ..., v_{n_0}), (v_{1_1}, v_{2_1}, ..., v_{n_1}), ..., (v_{1_n}, v_{2_n}, ..., v_{n_n})$ avec $v_i \in V_i$, $i \in \mathbb{N}$, $1 \leqslant i \leqslant n$. On définit alors $proj_{V_i}(\pi)$ la projection du chemin π par rapport à V_i par :*

$$proj_{V_i}(\pi) = v_{i_0}, v_{i_1}, ..., v_{i_n}$$

Théorème 2 (Contraintes sur les langages) *Soit $M = \langle S, S_0, P, \mathcal{L}, T, \Sigma, \varepsilon \rangle$ une structure de Kripke étiquetée et $B = \langle S_B, S_{0_B}, P \cup \Sigma, \mathcal{L}_B, T_B, F \rangle$ l'automate de Büchi représentant la formule ϕ. On montre alors que :*

$$proj_S(L_\omega(M \otimes B_\phi)) = \{\pi \in L(M), \pi \models \phi\} = L(M)_{\setminus \phi}$$

Nous allons donner une preuve commentée de ce résultat.

1. Par construction de l'automate de Büchi B qui accepte exactement les séquences infinies vérifiant ϕ et comme le montrent Vardi et Wolper dans [VW86], pour tout chemin $\pi = b_0 b_1 ...$ reconnu par B on peut construire un chemin $\pi' = b'_0 b'_1 ...$ tel que $\pi' \models \phi$ et $\forall i, b'_i$ est étiqueté par des propositions atomiques de la formule ϕ et non plus les sous-formules de ϕ.
2. Par construction de $M \otimes B$, la projection sur S des chemins $\sigma = (s_0, b_0), (s_1, b_1) ...$ reconnus par $M \otimes B$ sont tels que $\forall i, (\widetilde{\mathcal{L}(s_i)} \wedge \tilde{x_i}) \Rightarrow \mathcal{L}_B(b_i)$ or $\mathcal{L}_B(b'_i) \subset \mathcal{L}_B(b_i)$ par conséquent $\forall i, (\widetilde{\mathcal{L}(s_i)} \wedge \tilde{x_i}) \Rightarrow \mathcal{L}_B(b'_i)$. Donc σ est une extension de π' et par conséquent satisfait ϕ.

4.4.3 Composition d'un automate AltaRica et d'une propriété

La principale propriété attendue d'une redondance est la tolérance à la faute. En effet, tant que les deux composants ne sont pas défaillants le système global doit fonctionner correctement (i.e. dans notre cas il fournit une sortie). Malheureusement, sans hypothèse sur l'activation de C_2, notre exemple présenté précédemment ne respecte pas cette règle. La propriété garantie par l'automate de la figure 3.3 est $\phi_1 : \neg(Ffail_1) \Rightarrow Go$. Pour garantir cette tolérance nous devons également montrer que $\phi_2 : \neg(Ffail_2) \Rightarrow Go$. Pour cela nous devons faire une hypothèse sur l'activation de C_2 garantissant qu'une défaillance de C_1 implique toujours l'activation de C_2 après reconfiguration. On modélisera cette reconfiguration par l'opérateur temporel X montrant ainsi un décalage entre la perte d'un composant et l'activation du composant redondant. La traduction de cette hypothèse en SE-LTL donne alors $\phi_3 : G(\neg s_1 \Rightarrow Xa_2)$ et l'automate de Büchi associé est présenté sur la figure 4.10. La propriété garantie résultante sera alors $\phi : \neg(Ffail_1 \wedge Ffail_2) \Rightarrow G(o \vee Xo)$ qui est satisfaisante.

L'automate de Büchi correspondant à la formule ϕ_3 est représenté sur la figure 4.10. Le LKS correspondant au modèle comportemental de la redondance est représenté sur la figure 4.9 et ses caractéristiques sont les suivantes :

- $S = \{m_1, m_2, m_3, m_4, m_5, m_6\}$
- L'état initial est : m_1
- Les propositions atomiques sont : $P = \{s_1, s_2, a_2, o\}$
- La fonction d'étiquetage des états est :

$$\mathcal{L} = \begin{cases} m_1 \mapsto \{s_1, s_2, o\} \\ m_2 \mapsto \{s_2\} \\ m_3 \mapsto \{s_1, o\} \\ m_4 \mapsto \emptyset \\ m_5 \mapsto \{s_2, a_2, o\} \\ m_6 \mapsto \{a_2\} \end{cases}$$

- Les transitions sont identiques à celles de la section précédente : $(m_1, m_2), (m_1, m_3)$, $(m_3, m_4), (m_2, m_4), (m_2, m_5)$ et (m_5, m_6)
- Les événements sont : $\Sigma = \{fail_1, fail_2, activate_2\}$
- La fonction d'étiquetage des transitions est :

$$\varepsilon = \begin{cases} (m_1, m_2) \mapsto \{fail_1\} \\ (m_1, m_3) \mapsto \{fail_2\} \\ (m_3, m_4) \mapsto \{fail_1\} \\ (m_2, m_4) \mapsto \{fail_2\} \\ (m_2, m_5) \mapsto \{activate_2\} \\ (m_5, m_6) \mapsto \{fail_2\} \end{cases}$$

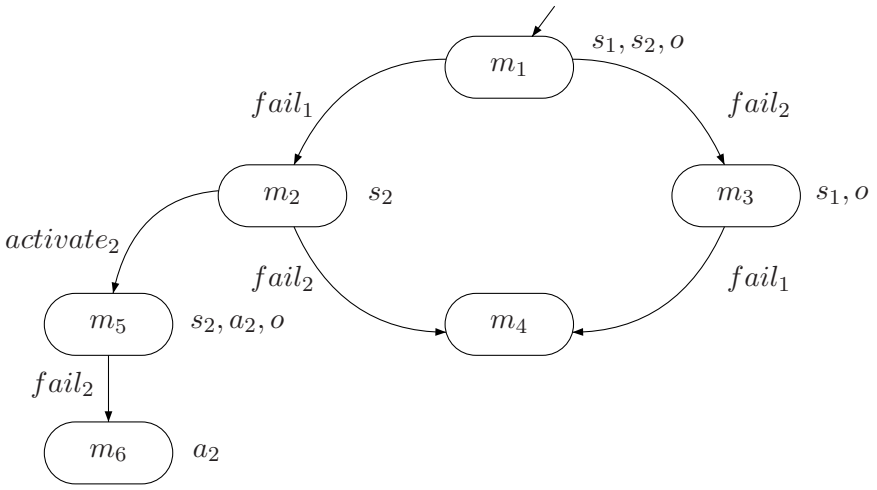


FIG. 4.9 – LKS de la redondance passive

Voici le code des deux nœuds que nous souhaitons composer.

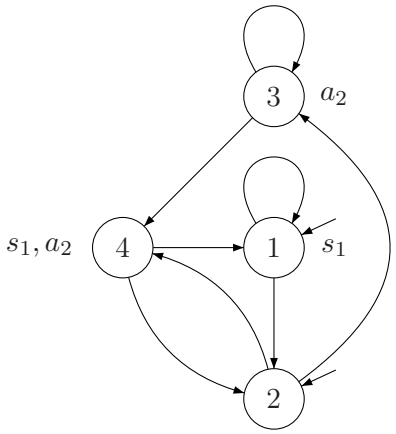


FIG. 4.10 – Automate de Büchi de ϕ_3

```
node cold_redundancy
 state a2, s2:bool;
 flow o:out:bool;
 event fail2, activate2;
 sub c1:source;
 trans s2 |- fail2 -> s2:=false;
  not(c1.s1) |- activate2 -> a2:=true;
 assert o=(c1.s1 or (s2 and a2));
 init a2:=true, s2:=true;
 extern guaranteed not(Fc1.fail) -> Go;
edon
```

```
node redundancy_activation
 state s1,a2:bool;
 assert G(not(s1) -> Xa2);
 init s1:=true, a2:=false;
edon
```

Calculons maintenant le produit de ces deux automates. Notons M le LKS, B l'automate de Büchi et (m_i, b_i) le i^{eme} état de l'automate résultant.

m_i	$\mathcal{L}(m_i)$	$\widetilde{\mathcal{L}(m_i)}$
m_1	$s_1 \wedge s_2 \wedge o$	$s_1 \wedge s_2 \wedge o \wedge \neg a_2$
m_2	s_2	$\neg s_1 \wedge s_2 \wedge \neg o \wedge \neg a_2$
m_3	$s_1 \wedge o$	$s_1 \wedge \neg s_2 \wedge o \wedge \neg a_2$
m_4	T	T
m_5	$s_2 \wedge o \wedge a_2$	$\neg s_1 \wedge s_2 \wedge o \wedge a_2$
m_6	a_2	$\neg s_1 \wedge \neg s_2 \wedge \neg o \wedge a_2$

TAB. 4.2 – Eléments de calcul des états de $M \otimes B$ (1)

Calcul des états de $M \otimes B$

Les tables 4.2, 4.3 et 4.4 nous permettent de calculer l'ensemble S' des états de $M \otimes B$. On trouve alors : $S' = \{(m_1, b_1), (m_1, b_2), (m_2, b_2), (m_3, b_1), (m_3, b_2), (m_4, b_2), (m_5, b_2), (m_5, b_3), (m_6, b_2), (m_6, b_3)\}$.

(m_i, m_j)	$\widetilde{\mathcal{L}(m_i)} \wedge \widetilde{x_i}$
(m_1, m_2)	$s_1 \wedge s_2 \wedge o \wedge \neg a_2 \wedge fail_1 \wedge \neg fail_2 \wedge \neg activate_2$
(m_1, m_3)	$s_1 \wedge s_2 \wedge o \wedge \neg a_2 \wedge \neg fail_1 \wedge fail_2 \wedge \neg activate_2$
(m_2, m_4)	$\neg s_1 \wedge s_2 \wedge \neg o \wedge \neg a_2 \wedge \neg fail_1 \wedge fail_2 \wedge \neg activate_2$
(m_2, m_5)	$\neg s_1 \wedge s_2 \wedge \neg o \wedge \neg a_2 \wedge \neg fail_1 \wedge \neg fail_2 \wedge activate_2$
(m_3, m_4)	$s_1 \wedge \neg s_2 \wedge o \wedge \neg a_2 \wedge fail_1 \wedge \neg fail_2 \wedge \neg activate_2$
(m_5, m_6)	$\neg s_1 \wedge s_2 \wedge o \wedge a_2 \wedge \neg fail_1 \wedge fail_2 \wedge \neg activate_2$

TAB. 4.3 – Eléments de calcul des transitions de $M \otimes B$

b_i	$\mathcal{L}_B(b_i)$	$\exists\Sigma.\mathcal{L}_B(b_i)$
b_1	$s_1 \wedge \neg a_2$	$s_1 \wedge \neg a_2$
b_2	T	T
b_3	$\neg s_1 \wedge a_2$	$\neg s_1 \wedge a_2$
b_4	$s_1 \wedge a_2$	$s_1 \wedge a_2$

TAB. 4.4 – Eléments de calcul des états de $M \otimes B$ (2)

Calcul des transitions de $M \otimes B$
On trouve également les transitions suivantes : $((m_1, b_1), (m_2, b_2)), ((m_1, b_1), (m_3, b_2)), ((m_1, b_2), (m_2, b_2)), ((m_1, b_1), (m_3, b_1)), ((m_3, b_1), (m_4, b_2)), ((m_2, b_2), (m_5, b_3)), ((m_5, b_3), (m_6, b_3)), ((m_5, b_2), (m_6, b_3))$.

Calcul des états initiaux de $M \otimes B$
Le seul état initial est (m_1, b_1).

Calcul des états acceptants de $M \otimes B$
Etant donné ϕ_3, tous les états sont acceptants.

L'automate issu de la composition de ces deux automates est présenté sur la figure 4.11. Notons que le produit $M \otimes B_{\phi_3}$ a contraint le système en supprimant l'indéterminisme qui existait dans l'état m_2 en enlevant la transition (m_2, m_4).

En observant l'automate de la figure 4.11, on note qu'il existe bien des chemins menant de l'état initial aux états acceptants. Par conséquent $L_\omega(M \otimes B_{\neg\phi}) \neq \emptyset$ d'où $\exists s \in S/M, s \models \phi$.

La syntaxe ALTARICA étendue résultant de cette composition est alors la suivante :

```
node cold_redundancy
 state a2, s2:bool;
 flow o:out:bool;
 event fail2, activate2;
 sub c1:source;
 trans s2 |- fail2 -> s2:=false;
```

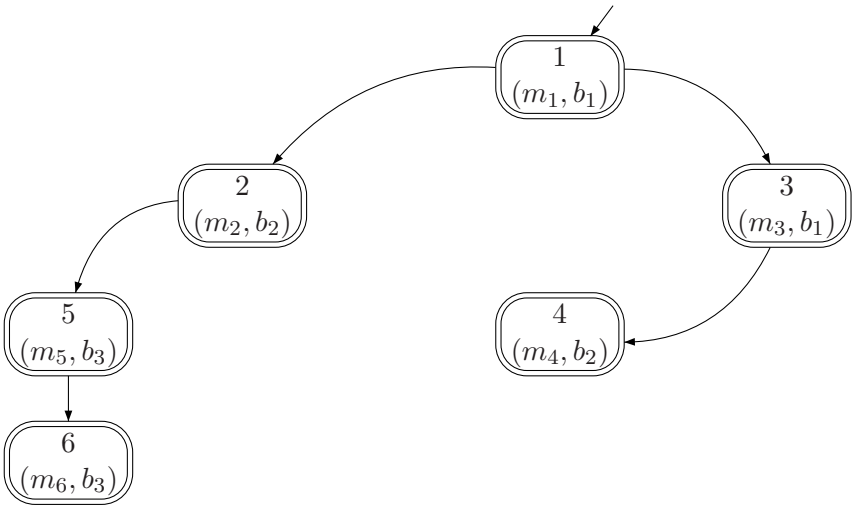


FIG. 4.11 – Automate $M \otimes B_{\phi_3}$

```
  not(c1.s1) |- activate2 -> a2:=true;
 assert o=(c1.s1 or (s2 and a2));
 init a2:=true, s2:=true;
 extern
  environment G(not(c1.s1) -> Xa2);
  guaranteed not(Fc1.fail and Ffail2) -> G(o or Xo);
edon
```

Après avoir défini les motifs d'architecture en spécifiant leur syntaxe ainsi que leur sémantique, nous allons à présent les définir formellement et décrire comment les valider. En effet, avant toute utilisation d'un motif il est nécessaire de vérifier sa cohérence, c'est à dire la consistance des différents attributs définis au chapitre précédent, indépendamment les uns des autres puis globalement.

4.4.4 Définition des motifs

Définition 17 (Motif d'architectures de sûreté) *Soit M un automate de mode, K_M son LKS équivalent et P un ensemble de formules de la logique temporelle linéaire propositionnelle à états et événements. On définit alors un motif d'architectures de sûreté par le couple $C = (M, P)$ avec :*

- *M le modèle comportemental du motif,*
- *$P = \{A_e, A_b, P_g\}$ avec A_e un ensemble d'hypothèses sur l'environnement, A_b un ensemble d'hypothèses internes et P_g un ensemble de propriétés garanties par le motif.*

Le motif C est bien formé s'il satisfait les conditions suivantes :

1. *$\forall \phi \in P$, ϕ est plausible,*
2. *M est plausible,*

3. $A_b \cup A_e$ *et* K_M *sont compatibles,*

4. $\forall p \in P_g$, p *est garantie par le motif.*

M et P sont liés et se contraignent mutuellement. (A_e, P_g) peut être vu comme la spécification du motif et (M, A_b) comme une implantation possible. Notons que si $P = \emptyset$ alors C est un composant AltaRica classique.

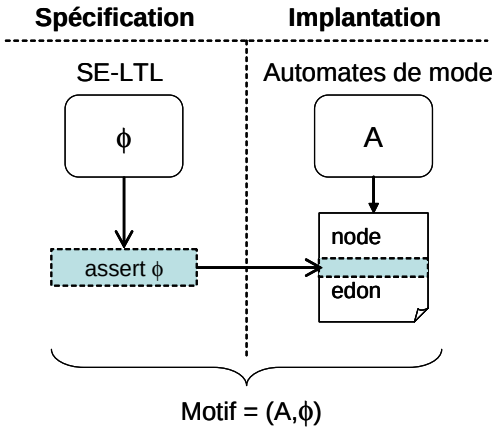


Fig. 4.12 – Motifs d'architectures de sûreté

Exemple Reprenons l'automate de mode de la figure 3.3 de la redondance passive, que nous noterons M. Nous avons vu au chapitre précédent qu'une propriété trivialement garantie par cet automate est $\phi_1 : \neg(Ffail_1) \Rightarrow Go$. $M \models \phi_1$, par conséquent le couple (M, ϕ_1) est un motif d'architecture de sûreté d'une redondance passive.

Nous allons désormais définir les termes *plausible*, *compatible* et *garanti* employés précédemment.

4.4.5 Consistance des attributs

La validation des hypothèses et propriétés d'un motif est importante car les motifs ont vocation à être réutilisés et il serait catastrophique de réutiliser des composants incorrects. Les hypothèses et propriétés d'un motif sont des formules de la logique temporelle linéaire à états et événements. Notons que pour un motif, les ensembles de formules sont soit vide, soit un ensemble plausible.

Définition 18 (Plausibilité d'une formule) *Soit p une formule de SE-LTL. p est une hypothèse ou une propriété plausible d'un motif ssi :*

- *p n'est pas une tautologie : $\not\models p$,*
- *p est consistante : $\not\models \neg p$.*

On donne alors la définition de la plausibilité d'un ensemble de formules.

Définition 19 (Plausibilité d'un ensemble de formules) *On dit qu'un ensemble de formules est plausible ssi :*

- *chaque formule p est plausible,*
- *la conjonction des formules est consistante avec K_M.*

Définition 20 (Plausibilité d'un modèle) *Il est nécessaire de définir la plausibilité d'un modèle. Elle se définit par la vérification syntaxique du code* ALTARICA *ainsi que la vérification de l'absence de blocage du modèle. Cette dernière a pour but de vérifier que le code n'est pas "mort", c'est à dire qu'il existe bien au moins un état initial dans M ($S_{0_M} \neq \emptyset$) et une transition partant de cet état ($\exists s_1/(s_0, s_1) \in T_M$). Notons que ce critère est faible mais nécessaire.*

4.4.6 Compatibilité d'une formule et d'un automate

Définition 21 (Compatibilité formule/automate) *Soit A un ensemble de formules plausibles et K_M le LKS d'un automate de mode plausible. A et K_M sont compatibles ssi $L_\omega(K_M \otimes B_{\bigwedge_{p\in A} p}) \neq \emptyset$ avec $B_{\bigwedge_{p\in A_b} p}$ l'automate de Büchi de la conjonction des hypothèses internes.*

Dans le premier cas, le motif est dit "complet", c'est à dire qu'il possède à la fois son modèle comportemental et ses hypothèses internes. On réalise alors la validation des hypothèses A_b en vérifiant leur consistance avec K_M, ce qui revient à montrer que le langage du produit du modèle comportemental M par l'automate de Büchi correspondant à la conjonction des hypothèses internes

$$B_{\bigwedge_{p\in A_b} p}$$

n'est pas vide.

Exemple : cas d'un motif "incomplet"
Reprenons encore une fois l'exemple de la redondance présenté dans les sections précédentes. Le LKS correspondant au modèle comportemental de la redondance est représenté sur la figure 4.9 et le code de ce modèle est le suivant :

```
node cold_redundancy
 state a2, s2:bool;
 flow o:out:bool;
 event fail2, activate2;
 sub c1:source;
 trans s2 |- fail2 -> s2:=false;
  not(c1.s1) |- activate2 -> a2:=true;
 assert o=(c1.s1 or (s2 and a2));
 init a2:=true, s2:=true;
edon
```

Aprés vérification de la correction syntaxique du code, l'étude du LKS nous permet de voir que ce code n'est pas mort.

Exemple : cas d'un motif "complet"
Reprenons le cas précédent en ajoutant la propriété L'automate de Büchi correspondant à la formule `not(Fc1.fail) -> Go` a été précédemment sur la figure 3.8 et le LKS est le même que dans le cas précédent.

```
node cold_redundancy
 state a2, s2:bool;
 flow o:out:bool;
 event fail2, activate2;
 sub c1:source;
 trans s2 |- fail2 -> s2:=false;
  not(c1.s1) |- activate2 -> a2:=true;
 assert o=(c1.s1 or (s2 and a2));
  not(Fc1.fail) -> Go;
 init a2:=true, s2:=true;
edon
```

Calculons le produit de ces deux automates. Notons M le LKS, B l'automate de Büchi et (m_i, b_i) le i^{eme} état de l'automate résultant.

Calcul des états de $M \otimes B$
(m_1, b_1) est-il un état de $M \otimes B$? Oui, car :
$\widetilde{\mathcal{L}(m_1)} = s_1 \wedge s_2 \wedge o \wedge \neg a_2$ et $\exists \Sigma . \mathcal{L}_B(b_1) = \top$.

De la même manière (m_2, b_4) est-il un état de $M \otimes B$? Non, car :
$\widetilde{\mathcal{L}(m_2)} = \neg s_1 \wedge s_2 \wedge \neg o \wedge \neg a_2$ et $\exists \Sigma . \mathcal{L}_B(b_4) = o$.

On trouve finalement que $S' = \{(m_1, b_1), (m_2, b_2), (m_2, b_3), (m_3, b_2), (m_3, b_3), (m_3, b_4), (m_4, b_2), (m_4, b_3), (m_5, b_2), (m_5, b_3), (m_6, b_2)\}$.

Calcul des transitions de $M \otimes B$
La transition $(m_1, b_1) \longrightarrow (m_2, b_2)$ existe-t-elle ? Oui, car :
L'événement $fail_1$ autorise la transition de m_1 à m_2, de plus la transition de b_1 à b_2 existe dans B et enfin $(\widetilde{\mathcal{L}(m_1)} \wedge \widetilde{fail_1}) = s_1 \wedge s_2 \wedge o \wedge \neg a_2 \wedge fail_1 \wedge \neg fail_2 \wedge \neg activate_2$ et $\mathcal{L}_B(b_1) = \top$.

La transition $(m_2, b_3) \longrightarrow (m_4, b_3)$ existe-t-elle ? Non, car :
L'événement $fail_2$ autorise la transition de m_2 à m_3, la transition de b_3 à b_3 existe dans B mais $(\widetilde{\mathcal{L}(m_2)} \wedge \widetilde{fail_2}) = \neg s_1 \wedge s_2 \wedge \neg o \wedge \neg a_2 \wedge \neg fail_1 \wedge fail_2 \wedge \neg activate_2$ et $\mathcal{L}_B(b_3) = fail_1$.

On trouve finalement les transitions suivantes : $((m_1, b_1), (m_2, b_2)), ((m_1, b_1), (m_2, b_3)), ((m_1, b_1), (m_3, b_2)), ((m_1, b_1), (m_3, b_3)), ((m_1, b_1), (m_3, b_4)), ((m_2, b_2), (m_4, b_2)), ((m_2, b_2), (m_4, b_3)), ((m_2, b_2), (m_5, b_2)), ((m_2, b_2), (m_5, b_3)), ((m_3, b_3), (m_4, b_3)), ((m_5, b_2), (m_6, b_2))$ et $((m_5, b_2), (m_6, b_3))$.

Calcul des états initiaux de $M \otimes B$
Le seul état initial est (m_1, b_1).

Calcul des états acceptants de $M \otimes B$
Les états acceptants sont (m_2, b_3), (m_3, b_3), (m_4, b_3), (m_5, b_3), (m_6, b_3) et (m_3, b_4).

Il existe bien des chemins menant de l'état initial aux états acceptants. Par conséquent $L(M) \cap L(B) \neq \emptyset$ et la propriété représentée par B est consistante avec M.

4.4.7 Correction de P_g

Les propriétés garanties le sont uniquement si elles peuvent être démontrées à partir des hypothèses internes et d'environnement. De plus, toutes les hypothèses sont utiles, c'est à dire que toutes contribuent à la preuve. Ces conditions de validité des propriétés garanties sont formalisées ci-dessous.

Définition 22 (Correction de P_g) *Les propriétés P_g sont garanties ssi :*

1. $K_M \models \bigwedge_{p \in A_b} p \wedge \bigwedge_{p' \in A_e} p' \Rightarrow \bigwedge_{q \in P_g} q$. *Notons que si* $A_i = \emptyset$, $\bigwedge_{p \in A_i} p = \top$
2. *Si* $A_b \neq \emptyset, K_M \not\models \bigwedge_{p \in A_b} p \Rightarrow \bigwedge_{q \in P_g} q$
3. *Si* $A_b \neq \emptyset, K_M \not\models \bigwedge_{p' \in A_e} p' \Rightarrow \bigwedge_{q \in P_g} q$

L'utilisation d'un outil de vérification formelle comme SMV permet de simplifier la vérification de la correction de P_g. Notons que SMV permet de détecter les définitions circulaires qui conduiraient faussement à la preuve de P_g. Comme le montre la figure 4.13, il n'est pas autorisé de déduire la propriété C de A et B sachant que A ou B est défini à partir de C. Un exemple d'utilisation de cet outil sera présenté dans la suite.

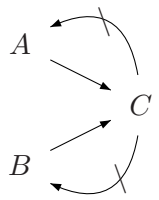


FIG. 4.13 – Définition circulaire

Plusieurs exemples de motifs de redondances sont présentés en Annexe B.

Chapitre 5

Validation d'exigences à base de motifs

5.1 Introduction

L'approche de conception d'architectures par motifs que nous proposons doit supporter les développements *top-down* et *bottom-up*. L'approche *top-down* est utilisée pour développer une architecture détaillée sûre par raffinement d'une architecture de plus haut niveau à base de motifs. En effet, chaque motif satisfait un ensemble de propriétés garanties. Par conséquent, le raffinement d'un modèle abstrait M_A composé de motifs permettra la création d'un ensemble d'architectures concrètes M_{C_i} vérifiant ces mêmes propriétés. L'utilisation des motifs pour la conception d'architectures est donc orientée "exigences", comme le montre la figure 5.1.

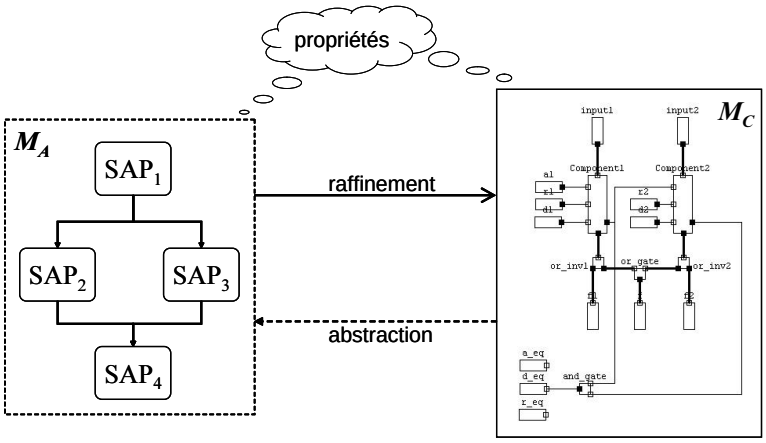


FIG. 5.1 – Conception et validation d'architectures

L'approche *bottom-up* est utilisée pour abstraire un ensemble de composants en un composant de plus haut niveau afin de valider une architecture. L'objectif est de montrer qu'une architecture concrète vérifie bien un ensemble d'exigences. Une architecture abstraite composée de motifs est créée à partir de l'architecture concrète. En montrant qu'il existe une relation d'équivalence entre M_A et M_C nous montrons que les propriétés satisfaites par M_A sont également satisfaites par M_C.

Ce chapitre a pour objectif d'introduire quelques opérations réalisables sur les motifs d'ar-

chitectures de sûreté. Une fois la validation d'un motif effectuée, nous pourrons chercher à instancier un motif au sein d'une architecture, reconnaître un motif dans une architecture déjà développée afin d'hériter de ses propriétés garanties ou encore raffiner un motif en un morceau d'architecture de plus bas niveau garantissant les mêmes propriétés que le motif. Nous montrerons ensuite comment faire de la preuve de propriétés d'une architecture à partir des propriétés des motifs. Enfin, nous présenterons quelques outils disponibles pour réaliser ces opérations.

5.2 Reconnaissance de motifs

La reconnaissance de motifs au sein d'une architecture concrète consiste à identifier la partie architecturale du motif et à vérifier que les hypothèses internes et d'environnement sont satisfaites. Afin d'effectuer l'identification du modèle comportemental, nous devons définir une relation permettant de comparer deux modèles. C'est ce que nous présentons dans cette section avec la définition de différentes relations d'équivalence.

5.2.1 Relations de raffinement

Les motifs d'architectures de sûreté sont des abstractions (M_A) de modèles concrets plus détaillés (M_C). Les propriétés démontrées sur M_A doivent être valides sur M_C. Se pose alors la question de la comparaison de ces deux modèles. Celle-ci peut être basée sur la définition d'une relation $\mathcal{R}$ de raffinement ou d'implication entre les modèles : $M_A \xrightarrow{\mathcal{R}} M_C$. En effet, la substitution/reconnaisance d'une architecture réelle par un pattern d'architecture nécessite de vérifier la compatibilité des deux modèles, i.e. vérifier qu'ils ont un comportement similaire. Différentes relations, traduisant cette notion de similarité de comportement de deux automates, existent. La bisimulation, le préordre et l'équivalence sont les relations d'équivalence les plus courantes. Dans cette partie nous présenterons ces différentes relations d'équivalence à notre disposition ainsi que le type de propriétés qu'elles préservent.

Un compromis entre la puissance de l'abstraction et la préservation de l'expressivité de la logique est nécessaire lors du choix d'une relation de raffinement. La bisimulation définie par Park est la manière la plus simple de définir l'équivalence de deux systèmes de transitions.

Définition 23 (Bisimulation) *Soient* $A_1 = \langle D, S_1, F^{in}, F^{out}, dom, \Sigma, \delta_1, \sigma_1, S_0\rangle$ *et* $A_2 = \langle D, S_2, F^{in}, F^{out}, dom, \Sigma, \delta_2, \sigma_2, S_0\rangle$ *deux automates à contraintes,* $\mathcal{R} \subset S_1 \times S_2$ *est une relation de bisimulation si :*

1. $\forall s_1 \in S_1, \exists s_2 \in S_2 \mid (s_1, s_2) \in \mathcal{R}$
2. $\forall s_2 \in S_2, \exists s_1 \in S_1 \mid (s_1, s_2) \in \mathcal{R}$
3. $\forall (s_1, s_2) \in \mathcal{R}, f_{in} \subseteq F^{in}, \sigma_1(s_1, f_{in}) = \sigma_2(s_2, f_{in})$
4. $\forall s'_1 \in \delta_1(s_1, f_{in}, e), e \in \Sigma, s_2 \in S_2, (s_1, s_2) \in \mathcal{R} \Rightarrow \exists s'_2 \in \delta_2(s_2, f_{in}, e), (s'_1, s'_2) \in \mathcal{R}$
5. $\forall s'_2 \in \delta_2(s_2, f_{in}, e), e \in \Sigma, s_1 \in S_1, (s_1, s_2) \in \mathcal{R} \Rightarrow \exists s'_1 \in \delta_1(s_1, f_{in}, e), (s'_1, s'_2) \in \mathcal{R}$

On notera $s_1 \equiv s_2$ *pour* $\mathcal{R}(s_1, s_2)$ *et* $A_1 \equiv A_2$ *s'il existe une telle relation de bisimulation entre les automates* A_1 *et* A_2*.*

Théorème 3 (Préservation des propriétés par bisimulation) *Si $A_1 \equiv A_2$ alors, pour toute formule f de CTL*, $A_1 \models f \Leftrightarrow A_2 \models f$.*

La bisimulation permet à l'abstraction de préserver fortement[1] les propriétés vérifiées [Gru03]. Cependant cet avantage est également un inconvénient car la préservation forte limite grandement le niveau d'abstraction autorisé. Une alternative à ce problème consiste à utiliser une relation d'équivalence plus faible comme le préordre de simulation.

Définition 24 (Préordre de simulation) *Une relation $\mathcal{R} \subseteq S_1 \times S_2$ est une relation de simulation entre A_1 et A_2 si elle vérifie les propriétés 1., 3. et 4. précédentes.*

On notera $s_1 \prec s_2$ pour $\mathcal{R}(s_1, s_2)$ et on dira que A_2 simule A_1 (noté $A_1 \prec A_2$) s'il existe une telle relation de simulation entre A_1 et A_2.

Remarque 1 *La relation $\equiv$ est une relation d'équivalence sur l'ensemble des modèles alors que $\prec$ n'est qu'un préordre.*

Théorème 4 (Préservation des propriétés par préordre de simulation) *Si $A_1 \prec A_2$ alors :*

- *pour toute formule f de ACTL*, $A_2 \models f \Rightarrow A_1 \models f$*
- *pour toute formule f de ECTL*, $A_1 \models f \Rightarrow A_2 \models f$*

Exemple Considérons maintenant deux composants non réparables en redondance active (i.e. les deux composants assurent la même fonction simultanément). En cas de défaillance d'un composant, le composant restant assure seul le fonctionnement du système. La défaillance du composant restant amène le système dans un état de défaillance global. Le système de transition A associé à cette spécification est représenté à gauche sur la figure 5.2.

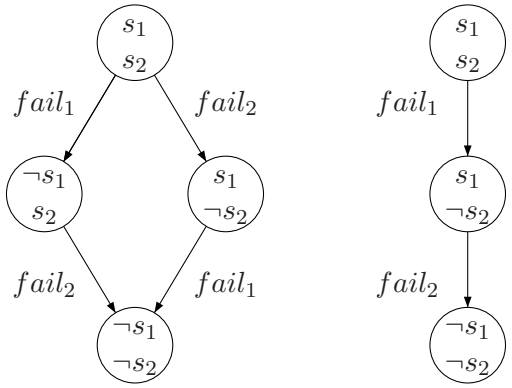


FIG. 5.2 – Deux systèmes "équivalents" ?

Le système de transitions comporte quatre états. Est-il possible de le substituer par une abstraction (un motif par exemple) ? L'utilisation de ce motif qui agrègerait les états $\{\neg s_1, s_2\}$ et $\{s_1, \neg s_2\}$ en un unique état $\{s_1, \neg s_2\}$, permettrait de réduire le nombre d'états du système de

[1] La valeur de vérité d'une propriété de l'abstraction est la même pour le modèle concret

transitions. Cette abstraction signifie que l'on ne différencie plus les composants à l'intérieur du système comme le montre le système de transition B à droite sur la figure 5.2. Mais ces deux systèmes de transitions sont-ils "équivalents" ? Nous allons montrer qu'ils sont similaires.

Si une relation de simulation R existe entre ces deux systèmes elle doit être la suivante : $\{s_1, s_2\}_A R \{s_1, s_2\}_B, \{\neg s_1, s_2\}_A R \{\neg s_1, s_2\}_B, \{s_1, \neg s_2\}_A R \{\neg s_1, s_2\}_B, \{\neg s_1, \neg s_2\}_A R \{\neg s_1, \neg s_2\}_B$

En donnant les classes d'équivalence suivantes : $\{s_1\}_B = (\{s_1\}_A \vee \{s_2\}_A)$ et $\{s_2\}_B = (\{s_1\}_A \wedge \{s_2\}_A)$ on remarque que les propriétés 1., 2., 3. et 4. de la définition 23 sont vérifiées. On obtient alors que $A \prec B$. Prenons maintenant l'exemple d'une propriété f spécifiant qu'il est possible de passer de l'état $\{s_1, \neg s_2\}$ à $\{\neg s_1, \neg s_2\}$ il est trivial de montrer $B \models f$. Alors, étant donné que $A \prec B$, on peut conclure que $A \models f$.

Définition 25 (Equivalence de simulation) *On dit que A_1 est équivalent en simulation à A_2 ssi $A_1 \prec A_2$ et $A_2 \prec A_1$.*

Théorème 5 (Préservation des propriétés par équivalence de simulation) *Si $A_1 \prec A_2$ et $A_2 \prec A_1$ alors, pour toute formule f de ACTL*, $A_2 \models f \Leftrightarrow A_1 \models f$*

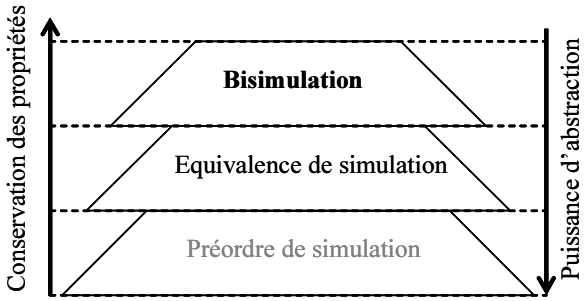


FIG. 5.3 – Classement des relations d'équivalence

La bisimulation, la simulation et l'équivalence de simulation préservent le μ-calcul ce qui est intéressant pour l'utilisation du vérificateur de modèles MecV.

Définition 26 (Reconnaissance de motifs) *La reconnaissance de l'instanciation d'un motif d'architecture de sûreté A par un modèle concret B est réalisée en montrant l'équivalence de simulation de ces deux modèles, i.e. $A \prec B$ et $B \prec A$. La relation d'équivalence de simulation exprime l'équivalence, basée sur la partie observable, entre deux états.*

Pour valider cette reconnaissance nous utiliserons des outils formels de vérification de modèles. Leur but est de vérifier la tenue de propriétés critiques de sûreté sur des modèles. La vérification de modèles permet de vérifier de manière exhaustive qu'un modèle de système, représenté sous la forme d'un automate à états finis, satisfait la propriété qu'on lui impose, représentée sous la forme d'une formule de logique. Par exemple, on pourra chercher à vérifier la propriété "deux sources électriques ne sont jamais en opposition" sur un modèle de système électrique. Le principal problème que pose cette approche est la taille limitée des modèles que l'on peut

vérifier. En effet, le nombre d'états d'un automate croît de manière exponentielle avec le nombre de composants. Or la vérification énumérative classique de modèles construit l'espace d'états complet de l'automate afin de vérifier que toutes les exécutions complètes de l'automate satisfont bien la propriété. Ce problème, bien connu, est communément appelé l'explosion combinatoire du nombre d'états. Une manière d'atténuer ce problème d'explosion consiste à encoder les états et transitions d'un automate par des formules booléennes, puis d'utiliser des structures de données efficaces comme les diagrammes de décision binaire pour les manipuler. C'est le principe de la vérification symbolique de modèles, utilisé entre autre par le célèbre Symbolic Model Verifier (SMV) de Mac Millan [Mac99]. Un article reconnu [BCM+90] montre que cette technique peut être utilisée pour vérifier des systèmes avec plus de 10^{20} états, des systèmes avec 10^{120} états ont même été vérifiés exhaustivement avec succès.

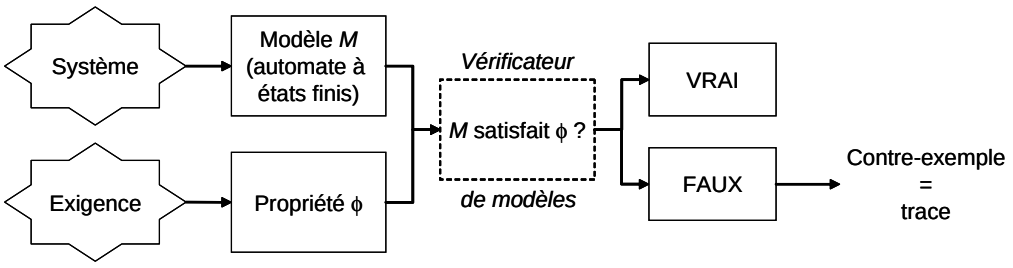


FIG. 5.4 – La vérification de modèles

Dans le cas où la formule n'est pas vérifiée, un contre-exemple permettant de la violer est fourni par l'outil. Cette fonctionnalité est très utile pour déboguer un système. Elle permet également d'utiliser la vérification de modèles comme un générateur de séquences "pas à pas". En effet, la propriété temporelle en entrée du vérificateur permet d'exprimer une propriété redoutée dynamique du système, menant à sa perte définitive par exemple. On va ensuite chercher à vérifier la négation de cette propriété. Si la propriété est vraie cet événement redouté n'apparaît pas dans le modèle. Sinon, le contre-exemple nous donne une suite ordonnée d'états menant d'un état initial à l'état redouté.

La vérification des logiciels est également le domaine des *vérificateurs de théorème*. Leur principal inconvénient par rapport aux vérificateurs de modèles est la nécessité d'intéraction avec l'utilisateur pour guider la preuve et donc la nécessité d'avoir des utilisateurs aguerris à la preuve formelle et à la logique mathématique, ce qui est rédhibitoire pour une utilisation en contexte industriel.

5.2.2 Utilisation de MecV

MecV est un vérificateur de modèles ALTARICA puissant développé au LaBRI par Aymeric Vincent [Vin03]. Un avantage non négligeable de cet outil est qu'il prend en entrée des nœuds AltaRica et évite ainsi l'utilisation d'une passerelle, comme Alta2Smv, qui alourdit le processus. MecV va nous permettre de calculer directement leur équivalence de simulation car son langage de spécification (μ-calcul de Park) est beaucoup plus expressif que les logiques temporelles comme LTL.

Dans Mec, les variables utilisées prennent leur valeur dans les mêmes domaines finis qu'en AltaRica : booléens, intervalles finis d'entiers, énumérations ou encore sur les configurations d'un nœud A (noté A !c) ou les vecteurs d'événements (i.e. synchronisation) d'un nœud A (noté A !ev). Pour ces vecteurs d'événements on notera, pour e $\in$ A !ev, "e" l'ensemble du vecteur d'événements et "e ." l'événement local. On définit également la relation de transition du nœud A, notée A !t, et de type A !c $\times$ A !ev $\times$ A !c. De même, les configurations initiales de A seront notées A !init et sont de type A !c. Les opérateurs disponibles sont les opérateurs booléens classiques de conjonction (noté `&`), de disjonction (noté `|`), de négation (noté `~`) ainsi que les opérateurs arithmétiques usuels (+,-), de comparaison (<,<=,>,>=,=). Mec permet le calcul de relations par point fixe. Il utilise la notation += pour le calcul du plus petit point fixe et -= pour celui du plus grand point fixe. Il est également possible d'introduire dans les expressions des variables quantifiées. On notera [x] le quantificateur universel sur la variable x et <x> le quantificateur existentiel sur cette même variable.

Revenons sur le système à deux composants de l'exemple précédent et montrons à l'aide de Mec que l'on a effectivement $A \prec B$ avec B un motif de A. Pour cela nous devons fournir à Mec les spécifications AltaRica des deux systèmes.

```
node Composant
  state etat : bool ;
  event faute ;
  trans etat = true |- faute -> etat := false ;
  init etat := true ;
edon

node A
  sub C1, C2 : Composant ;
  state s1,s2 : bool ;
  event fail1, fail2 ;
  trans s1 = false |- fail1 -> s1 := true ;
        s2 = false |- fail2 -> s2 := true ;
  sync <fail1, C1.faute> ;
       <fail2, C2.faute> ;
  init s1 := false ;
        s2 := false ;
edon

node B
  state s1,s2 : bool ;
  event fail ;
  trans s1 = false |- fail -> s1 := true ;
        s1 = true |- fail -> s2 := true ;
  init s1 := false ;
        s2 := false ;
edon
```

A l'aide de MecV nous calculons l'ensemble des états accessibles (`ReachSi`) ainsi que la relation

de transition (`ReachTi`) de chacun des modèles, restreintes aux états accessibles. Voici le calcul pour l'architecture initiale.

```
[Mec] ReachS1(t) += A!init(t) | <s><e>
. (A!t(s,e,t) & ReachS1(s));
A!t: (A!c, A!ev,
A!c) -> bool
A!init: (A!c) -> bool
ReachS1: (A!c) -> bool

[Mec] ReachT1(s,e,t) := A!t(s,e,t) & ReachS1(s);
ReachT1: (A!c, A!ev, A!c) -> bool
```

On procède de la même manière avec le motif pour obtenir `ReachS2` et `ReachT2`. Nous allons considérer dans cette relation de simulation la partie visible des configurations en plus des événements. Nous devons donc définir une relation sur les couples de configurations et une relation sur les couples d'événements.

```
[Mec] stateEq(a,a') := ReachS1(a) & ReachS2(a') & (a'.s1 = (a.s1 |
. a.s2)) & (a'.s2 = (a.s1 & a.s2));
stateEq: (A!c, B!c) -> bool

[Mec] eventEq(e:A!ev,e':B!ev) := (e. = fail1 & e'. = fail) |
. (e. = fail2 & e'. = fail) | (e. = " " & e'. = " ") ;
eventEq: (A!ev, B!ev) -> bool
```

Il reste maintenant à écrire la définition de la relation de simulation entre A1 et A2.

```
[Mec] sim(a,a') -= stateEq(a,a') & ([e][b] (ReachT1(a,e,b) =>
. <e'><b'> (ReachT2(a',e',b') & eventEq(e,e') & sim(b,b')))) ;
sim: (B!c, A!c) -> bool
```

Montrons que les configurations initiales sont également similaires.

```
[Mec] iSim(x) := x = (([a] (A!init(a) => <a'> (B!init(a') &
. sim(a,a'))))));
iSim: (bool) -> bool

[Mec] :display iSim
(true)
```

Nous obtenons donc bien que $A \prec B$. Il reste à montrer qu'il existe une relation R' telle que A simule B pour montrer qu'il y a équivalence de simulation.

Après avoir défini les trois principales relations d'équivalence et établi un classement en fonction de deux critères complémentaires, l'équivalence de simulation nous semble convenir pour la validation de la substitution/reconnaissance d'un SAP. En effet, plus l'abstraction est

forte et moins elle préserve les propriétés du modèle de départ. Ainsi, si la bisimulation préserve fortement CTL*, la simulation ne préserve que faiblement[2] ACTL*. En ce qui concerne l'équivalence de simulation, elle préserve fortement ACTL* et donc LTL. La bisimulation nous semble trop restrictive au niveau de l'abstraction et la simulation trop faible du point de vue de la conservation des propriétés. Nous utiliserons par conséquent l'équivalence de simulation pour valider la substitution d'une architecture réelle par un pattern d'architecture de sûreté. Celle-ci, bien que plus faible que la bisimulation, préserve fortement la logique qui nous intéresse tout en admettant un niveau d'abstraction correct.

5.3 Raffinement et implantation de motifs

Le raffinement [Dij76, Bac78, Bac88] permet de construire progressivement des implantations de plus en plus détaillées d'une spécification. C'est un concept très important dans le domaine de la spécification formelle. D'ailleurs, le processus standard de développement d'achitectures préconise la conception, dans un premier temps, d'un modèle abstrait puis son raffinement en un modèle concret. Deux formes principales de raffinement se distinguent. Le raffinement algorithmique qui permet le passage d'une formulation abstraite des spécifications à une description de plus en plus proche du code, et le raffinement de détails qui ajoute des détails comme de nouveaux événements ou variables d'état. Ces deux types de raffinement préservent la correction de nouveaux programmes vis-à-vis de la spécification initiale.

La vérification du raffinement d'un motif en une architecture est une vérification "forte" qui consiste à vérifier que tous les comportements de cette architecture sont similaires aux comportements du motif. La méthode la plus courante consiste donc à chercher une relation de simulation entre le motif et son raffinement éventuel. Les propriétés sont donc préservées. Nous pouvons donc à nouveau utiliser l'outil MecV pour vérifier ce raffinement ou bien utiliser l'outil SMV dans lequel ce type de vérification est implanté.

Le raffinement d'un motif en une construction de plus bas niveau consiste à dériver, à partir des propriétés du motif, des exigences concrètes sur la construction. Cette dérivation peut inclure un renommage des variables, adapté à l'implantation que l'on souhaite faire du motif.

5.3.1 Utilisation de SMV

SMV est un model checker symbolique de formules LTL et CTL sur une structure de Kripke finie. Il a été développé par Mc Millan du *Cadence Berkeley Laboratories*. Une spécificiation en SMV est composée d'un ou plusieurs automates déclarés par le mot clé `module`. Le `module` principal est appelé `main`. Comme nous allons le voir, la syntaxe de SMV est assez proche de celle d'AltaRica. Une passerelle entre ces deux langages a été développée à l'ONERA. Nous allons maintenant étudier la syntaxe de SMV à travers notre exemple de source électrique, présenté en 3.4.2 auquel nous avons ajouté un contacteur. La traduction en SMV de cet assemblage est la suivante :

```
module source(fail,fail_ct,push,o) {
 input fail, fail_ct, push : boolean;
```

[2]Une propriété vraie pour l'abstraction l'est aussi pour le modèle concret.

```
output o : boolean;
fail_assembly, open_ct, status : boolean;

init(status) := 1;
init(open_ct) := 1;
next(status) := case { fail_assembly : 0;
 default : status ;};
next(open_ct= := case { push : ~open_ct;
 default : open_ct ;};
fail_assembly := (fail | fail_ct);
o := status & open_ct ;}
```

Cette fois, un contacteur pouvant être en position ouverte ($open_ct = 1$) ou fermée par le "tirage d'un événement" `push`, permet de contraindre la sortie globale `o` de l'assemblage source électrique/contacteur. Notons que la structure même du code est identique à celle du code ALTARICA. On commence par déclarer les variables d'entrée (`input`), de sortie (`output`) et internes et leur type. Les variables en SMV peuvent être booléennes ou de type énuméré. La valeur d'une variable de type énuméré est convertie par l'outil à l'aide de variables booléennes de façon à ce que les transitions concernées puissent être encodées en BDD . Dans cet exemple, toutes les variables sont (directement) booléennes. On retrouve ensuite la partie automate avec l'initialisation des états et les changements d'état. Ici, la valeur de la variable d'état `ok` est initialement `true`. Elle prend ensuite la valeur `false` (ou 0) si l'entrée faute est vraie, sinon `ok` conserve sa valeur. Enfin, la dernière partie concerne les assertions où l'on spécifie la valeur de la sortie `o`.

Parmi les nombreuses possibilités offertes par cet outil, la vérification de raffinement nous intéresse tout particulièrement. Cette vérification permet de montrer que les fonctions du modèle d'un système abstrait sont correctement implantées par une architecture de plus bas niveau. Nous allons utiliser cette capacité de vérification pour valider le raffinement d'un motif en un assemblage de composants de plus bas niveau. Pour réaliser cette vérification, la notion de couche, ou `layer`, est introduite. Une couche permet de définir la spécification que devront garantir toutes les implantations. Nous allons montrer que l'assemblage ci-dessus est bien une implantation du motif fonctionnel élementaire. Voici le code SMV permettant de montrer la vérification de ce raffinement :

```
module main(fail,fail_ct,push,o){
 input fail,fail_ct,push : boolean;
 output o : boolean;
 i, a, r, f, s : boolean;
 fail_assemblage, open_ct : boolean;


 layer spec: {
 /* Spécification : le motif fonctionnel élémentaire */
 init(s) := 1;
 next(s) := case { fail : 0;
  default : s; };
```

```
 o := i & a & r & s;

}

 /* Raffinement : assemblage source électrique/contacteur */
 init(s) := 1;
 init(open_ct) := 1;
 next(s) := case { fail_assemblage : 0;
  default : s; };
 next(open_ct) := case { push : ~open_ct;
  default : open_ct; };
 /* Classes d'équivalence */
 o := s & open_ct;
 i := 1;
 a := open_ct;
 r := 1;
 fail_assemblage := (fail | fail_ct);

}
```

Les classes d'équivalence sont visibles dans la seconde partie du code ci-dessus. Cette validation nous permet d'utiliser l'assemblage source électrique/contacteur en lieu et place du motif fonctionnel élémentaire.

5.4 Instanciation de motifs

L'instanciation d'un motif consiste à fournir un environnement au motif en liant ses entrées et sorties aux entrées et sorties de composants concrets. A titre de comparaison, on peut considérer l'instanciation de motifs comme une instanciation explicite de fonction en langage C. En effet, il est nécessaire de spécifier explicitement tous les paramètres à utiliser et de les adapter au nouvel environnement. Un développement futur pourrait consister à "automatiser" cette instanciation, pour se rapprocher de l'instanciation implicite, en faisant de l'inférence de type par exemple. Pour une instanciation correcte d'un motif, il est nécessaire de vérifier les deux conditions suivantes :

- Le typage de chaque couple entrée/sortie est compatible.
- L'architecture concrète dans laquelle on instancie le motif doit fournir les services attendus : les hypothèses d'environnement instanciées du motif doivent être vérifiées par l'architecture. Les hypothèses d'environnement instanciées sont obtenues après identification et renommage des variables génériques par les variables utilisées dans l'architecture hôte. Ce procédé est egalement valable pour les hypothèses internes et les propriétés garanties.

⇒ Dans ces conditions, l'architecture hôte hérite des propriétés garanties du motif instancié.

Par conséquent, la validation de l'instanciation de motifs ne se limite pas à la cohérence des types lors de la liaison des interfaces, mais nécessite également de décharger certaines obli-

gations de preuve. Bien que ces preuves puissent être assez lourdes à vérifier, et ce malgré l'utilisation des capacités de preuve modulaire de l'outil SMV, ceci reste acceptable puisque, en général, cette vérification ne doit être réalisée qu'une seule fois. En contrepartie les propriétés garanties sont préservées. Par contre, cette vérification doit être réalisée à chaque utilisation dans le cas des hypothèses sur l'environnement. On pourra également utiliser la vérification de modèles dans ce cas.

Exemple Nous souhaitons implanter un motif fonctionnel élémentaire au sein de l'architecture d'un système électrique (figure 5.5). Ce système est basique puisqu'il contient une ou plusieurs sources électriques qui permettent d'alimenter un système d'éclairage. Nous ne considèrerons pas la propagation d'un flux complexe (du type énuméré en ALTARICA par exemple) dans cette architecture, mais uniquement un flux d'information booléen et monodirectionnel. L'exigence imposée à cette architecture est l'absence d'une panne simple entraînant la perte de l'éclairage. Les différentes étapes autorisant l'instanciation sont les suivantes :

1. Identification des motifs permettant de répondre au problème
2. Adaptation du motif aux spécificités d'un système électrique
3. Vérification du raffinement
4. Instanciation du motif raffiné

Le système étant clairement décomposé en deux zones (une zone de génération et une zone de consommation d'électricité) et l'exigence n'imposant que la tolérance à une défaillance, notre choix s'oriente vers l'utilisation de motifs de redondance. Nous choisissons donc de redonder passivement la génération et activement la consommation électrique (d'autres solutions sont bien entendu possibles). Le motif de redondance passive est composé de deux motifs fonctionnels élémentaires et d'une sortie de type *merger*. Celui de la redondance active est constitué de deux motifs fonctionnels élémentaires ainsi que d'une entrée de type *splitter*.

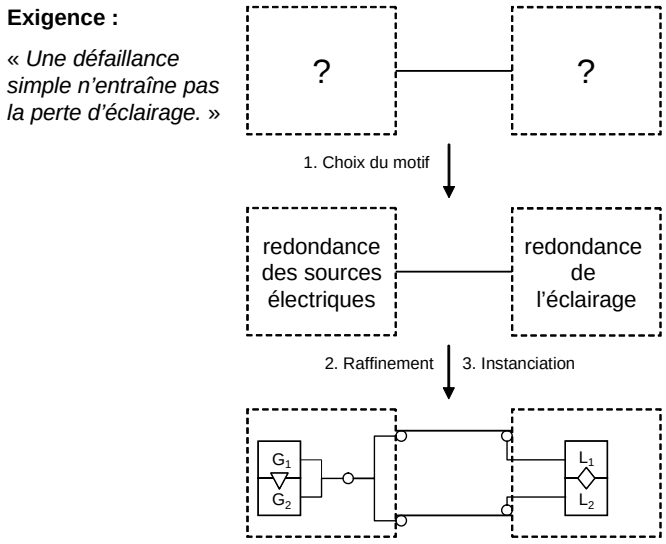


FIG. 5.5 – Processus d'instanciation d'un motif

L'étape concernant la vérification du typage est triviale puisque nous nous plaçons dans un cas simple où à la fois le motif et l'architecture d'accueil appartiennent au monde booléen. La seconde étape de l'instanciation concerne la validation des hypothèses d'environnement du motif par l'architecture d'accueil. Cette propriété n'est pas vérifiée dans l'état courant du développement de l'architecture. Elle devient par conséquent une exigence dérivée que l'architecture doit satisfaire pour hériter des propriétés garanties par le motif.

5.5 Preuve de propriétés et allocation d'exigences

La construction d'architectures à base de motifs permet le prototypage rapide d'architectures sûres de fonctionnement, ou tout du moins permet de prendre en compte l'aspect sûreté dès la phase de prototypage. En effet, comme nous l'avons vu, chaque motif est caractérisé par un ensemble d'exigences de sûreté : exigences dérivées sur les composants en interface (hypothèses d'environnement), exigences intrinsèques (hypothèses propres au motif) et exigences remplies (propriétés garanties). Donc chaque motif permet de mettre en évidence l'allocation des exigences de sûreté dans l'architecture puisque l'on est en mesure d'associer à chaque composant, ou groupe de composants, quelles propriétés doit tenir pour que l'exigence globale soit tenue. Ceci permet d'avoir une conception d'architectures de systèmes orientée par les exigences de sûreté que l'on souhaité vérifier. Quand une architecture est créée par composition de motifs, une exigence globale de sûreté peut être extraite selon deux stratégies possibles :

Stratégie 1 : l'analyse est basée sur les hypothèses intrinsèques et d'environnement de chaque motif. Cette stratégie est toujours possible.

Stratégie 2 : l'exigence globale ne résulte que des propriétés garanties. Cette stratégie peut ne pas toujours être possible. En effet, les propriétés garanties sont des abstractions des hypothèses intrinsèques du motif et peuvent omettre des détails importants lorsque l'on cherche à prouver une exigence globale spécifique. Cependant, chaque fois que cette stratégie est possible, elle fournit une preuve de la robustesse de l'architecture : c'est une garantie qu'un niveau de détail suffisant a été couvert par l'allocation des exigences.

Dans ces deux stratégies, les hypothèses et les propriétés sont assimilées à des lemmes pour la démonstration d'une exigence.

Définition 27 (Preuve de propriétés) *Soit R une exigence, ou une conjonction d'exigences, sur un modèle M, formalisée sous la forme d'une formule de logique temporelle linéaire à états et événements. Notons S l'ensemble des motifs de M. La tenue de R est obtenue de la manière suivante :*

$$\textit{Stratégie 1 :}\ M \models (\bigwedge_{s \in S} A_{b_s} \wedge \bigwedge_{s \in S} A_{e_s}) \rightarrow R$$

$$\textit{Stratégie 2 :}\ M \models \bigwedge_{s \in S} P_{g_s} \rightarrow R$$

Exemple Reprenons l'exemple précédent. La traduction en SMV du système On introduit la variable d'entrée `fail_event` qui est un entier prenant sa valeur dans l'intervalle [0,4]. Ceci nous permet de conserver la propriété ALTARICA qui impose de ne tirer qu'un événement

à la fois. La table suivante permet faire correspondre les valeurs de cette variable avec les défaillances des composants.

```
fail_G1 := (fail_event = 1) ? 1 : 0;
fail_G2 := (fail_event = 2) ? 1 : 0;
fail_L1 := (fail_event = 3) ? 1 : 0;
fail_L2 := (fail_event = 4) ? 1 : 0;
```

Nous introduisons également un compteur de défaillances qui facilite l'expression en SE-LTL de l'exigence à vérifier. Ce compteur s'exprime simplement de la façon suivante :

```
/* Failure counter */
init(counter) := 0;

next(counter) := case { ~(fail_event = 0) : counter+1;
 default : counter; };
```

Le comportement des sources électriques et des lampes est présenté ci-dessous. La variable `o` correspond à la sortie du *merger* de la redondance des sources électriques. La propagation de cette valeur aux lampes permet de les informer sur la présence ou l'absence de tension aux bornes des sources électriques. La variable `lighting` nous indique si l'éclairage est en fonctionnement ou non. Voici le code SMV correspondant au raffinement des motifs :

```
module main(fail_event, activate_G2, lighting) {
input fail_event : 0..4;
input activate_G2 : boolean;
output lighting : boolean;
fail_G1, fail_G2, fail_L1, fail_L2, G1.s, G2.s, G2.a, L1.s, L2.s, o : boolean;
counter : 0..15;

/* Generation */
init(G1.s) := 1;
init(G2.s) := 1;
init(G2.a) := 0;

next(G1.s) := case { fail_G1 : 0;
 default : G1.s; };
next(G2.s) := case { fail_G2 : 0;
 default : G2.s; };
next(G2.a) := case { activate_G2 : 1;
 default : G2.a; };

o := (G1.s | (G2.s & G2.a));

/* Consumption */
init(L1.s) := 1;
init(L2.s) := 1;
```

```
next(L1.s) := case { fail_L1 : 0;
 default : L1.s; };
next(L2.s) := case { fail_L2 : 0;
 default : L2.s; };

lighting := (o & (L1.s | L2.s));
```

Les hypothèses garanties instanciées de cette architecture sont les suivantes :

Redondance sur la génération :

$$\text{generation\_guaranteed} : P_{g_1} : \neg(F(fail\_G_1 \vee fail\_ct_1) \wedge F(fail\_G_2 \vee fail\_ct_2)) \Rightarrow G(o \vee Xo)$$

Exigence dérivée :

$$\text{derived\_requirement} : G(\neg G_1.s \Rightarrow XG_2.a)$$

Redondance sur la consommation :

$$\text{consumption\_guaranteed} : P_{g_2} : \neg(Ffail\_L_1 \wedge Ffail\_L_2) \Rightarrow G((L_1.s \vee L_2.s) \vee X(L_1.s \vee L_2.s))$$

La formalisation de l'exigence sur le système électrique peut se traduire de la manière suivante en SE-LTL :

$$\boxed{G(counter \leqslant 1) \Rightarrow G(F(lighting))}$$

La commande suivante nous permet de vérifier la satisfaction de l'exigence initiale :

```
assume generation_guaranteed, derived_requirement, consumption_guaranteed;
using generation_guaranteed, derived_requirement, consumption_guaranteed prove
requirement;
```

L'exigence initiale est rapidement vérifiée sur cette architecture. Il est maintenant possible de raffiner les motifs afin d'obtenir l'architecture d'un système électrique de plus bas niveau. La nouvelle architecture ainsi obtenue préservera l'exigence que nous venons de vérifier sur l'architecture de motifs. Nous avons montré en 5.3.1 que l'assemblage source électrique/contacteur était un raffinement (une spécialisation) du motif élémentaire fonctionnel. Par conséquent, il est possible d'utiliser cet assemblage au sein d'un motif de redondance pour obtenir une redondance de sources électriques munis de leurs contacteurs. Il reste alors ensuite à instancier le motif raffiné dans l'architecture d'accueil. De la même manière, il est possible de montrer que le modèle simple d'une lampe à incandescence est également un raffinement du motif fonctionnel élémentaire. Ceci nous permet d'utiliser un motif de redondance active de lampes dans notre architecture.

Afin de spécialiser les motifs au domaine électrique, nous pouvons, par exemple, ajouter un mode de défaillance courant dans ce type de systèmes : le court-circuit, et ajouter une variable permettant de propager cette information aux autres motifs de l'architecture. Dans ces conditions, on obtient le raffinement du motif fonctionnel élémentaire suivant :

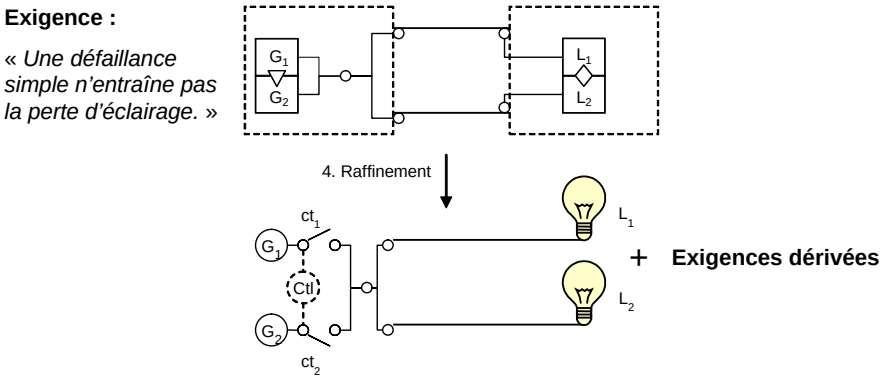


FIG. 5.6 – Raffinement d'une architecture à base de motifs

```
node source
 state s, sc:bool;
 flow o, export_sc:out:bool;
 event fail, fail_sc;
 trans s |- fail -> s:=false;
  not(sc) |- fail_sc -> sc:=true;
 assert o=(s and not(sc));
  export_sc=sc;
 init s1:=true;
  sc:=false;
edon
```

Chapitre 6

De la théorie à la pratique : application à un système avion

6.1 Introduction

Le choix du cas d'étude a été réalisé selon deux critères principaux. Le premier critère considéré est la complexité du système. En effet, nous souhaitions modéliser et analyser un système nous permettant de tester la robustesse de l'approche et son applicabilité à des cas non triviaux. Le second critère concerne la criticité de ce même système car notre objectif était d'étudier un système avec de fortes exigences de sûreté. Nous nous sommes donc focalisés sur deux systèmes appartenant à cette classe des systèmes complexes critiques : les générations et distributions hydraulique et éléctrique d'un avion de type A320. Dans la suite de ce chapitre nous avons choisi de présenter uniquement le système électrique. L'étape de modélisation a débuté par l'étude des différents standards de conception de systèmes aéronautiques. Ces standards sont principalement l'ARP4754 qui décrit le processus de développement utilisable comme support pour la certification (il s'adresse aux autorités de certification ainsi qu'aux candidats à la certification) et l'ARP4761 qui traite de la prise en compte de la sûreté de fonctionnement au cours du développement en définissant les analyses de sécurité qui peuvent être faites en vue de la certification d'un avion et en identifiant les techniques pouvant être utilisées pour ces différentes analyses. Ces recommandations introduisent un processus d'évaluation de la sûreté de fonctionnement d'un système basé sur trois analyses principales : la FHA (Functional Hazard Assessment), la PSSA (Preliminary System Safety Assessment) et la SSA (System Safety Assessment) et sur la rédaction de documents d'exigences appelés T112 chez Airbus. Grossièrement, la FHA considère les fonctions avion et identifie les conditions de panne et leurs effets. La PSSA doit montrer, entre autre, que la liste des conditions de panne issue de la FHA est complète ou doit la compléter. Enfin, la SSA évalue les fonctions du système implémenté pour démontrer que les objectifs de sécurité sont atteints. L'étude de ces documents a été nécessaire pour la compréhension globale du système, des interactions entre les composants et la définition des modes de défaillances, de leurs effets et de leur propagation.

6.2 Présentation du système

Le rôle du système électrique est fondamental dans un aéronef puisqu'il alimente de nombreux éléments critiques comme les servocommandes de gouvernes ou encore les calculateurs

embarqués. Comme l'absence d'énergie électrique peut aboutir à la perte du contrôle de l'avion, sa perte totale est classée catastrophique. Le taux d'occurrence de ce mode de défaillance doit donc être inférieur à 10^{-9} par heure de vol[1], et une seule défaillance ne doit pas y aboutir.

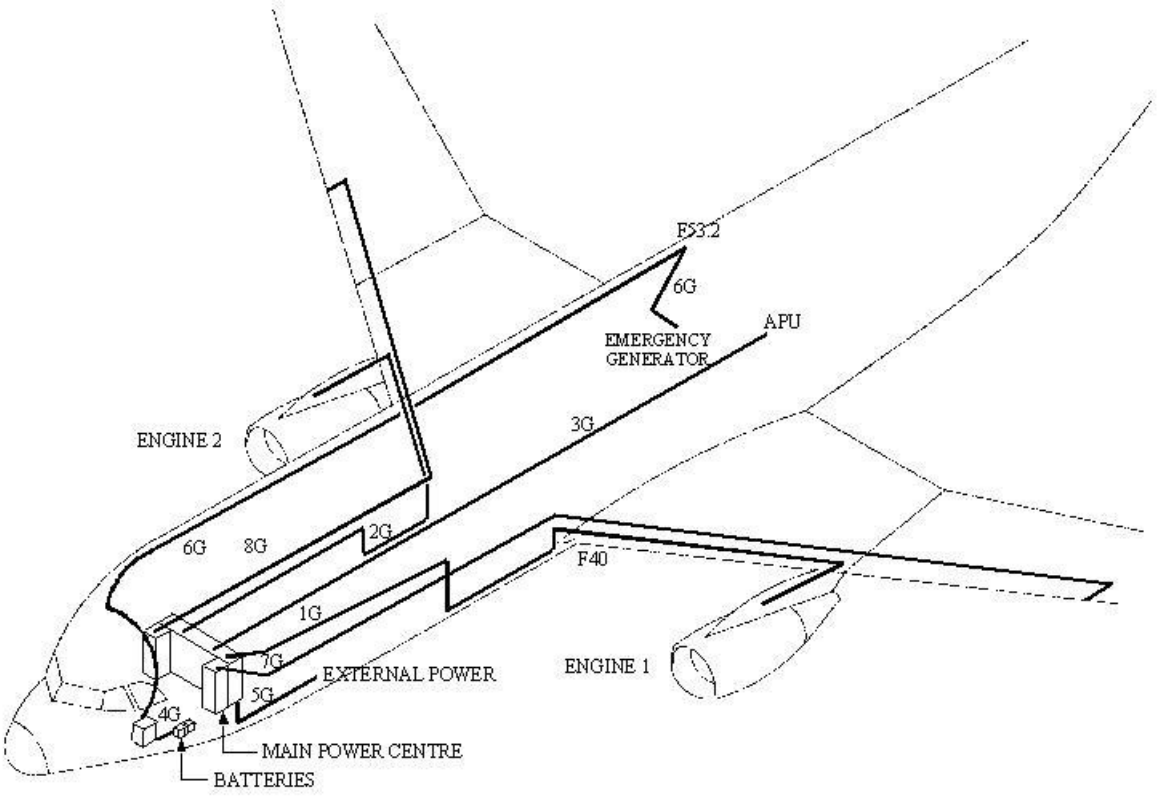


FIG. 6.1 – Ségrégation des routes électriques (schéma Airbus)

Deux solutions complémentaires permettent de garantir cette exigence forte. La première concerne l'implantation géographique du système et la seconde concerne son architecture. En effet, il est capital de ségréguer les routes électriques dans l'appareil afin d'empêcher qu'un événement unique, comme l'explosion d'un réacteur ou encore l'éclatement d'un pneu, n'entraîne la coupure d'une combinaison de câbles électriques entraînant la perte totale de l'alimentation. Ensuite, toujours pour respecter cette exigence, l'architecture du système électrique d'un avion comme celui de l'A320 est organisé en deux sous-systèmes redondants : un système primaire qui assure la fourniture d'électricité dans les conditions normales et un système secondaire, de secours, qui permet de prendre le relais en cas de défaillance du premier. Chacun des sous-systèmes est un circuit électrique qui comprend des générateurs, des barres d'alimentation permettant d'alimenter les différents consommateurs d'électricité embarqués, des contacteurs permettant de fermer ou d'ouvrir un circuit électrique, des disjoncteurs utilisés en

[1]La durée de vie de la cellule de l'A320 est d'environ 6.10^4 heures de vol. Par conséquent, en imposant un taux de défaillance inférieur à 10^{-9} à cet événement, il n'est pas sensé se produire au cour de la vie de l'appareil.

cas de court-circuit, des transformateurs qui comme leur nom l'indique transforment le courant alternatif en courant continu, et enfin des jonctions. Le système primaire est composé de deux générateurs principaux GEN1 entraîné par le réacteur 1, GEN2 entraîné par le réacteur 2 et une unité de puissance auxiliaire APU qui est une petite turbine située dans la queue de l'avion. Le système de secours est composé d'un générateur de secours CSMG (Constant Speed Motor Generator) alimenté par le système hydraulique et la RAT (Ram Air Turbine) qui est une éolienne de secours qui se déploie automatiquement en cas de perte des générateurs principaux. La puissance électrique est fournie aux charges électriques par le biais des quatre barres de distribution pour le système primaire : ACside1 et ACside2 pour le courant alternatif, DCside1 et DCside2 pour le courant continu et par le biais de deux barres de distribution essentielles dans le système de secours : ACess pour l'alternatif essentiel et DCess pour le continu essentiel. La conversion du courant alternatif vers le courant continu est réalisée par les transformateurs TR1, TR2 et TRess. Le système comprend également des disjoncteurs de façon à limiter la propagation de courts-circuits dans le système.

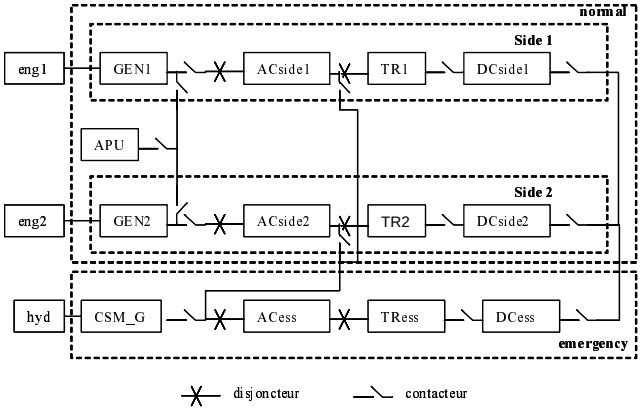


FIG. 6.2 – Architecture du système électrique

Les contacteurs sont contrôlés de façon à implanter plusieurs types de reconfigurations. Par exemple, tous les générateurs du système normal (GEN1, GEN2 or APU) peuvent être utilisés pour fournir de l'électricité à toutes les barres de distribution des systèmes primaire ou de secours lorsque un ou deux générateurs ne sont pas utilisables. D'autres règles de reconfiguration s'appliquent lorsque un des transformateurs est perdu. Finalement, lorsque tous les générateurs du système normal sont perdus, le générateur de secours fournit de l'électricité exclusivement aux barres de distribution essentielles (ACess et DCess) évitant ainsi d'alimenter des consommateurs non prioritaires, comme les *galleys* qui regroupent des équipements commerciaux.

Dans la suite de ce chapitre nous allons utiliser les motifs selon les approches *bottom-up* et *top-down* que nous allons maintenant présenter.

6.3 Approche *Top-Down*

L'approche *top-down* consiste, après réalisation d'une analyse fonctionnelle et dysfonctionnelle (une AMDEC par exemple), à réaliser un modèle abstrait d'une architecture de système

en utilisant des motifs de sûreté. Le choix des motifs est guidé par les exigences imposées par les analyses préliminaires de sûreté. On dit alors que la conception est "orientée exigences". Le modèle abstrait ainsi obtenu est ensuite raffiné pour obtenir un jeu de prototypes satisfaisant également les exigences initiales. Le choix final du prototype retenu dépend ensuite de facteurs tels que la technologie employée, la masse minimale des composants implantés etc.

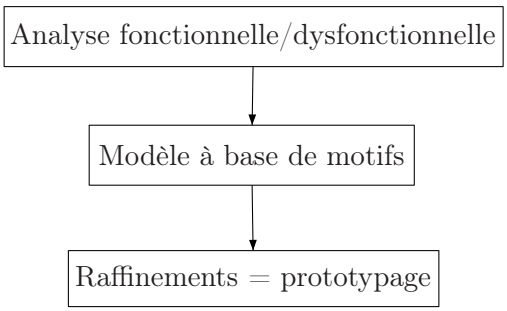


FIG. 6.3 – Chronologie de la conception *Top-down*

Comme nous l'avons déjà mentionné dans la présentation du système électrique, la principale exigence de sûreté qui nous intéresse est que la perte totale d'énergie électrique (c'est-à-dire la perte de trois barres de distribution) est considérée catastrophique. Nous nous sommes intéressés à deux autres exigences du même type, imposant que la perte de deux barres de distribution soit considérée comme majeure et la perte d'une barre de distribution soit considérée comme mineure.

Nous associons à chacune de ces exigences une exigence qualitative de la forme : "s'il y a moins de N défaillances individuelles, alors la perte de $N+1$ barres de distribution ne doit pas arriver" avec $N = 0, 1, 2$. Ce type d'exigences qualitatives permet de détecter de grosses anomalies, comme une panne simple qui causerait la perte de deux ou trois barres de distribution, en phase préliminaire de conception d'une architecture. Puis, lorsque les taux de défaillances des constituants du modèle sont connus, l'analyse de l'architecture peut être affinée en évaluant la tenue des exigences quantitatives.

La tenue de ces exigences impose implicitement une architecture de base au système. Le système électrique que nous cherchons à modéliser contient trois classes de barres de distribution. Nous pouvons déduire de la règle présentée précédemment que le système doit tolérer deux défaillances. Si l'on considère que cette architecture est basée sur une forme classique d'architecture du type Génération → Distribution → Consommation, alors chacune de ces composantes doit satisfaire cette exigence. Nous proposons par conséquent une architecture du type de celle présentée sur la figure 6.4.

Cette architecture est composée d'une génération tripliquée. Au sein de cette triplication, deux générateurs se redondent activement tandis que le troisième est passif. Ce choix est motivé par le fait que les générateurs sont généralement intégrés aux réacteurs et, par conséquent, deux générateurs sont actifs simultanément. On choisit également une triplication active en ce qui

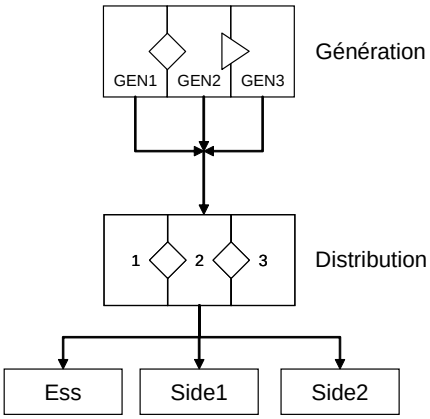


FIG. 6.4 – Architecture à base de motifs en top-down

concerne la distribution. Ceci correspond de manière concrète à une ségrégation des routes électriques menant à un même consommateur au sein du système. Cette architecture abstraite est satisfaisante vis à vis des exigences considérées. Le raffinement de cette abstraction permet de prototyper un ensemble d'architectures concrètes, comme par exemple celle de la figure 6.5.

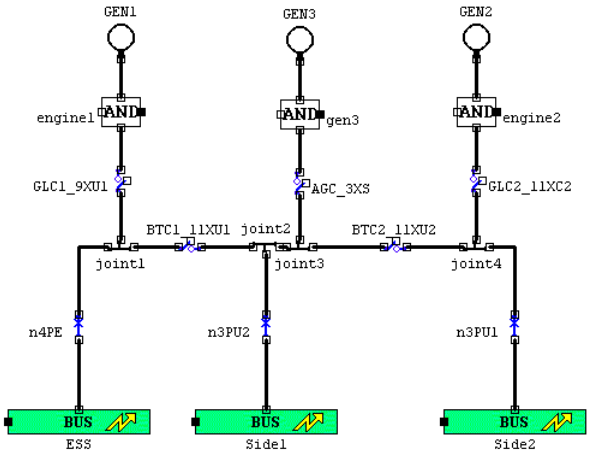


FIG. 6.5 – Concrétisation de l'architecture abstraite

Notons que par soucis de simplicité nous n'avons modélisé que trois "méta"-barres de distribution regroupant les barres de ces trois classes (i.e. consommateurs essentiels, side1 et side2). Notons également que l'architecture concrète réelle du système électrique n'est pas basé sur des duplications de la génération et de la distribution. En effet, Airbus a choisi un surdimensionnement du système, dépassant ainsi les exigences imposées par la réglementation. Ainsi la génération électrique, par exemple, est assurée par pas moins de quatre générateurs redondants : GEN1, GEN2, APU et CSMG (bien que l'APU ne soit pas à la base un élément de sûreté dans ce système, il peut néanmoins le devenir en cas de problème) et une triplication des routes électriques.

6.4 Approche *Bottom-Up*

Dans cette approche, l'objectif est de construire une architecture de système à partir d'une bibliothèque de composants, à créer ou pré-existante, et de la valider en pratiquant la reconnaissance de motifs. La phase préliminaire à cette modélisation consiste bien entendu à réaliser une analyse fonctionnelle et dysfonctionnelle. La reconnaissance de motifs au sein de l'architecture permet d'exhiber les propriétés de sûreté satisfaites par l'architecture. A la différence de l'approche précédente, la conception *bottom-up* n'est pas "orientée exigences" et il est nécessaire de procèder par tâtonnement afin de valider l'architecture.

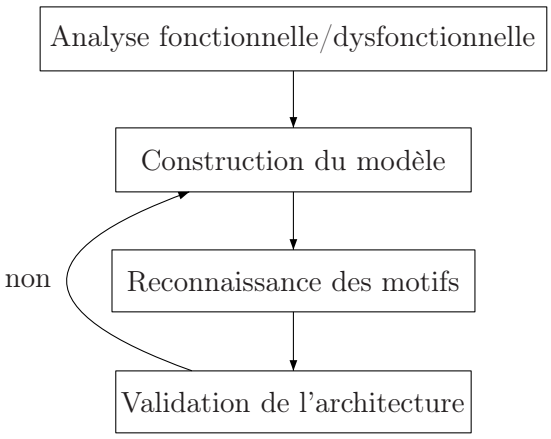


FIG. 6.6 – Chronologie de la conception *bottom-up*

6.4.1 Développement de la bibliothèque

Nous avons tout d'abord développé une bibliothèque de modèles de composants électriques de base comme des générateurs, disjoncteurs, transformateurs, contacteurs, barre de distribution et charges électriques. Pour chaque composant, nous avons modélisé l'effet des modes de défaillance sur son comportement interne ainsi que la propagation des défaillances dans son environnement. Nous nous sommes intéressés à des modes de défaillance pouvant aboutir à la perte totale ou partielle de puissance électrique. Nous supposons que tous les composants peuvent ne plus être capable de générer, transmettre ou fournir de l'électricité et que des courts-circuits peuvent survenir dans les barres d'alimentation. Finalement, les contacteurs et les disjoncteurs peuvent rester bloqués en position ouverte ou fermée. La table 6.1 résume les modes de défaillance associés aux composants de notre bibliothèque.

Nous utilisons les flux des composants du système électrique pour modéliser la propagation des défaillances au sein du système électrique. Il faut donc que les flux rendent compte de la présence ou de l'absence de tension et formaliser la propagation des courts-circuits. Etant donné l'architecture du système électrique étudié, il n'est pas possible de supposer que la propagation de la tension entre les bornes d'un composant aura toujours le même sens. En effet, en fonction des configurations des contacteurs, une borne ou l'autre imposera la tension

Composant	Modes de défaillance
contacteur	perte, bloqué
disjoncteur	perte, bloqué
transformateur	perte, court-circuit
charge électrique	perte, court-circuit
barre de distribution	perte, court-circuit
générateur	perte, court-circuit

TAB. 6.1 – Modes de défaillance pris en compte

au composant. Nous avons donc considéré que chaque borne des composants était susceptible d'avoir une tension en entrée et une tension en sortie. En revanche, la présence simultanée de tension en entrée sur plusieurs bornes d'un même composant est assimilée à un court-circuit du composant. En ce qui concerne les courts circuits nous avons ajouté des flux qui leurs sont dédiés à tous les composants de la bibliothèque, même à ceux auxquels ce mode de défaillance n'a pas été associé (comme les contacteurs et les disjoncteurs). Comme pour la tension, chaque borne du composant peut avoir un signal de court-circuit en entrée et en sortie. Ceci nous permet de propager le court-circuit dans l'ensemble du système électrique.

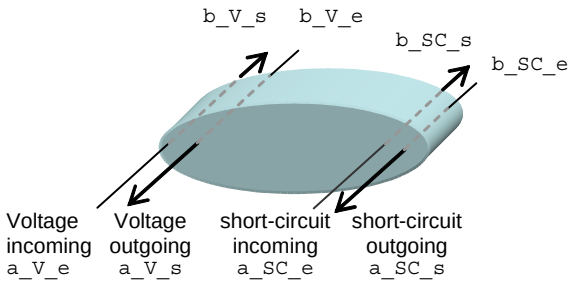


FIG. 6.7 – Flux associés à une barre de distribution

L'exemple suivant montre en détail comment nous modélisons la propagation des défaillances pour une barre de distribution. La barre a deux bornes : `a` et `b`, à chaque borne les quatre flux sont définis.

```
node bus_bar
 flow
  a_SC_e : bool : in; // port a incoming SC
  a_SC_s : bool : out; // port a outgoing SC
  a_V_e : bool : in; // port a incoming V
  a_V_s : bool : out; // port a outgoing V
  b_SC_e : bool : in; // port b incoming SC
  b_SC_s : bool : out; // port b outgoing SC
  b_V_e : bool : in; // port b incoming V
  b_V_s : bool : out; // port b outgoing V
```

```
state
  status : {ok,sc,lost};
 event
   fail_loss,fail_SC,update;
 trans
    status=ok) |- fail_loss -> status := lost;
   (status=ok) |- fail_SC -> status := sc;
   (status=ok) and ((a_V_e and b_SC_e) or
   (b_V_e and a_SC_e) or (a_V_e and b_V_e)) |- update -> status := sc;
 assert
   a_V_s = case((status=ok): b_V_e,
                 else : false);
   b_V_s = case((status=ok): a_V_e
                 else : false);
   a_SC_s = case((status=sc):  a_V_e,
                 (status=lost) : false,
                (status=ok): b_SC_e and a_V_e);
   b_SC_s = case((status=sc):  b_V_e,
                 (status=lost) : false,
                 (status=ok): a_SC_e and b_V_e);
 init
   status := ok;
edon
```

La barre de distribution peut être dans l'état nominal (`status=ok`), dans l'état perdu (`status=lost`) ou dans l'état court-circuit (`status=sc`). Les deux premières transitions décrivent le fait que la barre passe de l'état nominal vers les états perdus ou court-circuit lors de l'occurrence d'un événement de défaillance (`fail_loss` ou `fail_sc`). La dernière transition décrit trois cas pour lesquels le composant passe aussi dans l'état court-circuit. Ces cas correspondent à la propagation d'une défaillance dans l'environnement de la barre de distribution. Si la barre est dans l'état nominal et si sur une borne une tension en entrée est présente et sur la borne opposée un court-circuit en entrée est présent alors après occurrence de l'événement de mise à jour update le composant passe dans l'état court-circuit. De même, si la barre est dans l'état nominal et si une tension en entrée est simultanément présente sur les deux bornes de la barre alors le composant passe dans l'état court-circuit.

Les assertions associées aux signaux de présence de tension en sortie sur la borne `a` (`a_V_s`) et sur la borne `b` (`b_V_s`) modélisent la règle suivante : quand la barre est dans un état nominal elle propage la tension d'une borne vers l'autre, dans tous les autres modes nous supposons que la tension n'est pas propagée. Les assertions associées aux signaux de présence de court-circuit en sortie sur la borne `a` (`a_SC_s`) et sur la borne `b` (`b_SC_s`) modélisent la règle suivante : Quand la barre est dans l'état court-circuit, elle envoie un signal de court-circuit vers une borne si une tension en entrée est présente sur cette borne, quand la barre est perdue elle ne propage pas les courts-circuits et dans le mode nominal elle propage le signal de court-circuit reçu sur une borne vers l'autre borne si une tension en entrée est présente sur cette dernière.

6.4.2 Construction du modèle

Grâce aux capacités de l'éditeur graphique de modèles de l'environnement Cecilia OCAS, le modèle de l'architecture a été obtenu rapidement en interconnectant les composants puisés dans la bibliothèque. Le système électrique que nous avons modélisé utilise vingt classes de composants, il contient environ soixante-dix instances de composants.

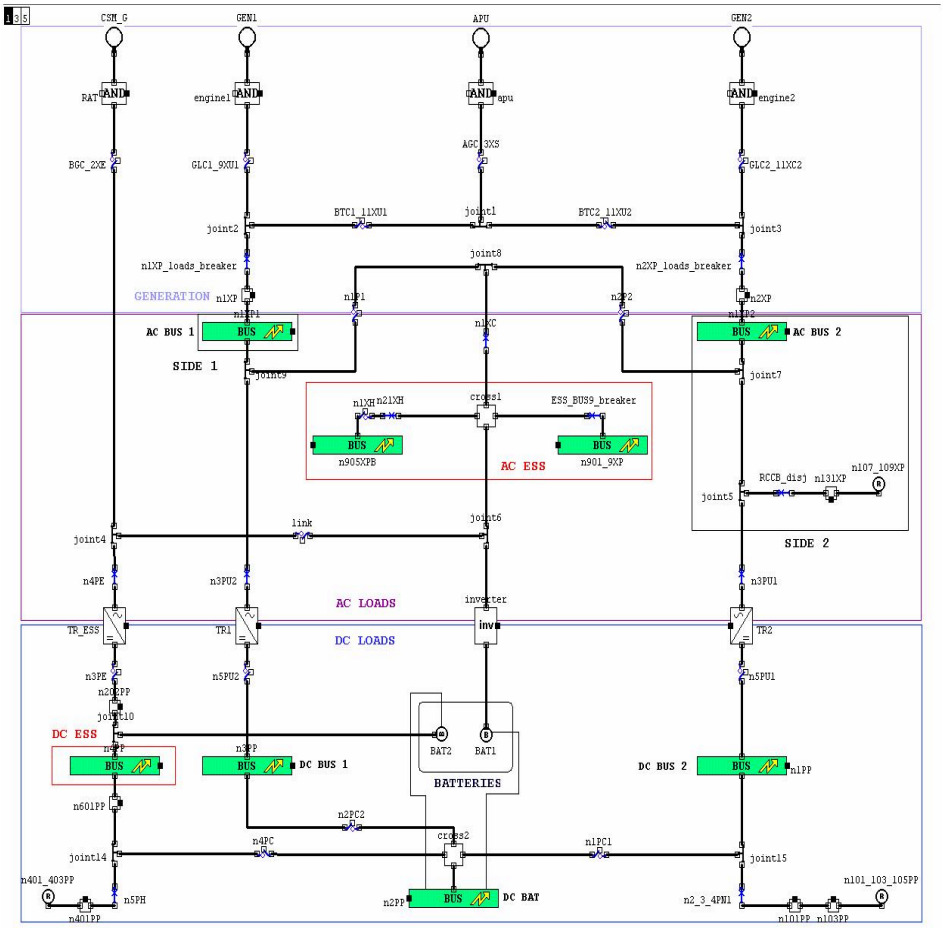


FIG. 6.8 – Vue graphique du modèle de système électrique

Le système comporte quinze contacteurs qui permettent un grand nombre de reconfigurations possibles. L'architecture du système électrique nous a été proposée par Airbus sans la description du contrôle des contacteurs. Nous avons donc développé une composant contrôleur qui délivre un ordre d'ouverture ou de fermeture à chacun des contacteurs du modèle en fonction de l'observation de l'état courant des générateurs et des transformateurs. Nous avons validé le modèle avec Airbus en utilisant le simulateur graphique de Cecilia OCAS. Pour cela nous avons déclenché des événements de défaillances dans un ou plusieurs composants, puis nous avons observé la propagation des défaillances dans l'ensemble du système électrique.

Le modèle ainsi créé comporte des circularités de calcul sur les flux de données dues à des

boucles physiques dans l'architecture du système électrique (i.e. des chemins tels que l'entrée d'un composant dépend de sa sortie). Nous avons cherché à limiter ces circularités car bien que le simulateur OCAS soit capable de traiter des modèles avec circularité, les techniques de vérification d'exigences (model-checker SMV ou SCADE, générateur automatique d'arbre de défaillance associés à AltaRica) ne sont pas capables de traiter ce type de modèles. Pour parvenir à casser ces boucles, nous avons introduit des délais de propagation pour les signaux de tension et de court-circuit. Par exemple, reprenons le modèle `bus_bar` présenté plus haut. En mode nominal le court-circuit se propage instantanément entre les bornes.

```
a_SC_s = case((status = sc):  a_V_e,
              else : false);
b_SC_s = case((status = sc):  b_V_e,
              else : false);
```

Si l'on supprime le cas du mode `ok` dans les assertions associées aux flux `a_SC_s` et `b_SC_s` alors le court-circuit ne se propage plus instantanément. En effet, si un court-circuit est présent sur une des bornes, ce n'est qu'après un événement `update` que la barre passe dans l'état `sc`, et que le court-circuit peut se propager sur l'autre borne.

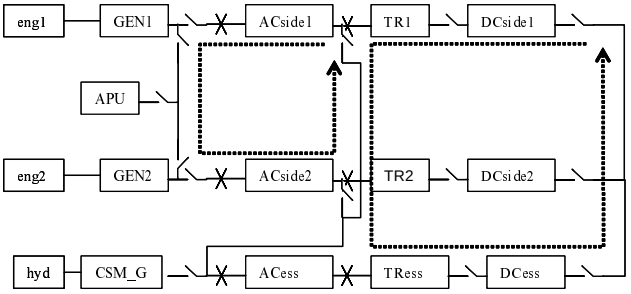


FIG. 6.9 – Cycles dans le système électrique

Pour éliminer toutes les définitions circulaires il n'est pas nécessaire de modifier tous les composants. Nous nous sommes contentés d'introduire des délais de propagation pour les signaux de tension et de court-circuit au niveau des composants de jonction présents dans toutes les boucles du modèle. Le modèle modifié est de type flot de données donc il peut être traité par les outils de vérification de modèles comme SMV ou SCADE mais la propagation des signaux électriques n'est pas instantanée.

6.4.3 Processus d'identification

Nous allons chercher à identifier des motifs élémentaires au sein de l'architecture du système électrique que nous venons de développer.

Considérons une identification simple. Celle-ci consiste à considérer des éléments de base, ou un petit groupe d'éléments de base, de l'architecture concrète comme des motifs élémentaires. On peut, par exemple, essayer de montrer qu'un générateur parfait (`S1`) et son contacteur associé (`ct1`) correspondent à un motif élémentaire appelé *block*, présenté ci-dessous, pour

Motif		Architecture concrète
i	→	true
a	→	ct1.activation
r	→	true
f	→	S1.failure or ct1.stucked
o	→	o

TAB. 6.2 – Identification des variables

lequel l'entrée, l'activation et la ressource sont toujours disponibles. L'activation de ce motif correspondra alors à l'état du contacteur.

```
node block
 flow a, f, i, r:in:bool;
  o, s:out:bool;
 assert s = not(f),
 extern environment G(a and not(f) -> r);
  guaranteed G(a and i and r and not(f) -> o);
edon
```

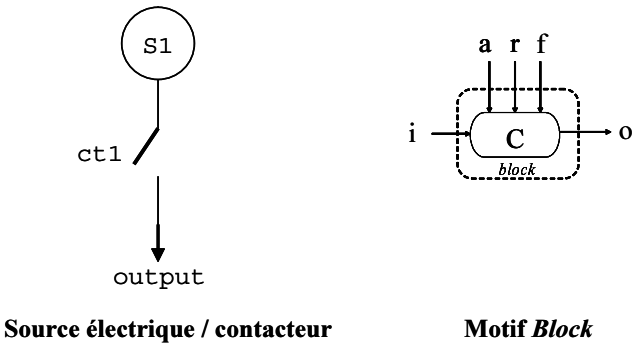


FIG. 6.10 – Partie d'architecture à substituer

Nous devons tout d'abord associer aux variables du motif les variables, ou combinaison de variables, de l'architecture réelle. Cette identification des variables est présentée dans le tableau 6.2 ci-dessous.
Par soucis de simplification nous avons forcé l'entrée et la ressource à la valeur "vrai". Cette simplification peut être supprimée si l'on considère que le réacteur qui alimente le générateur a un réservoir fini. Dans ce cas, le paramètre qui correspond à la ressource du motif *block* doit avoir en entrée le modéle du comportement de ce réservoir. Nous simplifions également en considérant que la défaillance du générateur correspond à la disjonction de la perte simple et d'un court-circuit. Nous devons montrer que l'architecture concrète est un raffinement du motif élémentaire. Pour cela nous avons le choix entre les vérificateurs de modèles SMV et MecV. La substitution de l'architecture initiale par le motif permet de passer d'un système à huit états (figure 6.11) à un système à quatre états (figure 6.12). Pour cela il est nécessaire de

montrer qu'une relation de simulation $\mathcal{R}$ existe entre ces deux systèmes ($A \prec B$). Les classes d'équivalence considérées sont les suivantes. Pour les variables d'état :

$$\{ok\}_B = (\{ok\}_A \wedge \{\neg bloque\}_A,\ \{a\}_B = \{\neg ouvert\}_A$$

Et pour les événements :

$$faute_A = d_B,\ bloque\_ouvert_A = d_B,\ bloque\_ferme_A = \epsilon_B,\ push_A = act_B \text{ et } \epsilon_A = \epsilon_B$$

Sous ces classes d'équivalence, le motif *block* simule l'assemblage source électrique/contacteur. L'existence de cette relation d'équivalence permet la validation de la première étape de la substitution.

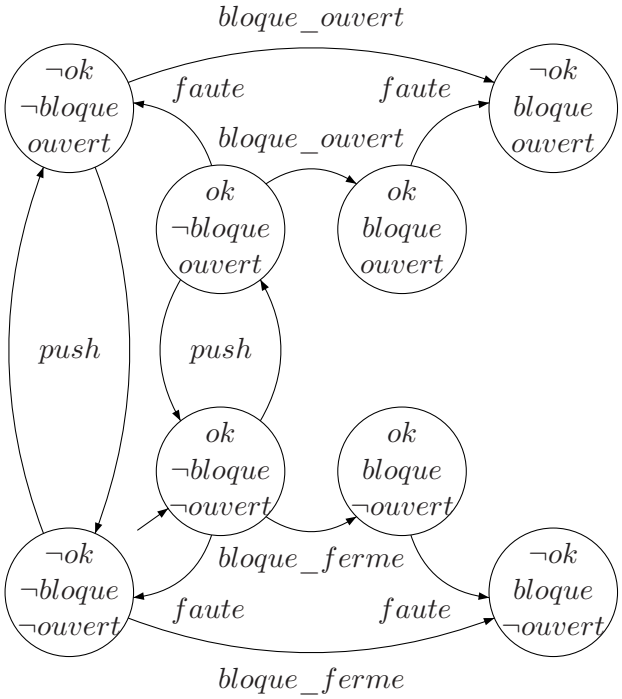


FIG. 6.11 – Source électrique/contacteur (A)

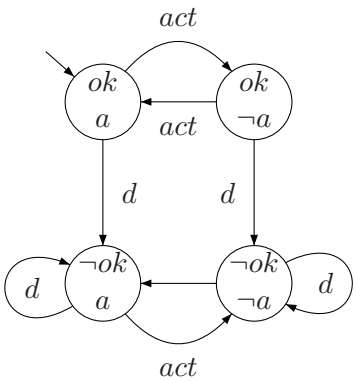


FIG. 6.12 – Motif *Block* (B)

L'identification des motifs n'est pas assistée. Par contre il est possible d'assister la validation de cette identification. Par la suite, nous avons utilisé le vérificateur de modèles SMV pour valider les différentes identifications, comme cela a été présenté au chapitre 5.
A la fin du processus d'identification/validation/substitution nous obtenons une architecture abstraite à base de motifs comme celle représentée sur la figure 6.13. Trois défaillances sont nécessaires pour faire tomber le système avec cette architecture ce qui signifie que l'on doit tenir la probabilité de perte totale de 10^{-9} imposée.

6.4.4 Propriétés instanciées

Le processus d'identification, et la validation qui suit, permet d'obtenir un ensemble de propriétés instanciées. Ces propriétés vont permettre une validation globale de l'architecture. Comme nous venons de le montrer, l'assemblage générateur/contacteur est un raffinement du motif élémentaire *block*. Une étape de la phase d'instanciation du motif consisté à vérifier la tenue des hypothèses d'environnement instanciées. Dans notre cas, l'hypothèse à vérifier est la suivante :

$$G((ct1.activation \wedge \neg(S1.failure \vee ct1.failure)) \Rightarrow true)$$

Cette propriété instanciée est une tautologie. Par conséquent, la substitution de ce morceau d'architecture permet de mettre en évidence la propriété garantie instanciée suivante :

$$G((ct1.activation \wedge \neg(S1.failure \vee ct1.failure)) \Rightarrow o)$$

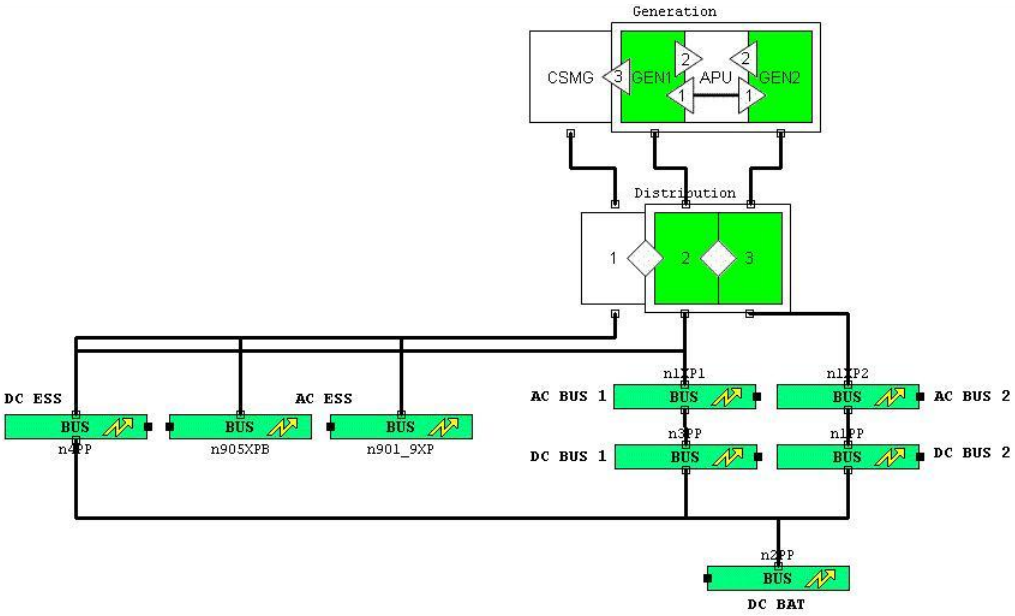


FIG. 6.13 – Architecture à base de motifs en bottom-up

6.5 Analyse par l'approche SAP et résultats

Ce test de performance a été réalisé sur un ensemble de propriétés formalisant quelques exigences. Nous ne présenterons ici que les exigences concernant la perte des barres électriques. Les exigences concernant les barres électriques peuvent être simplifiées comme suit :

- La perte d'une barre électrique doit être causée par au moins une défaillance
- La perte de deux barres doit être causée par au moins deux défaillances

Pour pouvoir vérifier ces exigences, un nœud jouant le rôle d'observateur ainsi qu'un compteur de défaillance ont été ajoutés dans chacun des modèles AltaRica afin de tester la tension en sortie des barres électriques. Les sorties de l'observateur sont les suivantes :

- `bus_bars_ok` est vraie lorsque toutes les barres "normales" fournissent de la tension
- `ESS_bars_ok` est vraie si toutes les barres "essentielles" fournissent de la tension
- `DCside1_DCside2_ok` est vraie lorsque la barre de courant continu située sur la ligne `side1` ou la barre de courant continu située sur `side2` fournit de la tension
- `DCside1_DCess_ok` est vraie lorsque la barre de courant continu située sur la ligne `side1` ou la barre de courant continu située sur la ligne `ess` fournit de la tension
- `DCside2_DCess_ok` est vraie lorsque la barre de courant continu située sur la ligne `side2` ou la barre de courant continu située sur la ligne `ess` fournit de la tension

Idem pour les barres de courant alternatif :

- `ACside1_ACside2_ok`
- `ACside1_ACess_ok`
- `ACside2_ACess_ok`

Les exigences précédentes ont été traduites en Logique Temporelle Linéaire. Celles concernant la perte d'une barre ont été traduites de la façon suivante :

- $G(upto\_0\_failure) \Rightarrow Gbus\_bars\_ok$
- $G(upto\_0\_failure) \Rightarrow GESS\_bars\_ok$

La traduction des exigences qui concernent la perte de deux barres est :

- $G(upto\_1\_failure) \Rightarrow GF(ACside1\_ACside2\_ok)$
- $G(upto\_1\_failure) \Rightarrow GF(ACside1\_ACess\_ok)$
- $G(upto\_1\_failure) \Rightarrow GF(ACside2\_ACess\_ok)$
- $G(upto\_1\_failure) \Rightarrow GF(DCside1\_DCside2\_ok)$
- $G(upto\_1\_failure) \Rightarrow GF(DCside1\_DCess\_ok)$
- $G(upto\_1\_failure) \Rightarrow GF(DCside2\_DCess\_ok)$

L'utilisation de l'opérateur temporel F est imposée par le temps de reconfiguration nécessaire en cas de défaillance d'un composant du système. Les résultats obtenus pour la vérification

de ces propriétés avec l'outil SMV sont représentés dans la table 6.3[2] et sur la figure 6.14.

Propriétés	TU M_C	NBA M_C	TU M_{SAP}	NBA M_{SAP}
ACside1_ACess_ok	1,272	123449	0,23	27399
ACside1_ACside2_ok	1,192	111098	0,68	27553
ACside2_ACess_ok	1,402	314498	0,71	28020
DCside1_DCess_ok	1,692	181866	0,73	28400
DCside1_DCside2_ok	1,542	124500	0,72	27800
DCside2_DCess_ok	1,352	131634	0,70	27794
Ess_bars_ok	0,7911	81652	0,45	15379
bus_bars_ok	0,6609	61907	0,48	15897

TAB. 6.3 – Performances (1)

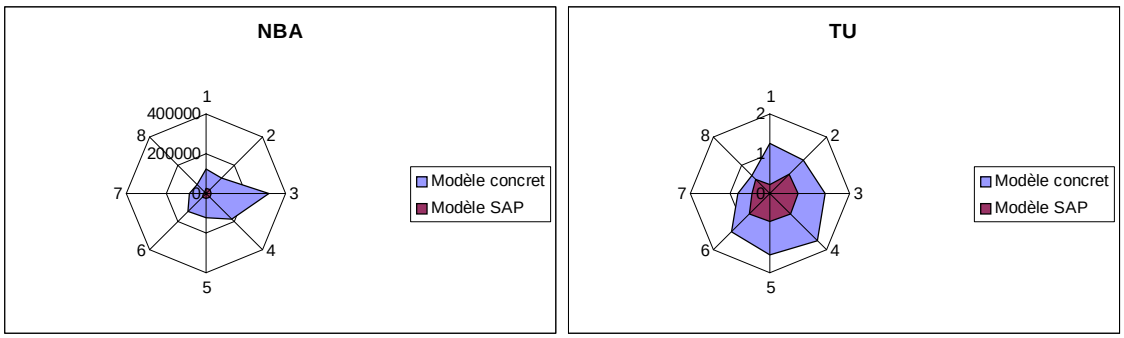


FIG. 6.14 – Performances (2)

Comme nous venons de le voir, les intérêts des motifs d'architectures de sûreté sont nombreux. Ils permettent, entre autre, de présenter une vue synthétique de l'architecture du système. Sur notre exemple de circuit électrique l'utilisation des motifs a entraîné une réduction de près de 80% du nombre de composants utilisés. Les résultats de cet allègement (présentés dans le tableau 6.3 ci-dessus[3]) montrent qu'outre une vision plus claire et fonctionnelle du système ils permettent un gain de temps grâce à une réduction de la taille des BDD manipulés (ce qui est logique puisque nous manipulons des abstractions). Ils permettent la vérification de la tenue des exigences de sûreté soit en créant directement le modèle à l'aide de motifs, soit en réalisant la reconnaissance de motifs dans un modèle existant. De plus, ils facilitent la création rapide de nouvelles architectures et favorisent ainsi le prototypage et donc la conception d'architectures sûres.

[2] M_C : Modèle concret, M_{SAP} : Modèle SAP, NBA : Nœud BDD alloués et TU : Temps utilisateur en secondes

[3] Matériel utilisé : SunBlade 1500, 1GHz, 1Go de RAM

Quatrième partie

Conclusion

Chapitre 7

Conclusion

7.1 Synthèse

Le domaine de cette thèse est celui de l'évaluation de la sûreté de fonctionnement de systèmes industriels complexes. Considérons, par exemple, un composant dont la sémantique serait représentée par automate à 10 états. La composition parallèle de 80 de ces composants au sein d'un système (complexe) produit un automate à 10^{80}, soit plus que le nombre d'atomes dans l'univers. Même si nous ne manipulons pas tous les jours de telles structures, l'illustration précédente a pour objectif de montrer l'importance d'abstraire pour analyser et de vérifier compositionnellement les systèmes complexes. C'est ce que nous avons chercher à faire au cours de cette thèse.

L'idée directrice, à la base de notre approche compositionnelle, est d'utiliser ou de réutiliser des composants génériques, abstractions d'architectures de sûreté, dont les propriétés ont été préalablement vérifiées. La conception (ou le prototypage) et l'analyse d'architectures de systèmes complexes sont souvent difficiles du fait de leur taille. Notre expérience de modélisation en AltaRica de plusieurs systèmes avion nous a permis de mettre en évidence un certain nombre d'assemblages de composants ayant pour but d'assurer la sûreté de l'architecture. La réutilisation de ces assemblages, rendus génériques et que l'on appelle motifs d'architecture de sûreté, permet de simplifier ces différentes tâches. Cette approche s'intégre parfaitement dans la tendance actuelle de l'ingéniérie qui consiste à construire des systèmes à partir de bibliothèques de composants. Nous avons procédé en trois étapes :

1. Nous avons défini les motifs d'architectures de systèmes pour la sûreté de fonctionnement. Cette définition contient une partie formelle caractérisant son comportement ainsi que ses propriétés et exigences, et une partie informelle sous forme d'attributs permettant une meilleure utilisation des motifs. Une notation graphique a également été définie afin de normaliser leur représentation au sein d'une architecture.

2. Nous avons cherché à formaliser ces motifs en définissant un cadre formel unique aux motifs d'architectures de sûreté. Ils possèdent en effet une double formalisation provenant des deux parents dont ils sont les enfants. Tout d'abord la sûreté de fonctionnement avec un aspect comportemental (orienté dysfonctionnel) qui se traduit en terme d'automates à contraintes, puis la logique mathématique avec un ensemble de propriétés

se formalisant en formules de logique temporelle et traduisant des exigences de sûreté. L'unification a été envisagée en utilisant le résultat des travaux de Chakine, Clarke *et al.* qui ont défini une logique dérivée de LTL, appelée SE-LTL, permettant de prendre en compte les événements comme des propositions atomiques et dont la sémantique est celle des structures de Kripke étiquetées. Nous avons ainsi présenté la traduction des automates à contraintes en LKS, puis leur produit avec des automates de Büchi issus de formules de SE-LTL. Ce cadre convient aux besoins des motifs comme le montre les illustrations présentées tout au long de ce mémoire.

3. Nous avons étudié des scénarios d'utilisation des motifs avec les approches *top-down* et *bottum-up*. La première approche consiste à prototyper des architectures à partir d'un modèle de plus haut niveau sûr. La seconde approche permet de valider une architecture concrète en pratiquant la reconnaissance de motifs. Nous avons également dégagé les différentes contraintes formelles à vérifier pour ces différents scénarios.

4. Nous avons appliqué cette approche sur plusieurs cas d'étude concrets, dont un a été présenté dans ce mémoire. L'utilisation des motifs rend la vue globale du système plus concise et facilite l'extraction des propriétés satisfaites. Sur notre exemple de circuit électrique l'utilisation des motifs a entraîné une réduction importante de la taille du modèle. Les résultats de cet allègement montrent qu'outre une vision plus claire et fonctionnelle du système ils permettent un gain de temps grâce à une réduction de la taille des BDD manipulés (car ce sont des abstractions).

Comme nous l'avons vu au cours de cette thèse, les intérêts des motifs d'architectures nous paraîssent nombreux. Les modèles basés motifs permettent la capitalisation du savoir faire des experts et de pratiquer la réutilisation d'éléments sûrs pour la conception de nouvelles architectures. Nous attendons de cette approche qu'elle offre une meilleure et plus large compréhension de la sûreté des systèmes et permettra un gain de temps lors de la phase préliminaire d'analyse de sûreté. Un autre bénéfice attendu est la possibilité de prototypage rapide de nouvelles architectures. En effet, l'utilisation de la bibliothèque de motifs a montré que la conception d'architectures est facilitée et accélérée. Le concepteur peut ainsi comparer plusieurs solutions d'architecture d'un même système, en termes de faisabilité ou de coût par exemple. Enfin, ils permettent la vérification de la tenue des exigences de sûreté soit en créant directement le modèle à l'aide de motifs, soit en réalisant la reconnaissance de motifs dans un modèle existant. Les exigences de sûreté sont analysées quasi-automatiquement grâce aux technologies de vérification disponibles. Nous avons utilisé au cours de notre étude le simulateur graphique Cecilia OCAS de Dassault couplé aux vérificateurs de modèles SMV et MecV.

7.2 Perspectives

Les perspectives de ce travail sont nombreuses. Nous les avons regroupé au sein des trois catégories suivantes :

1. Le catalogue actuel, tel que présenté en Annexe B, est incomplet. L'extension du catalogue actuel de motifs est donc nécessaire. A l'heure actuelle il ne contient que des motifs du type redondance et il est important d'étendre ce catalogue aux éléments de sûreté

plus complexes, comme les architectures COM-MON par exemple. Plus ce catalogue sera étoffé, plus les possibilités de prototypage et les moyens de validation des architectures seront forts.

2. La formalisation des motifs que nous avons proposée, basée sur un mélange entre les structures de Kripke étiquetées et une variante de la logique temporelle linéaire, n'est actuellement supportée par aucun outil. L'outillage constitue donc une perspective importante de notre approche. A court terme, il consiste au développement d'outils permettant la substitution (ou l'assistance à la substitution) d'un morceau d'architecture concret par un motif tout en évaluant la correction de cette substitution. A plus long terme, la reconnaissance automatique des motifs d'un catalogue à partir de la modélisation d'un système pourra aussi être envisagée. Elle pourra, entre autre, s'inspirer des travaux réalisés à l'Ecole des Mines de Nantes avec l'outil PTIDEJ en génie logiciel et présentés succintement au début de ce mémoire. Cette reconnaissance est loin d'être triviale puisque, comme nous l'avons vu tout au long de cette thèse, un motif est une structure complexe rassemblant des données hétérogènes. Nous pensons utiliser XML [Hoq00] comme support des outils autour des motifs d'architectures de sûreté. XML nous permettra de coder les méta-modèles, comme les diagramme de classe et modèles comportementaux, afin de faciliter la manipulation et les transformations des fichiers. Nous pensons que XML permettrait de réaliser un "pattern matching" simple et efficace.

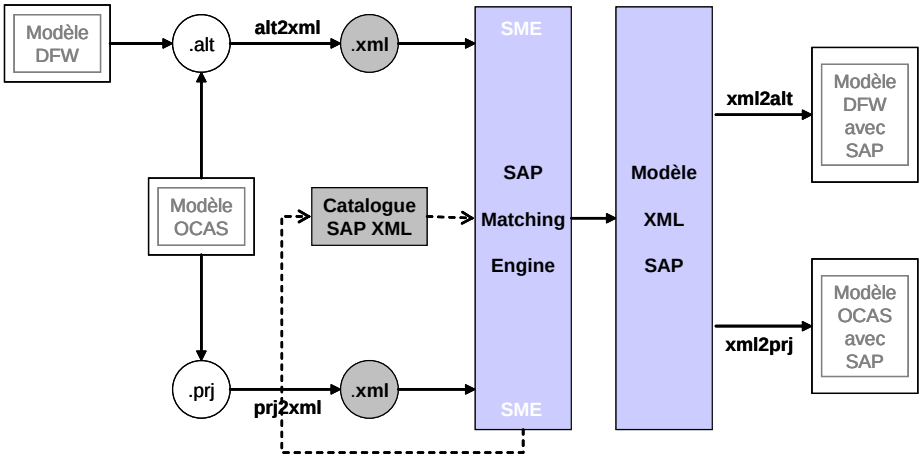


FIG. 7.1 – Pattern matching

La figure 7.1 montre une possibilité pour le processus de reconnaissance automatique de motifs utilisant XML. Partant d'une architecture ALTARICA classique et éventuellement de la structure de l'architecture en Cecilia OCAS, des passerelles permettent d'obtenir leur traduction en XML. Un moteur de reconnaissance réalise ensuite une comparaison des composants, ou de groupes de composants, de l'architecture en XML avec le catalogue de motifs. Puis, un processus automatique de validation (respectant les étapes de validation des motifs) autorise la substitution et construit la nouvelle structure XML. Enfin, de nouvelles passerelles permettent de revenir à une architecture ALTARICA classique et à un projet OCAS. A l'heure actuelle deux passerelles ont été réalisées : ALTARICA → XML (outil alt2xml) et architecture OCAS → XML (outil prj2xml).

Concernant les outils de vérification, les traductions possibles vers SMV puis la transformation des propriétés SE-LTL en LTL ou bien l'utilisation d'autres outils, comme Tina [BRV04] qui permet de traiter les formules de SE-LTL, nous semblent être des pistes prometteuses. Enfin, la simulation d'architectures à bases de motifs constitue également une perspective à étudier.

3. L'impact des motifs d'architecture de sûreté sur la structuration générale des arbres de défaillances nous semble intéressant à étudier. L'utilisation de motifs ou plus précisément l'allocation de propriétés au sein d'une architecture réalisée à l'aide de motifs, simplifie t-elle la construction ou l'analyse d'arbres sur ce type de systèmes ? Ces motifs, ou assemblages de motifs, ne constitueraient-ils pas les feuilles d'un arbre de défaillance dynamique ?

Cinquième partie

Annexes

Annexe A

La grammaire $AltaRica_{SAP}$

Remarque 2 *La grammaire du langage* AltaRica$_{/SAP}$ *est présentée au format Backus-Naur. Elle correspond à celle proposée par Aymeric Vincent dans sa thèse [Vin03] avec une extension aux expressions temporelles de la logique temporelle linéaire au niveau des assertions et des clauses externes. Ces ajouts apparaîtront en* ***gras*** *dans la grammaire. Notons que certains mots ne peuvent pas être utilisés comme nom d'identificateur, ces mots clés sont les suivants :*

```
and, assert, bool, case, const, clock, domain, else,
environment, event, extern, false, flow, guaranteed, if,
imply, init, min, max, node, not, or, sap, state, sub, sync,
then, trans, true
```

fichier-altarica :
    définition ; fichier-altarica
    ε

définition :
    définition-de-domaine
    déclaration-de-constante
    définition-de-nœud

définition-de-domaine :
    `domain` *identificateur* `=` *spécification-de-domaine*

spécification-de-domaine :
    `bool`
    `[` *expression, expression* `]`
    `{` *liste-d-identificateurs* `}`
    identificateur

définition-de-constante :
    `const` *identificateur* `=` *expression*

A.1 Définition d'un composant

définition-de-nœud :
`node` *identificateur liste-de-champs-de-nœud* `edon`

liste-de-champs-de-nœud :
champ-de-nœud ; liste-de-champ-de-nœud
directives-externes

champ-de-nœud :
définition-de-variables
déclaration-d-événements
définition-de-transitions
assertions
définition-de-sous-nœuds
définition-de-vecteurs-de-diffusion
définition-d-état-initial

A.2 Déclaration des variables et événements

définition-de-variables :
`state` *liste-de-variables*
`flow` *liste-de-variables*

liste-de-variables :
variable ; liste-de-variables
variable

variable :
liste-d-identificateurs : *domaine*

déclaration-d-événements :
`event` *liste-d-événements*

liste-d-événements :
liste-de-comparaisons-d-événements ; liste-d-événements
liste-de-comparaisons-d-événements

liste-de-comparaisons-d-événements :
comparaison-d-événements ; liste-de-comparaisons-d-événements
comparaison-d-événements

comparaison-d-événements :
comparaison-d-événements `<` *atome-de-comparaison-d-événements*
comparaison-d-événements `>` *atome-de-comparaison-d-événements*

atome-de-comparaison-d-événements

atome-de-comparaison-d-événements :
{*liste-de-comparaisons-d-événements*}
identificateur

A.3 Déclaration des transitions et assertions

définition-de-transitions :
`trans` *liste-de-transitions*

liste-de-transitions :
transition ; liste-de-transitions
transition ;

transition :
expression liste-de-successeurs

liste-de-successeurs :
successeur liste-de-successeurs
successeur

successeur :
`|-` *liste-d-identificateurs* `->` *liste-d-affectations*

liste-d-affectations :
affectation liste-d-affectations
affectation

affectation :
identificateur `:=` *expression*

assertions :
`assert` *liste-d-assertions*

liste-d-assertions :
liste-d-expressions
liste-d-expressions-temporelles

liste-d-expressions :
expression, liste-d-expressions
expression

***liste-d-expressions-temporelles* :**
expression-temporelle, liste-d-expressions-temporelles
expression-temporelle

A.4 Définition de sous nœuds

définition-de-sous-nœuds :
`sub` *liste-de-sous-nœuds*

liste-de-sous-nœuds :
sous-nœuds, liste-de-sous-nœuds
sous-nœuds

sous-nœuds :
liste-d-identificateurs : identificateur

définition-de-vecteurs-de-diffusion :
`sync` *liste-de-vecteurs-de-diffusion*

liste-de-vecteurs-de-diffusion :
vecteurs-de-diffusion, liste-de-vecteurs-de-diffusion
vecteurs-de-diffusion

vecteurs-de-diffusion :
`<` *liste-de-diffusions* `>` *contrainte-de-diffusion politique-de-diffusion*

liste-de-diffusions :
diffusion, liste-de-diffusions
diffusion

diffusion :
chemin ?

contrainte-de-diffusion :
`<` *entier*
`<=` *entier*
`>` *entier*
`>=` *entier*

politique-de-diffusion :
`min`
`max`

A.5 Définition de l'état initial et des directives externes

définition-d-état-initial :
`init` *liste-d-affectations-de-chemins*

liste-d-affectations-de-chemins :
affectation-de-chemin ; liste-d-affectations-de-chemins

affectation-de-chemin, liste-d-affectations-de-chemins
affectation-de-chemin

affectation-de-chemin :
chemin `:=` *expression*

directives-externes :
`extern` *liste-de-clauses*

liste-de-clauses :
liste-hypotheses-d-environnement
liste-proprietes-garanties
...

liste-hypotheses-d-environnement :
hypothese-d-environnement, liste-hypotheses-d-environnement
hypothese-d-environnement

liste-proprietes-garanties :
propriete-garantie, liste-proprietes-garanties
propriete-garantie

hypothese-d-environnement :
`environment` ***expression-temporelle***

propriete-garantie :
`guaranteed` ***expression-temporelle***

A.6 Expression

expression-temporelle :
expression-disjonction
`~` ***expression-temporelle***
expression-temporelle `&` ***expression-temporelle***
expression-temporelle `|` ***expression-temporelle***
expression-temporelle `=>` ***expression-temporelle***
expression-temporelle `=` ***expression-temporelle***
`G` ***expression-temporelle***
`F` ***expression-temporelle***
`X` ***expression-temporelle***
expression-temporelle `U` ***expression-temporelle***

expression :
expr-conditionnelle

expr-conditionnelle :

`if` *expression* `then` *expression* `else` *expression*
expr-case
expr-disjonction

expr-case :
`case {` *expr-liste-de-cas* `}`

expr-liste-de-cas :
expr-un-cas, expr-liste-de-cas
`else` *expression*

expr-un-cas :
expression `:` *expression*

expr-disjonction :
expr-disjonction `|` *expr-conjonction*
expr-conjonction

expr-conjonction :
expr-conjonction `&` *expr-comparaison-logique*
expr-comparaison-logique

expr-comparaison-logique :
expr-comparaison-logique `=` *expr-comparaison-arithmétique*
expr-comparaison-logique `!=` *expr-comparaison-arithmétique*
expr-comparaison-logique `=>` *expr-comparaison-arithmétique*
expr-comparaison-arithmétique

expr-comparaison-arithmétique :
expr-comparaison-arithmétique `<` *expr-additive*
expr-comparaison-arithmétique `<=` *expr-additive*
expr-comparaison-arithmétique `>` *expr-additive*
expr-comparaison-arithmétique `>=` *expr-additive*

expr-additive :
expr-additive `+` *expr-multiplicative*
expr-additive `-` *expr-multiplicative*
multiplicative

expr-multiplicative :
expr-multiplicative `*` *expr-unaire*
expr-multiplicative `/` *expr-unaire*
expr-unaire

expr-unaire :
`-` *expr-unaire*
`~` *expr-unaire*

expr-atomique

expr-atomique :
`(` *expression* `)`
chemin
entier
`true`
`false`

attributs :
`:` *liste-d-identificateurs*

entier :
*[0-9][0-9]**

identificateur :
$[a-zA-Z\_][0-9a-zA-Z\_]*$
'[^.][^.]'*

chemin :
identificateur . chemin
identificateur

liste-d-identificateurs :
identificateur, liste-d-identificateurs
identificateur

Annexe B

Quelques motifs

Le fragment de catalogue que nous présentons ici contient des motifs de redondances sous forme réduite, c'est à dire sans leurs attributs informels. Ces motifs ont été développés sur la base des quatre types d'architectures de patterns de la figure B.1.

Remarque 3 *Dans la suite de ce catalogue nous considérerons que les motifs ont plusieurs variables d'entrée et de sortie par défaut. Ces variables sont :*

Nom	*Type*	*Signification*
a	*entrée*	*activation*
f	*entrée*	*défaillance*
i	*entrée*	*entrée*
o	*sortie*	*sortie*
r	*entrée*	*ressource*
s	*sortie*	*état*

Toutes ces variables sont toutes booléennes (sauf si précisé autrement).

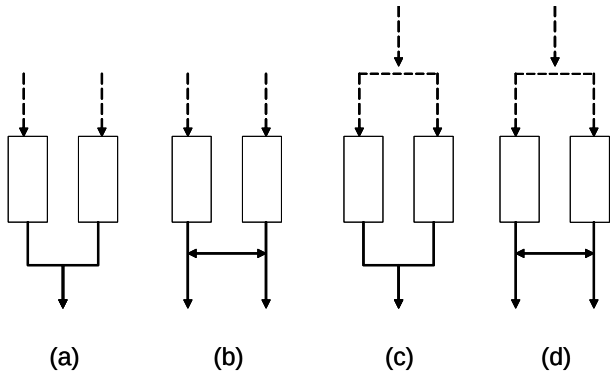


FIG. B.1 – Architecture en "Y" (a), "H" (b), "ϕ" (c) et "ϕH" (d)

B.1 Motifs élémentaires

B.1.1 SAP *Block*

Le motif *block* est un pattern d'architecture élémentaire permettant, par opération de composition, de créer des motifs plus complexes. Il n'est pas tolérant à la faute, par conséquent une défaillance simple entraîne sa perte. Il peut être utilisé comme vue abstraite de canaux d'information ou encore d'assemblages de composants.

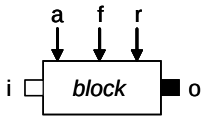


FIG. B.2 – SAP élémentaire Block

Déclaration

```
node block
 flow
  i, a, r, f : bool in;
  o : bool : out;
 assert
  G(i and a and r and not(f) -> o);
 extern
  guaranteed G(i and a and r and not(f) -> o);
edon
```

Modèle spécialisé SMV

```
/* BLOCK PROPERTIES VALIDATION */

module main(i, a, r, f, o){
input i, a, r, f : boolean;
output o : boolean;

/* BODY */
body : assert G(i & a & r & ~f <-> o)

/* ENVIRONMENT */
/* void */

/* GUARANTEED */
guaranteed : assert G(i & a & r & ~f <-> o);}

using body prove guaranteed;
```

B.2 Motifs composés "élémentaires"

L'appellation de ces patterns nécessite un éclaircissement. En effet, ces patterns sont considérés comme composés car tous utilisent une brique élémentaire dans leur solution. Cependant ils sont également qualifiés d'élémentaires car, par composition, ces huit motifs permettent de retrouver l'ensemble des architectures de sûreté que nous avons pu rencontrer lors de la modélisation de nos cas d'étude.

Pour chaque catégorie de redondance nous ne présenterons que les quatre types d'architectures vues précédemment, sous forme réduite, graphique fonctionnelle et graphique simplifiée. Nous donnons également la version spécialisée en SMV ayant permis la validation de chacun des motifs.

B.2.1 Redondance passive : *Cold Spare SAP*

Redondance en "H"

Déclaration

```
node cossap
 flow
  o', o'' : bool: out;
 sub
  B1, B2: block;
 assert
  G((B1.o or B2.o) <-> (o' and o''));
  G(not(B1.f) <-> F B1.a);
  G(not(B2.f) and B1.f <-> F B2.a);
 extern
  environment G(B1.a <-> F B1.r),
   G(B2.a <-> F B2.r),
   G(B1.i = true),
   G(B2.i = true),
   G(~(B1.f and B2.f));
   guaranteed G(F (o' or o''));
edon
```

Modèle spécialisé SMV

```
/* CoSSAP PROPERTIES VALIDATION */

module main(i1, r1, i2, r2, o', o''){
input i1, r1, i2, r2 : boolean;
output o', o'' : boolean;
a1, f1, o1, a2, f2, o2 : boolean;

/* BODY */
guaranteed_block1 : assert G(i1 & a1 & r1 & ~f1 <-> o1);
```

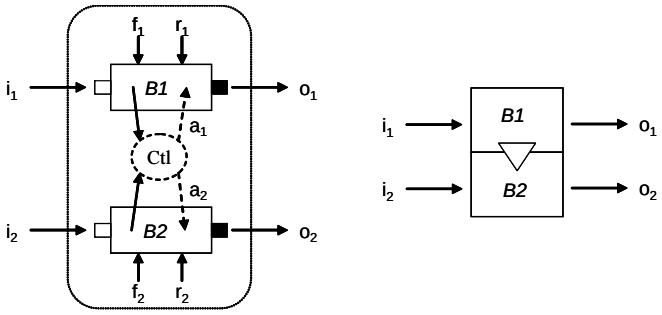


FIG. B.3 – CoSSAP en "H"

```
guaranteed_block2 : assert G(i2 & a2 & r2 & ~f2 <-> o2);
B1B2_merge : assert G((o1 | o2) <-> (o' & o''));
B1_activation : assert G(~f1 <-> F a1);
B2_activation : assert G(~f2 & f1 <-> F a2);

/* ENVIRONMENT */
B1_resource : assert G(a1 <-> F r1);
B2_resource : assert G(a2 <-> F r2);
B1_input : assert G(i1 = 1);
B2_input : assert G(i2 = 1);
B1B2_fail : assert G(~(f1 & f2));

/* GUARANTEED */
guaranteed : assert G(F (o' | o''));

using guaranteed_block1, guaranteed_block2, B1B2_merge,
B1_activation, B2_activation, B1_resource,
B2_resource, B1_input, B2_input, B1B2_fail prove guaranteed; }
```

Redondance en "Y"

Déclaration

```
node cossap
 flow
  o : bool: out;
 sub
  B1, B2 : block;
 assert
  G((B1.o or B2.o) <-> o);
  G(not(B1.f) <-> F B1.a);
  G(not(B2.f) and B1.f <-> F B2.a);
 extern
  environment G(B1.a <-> F B1.r),
   G(B2.a <-> F B2.r),
```

```
    G(B1.i = true),
    G(B2.i = true),
    G(not(B1.f and B2.f));
   guaranteed G(F o);
 edon
```

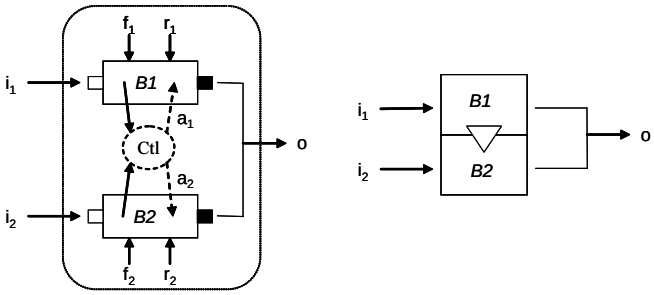


FIG. B.4 – CoSSAP en "Y"

Modèle spécialisé SMV

```
/* CoSSAP PROPERTIES VALIDATION */

module main(i1, r1, i2, r2, o){
input i1, r1, i2, r2 : boolean;
output o : boolean;
a1, f1, o1, a2, f2, o2 : boolean;

/* BODY */
guaranteed_block1 : assert G(i1 & a1 & r1 & ~f1 <-> o1);
guaranteed_block2 : assert G(i2 & a2 & r2 & ~f2 <-> o2);
B1B2_merge : assert G((o1 | o2) <-> o);
B1_activation : assert G(~f1 <-> F a1);
B2_activation : assert G(~f2 & f1 <-> F a2);

/* ENVIRONMENT */
B1_resource : assert G(a1 <-> F r1);
B2_resource : assert G(a2 <-> F r2);
B1_input : assert G(i1 = 1);
B2_input : assert G(i2 = 1);
B1B2_fail : assert G(~(f1 & f2));

/* GUARANTEED */
guaranteed : assert G(F o);

using guaranteed_block1, guaranteed_block2, B1B2_merge,
B1_activation, B2_activation, B1_resource,
B2_resource, B1_input, B2_input, B1B2_fail prove guaranteed; }
```

Redondance en "ϕ"

Déclaration

```
node cossap
 flow
  o : bool: out;
 sub
  B1, B2 : block;
 assert
  G((B1.o or B2.o) <-> o);
  G(not(B1.f) <-> F B1.a);
  G(not(B2.f) and B1.f <-> F B2.a);
 extern
  environment G(B1.a <-> F B1.r),
   G(B2.a <-> F B2.r),
   G(B1.i = true),
   G(B2.i = true),
   G(not(B1.f and B2.f));
  guaranteed G(F o);
edon
```

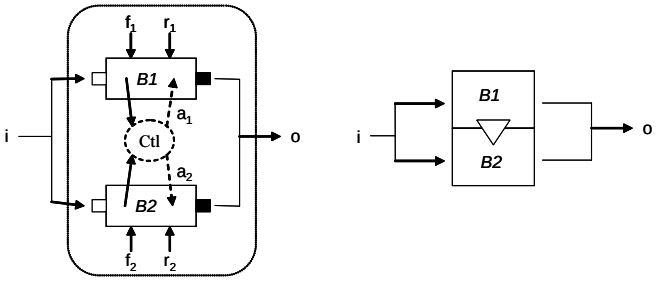


FIG. B.5 – CoSSAP en "ϕ"

Modèle spécialisé SMV

```
/* CoSSAP PROPERTIES VALIDATION */

module main(i, r1, r2, o){
input i, r1, r2 : boolean;
output o : boolean;
a1, f1, o1, a2, f2, o2 : boolean;

/* BODY */
guaranteed_block1 : assert G(i & a1 & r1 & ~f1 <-> o1);
guaranteed_block2 : assert G(i & a2 & r2 & ~f2 <-> o2);
B1B2_merge : assert G((o1 | o2) <-> o);
```

```
B1_activation : assert G(~f1 <-> F a1);
B2_activation : assert G(~f2 & f1 <-> F a2);

/* ENVIRONMENT */
B1_resource : assert G(a1 <-> F r1);
B2_resource : assert G(a2 <-> F r2);
B1_input : assert G(B1.i = 1);
B2_input : assert G(B2.i = 1);
B1B2_fail : assert G(~(f1 & f2));

/* GUARANTEED */
guaranteed : assert G(F o);

using guaranteed_block1, guaranteed_block2, B1B2_merge,
B1_activation, B2_activation, B1_resource,
B2_resource, B1_input, B2_input, B1B2_fail prove guaranteed; }
```

Redondance en "ϕH"

Déclaration

```
node cossap
 flow
  o', o'' : bool: out;
 sub
  B1, B2 : block;
 assert
  G((B1.o or B1.o) <-> (o' and o''));
  G(not(B1.f) <-> F B1.a);
  G(not(B2.f) and B1.f <-> F B2.a);
 extern
  environment G(B1.a <-> F B1.r),
   G(B2.a <-> F B2.r),
   G(B1.i = true),
   G(B2.i = true),
   G(not(B1.f and B2.f));
  guaranteed G(F (o' or o''));
edon
```

Modèle spécialisé SMV

```
/* CoSSAP PROPERTIES VALIDATION */

module main(i, r1, r2, o', o''){
input i, r1, r2 : boolean;
output o', o'' : boolean;
a1, f1, o1, a2, f2, o2 : boolean;
```

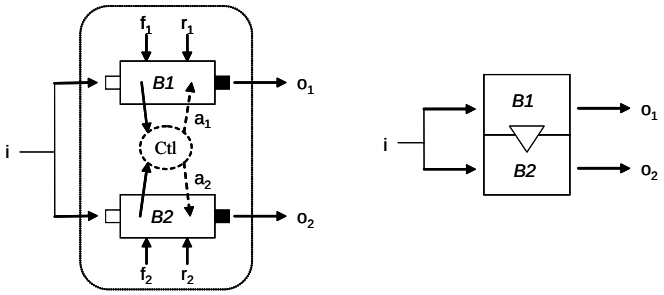


FIG. B.6 – COSSAP en "ϕH"

```
/* BODY */
guaranteed_block1 : assert G(i & a1 & r1 & ~f1 <-> o1);
guaranteed_block2 : assert G(i & a2 & r2 & ~f2 <-> o2);
B1B2_merge : assert G((B1.o | B2.o) <-> (o' | o''));
B1_activation : assert G(~f1 <-> F a1);
B2_activation : assert G(~f2 & f1 <-> F a2);

/* ENVIRONMENT */
B1_resource : assert G(a1 <-> F r1);
B2_resource : assert G(a2 <-> F r2);
B1_input : assert G(B1.i = 1);
B2_input : assert G(B2.i = 1);
B1B2.fail : assert G(~(f1 & f2));

/* GUARANTEED */
guaranteed : assert G(F (o' | o''));

using guaranteed_block1, guaranteed_block2, B1B2_merge,
B1_activation, B2_activation, B1_resource,
B2_resource, B1_input, B2_input, B1B2_fail prove guaranteed; }
```

B.2.2 Redondance active : *Hot Spare SAP*

Redondance en "H"

Déclaration

```
node hossap
 flow
  o', o'' : bool: out;
 sub
  B1, B2 : block;
 assert
  G((B1.o or B2.o) <-> (o' and o''));
  G(not(B1.f) <-> F B1.a);
```

```
   G(not(B2.f) and B1.f <-> F B2.a);
  extern
   environment G(B1.a <-> F B1.r),
    G(B2.a <-> F B2.r),
    G(B1.i = true),
    G(B2.i = true),
    G(not(B1.f and B2.f)),
   guaranteed G(F (o' or o''));
 edon
```

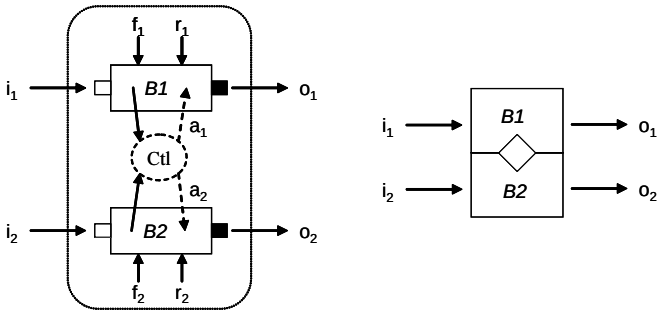


FIG. B.7 – HOSSAP en "H"

Modèle spécialisé SMV

```
/* HoSSAP PROPERTIES VALIDATION */

module main(i1, r1, i2, r2, o', o''){
input i1, r1, i2, r2 : boolean;
output o', o'' : boolean;
a1, f1, o1, a2, f2, o2 : boolean;

/* BODY */
guaranteed_block1 : assert G(i1 & a1 & r1 & ~f1 <-> o1);
guaranteed_block2 : assert G(i2 & a2 & r2 & ~f2 <-> o2);
B1B2_merge : assert G((B1.o | B2.o) <-> (o' & o''));
B1_activation : assert G(~f1 <-> F a1);
B2_activation : assert G(~f2 <-> F a2);

/* ENVIRONMENT */
B1_resource : assert G(a1 <-> F r1);
B2_resource : assert G(a2 <-> F r2);
B1_input : assert G(B1.i = 1);
B2_input : assert G(B2.i = 1);
B1B2_fail : assert G(~(f1 & f2));

/* GUARANTEED */
```

```
guaranteed : assert G(F (o' | o''));

using guaranteed_block1, guaranteed_block2, B1B2_merge,
B1_activation, B2_activation, B1_resource,
B2_resource, B1_input, B2_input, B1B2_fail prove guaranteed; }
```

Redondance en "Y"

Déclaration

```
node hossap
 flow
  o : bool: out;
 sub
  B1, B2 : block;
 assert
  G((B1.o or B2.o) <-> o);
  G(not(B1.f) <-> F B1.a);
  G(not(B2.f) and B1.f <-> F B2.a);
 extern
  environment G(B1.a <-> F B1.r),
   G(B2.a <-> F B2.r),
   G(B1.i = true),
   G(B2.i = true),
   G(not(B1.f and B2.f));
  guaranteed G(F o);
edon
```

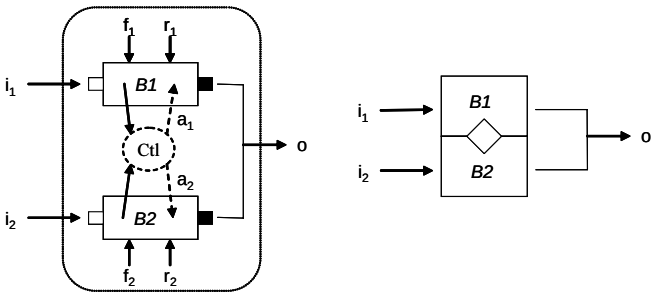


FIG. B.8 – HOSSAP en "Y"

Modèle spécialisé SMV

```
/* HoSSAP PROPERTIES VALIDATION */

module main(i1, r1, i2, r2, o', o''){
input i1, r1, i2, r2 : boolean;
output o : boolean;
```

```
a1, f1, o1, a2, f2, o2 : boolean;

/* BODY */
guaranteed_block1 : assert G(i1 & a1 & r1 & ~f1 <-> o1);
guaranteed_block2 : assert G(i2 & a2 & r2 & ~f2 <-> o2);
B1B2_merge : assert G((B1.o | B2.o) <-> o);
B1_activation : assert G(~f1 <-> F a1);
B2_activation : assert G(~f2 <-> F a2);

/* ENVIRONMENT */
B1_resource : assert G(a1 <-> F r1);
B2_resource : assert G(a2 <-> F r2);
B1_input : assert G(B1.i = 1);
B2_input : assert G(B2.i = 1);
B1B2_fail : assert G(~(f1 & f2));

/* GUARANTEED */
guaranteed : assert G(F o);

using guaranteed_block1, guaranteed_block2, B1B2_merge,
B1_activation, B2_activation, B1_resource,
B2_resource, B1_input, B2_input, B1B2_fail prove guaranteed; }
```

Redondance en "ϕ"

Déclaration

```
node hossap
 flow
  o : bool: out;
 sub
  B1, B2 : block;
 assert
  G((B1.o or B2.o) <-> o);
  G(not(B1.f) <-> F B1.a);
  G(not(B2.f) and B1.f <-> F B2.a);
 extern
  environment G(B1.a <-> F B1.r),
   G(B2.a <-> F B2.r),
   G(B1.i = true),
   G(B2.i = true),
   G(not(B1.f and B2.f));
  guaranteed G(F o);
edon
```

Modèle spécialisé SMV

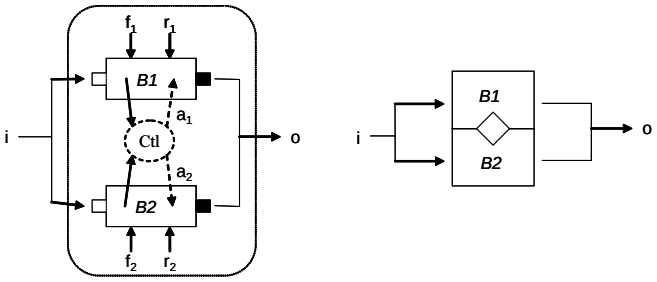


FIG. B.9 – CoSSAP en "ϕ"

```
/* HoSSAP PROPERTIES VALIDATION */

module main(i, r1, r2, o', o''){
input i, r1, r2 : boolean;
output o : boolean;
a1, f1, o1, a2, f2, o2 : boolean;

/* BODY */
guaranteed_block1 : assert G(i & a1 & r1 & ~f1 <-> o1);
guaranteed_block2 : assert G(i & a2 & r2 & ~f2 <-> o2);
B1B2_merge : assert G((B1.o | B2.o) <-> o);
B1_activation : assert G(~f1 <-> F a1);
B2_activation : assert G(~f2 <-> F a2);

/* ENVIRONMENT */
B1_resource : assert G(a1 <-> F r1);
B2_resource : assert G(a2 <-> F r2);
B1_input : assert G(B1.i = 1);
B2_input : assert G(B2.i = 1);
B1B2_fail : assert G(~(f1 & f2));

/* GUARANTEED */
guaranteed : assert G(F o);

using guaranteed_block1, guaranteed_block2, B1B2_merge,
B1_activation, B2_activation, B1_resource,
B2_resource, B1_input, B2_input, B1B2_fail prove guaranteed; }
```

Redondance en "ϕH"

Déclaration

```
 node hossap
  flow
   o', o'' : bool: out;
  sub
```

```
   B1, B2 : block;
  assert
   G((B1.o or B2.o) <-> (o' and o''));
   G(not(B1.f) <-> F B1.a);
   G(not(B2.f) and B1.f <-> F B2.a);
  extern
   environment G(B1.a <-> F B1.r),
    G(B2.a <-> F B2.r),
    G(B1.i = true),
    G(B2.i = true),
    G(not(B1.f and B2.f));
   guaranteed G(F (o' or o''));
 edon
```

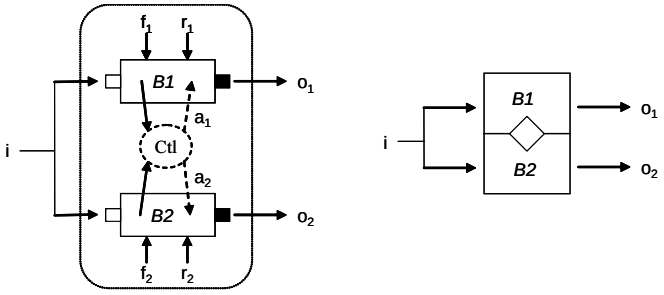


FIG. B.10 – CoSSAP en "ϕH"

Modèle spécialisé SMV

```
/* HoSSAP PROPERTIES VALIDATION */

module main(i, r1, r2, o', o''){
input i, r1, r2 : boolean;
output o', o'' : boolean;
a1, f1, o1, a2, f2, o2 : boolean;

/* BODY */
guaranteed_block1 : assert G(i & a1 & r1 & ~f1 <-> o1);
guaranteed_block2 : assert G(i & a2 & r2 & ~f2 <-> o2);
B1B2_merge : assert G((B1.o | B2.o) <-> (o' & o''));
B1_activation : assert G(~f1 <-> F a1);
B2_activation : assert G(~f2 <-> F a2);

/* ENVIRONMENT */
B1_resource : assert G(a1 <-> F r1);
B2_resource : assert G(a2 <-> F r2);
B1_input : assert G(B1.i = 1);
```

```
B2_input : assert G(B2.i = 1);
B1B2_fail : assert G(~(f1 & f2));

/* GUARANTEED */
guaranteed : assert G(F (o' | o''));

using guaranteed_block1, guaranteed_block2, B1B2_merge,
B1_activation, B2_activation, B1_resource,
B2_resource, B1_input, B2_input, B1B2_fail prove guaranteed; }
```

Annexe C

Articles publiés

1. C. Kehren, C. Seguin - *Evaluation qualitative de systèmes physiques pour la sûreté de fonctionnement* - Formalisation des activités concurrentes (FAC03), Toulouse, France, Mars 2003

2. M. Bozzano, A. Villafiorita, O. Akerlund, P. Bieber, C. Bougnol, E. Böde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, A. Cimatti, A. Griffault, C. Kehren, B. Lawrence, A. Lüdtke, S. Metge, C. Papadopoulos, R. Passarello, T. Peikenkamp, P. Persson, C. Seguin, L. Trotta, L. Valacca et G. Zacco - *ESACS : an integrated methodology for design and safety analysis of complex systems* - European Safety and Reliability Conference (ESREL03), Maastricht, Pays-Bas, Juin 2003

3. C. Kehren, C. Seguin, P. Bieber et C. Castel - *Safety Architecture Patterns : une introduction* - Formalisation des activités concurrentes (FAC04), Toulouse, France, Mars 2004

4. C. Kehren, C. Seguin, P. Bieber, C. Castel, C. Bougnol, J.-P. Heckmann et S. Metge - *Architecture patterns for safe design* - AAAF International Complex and Safe Systems Engineering (CS2E04), Arcachon, France, Juin 2004

5. C. Kehren, C. Seguin, P. Bieber, C. Castel, C. Bougnol, J.-P. Heckmann et S. Metge - *Advanced multi-system simulation capabilities with AltaRica* - Actes de International System Safety Conference (ISSC04), p489-498, Rhode Island, Etats-Unis, Août 2004

6. C. Kehren, C. Seguin, P. Bieber, C. Castel - *Analyse des exigences de sûreté d'un système électrique par model-checking* - Actes du Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement (lm14), p492-497, Bourges, France, Octobre 2004

7. P. Bieber, C. Kehren, C. Seguin, C. Castel, C. Bougnol, J.-P. Heckmann et S. Metge - *Safety assessment with AltaRica. Lessons learnt based on two aircraft system studies* - Actes du World Computer Congress (WCC04), p505-510, Toulouse, France, Août 2004. Kluwer Academic Publishers

8. C. Kehren et C. Seguin - *Automates AltaRica de propriétés* - Formalisation des activités concurrentes (FAC05), Toulouse, France, Mars 2005

9. C. Kehren et C. Seguin - *Des automates à contraintes aux structures de Kripke étiquetées* - Colloque annuel des doctorants de l'école doctorale informatique et télécommunications, UPS, (EDIT05), Toulouse, France, Avril 2005

10. P. Bieber, C. Castel, L. Cholvy, H. Demmou, C. Kehren, M. Medjoudj, N. Rivière, C. Seguin, R. Valette - *Qualtitative formalisation of critical scenarios wrt dynamic system models* - Actes du Congrès Qualita 2005, Bordeaux, France, 2005

Bibliographie

[92667] Aerospace Recommended Pratice 926. *Design analysis procedure for failure mode, effects and criticality analysis*. Society of Automotive Engineers - Aerospace, 1967.

[AA03] H. Albin-Amiot. *Idiomes et patterns Java : Application à la synthèse de code et à la détection*. PhD thesis, Ecole des Mines de Nantes and Université de Nantes, February 2003.

[AACGJ01] H. Albin-Amiot, P. Cointe, Y.-G. Gueheneuc, and N. Jussien. Instantiating and detecting design patterns : putting bits and pieces together. In *Proceedings of 16th IEEE conference on Automated Software Engineering*, 2001.

[ACD98a] G.S. Avrunin, J.C. Corbett, and M.B. Dwyer. Patterns in property specifications for finite-state verification. Technical Report UM-CS-1998-035, 1998.

[ACD98b] G.S. Avrunin, J.C. Corbett, and M.B. Dwyer. Property specification patterns for finite-state verification. In *Proceedings of Formal Methods in Software Practice*, 1998.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, July 1997.

[Ale77] C. Alexander. *A pattern language*. Oxford university press edition, 1977.

[All97] R. Allen. *A formal approach to software architecture*. PhD thesis, Carnegie Mellon University, May 1997.

[APGR00] A. Arnold, G. Point, A. Griffault, and A. Rauzy. The altarica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40 :109–124, 2000.

[Arn94] A. Arnold. *Finite transition systems*. Prentice-hall edition, 1994.

[Bac78] R.J. Back. *On the correctness of refinement in program development*. PhD thesis, University of Helsinky, 1978.

[Bac88] R.J. Back. *A calculus of refinements for program derivations*, volume 25, pages 593–624. Acta informatica edition, 1988.

[Bat86] J. Batut. Fiabilité prévisionnelle du réseau à très haute tension d'edf. In *5eme Colloque international de fiabilité et de maintenabilité, Biarritz, France*, October 1986.

[BBDE+01] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In Springer, editor, *Proceedings of Conference on Computer Aided Verification (CAV)*, volume 2102, 2001.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 10^{20} states and beyond. In *Proceedings of 5th Annual IEEE Symposium on Logic in Computer Science*, 1990.

[Bec94] K. Beck. Patterns generate architectures. In *European Conference on Object Oriented Programming*, 1994.

[BEJV96] P. Binns, M. Engellheart, M. Jackson, and S. Vestal. Domain specific software architectures for guidance, navigation and control. *International Software and Knowledge Engineering*, 6(2), 1996.

[Bou02] M. Bouissou. Boolean logic driven markov processes : a powerful new formalism and solving very large markov models. In *Proceedings of Probabilistic Safety Assessment and Management*, 2002.

[BR94] S. Brlek and A. Rauzy. Synchronization of constrained transition systems. In *Proceedings of the First International Symposium on Parallel Symbolic Computation*, pages 54–62. World Scientific Publishing, 1994.

[BRV04] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool tina : construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14), July 2004.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8) :677–691, August 1986.

[BSSR99] F. Buschmann, D. Schmidt, M. Stal, and H. Rohnert. Pattern-oriented software architecture, volume 2. In *Wiley series in software design patterns*, 1999.

[CCD98] L. Cholvy, F. Cuppens, and R. Demolombe. Logiques modales et bases de données. *Revue Technique et Sciences Informatiques*, 17, 1998.

[CCO+04] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event based software model checking. In LNCS, editor, *Proceedings of IFM*, volume 2004, 2004.

[Cha00] E. Chatelet. *Sûreté de fonctionnement : méthodes et outils de base*. Université de technologie de troyes edition, 2000.

[CL96] Y. Caseau and F. Laburthe. Claire : Combining objects and rules for problem solving. In *JICSLP workshop on multi-paradigm logic programming*, 1996.

[CM02] M. Cepin and B. Mavko. A dynamic fault tree. *Reliability Engineering and System Safety*, 75 :83–91, 2002.

[Dij76] E.W. Dijkstra. *A discipline of programming*. Prentice-hall edition, 1976.

[Dou98] B.P. Douglass. *Real-Time UML : Developing efficient objects for embedded systems*. Addison-Wesley, 1998.

[DS64] C.F. Desieno and L.L. Stine. A probability method for determining the reliability of electric power systems. *IEEE Transaction on Power Aparatus and Systems*, 83 :174–181, February 1964.

[Eme95] E.A. Emerson. *Temporal and modal logic*. Computer Science Handbook. March 1995.

[FD93] K. Forbus and J. DeKleer. *Building problem solvers*. The MIT Press, 1993.

[Fre78] E. Freuder. Synthesizing constraint expressions. *Communications of the Association for Computing Machinery*, 21(11) :958–966, November 1978.

[GHJV99] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Catalogue de modèles de conceptions réutilisables*. Vuibert edition, 1999.

[GMW97] D. Garlan, R.T. Monroe, and D. Wile. An architecture description interchange language. In *Proceedins of GACON '97, Toronto, Canada*, pages 169–183, Novembre 1997.

[GMW00] D. Garlan, R.T. Monroe, and D. Wile. Acme : architecture description of component-based systems. In G.T. Leavens and M. Sitaraman, editors, *Foundation of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[Gro00] IEEE Architecture Working Group. Recommended practice for architectural description of software-intensive systems. Technical Report Std 1471-2000, IEEE, 2000.

[Gué05] Y-G. Guéhéneuc. Ptidej : Promoting patterns with patterns. In *European Conference on Object Oriented Programming, workshop on Building a System with Patterns, Glasgow, Scotland*, 2005.

[HM99] A. Hussey and M. Mahemoff. Patterns for designing safety-critical interactive systems. http ://www.cs.mu.oz.au/tr_submit/test/cover_db/mu_TR_1999_25.html, 1999.

[Hoa03] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 2003.

[Hoq00] R. Hoque. *XML for real programmers*. Morgan Kaufmann, 2000.

[How60] R.A. Howard. *Dynamic programming and Markov processes*. T. MIP Press, Massachusetts, 1960.

[JB00] N. Jussien and V. Barichard. The palm system : explanation-based constraint programming. In *Techniques for Implementing Constraint programming Systems (TRICS), Singapore*, pages 118–133, September 2000.

[JM72] W.E. Jordan and G.C. Marshall. Failure modes, effects and criticality analysis. In *Annual Reliability and Maintenability Symposium, San Francisco, USA*, 1972.

[Kri63] S. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16, 1963.

[KSB+04] C. Kehren, C. Seguin, P. Bieber, C. Castel, C. Bougnol, J.-P. Heckmann, and S. Metge. Architecture patterns for safe design. In *Proceedings of Complex and Safe Systems Engineering*, 2004.

[KV98] E. Kindler and T. Vesper. Estl : a temporal approach logic for events and states. In LNCS, editor, *Proceedings of ATPN*, pages 365–383, 1998.

[Lab00] F. Laburthe. Choco : implementing a cp kernel. In *Techniques for Implementing Constraint programming Systems (TRICS), Singapore*, September 2000.

[LKA+95] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4) :336–355, April 1995.

[Lut97] R R. Lutz. Reuse of a formal model for requirements validation. In *4th NASA Langley Formal Methods Workshop, Hampton, VA, USA*, 1997.

[Lut99] R.R. Lutz. Towards safe reuse of product family specifications. In *5th Symposium on Software Reusability, Los Angeles, CA, USA*, May 1999.

[LV95] D.C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9) :717–734, September 1995.

[Mac99] K.L. MacMillan. The smv language. Technical report, Cadence Berkeley Labs, 1999.

[MDCS98] R. Manian, J.B. Dugan, D. Coppit, and K.J. Sullivan. Combining various techniques for dynamic fault tree analysis of computer systems. In *Proceedings of International Symposium on High Assurance System Engineering*, 1998.

[MK97] J.A. McDermid and T.P. Kelly. Safety case construction and reuse using patterns. In *16th International Conference on Computer Safety and Reliability, York, UK*. www.users.cs.york.ac.uk/tpk/patterns.pdf, 1997.

[MP92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-verlag edition, 1992.

[MR98] F. Maraninchi and Y. Rémond. Mode automata : about modes and states for reactive systems. *Lecture Notes in Computer Science*, 1381 :185–196, 1998.

[MT97] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the Sixth European Software Engineering Conference*, pages 60–76, 1997.

[Pag04] C. Pagetti. *Extension temps réel d'AltaRica*. PhD thesis, Ecole Centrale de Nantes, April 2004.

[Pnu81] A. Pnueli. A temporal logic of concurrent programs. *Theoritical Computer Science*, 13 :45–60, 1981.

[Poi00] G. Point. *AltaRica : contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement*. PhD thesis, Université de Bordeaux, January 2000.

[PR99] G. Point and A. Rauzy. Altarica : constraint automata as a description language. *Journal Européen des Systèmes Automatisés*, 33 :1033–1052, 1999.

[Rau02] A. Rauzy. Mode automata and their compilation into into fault trees. *Reliability Engineering and System Safety*, 78 :1–12, 2002.

[SAE96] SAE. *Certification considerations for highly-integrated or complex aircraft systems - ARP4754*. SAE Systems Integration Requirements Task Group, June 1996.

[San02] M. Sand. Modelling dependable systems with patterns. In A. Bondavalli and P. Thevenod-Fosse, editors, *4th European Dependable Computing Conference, Toulouse, France*, October 2002.

[Uni98] Carnegie Mellon University. *The ACME architectural description language*. www-2.cs.cmu.edu/acme, 1998.

[Var96] M. Vardi. *An automata-theoretic approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-verlag edition, 1996.

[Ves96] S. Vestal. *MetaH programmer's manual, version 1.09*. Honeywell Technology Center, Avril 1996.

[Vil88] A. Villemeur. *Sûreté de fonctionnement des systèmes industriels*. Eyrolles edition, 1988.

[Vin03] A. Vincent. *Conception d'un vérificateur de modèles AltaRica*. PhD thesis, Université de Bordeaux, December 2003.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program. In *Symposium on Logic in Computer Science (LICS'86)*, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[Zim95] W. Zimmer. Relationships between design patterns. *Pattern languages of program design*, pages 345–364, 1995.

Motifs formels d'architectures de systèmes pour la sûreté de fonctionnement

Résumé : Cette thèse propose des méthodes assistant la modélisation et l'évaluation qualitative de l'architecture de sûreté de fonctionnement des systèmes embarqués complexes. Ces architectures sont souvent construites à partir de motifs généraux d'architectures de systèmes correspondant à des mécanismes de sûreté récurrents comme des redondances, des détections, etc. En s'inspirant des principes des "patrons de conception" développés en génie logiciel, nous avons proposé une modélisation de ces mécanismes et des attributs permettant leur réutilisation lors des analyses de sûreté de fonctionnement. Ces analyses nécessitent de raisonner sur le comportement des systèmes en présence de pannes qui peut être modélisé à l'aide de langages formels comme AltaRica. Dans notre cas, les motifs correspondent à des abstractions d'architectures concrètes et donc requièrent une modélisation plus déclarative. Les propriétés étudiées étant en général dynamiques, nous avons choisi d'utiliser une logique temporelle pour les exprimer. Les motifs sont donc constitués d'une partie AltaRica et d'une partie propriétés. Ce type de modélisation mixte possède plusieurs intérêts, notamment lors de la conception en phase amont d'architectures de systèmes où il est possible de manipuler à la fois des parties d'un système conçues de manière détaillée et des spécifications. Elle a également pour buts de faciliter l'allocation d'exigences pour la validation d'architectures ainsi que le prototypage. Nous avons donc défini une notation mixant ces aspects opérationnels et déclaratifs.

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

Systems architectures formal safety patterns

Abstract : This thesis aimed at providing methods to assist modelling and assessing qualitatively embedded complex systems safety architectures. These architectures are often based on generic systems architectures models corresponding to safety mechanisms such as redundancies, detections, etc. While taking as a starting point the principles of the design pattern approach used in the software community, we proposed a modeling of these mechanisms and attributes allowing their re-use during safety assessments. These analyses require to reason on the behavior of systems in the presence of failures which can be modelled using formal languages like AltaRica. In our case, patterns are corresponding to concrete architectures' abstractions and so require a more declarative modelling, using properties. Those properties being generally dynamic, we chose a temporal logic to model them. Safety patterns are therefore made of an AltaRica part and a property part. We believe this kind of mixed modelling to be of great interest, especially in the preliminary system architecture design phase where it is necessary to deal with detailed parts of systems as well as specifications. It should also ease the allocation of requirements and prototyping. A notation mixing operational and declarative views has been defined.

Key words : AltaRica, architecture design/validation, formal methods, patterns, safety engineering