THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par **l'Institut Supérieur de l'Aéronautique et de l'Espace**
Spécialité : Informatique

Présentée et soutenue par **Thomas BOCHOT**
le 17 décembre 2009

Vérification par Model Checking des commandes de vol : applicabilité industrielle et analyse de contre-exemples

---

JURY

M. Ioannis Parissis, président du jury, rapporteur
Mme Béatrice Bérard, rapporteur
M. Bruno Marre
M. Pierre Virelizier
Mme Hélène Waeselynck, co-directrice de thèse
Mme Virginie Wiels, directrice de thèse

École doctorale : **Mathématiques, informatique, télécommunications de Toulouse**

Unité de recherche : **Équipe d'accueil ISAE-ONERA MOIS**

Directrice de thèse : **Mme Virginie Wiels**
Co-directrice de thèse : **Mme Hélène Waeselynck**

Remerciements

Les travaux présentés dans ce mémoire ont été effectués conjointement au sein :
- de l'unité « Ingénierie des Systèmes Critiques » (ISC) du Département de Traitement de l'Information et modélisation (DTIM) de l'Office National d'Etude et de Recherche Aéronautique (ONERA) et,
- du service recherche du département système de commande de vol de la société Airbus Operations S.A.S.

Du côté de l'ONERA, je remercie Messieurs Jacques Cazin et Bernard Lécussan, Directeurs successifs du DTIM, de m'avoir accueilli au sein de leur département.

Du côté d'Airbus, je remercie Messieurs Jean-François Polchi et Jean-Jacques Aubert de m'avoir accueilli respectivement dans leur département et service.

Ces travaux a été conduit sous la direction de Madame Virginie Wiels, Ingénieur de recherche à l'ONERA et de Madame Hélène Waeselynck, Chargée de recherche au LAAS-CNRS. De plus, j'ai été encadré par Monsieur Pierre Virelizer, ingénieur Airbus.

Merci Virginie pour ta patience et ta clairvoyance. C'est trois années, à tes côtés, mon beaucoup apportées tant sur le plan scientifique que sur le plan relationnel. Merci Hélène pour ta sincérité et ta rigueur, j'ai notamment apprécié ta présence dans les derniers instants de la rédaction. Merci Pierre pour ta disponibilité, ta rigueur et ton expertise technique. Tes conseils ont été toujours précieux et pertinents.

J'ai beaucoup appris de vous trois tant humainement que professionnellement et je ne vous remercierez jamais assez pour la qualité d'encadrement que vous avez fournit tout au long de ma thèse. Je tiens à préciser que la clarté de ce manuscrit doit beaucoup à leurs nombreuses et nécessaires corrections. Je vous remercie aussi d'avoir participé à mon Jury de thèse.

J'exprime également ma gratitude à :
- Madame Béatrice Bérard, Professeur à l'université Pierre et Marie Curie (LIP6)
- Monsieur Ioannis Parissis, Professeur au Laboratoire de Conception et d'Intégration des Systèmes (LCIS)

d'une part, pour avoir accepté de rapporter sur mes travaux et pour tout l'intérêt que vous y avez porté, d'autre part, pour avoir participé à mon Jury.

Je tiens également à remercier Monsieur Bruno Marre, Chargé de recherche au CNRS d'avoir accepté de participé à mon Jury.

Je termine maintenant en remerciant Claire Pagetti par qui tout a commencé à la sortie d'un cours dans les couloirs de l'école.

Je remercie toutes les personnes qui ont participé de prés ou de loin à mes travaux. Merci à mes amis, merci à vous qui lisez ces mots.

à ma famille, à Michèle et Éric, à Patrick et Dominique...

à la personne la plus importante qui soit pour moi, mon amoureuse je dédie ces travaux.

Table des matières

Table des figures

Introduction

Cette thèse CIFRE a été réalisée dans le cadre d'une collaboration entre un acteur industriel, Airbus, et deux acteurs académiques, l'Onera et le LAAS-CNRS.

En guise de points de départ de cette thèse, deux constats peuvent être faits sur l'état de l'art et de la pratique de l'utilisation des méthodes formelles.

Côté *académique*, les techniques de vérification formelle existent depuis de nombreuses années et sont optimisées par de nombreuses équipes de recherche. Des outils existent et peuvent maintenant être appliqués à des applications de tailles conséquentes.

Côté industriel, les langages formels sont maintenant largement utilisés pour la conception des systèmes embarqués critiques. Un des facteurs essentiels ayant motivé leur utilisation est la capacité de génération automatique de code, qualifié dans certains cas, à partir de modèles. Un autre facteur est l'existence de langages formels adaptés aux domaines d'application considérés. Un exemple typique est l'utilisation de SCADE pour le développement du système de commandes de vol par Airbus.

Les utilisations industrielles opérationnelles des techniques de vérification formelles sont cependant encore rares [Laurent 2001]. De nombreux travaux de recherche existent [Miller 2006] mais le transfert effectif de ces techniques de vérification vers les ingénieurs n'a pas encore eu lieu.

Cette thèse se propose de contribuer à rendre ce transfert effectif dans le cas des systèmes de commandes de vol Airbus. Les travaux de thèse peuvent être structurés en trois parties.

La première partie tire le bilan des études passées d'Airbus sur l'application du Model Checking au système de commande de vol. Nous analysons notamment les caractéristiques des fonctions de CDV, et leur impact sur l'applicabilité de la technologie.

La deuxième partie complète la précédente par une nouvelle étude, expérimentant le model checking sur la fonction Ground Spoiler de l'A380. Les expérimentations ont permis de consolider notre analyse du positionnement du model checking dans le processus Airbus. Un des problèmes pratiques identifiés concerne l'exploitation des contre-exemples retournés par le model checker, en phase de mise au point d'un modèle.

La troisième partie propose une solution à ce problème, basée sur l'analyse structurelle des parties d'un modèle activées par le contre-exemple. Il s'agit d'une part d'extraire l'information pertinente pour expliquer la violation de la propriété cible, et d'autre part de guider le model checker vers l'exploration de comportements différents, activant d'autres parties du modèle. Un algorithme d'analyse structurelle est défini, et implémenté dans un prototype afin d'en démontrer le concept.

Le document est composé de six chapitres. Le chapitre 1 décrit le contexte indus-

triel de la thèse : le système de commandes de vol et son processus de développement. Il présente aussi le langage SCADE et son utilisation pour la conception du système de CDV. Il introduit enfin la technique du model checking qui permet de vérifier formellement des propriétés sur un modèle.

Le chapitre 2 synthétise les expérimentations réalisées par Airbus depuis une dizaine d'années sur l'utilisation du model checking. Il décrit également une expérimentation menée pendant la thèse. Il conclut sur les leçons tirées de ces études et des axes de travail pour améliorer l'industrialisation du model checking. Un de ces axes est la génération de plusieurs contre-exemples illustrant des scénarios de violation différents. Une première étape est de chercher à caractériser la violation observée en identifiant les parties du modèle activées au cours du temps.

Le chapitre 3 propose un état de l'art sur l'analyse structurelle de modèles. Après une brève présentation du test, le chapitre présente trois types de travaux : des travaux dans le domaine du test du matériel dont l'objectif est d'exciter une faute particulière et de la propager pour la rendre observable ; des travaux sur la définition de critères de couverture structurelle pour le langage Lustre ; et enfin des travaux concernant le diagnostic de fautes dans les modèles Lustre.

Les chapitres 4 et 5 présentent l'approche proposée pour l'analyse automatisée d'un contre-exemple. L'objectif est d'extraire du contre-exemple des informations pertinentes permettant de comprendre la violation de la propriété observée. Le chapitre 4 formalise le problème en définissant la notion de *cause* d'un contre-exemple à partir de la notion de chemin actif dans la structure du modèle. La formalisation est illustrée sur un petit exemple et justifiée par rapport aux besoins identifiés plus tôt.

Le chapitre 5 présente un algorithme permettant de calculer des causes de la violation de la propriété pour un contre-exemple donné. Ce chapitre contient également la preuve de correction de l'algorithme par rapport aux notions définies dans le chapitre 4.

Le chapitre 6 décrit le prototype implémentant l'algorithme du chapitre 5 dans l'environnement Simulink et illustre l'approche sur un cas d'étude.

Chapitre 1

Le contexte industriel

Sommaire

Ce chapitre introduit le contexte dans lequel les travaux de thèse se sont déroulés. Le chapitre est décomposé en six parties. La première partie décrit brièvement le système de commande de vol (CDV). La deuxième partie présente les contraintes appliquées au processus de développement de ce système. La troisième partie détaille le processus de développement du système de CDV chez Airbus. La quatrième partie introduit les modèles comportementaux du système de CDV. La cinquième partie présente les moyens de vérification des modèles. La sixième partie présente les travaux sur le model checking dans le domaine de l'aéronautique.

1.1 Description du système de commande de vol

Le département des *commandes de vol* est en charge de la spécification du système de CDV qui englobe le pilotage automatique et manuel. Ce système a pour objectif de piloter l'avion autour de son centre de gravité et de contrôler sa trajectoire. L'avion dispose de surfaces mobiles réparties sur la voilure et l'empennage [1]. Ces surfaces, dites primaires, sont représentées sur la figure 1.1.

1. Ensemble de plans fixes et mobiles qui assure la stabilité et la gouverne en tangage (profondeur) et en lacet (direction) de l'avion

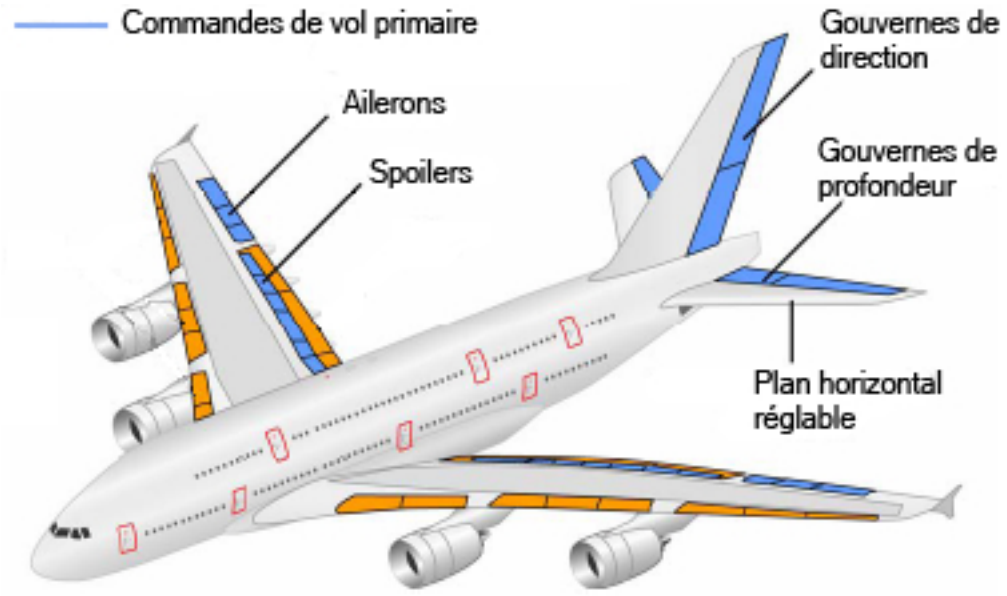


FIGURE 1.1 – Surfaces de contrôle primaires de l'A380

Les *surfaces primaires* sont composées des *ailerons*, *spoilers*, *gouvernes de profondeur*, *gouvernes de direction*, *plan horizontal réglable*. Elles sont destinées à modifier l'attitude de l'avion.

Le pilote dispose de commandes dans le cockpit qui lui permettent de commander l'attitude et la trajectoire de l'avion. Ces commandes incluent les *mini manches*[2], le *palonnier*, le levier des *aérofreins*, les boutons d'*équilibrage latéral (lacet) et en profondeur (tangage)* et le bandeau du *pilote automatique*. Les commandes et les dispositifs d'affichages d'un A380 sont illustrés en figure 1.2.

Dans le cas de l'A380, les informations provenant de ces organes de pilotage sont traitées par six calculateurs. Ces derniers calculent l'angle à appliquer aux surfaces primaires afin de répondre à la demande du pilote. Les calculateurs prennent en compte un certain nombre de paramètres, liés à la configuration[3] de l'avion et à son environnement, pour synthétiser les ordres. Les calculateurs ont aussi une fonction de protection qui limite les trajectoires de l'avion dans un domaine sécurisé. Nous allons voir comment est développé le logiciel embarqué dans ces calculateurs.

1.2 Les contraintes réglementaires du développement des CDV

Le processus de développement des CDV est soumis à des règlements. Les règles sont formulées par les autorités de certification. Pour l'Europe, l'autorité de certi-

2. ou sidesticks
3. tels que la masse et le centrage

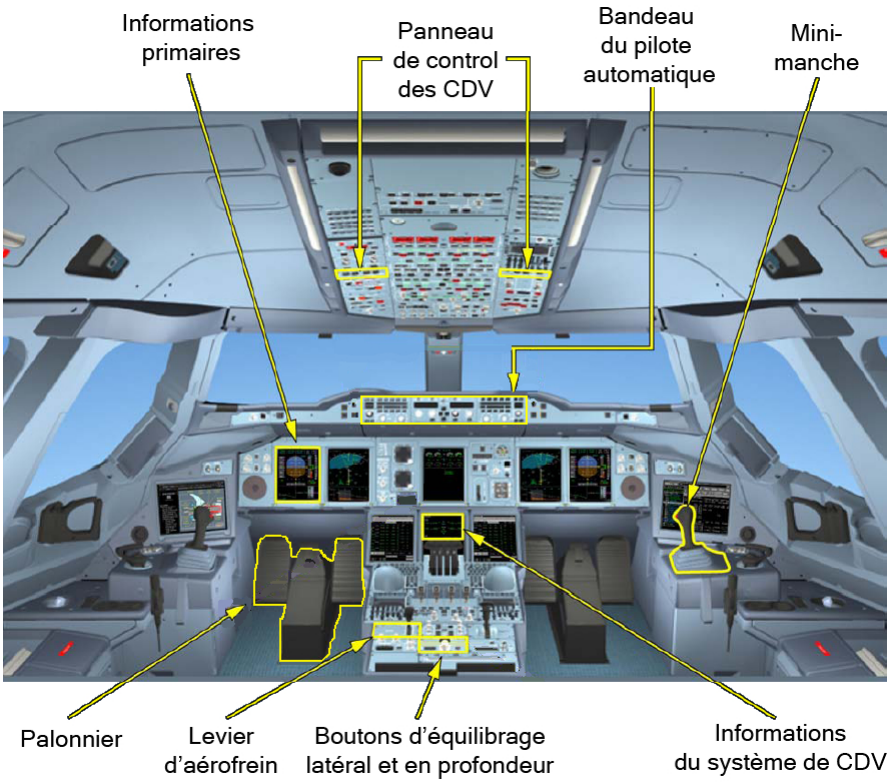


FIGURE 1.2 – Organes de pilotage primaires de l'A380

fication est l'EASA [4]. Cette agence est garante de la qualité et de la sécurité des avions construits. Cette notion de qualité étant non quantifiable sur l'avion, l'EASA s'assure de la qualité du processus de développement. Pour cela, elle établit des règles que l'avionneur doit suivre durant toute la phase de développement : JAR et CRI. Lorsque les exigences des autorités sont respectées, elle délivre un *certificat de type*. Ensuite, elle délivre pour chaque avion fabriqué, un *certificat de navigabilité* qui autorise l'avion à voler dans l'espace aérien qu'elle régit.

En particulier, les règlements de l'EASA conditionnent deux activités garantissant la sécurité des CDV : le processus de développement des systèmes, ainsi que l'analyse de sûreté de fonctionnement des systèmes.

Les ingénieurs d'Airbus se sont basés sur les recommandations qui sont définies dans le document ARP4754 [5][SAE 1996] et DO-178B/ED-12 pour définir des règles de développement en concordance avec les exigences des autorités de certification. La directive Airbus ABD200 spécifie le processus dédié à la vérification des systèmes.

La JAR décrit un processus d'évaluation de la *sûreté de fonctionnement* basé

4. European Aviation Safety Agency
5. Systems development processes

sur trois analyses principales : la FHA [6], la FMES [7] et la SSA [8].

FHA Cette analyse marque le début du processus. Elle identifie les différentes *situations redoutées* appelées FC [9] associées aux différentes fonctions réalisées par le système de CDV. Les FC sont classées selon leur gravité. La définition de la classification de gravité est donnée dans le tableau 1.1.

Gravité	Effet de la défaillance	Objectif
Catastrophique	L'avion ne peut voler de façon sûre, perte de l'avion, de l'équipage ou de passagers	$10^{-9}/hv$
Dangereux	Réduction importante des marges de sécurités ou des fonctions, augmentation importante de la charge de travail pour l'équipage, blessures sévères	$10^{-7}/hv$
Majeur	Réduction significative des marges de sécurité ou des fonctions, augmentation de la charge de travail pour l'équipage, inconfort blessures possibles	$10^{-5}/hv$
Mineur	Réduction légère des marges de sécurité ou des fonctions, augmentation de la charge de travail pour l'équipage, inconfort	$10^{-3}/hv$

TABLE 1.1 – Classification des défaillances

La FHA décline les FC pour chaque phase de vol et détermine les différentes exigences à respecter. Ces exigences se traduisent sous forme d'objectifs quantitatifs : taux de défaillance par heure de vol (/hv) (colonne de droite du tableau 1.1). Dans certains cas, cette exigence quantitative est complétée d'une exigence qualitative telle que : *Une panne simple ne doit pas conduire à une situation redoutée catastrophique.*

FMES Cette analyse permet de lister les pannes des composants du système et donne leur taux de défaillance.

SSA Cette analyse consiste à démontrer que l'architecture matérielle finale du système est en accord avec les objectifs fixés. Il s'agit de vérifier la cohérence entre les taux de défaillance à atteindre et ceux présentés par les composants matériels du système.

On peut noter que le développement du logiciel est impacté par la *sévérité* de l'équipement dans lequel il est implanté. Par exemple, la perte du système de CDV est classée *catastrophique*. Son architecture est telle qu'il est extrêmement improbable que ce système tombe en panne.

Dans ces analyses de sécurité, on fait toujours l'hypothèse que le logiciel est sans erreur. Pour couvrir cette hypothèse, le logiciel est vérifié et validé. L'intensité de

6. Functionnal Hazard Assessment
7. Failure Mode Effect Summary
8. System Safety Assessment
9. Failure Conditions

la vérification est graduée en fonction du *niveau de criticité* du logiciel. La norme ARP4754 fixe ce niveau en rapport avec la criticité de la fonction à laquelle le logiciel est rattaché. Nous verrons dans la section 3.3 que la norme DO-178B/ED-12 impose une contrainte sur la manière dont est vérifié le logiciel (voir section 3.3.1).

Le processus de développement du système de CDV est mené en suivant les règles définies par l'EASA. Nous allons présenter le processus de développement du logiciel du système de CDV.

1.3 Le processus de développement Airbus

Le cycle de développement complet de ce système est visible sur la figure 1.3. Ce cycle peut se décomposer en trois niveaux : *avion*, *système* et *équipement*.

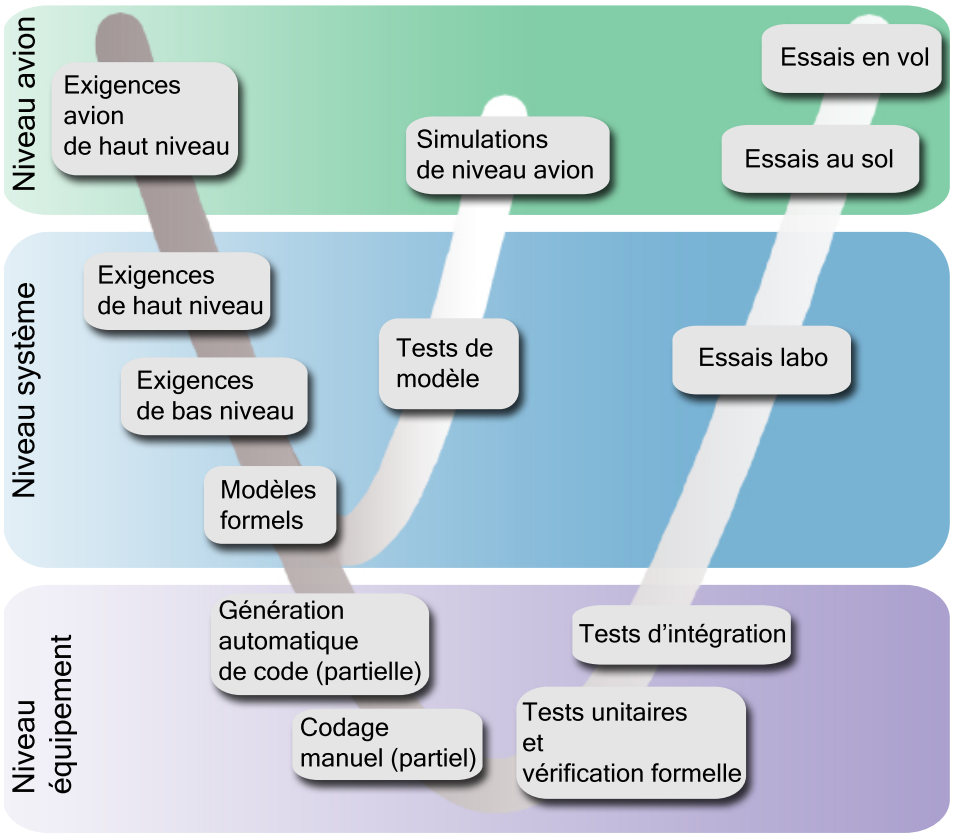


FIGURE 1.3 – Processus de développement des commandes de vol Airbus

Le niveau *avion* définit les exigences de l'avion et de tous les systèmes qui le composent. Les exigences du système de CDV sont transmises au niveau *système* qui a pour objectif de développer ce dernier. Les moyens de vérification du niveau *avion*, les *essais au sol* et *en vol*, seront abordés dans la section 1.3.3. Une vue plus détaillée de ce niveau n'est pas nécessaire pour positionner l'intérêt des travaux. En revanche, nous détaillerons plus les niveaux *système* et *équipement* dans les sous

sections ci-dessous.

1.3.1 Le niveau système

Le niveau système a pour objectif de spécifier le système de CDV selon les exigences du niveau *avion*. Les étapes suivantes sont réalisées conformément aux procédures agrées par les autorités de certification :

- Les **exigences de haut niveau** (SRD [10]) sont produites à partir des exigences du niveau avion.
- Les **exigences de bas niveau** (DTS [11]) sont produites à partir des SRD.
- Les **modèles formels** (FS [12]) du comportement du système sont produits à partir des DTS et SRD.

Les *modèles formels* sont ensuite transmis à un équipementier (niveau *équipement*) qui est responsable de la fabrication et des tests élémentaires du logiciel embarqué. Afin de réduire les coûts liés à la correction des erreurs, des vérifications sont donc réalisées au niveau du *modèle formel* :

Tests de modèle Ces tests ont pour objectif de valider l'aspect fonctionnel du modèle qui sera utilisé pour générer le code embarqué. Ce type de test est détaillé dans la section 1.5.1. Le CRI-F22 [13] est un document, complémentaire à l'ARP4754, qui précise la démarche de vérification et validation à suivre lorsqu'on utilise des modèles formels.

Les tests réalisés sur le calculateur (Essais labo, Essais au sol et en vol) seront décrits dans la section 1.3.3.

1.3.2 Le niveau équipement

Le niveau *équipement* a pour objectif de développer le calculateur spécifié au niveau *système*. Les étapes suivantes sont réalisées conformément aux attentes des autorités de certification :

- Les **exigences de haut niveau** (HLR) sont produites à partir des exigences du niveau système.
- Les **exigences de bas niveau** (LLR), et l'*architecture logicielle* sont ensuite produites à partir des HLR.
- Le **code source** est produit à partir des LLR et de l'architecture logicielle.
- Le code source est compilé pour produire le **code objet** qui est implanté dans la cible matérielle : le calculateur de CDV.

Dans le cas de l'A380, le *modèle formel* permet de générer plus de 90% du code source. Cependant, la partie restante (<10%) du programme source doit être réalisée manuellement. Le code source a donc deux origines :

10. System Requirement Documents
11. Detailed Technical Specifications
12. Formalized Specification
13. Certification Review Item

1. Une grande partie (>90%) provient de la génération automatique depuis le *modèle formel*. La génération est certifiée et garantit que le code source est conforme au modèle d'un point de vue fonctionnel.
2. La partie restante (<10%) est écrite manuellement selon les LLR du *niveau équipement*. Cette partie doit être vérifiée afin de garantir que le code, d'une part, satisfait les exigences d'un point de vue fonctionnel, et d'autre part, ne contient pas d'erreur d'exécution.

Les activités de vérification suivantes sont appliquées au code source écrit manuellement :

Test et preuve unitaire Cette étape a pour objectif de démontrer que le programme source produit manuellement satisfait les LLR. Dans 95% des cas, il est possible de démontrer formellement que le code satisfait les exigences [Duprat 2006, Randimbivololona 1999]. Les quelques 5% restants sont traités par du test. Il est important de préciser que le code est écrit en vu d'être vérifié, ce qui explique cette proportion. Les démonstrations utilisent des approches déductives basées sur la logique de Hoare [Hoare 1969] et le calcul de la plus faible précondition [Dijkstra 1976]. L'outil utilisé pour réaliser ces démonstrations est *CAVEAT* [webpage Caveat 2009, Baudin 2002]. Cet outil a été réalisé par le CEA [14]. Aujourd'hui, un nouvel outil est en train de voir le jour : *FramaC* [webpage frama c 2009].

Nous venons de voir que les fonctions produites manuellement ont un comportement correct de manière individuelle. Ce type de vérification n'a pas lieu d'être pour le code généré automatiquement. Le processus de génération automatique de code est certifié et garantit que le code source généré est sémantiquement équivalent au modèle qui a déjà été testé durant les tests de modèle. Cependant, il est nécessaire de vérifier le code source dans son ensemble. S'ajoute à cela la vérification d'exigences de précision des calculs.

Tests d'intégration Cette étape a pour objectif de vérifier que les fonctions, dans leur globalité, se comportent correctement. Des études sont en cours pour utiliser des méthodes formelles lors de cette étape.

Absence d'erreurs d'exécution ou RTE [15], les vérifications précédentes ne fonctionnent qu'en leur absence. Cette activité a pour objectif de détecter des erreurs telles que les *débordements de tableau*, les *divisions par zéro* ou encore le *débordement des calculs à virgule flottante*. Des techniques d'interprétation abstraite sont utilisées pour détecter ce type d'erreur et en démontrer l'absence. L'outil qui implémente ces techniques est Astrée [Cousot 2005], [Cousot 2007].

Précision des calculs flottants L'assurance de l'absence de RTE n'est pas suffisante. Il est aussi nécessaire de s'assurer de la précision des calculs numériques.

14. Commissariat à l'énergie atomique
15. Run Time Error

L'outil *Fluctuat* [Goubault 2001], réalisé par le CEA, permet de faire cette vérification de manière automatique.

Une fois ces vérifications réalisées, le code source est compilé. Le code objet résultant va être analysé afin de vérifier les temps de calculs.

Estimation du WCET [16] Le WCET est le pire temps d'exécution d'une tâche ou d'un programme. Cette activité a pour objectif d'estimer le temps d'exécution du logiciel. Aujourd'hui, l'outil *Ait* [Ferdinand 2001] fournit une estimation légèrement supérieure au WCET réel. Les résultats sont sûrs si les hypothèses sont respectées. Cette mesure du pire temps d'exécution est indispensable pour vérifier l'hypothèse de synchronisme faite au niveau *système* (voir 1.4.1).

Ces utilisations des méthodes formelles [Duprat 2006] ont été couronnées de succès et des extensions sont en cours d'étude.

1.3.3 Les moyens d'essais finaux

L'activité de vérification des calculateurs est la suivante :

Essais labo Ce sont les premiers tests sur l'équipement réel, ciblant le système de CDV uniquement ou son intégration avec plusieurs autres systèmes. On distingue plusieurs phases de test :

1. tests sur **banc partiel** [17] permettant la manipulation séparée des équipements (calculateurs, servo-contrôleurs, ...) dans un environnement simulé.
2. tests sur **banc d'intégration** permettant la manipulation des équipements connectés entre eux. Le banc est représentatif des contraintes opérationnelles d'un point de vue géométrique, câblage, source électrique et hydraulique.
3. tests sur **simulateur** [18] permettant de valider les lois de pilotage et les reconfigurations en cas de défaillance dans un environnement réel d'un point de vue système. Le simulateur est équipé d'un cockpit similaire à celui de l'avion et d'ordinateurs permettant la simulation de la mécanique du vol. De plus, le testeur dispose d'une réalité virtuelle qui lui fournit un retour visuel sur l'environnement simulé.

Cette activité est la dernière étape avant l'intégration des calculateurs dans l'avion réel. Une fois le calculateur intégré, les activités de vérification suivantes ont lieu.

Essais au sol Ce sont les premiers tests sur l'avion réel équipé d'instruments de mesures dédiés aux essais. Durant cette étape, tous les équipements composant l'avion sont testés au sol.

16. Worst Case Execution Time
17. que l'on appel aussi *avion-1* ou *Iron Bird*
18. que l'on appel aussi *avion0* ou aussi *Iron Bird*

Essais en vol C'est la dernière phase de test avant l'entrée en service de l'avion. Dans le cycle de développement de l'avion, cette phase dure généralement un an alors que les phases précédentes durent plusieurs années. Les avions d'essai en vol sont équipés d'instruments de mesure dédiés. Les données sont enregistrées à bord et une partie est envoyée à une station sol afin d'être analysée en temps réel.

Après cette vue d'ensemble du processus de développement, nous allons nous concentrer sur la modélisation formelle réalisée au *niveau système*. Nous considérerons le formalisme SCADE et l'outil de vérification : SCADE Design Verifier.

1.4 Description du modèle formel

Les modèles formels du système de CDV sont réalisés avec l'environnement SCADE[1] [19]. Cet environnement a été défini pour assister le développement des systèmes réactifs critiques. il est composé de plusieurs fonctionnalités :

- Édition graphique [Editor 2009].
- Simulation [Simulator 2009]
- Génération de code C ou ada.
- model-checking [Verifier 2009]

Dans la suite, nous nous intéresserons uniquement au langage et à la fonctionnalité de model-checking. Nous allons dans un premier temps décrire le langage. Nous aborderons la fonctionnalité de model-checking dans la section 1.5.2.

1.4.1 Le langage SCADE

Le formalisme SCADE est une notation graphique formelle basée sur le langage Lustre [Halbwachs 1991, Halbwachs 2002], ce qui en fait un langage synchrone à flot de données. Les langages synchrones se basent sur l'hypothèse de synchronisme. Cette hypothèse stipule que le système réagit à un événement instantanément. Cela sous-entend que le temps de calcul des algorithmes est nul. En pratique, il faut que le système réagisse à un événement externe avant l'acquisition de tout nouvel événement. Cette hypothèse est réaliste dès lors que l'on démontre que les algorithmes respectent leurs échéances. Ce qui est fait a posteriori, par les analyses de WCET, au niveau *équipement* (cf. section 1.3.2). En SCADE, une expression X représente un flot. Un flot est une séquence infinie $(x_0, x_1, \ldots, x_n, \ldots)$ de valeurs. Les systèmes que l'on considère ont un comportement cyclique et x_n est la valeur de x au n^{ime} cycle d'exécution. Un système est défini par un ensemble de planches SCADE possédant des entrées, des sorties et des variables locales. Une planche contient un ensemble d'équations caractérisant les sorties à partir des entrées et des variables internes. Une équation est un invariant temporel, par exemple $x = y + z$ qui définit

19. Safety Critical Application Development Environment commercialisé par Esterel Technologies

le flot $(x_0, x_1, \dots) = (y_0 + z_0, y_1 + z_1, \dots)$. La figure 1.4 montre comment cette équation est représentée graphiquement dans le langage SCADE.

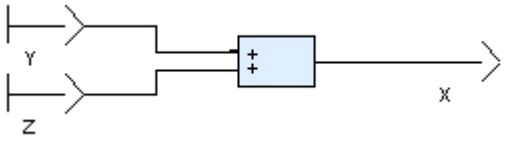


FIGURE 1.4 – Représentation SCADE de l'addition

Les équations sont écrites à partir des variables, de constantes, d'opérateurs arithmétiques, booléens, conditionnels et temporels. Par exemple, la figure 1.5 et 1.6 donnent respectivement la représentation graphique de l'opérateur ET et de l'opérateur conditionnel ITE [20].

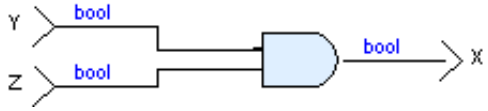


FIGURE 1.5 – Représentation SCADE de l'opérateur ET logique

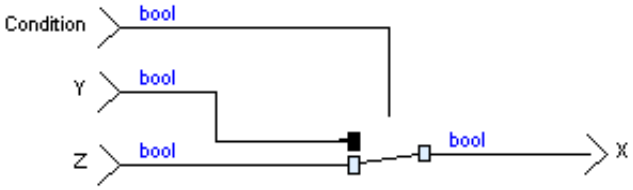


FIGURE 1.6 – Représentation SCADE de l'opérateur conditionnel

Trois opérateurs temporels permettent de considérer le temps dans les modèles. Il y a l'opérateur PRE, l'opérateur -> et l'opérateur *followed-by* (FBY) dont la sémantique peut se décrire à l'aide des deux premiers.

L'opérateur PRE permet de faire référence à l'instant passé d'un flot. Si le flot E définit $(e_1, e_2, e_3, \dots)$ alors : le flot PRE(E) définit le flot : $(nil, e_1, e_2, \dots)$ où *nil* traduit l'absence de valeur à l'instant 1.

Pour régler le problème du *nil*, l'opérateur PRE s'accompagne généralement de l'opérateur d'initialisation -> qui permet de définir la valeur du flot PRE(E) à l'instant 1. Par exemple, on l'utilise de la manière suivante : FAUX -> PRE(E) pour définir le flot (FAUX, e_1, e_2, ...).

L'opérateur FBY(E, n, F) est utilisé dans les modèles de CDV (pas le PRE, ni de ->). Cet opérateur a trois paramètres d'entrée :

1. le flot retardé : E,

20. If Then Else

2. le nombre de retards de la sortie : n,
3. le flot d'initialisation : F.

La sortie de FBY(E, n, F) a la valeur de F aux n premiers cycles puis prend la valeur de E retardée de n cycles. n est un paramètre constant. Par exemple, si E et F sont deux flots de même type, représentant respectivement les séquences (e_1, e_2, e_3, ...) et (f_1, f_2, f_3, ...), alors FBY$(E, 1, F)$ représente la séquence (f_1, e_1, e_2, ...). La représentation SCADE de cet opérateur est visible en figure 1.7.

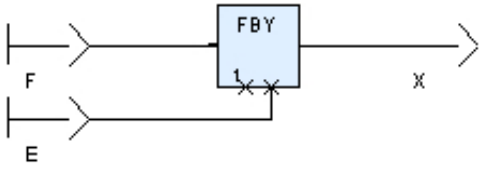


FIGURE 1.7 – Représentation SCADE de l'opérateur FBY

Remarque : FBY$(E, 1, F)$ est équivalent à l'expression Lustre : $F \rightarrow$PRE(E). Dans la suite nous considérerons uniquement des retards de 1 cycle. Le tableau 1.2 récapitule la sémantiques des opérateur PRE, -> et FBY.

Cycle	1	2	3
E	e_1	e_2	e_3
F	f_1	f_2	f_3
$PRE(E)$	*nil*	e_1	e_2
F -> E	f_1	e_2	e_3
F -> PRE(E)	f_1	e_1	e_2
FBY$(E, 1, F)$	f_1	e_1	e_2
FBY$(E, 2, F)$	f_1	f_2	e_1

TABLE 1.2 – Sémantique des opérateurs temporels

1.4.2 Spécification du système de commande de vol

Le système de CDV est formellement défini par des modèles SCADE appelés planches SCADE. Cependant, les deux éléments suivants ne sont pas spécifiés dans le formalisme SCADE.

L'ordonnancement est l'ordre dans lequel le calculateur exécute le code généré à partir des planches SCADE. L'ordonnancement des planches SCADE n'est pas spécifié dans le formalisme SCADE. Il est décrit dans un document textuel.

Les symboles Une bibliothèque d'opérateurs dédiés à la définition du système de CDV a été développée par Airbus. Ces opérateurs sont appelés *symboles* et sont très souvent utilisés dans les spécifications de CDV. Pour des raisons d'optimisation, les *symboles* sont spécifiés textuellement et directement implémentés dans un langage bas niveau tel que le code C ou l'assembleur. Les

symboles sont validés selon le niveau de DAL A, le niveau le plus sévère du standard de certification, qui sera introduit dans la section 3.3.1.

Nous verrons au chapitre 2.1 les problèmes que ces deux points posent pour l'utilisation d'un outil de model checking.

1.5 Vérification du modèle formel

La vérification de modèle a été introduite dans la section 1.3.1. Cette activité se base aujourd'hui sur des techniques de test.

1.5.1 L'approche classique : tests de modèle

Le test consiste à exécuter un modèle avec des entrées valuées et à vérifier la conformité des sorties par rapport au comportement attendu. Sa mise en œuvre nécessite de résoudre deux problèmes :

- la sélection des valeurs des entrées,
- l'*oracle* ou comment décider de l'exactitude des résultats observés, fournis par le programme en réponse aux entrées de test.

Afin de tester le modèle, une partie du code C est automatiquement générée à partir des planches SCADE. Ce code est ensuite compilé puis implanté sur une plate-forme [21] de simulation intégrant un ensemble de modèles tels que la mécanique du vol, les actionneurs, le réseau de bus numérique. La plate-forme comprend des organes de pilotage et permet la visualisation des paramètres de vol principaux. Une campagne de test peut ainsi être réalisée au plus tôt afin de valider une sous-partie fonctionnelle des exigences de haut et bas niveau (SRD et DTS). A partir de ces documents, les testeurs définissent des scénarios de test. Les scénarios sont exécutés sur le simulateur et les testeurs décident ensuite si le test est correct ou non. Il est important de noter que la cible sur laquelle est implantée le logiciel n'est pas représentative des calculateurs embarqués, d'où la présence des *essais labo* ultérieurs. Ces tests de modèle permettent de vérifier et de valider le comportement nominal du système mais aussi les éléments suivants :

- les limites du système,
- le fonctionnement du système en cas de panne,
- la synchronisation des calculateurs,
- les effets transitoires,
- l'asservissement des servo-controlleurs qui sont simulés,
- le pilote automatique.

Les *tests de non-régression* permettent de s'assurer qu'une évolution du modèle ne vient pas perturber les fonctions qui n'ont pas subi d'évolution. Des jeux de tests sont réutilisés automatiquement et un oracle humain est requis lorsque les résultats ne sont pas similaires à ceux avant modification.

21. que l'on appelle « OCASIME »

Pour de plus amples informations sur les techniques de test, nous ferons référence à [Ammann 2008, Beizer 1990].

1.5.2 Une approche formelle : le model checking

Le test n'est pas l'unique approche pour vérifier qu'un système satisfait ses exigences. Nous avons vu qu'au niveau du code, des techniques formelles sont utilisées. Nous allons nous intéresser à une technique formelle permettant de vérifier des propriétés sur les modèles : le *model checking*. Pour cela, il est nécessaire de posséder un modèle formel représentatif du système que l'on souhaite vérifier. Dans le processus de développement du système de CDV, le model checking peut être employé pour vérifier des propriétés sur les modèles SCADE.

Le model checking [Bérard 1999] est classé parmi les techniques de *vérification statique*. Le terme *statique* signifie que, contrairement au test et à la simulation, le modèle n'est pas exécuté. Les techniques de model checking sont souvent présentées comme *automatiques* dans le sens où l'utilisateur n'a pas à interagir durant l'analyse. Cependant, nous verrons au chapitre 2.1 que l'analyse nécessite en fait une participation relativement importante de la part de l'utilisateur. L'objectif de cette section n'est pas de décrire en détail les algorithmes de model checking mais de donner au lecteur des notions de base. On utilisera le terme *model checker* pour désigner l'outil utilisant une technique de model checking.

1.5.2.1 Notion de propriété

Considérons un modèle M, dont le comportement peut être représenté par un automate, un ensemble d'hypothèses H et une propriété ψ. Un model checker permet de décider si M satisfait ψ sous H. Si tel est le cas, on note :

$$M, H \models \psi$$

Nous allons nous intéresser à H et ψ pour le moment. Une propriété, tout comme une hypothèse, peut s'exprimer au travers d'une expression logique. Il existe plusieurs types de logique avec un pouvoir d'expression propre à chacune. Nous n'allons pas présenter ces différentes logiques, cependant nous pouvons citer deux logiques très connues : PLTL [22] [Pnueli 1979] et CTL [23] [Ben-Ari 1981]. La première est une extension de la logique propositionnelle qui utilise des combinateurs temporels permettant d'exprimer une notion du temps linéaire. Elle permet d'exprimer des événements sur une exécution unique du système (il n'y a qu'un seul futur). La seconde possède des quantificateurs de chemins et permet d'envisager plusieurs possibilités (il y a plusieurs futurs) et de les exprimer au travers d'une seule formule. Ainsi, on considère plusieurs exécutions du système avec une notion de temps arborescente.

On distingue quatre types de propriétés ou d'hypothèses que l'on exprime sur l'automate représentant le comportement du modèle :

22. Propositional Linear Temporal Logic ou Logique Temporelle Linéaire
23. Computational Tree Logic ou Logique Temporelle Arborescente

Atteignabilité énonce qu'une situation peut se produire. Depuis les états initiaux de l'automate, il existe au moins un état atteignable dans lequel l'expression de la propriété est vraie.

Sûreté énonce qu'une situation ne peut jamais se produire. Depuis les états initiaux, il n'existe aucun état de l'automate dans lequel la propriété est vraie.

Vivacité énonce qu'une situation finira par avoir lieu.

Équité énonce qu'une situation se produira (ou ne se produira pas) un nombre infini de fois.

Nous invitons le lecteur à lire [Audureau 1990] pour un approfondissement de la logique temporelle.

1.5.2.2 Principe de base du model checking

L'ensemble des comportements possibles de M peut se représenter sous la forme d'un automate. Le model checker parcourt les états de l'automate à la recherche d'un état ne satisfaisant pas ψ. S'il n'existe pas de tel état, on dira que ψ est valide sous H. Si ψ exprime une exigence, cela signifie que M satisfait cette exigence sous H. Dans le cas contraire, le model checker renvoie un *contre-exemple* de la propriété. Un *contre-exemple* est une exécution amenant le modèle dans un état violant ψ tout en respectant H.

La technique de base est l'*énumération explicite des états* atteignables. Ce type d'algorithme procède en deux temps :

1. construction de l'automate représentant les comportements du modèle,
2. parcours des états de l'automate à la recherche d'un état satisfaisant $\neg\psi$. Dans ce cas, l'algorithme se termine et renvoie un contre-exemple. S'il n'en existe pas, cela signifie que le modèle satisfait la propriété [24].

Par exemple, **Lesar** [Halbwachs 1992, Ratel 1992], un model checker pour les modèles **Lustre**, possède un algorithme de ce type.

Généralement un contre-exemple se présente sous la forme d'une séquence de valuations des variables d'entrées permettant d'atteindre l'état violant la propriété. Un utilisateur peut alors analyser ou utiliser un outil de simulation afin de comprendre les raisons de cette violation.

Lorsque l'on considère des systèmes de taille importante tel que le système de CDV, l'automate peut être très gros et non représentable totalement. Ce problème est connu sous le nom d'**explosion combinatoire**.

1.5.2.3 Le problème de l'explosion combinatoire

Plusieurs solutions ont été proposées afin de contourner les problèmes de performances des model checkers. Nous décrivons brièvement ces principales solutions.

24. Dans ce cas il est important de noter que le parcours des états aura été exhaustif.

La réduction d'ordre partiel Ce type de technique [Alur 1997] a pour objectif d'éviter de parcourir tous les chemins d'exécution lorsque les entrelacements d'événements sont équivalents. Ces techniques sont particulièrement efficaces pour la vérification des systèmes concurrents pour lesquels la commutativité des événements est fréquente.

Réduction par symétrie Ce type de technique [Clarke 1998] est efficace lorsque les modèles possèdent des parties symétriques qui induisent des relation d'équivalence entre certains états de l'automate. Lors de l'analyse de l'automate, il est alors possible de ne pas explorer un état s' si l'état équivalent s a déjà été exploré.

Les techniques d'abstraction Ces techniques ne considèrent pas directement le modèle mais une version abstraite de ce dernier. Il existe plusieurs types d'abstraction. Dans tous les cas, l'abstraction doit préserver le type de propriété que l'on considère. Nous ne parlerons que des abstractions qui représentent une sur-approximation des comportements. Dans la mesure où le modèle abstrait M' autorise plus de comportements que le modèle initial M, alors si M' vérifie la propriété de sûreté ψ, M la vérifiera aussi. En revanche, si M' ne vérifie pas ψ, il n'est pas possible de conclure sur M. La difficulté de ce type de technique est de trouver le bon niveau d'abstraction.

L'approche **CEGAR**[25] [Clarke 2000] propose une manière automatique de faire cela. La méthode consiste à (1) produire une abstraction M' à partir de M. (2) Vérifier que M' satisfait ψ. Si $M' \models \psi$ alors cela signifie que $M \models \psi$[26]. En revanche, s'il existe un contre-exemple de la propriété dans M', deux cas de figure peuvent se produire. (a) Soit le contre-exemple de M' est aussi un contre-exemple de M, dans quel cas il sera fourni à l'utilisateur. (b) Soit le contre-exemple n'existe pas dans M et il est donc le résultat de l'abstraction. Dans ce cas, l'abstraction est raffinée de telle manière à écarter le contre-exemple. Puis l'on recommence l'étape (2).

A partir d'un modèle concret et d'une relation d'abstraction, il est possible de créer un modèle abstrait en utilisant des techniques d'**interprétation abstraite** [Cousot 1977, Cousot 1979, Cousot 2007]. Ces techniques formalisent l'idée qu'une démonstration formelle peut être réalisée avec un certain degré d'abstraction permettant l'oubli d'informations inutiles.

A partir d'un modèle et d'un critère de découpage, il est également possible de créer un modèle plus petit, sémantiquement équivalent au modèle initial vis à vis du critère de découpage. Cette technique est connue sous le nom de **slicing** [Tip 1995]. Si le critère de découpage est lié à la propriété que l'on cherche à vérifier, on parle alors de *Property-based Slicing*.

La **réduction du cône d'influence** [Berezin 1998] a pour objectif de réduire la taille de l'automate en n'utilisant que les variables du modèle impliquées

25. Counterexample Guided Abstraction Refinement
26. Si l'abstraction conserve la propriété considérée

dans la propriété. La réduction est obtenue en éliminant les variables qui n'ont pas d'influence sur la propriété.

Une autre forme d'abstraction est appelée **Predicate abstraction**, où des prédicats de A sont convertis en variables booléennes dans A'. Par exemple, une comparaison de seuil : $Altitude > PreSelectAlt + AltCapBias$ où les variables peuvent varier entre 0 et 50000, peut être remplacée par une variable booléenne que l'on appellerait par exemple $Altitude\_Gt\_PreSelectAlt\_Plus\_Alt\_CapBias$.

Systèmes paramétrés L'idée ici est de démontrer la propriété par model checking en fixant la valeur de certaines variables dans un premier temps. Puis, dans un second temps, de démontrer par induction que la propriété est valide pour n'importe quelle valeur des variables.

La composition Lorsque que la taille du modèle est trop importante et qu'une vérification globale n'est pas envisageable, mais que le modèle peut se décomposer en sous-modèles indépendants, il est possible de procéder suivant une approche par *composition* [Berezin 1998]. Le principe est de démontrer des propriétés localement sur des parties du modèle afin d'en déduire la propriété sur le modèle global. Le problème est de déterminer les propriétés locales aux composants, permettant d'inférer la propriété globale. Les propriétés locales vont dépendre de la manière dont sont décomposées les parties du modèles. La décomposition du modèle est plus ou moins facile selon la structure. Il est donc important, lorsque l'on considère une telle approche, d'anticiper les vérifications dès la réalisation du modèle. L'approche **Assume-guarantee** [Grumberg 1991] est une technique manuelle qui consiste à vérifier des parties du modèles séparément. Le bon comportement des parties du modèles dépendent des parties voisines. Lors de la vérification d'une partie, il est nécessaire de spécifier des hypothèses d'environnement que les autres parties devront garantir.

Le model checking symbolique Une solution pour contourner le problème de l'explosion combinatoire est de représenter l'automate de manière plus efficace en mémoire. On utilise le terme de représentation symbolique. Les algorithmes de model checking capables de manipuler ce type de représentation sont classés dans la catégorie des techniques de model checking symbolique. Les BDD [27] [Bryant 1986, Bryant 1992] sont un exemple de représentation symbolique des structures de données. Les algorithmes de model checking symbolique ont démontré leur capacité à traiter des problèmes de tailles importantes. L'efficacité de cette approche dépend de l'ordre employé pour représenter les variables. Pour les très gros systèmes, la représentation sous forme de BDD ne suffit pas et le temps de calcul de l'ordre de représentation des variables explose.

Résolution SAT et SMT Plutôt que de considérer toutes les traces d'exécution d'un système, il est possible de se limiter à des traces finies, de longueur

27. Binary Decision Diagrams

bornée k. L'idée est de considérer uniquement des contre-exemples d'une certaine longueur k et de générer une formule de logique propositionnelle qui est satisfiable uniquement si un tel contre-exemple existe. Par exemple, l'algorithme de Stalmark [Sheeran 1998] peut permettre de démontrer la satisfiabilité d'une telle formule. Les algorithmes *bounded model checking* [Biere 1999] implémentent souvent un solveur SAT. Ce nom est du à la longueur bornée des contre-exemples qu'ils recherchent. Si, durant la recherche, la profondeur k est incrémentée progressivement, l'algorithme trouvera alors le contre-exemple le plus court. Ce type d'algorithme peut aussi démontrer des propriétés par k-induction (voir annexe A). Ces algorithmes ont été utilisés avec succès dans différents domaines tels que la vérification de composants, la logique modale, la vérification d'un système de contrôle de voies ferrées. A l'inverse des approches précédentes, cette méthode ne nécessite pas la construction d'un système de transition.

Une évolution récente est l'ajout, en plus de variables booléennes, de variables entières ou réelles. Les formules atomiques ne sont alors plus seulement des variables booléennes, mais des prédicats atomiques sur ces variables entières ou réelles. On parle alors de SMT [28] (par exemple, on pourra considérer comme prédicats atomiques les égalités et inégalités linéaires). Une approche consiste à remplacer les prédicats atomiques par des variables booléennes supplémentaires, et à résoudre le système via SAT. S'il n'y a pas de valuation vérifiant la formule booléenne, la formule originale n'était pas non plus satisfiable. S'il existe une valuation, il faut vérifier que celle-ci soit cohérente par rapport à la théorie. Par exemple, si on a remplacé $x < 5$ par un booléen b_1 et $x > 6$ par un booléen b_2, la valuation $b_1 = b_2 =$VRAI est incohérente par rapport à la théorie des inégalités linéaires. En pratique, il faudra donc savoir décider effectivement la satisfiabilité d'une conjonction de prédicats atomiques.

1.5.2.4 Présentation des model checkers les plus connus

Il existe plusieurs model checkers avec pour la plupart un langage d'entrée dédié permettant de modéliser un certain type de système. Le choix du model checker dépend donc de deux paramètres qui sont (a) le type de système que l'on souhaite modéliser et (b) le type d'exigence que l'on souhaite vérifier. Une démarche d'utilisation du model-checking est la suivante :

1. Quel type de propriété veut-on vérifier sur le système ?
2. Quel model checker peut traiter cette propriété ?
3. Exprimer le système et la propriété dans les formalismes de l'outil.

Nous présentons dans cette section les model checkers les plus connus. Pour plus de détails sur ces aspects, nous renvoyons le lecteur vers [Bérard 1999].

SPIN [Holzmann 1997, Ruys 2004] est un model checker développé par Bell Labs. Ce model checker est spécialisé dans la vérification de systèmes concurrents

28. satisfiabilité modulo une théorie

ou asynchrones. Le langage du modèle est PROMELA [29]. Les propriétés que l'on peut vérifier sont non fonctionnelles : absence d'état bloquant [30], ou fonctionnelles : *propriété de sûreté* ou de *vivacité*.

Kronos [Bozga 1998] est développé par le laboratoire Verimag. Ce model checker symbolique est dédié à la vérification de systèmes temps réel modélisés sous la forme d'automates temporisés [Alur 1994]. Il peut vérifier des *propriétés fonctionnelles de vivacité* ou des propriétés non fonctionnelles tel que l'absence de *zeno* ou les réponses à requêtes en temps fini. Kronos a été étendu et combiné avec d'autres outils de vérification dans l'environnement TAXYS [Bertin 2001].

SMV [SMV 2009] est un model checker symbolique développé par CMU [31]. Il comporte un langage de description d'automates dont le nombre d'états est fini. Le model checker de SMV permet de démontrer des propriétés écrites sous la forme de formules logiques LTL. L'utilisation des BDD dans cet outil a permis de vérifier des propriétés sur des modèles de taille importante : [Clarke 1994] présente une étude de cas d'un circuit logique dont l'automate possède 10^{1300} états.

NuSMV [Cimatti 1999] est une extension de SMV. Il permet la représentation d'automates finis synchrones et asynchrones. Les propriétés peuvent être exprimées au travers de formules LTL ou CTL.

Uppaal [Larsen 1997] est un outil de simulation et de vérification d'automates temporisés développé par Uppsala et Ȧlborg. Le model checker est capable de vérifier des *propriétés de sûreté* et d'*atteignabilité*.

Hytech [Alur 1996, Henzinger 1997] est un outil de vérification dédié à la modélisation et vérification des systèmes embarqués. Il est développé par l'université de Berkeley. Cet outil possède un model checker symbolique capable de calculer les paramètres pour lesquels le modèle satisfait des propriétés temporelles. Les modèles sont des automates hybrides [32].

SAL [de Moura 2004, Bensalem 2000] est un environnement de spécification et d'analyse de systèmes concurrents spécifiés sous forme d'automates. Son langage est une extension de celui proposé dans PVS [Owre 1997]. L'environnement SAL propose une série de techniques (abstraction, de génération d'invariants, de slicing) et d'outils (démonstrateurs de théorèmes et des model checkers) permettant d'analyser les automates. Quatre model checkers sont disponibles :

- un model checker symbolique (SMC),
- un model checker borné (BMC),
- un *witness* model checker (WMC),
- un *infini borné* model checker (infBMC).

29. Process Meta Language
30. ou absence de deadlock
31. Carnegie Mellon University
32. Un système est dit hybride s'il contient des composants discrets et continus.

SMC et WMC représentent l'automate de manière symbolique en utilisant les BDD. BMC utilise un solver SAT et permet d'analyser des systèmes définis avec des types finis. infBMC utilise le solveur SMT Yices [Dutertre 2006] qui permet aussi bien d'analyser des systèmes définis avec des types finis qu'avec des types infinis [33]. BMC et infBMC peuvent chercher des contre-exemples de longueur finie k. De plus, il peuvent démontrer des propriétés par k-induction.

Prover Plug-In est commercialisé par Prover Technology. Ce model checker est intégré aux environnements de travail de SCADE Suite et de SIMULINK. La transformation des modèles (SCADE ou SIMULINK) vers le modèle d'entrée du model checker est automatique. Prover Plug-In propose deux algorithmes :
- *Bounded model checking* [Clarke 2001] qui est adapté à la recherche d'erreurs,
- *Induction over time* [Sheeran 2000] qui est adapté à la démonstration de propriété.

Une description plus détaillée du model checker Prover Plug-In est disponible en annexe A.

Les spécifications du système de CDV sont réalisées en SCADE. Cependant, afin de procéder à un exercice de model checking, il est nécessaire :
- d'utiliser les modèles des symboles,
- de modéliser les liaisons entre les planches SCADE,
- de modéliser le séquencement des planches SCADE.

L'absence de passerelle SCADE vers d'autre model checker nous impose d'utiliser l'outil *Scade Design Verifier* utilisant l'outil Prover Plug-In.

Dans la suite du document nous considérerons que nos modèles sont des modèles SCADE. Les hypothèses et les propriétés seront exprimées en SCADE/Lustre. La logique propre au langage Lustre appelé SL dans [Halbwachs 1993a] permet d'exprimer des *propriétés de sûreté* et des *hypothèses de sûreté*.

1.5.2.5 Observateur synchrone

L'utilisation du model checker pour SCADE nécessite la réalisation d'un *observateur synchrone* [Halbwachs 1993b]. De manière informelle, un observateur « observe » le comportement du modèle et décide s'il respecte les propriétés. Techniquement, un observateur est un modèle connecté de façon synchrone au modèle en cours de vérification. La figure 1.8 montre une architecture typique de vérification. Elle inclut deux catégories d'observateur :
- un pour la propriété à vérifier,
- un pour des hypothèses sur l'environnement du modèle.

Les observateurs et leurs connexions avec le modèle à vérifier sont réalisés dans une planche SCADE.

Les hypothèses sont utilisées pour contraindre le domaine d'entrée du modèle tout au long de l'analyse. L'observateur de la propriété calcule un flot booléen p

33. tels que les entiers et les réels

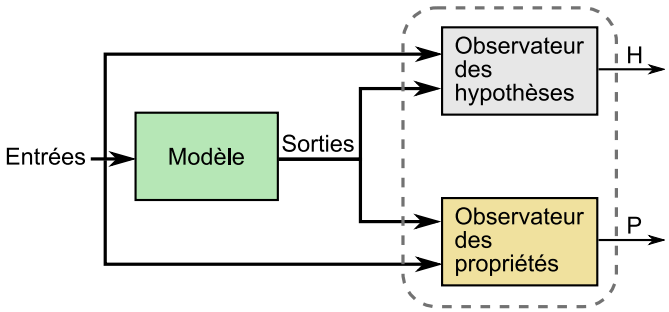


FIGURE 1.8 – Schéma de principe des observateurs

qui est VRAI tant que le modèle satisfait la propriété. L'observateur des hypothèses calcule un flot booléen h que le model checker aura pour contrainte de maintenir à VRAI tout au long de l'analyse de p.

1.5.2.6 Exemple

Pour illustrer les notions que nous venons d'introduire, nous présentons un exemple purement illustratif : une fonction de contrôle d'aérofreins. Cet exemple n'est pas représentatif des modèles Airbus. Le modèle du système d'aérofreinage utilise un symbole de la bibliothèque de CDV Airbus : la bascule R*S. Cet opérateur a deux entrées *Set* et *Reset* et une sortie *O*. A l'instant initial, *O* est FAUX. La sortie devient VRAI dès que *Set* est VRAI et *Reset* est FAUX, et reste VRAI jusqu'à ce que *Reset* soit VRAI. Le modèle de la bascule R*S est visible en figure 1.9.

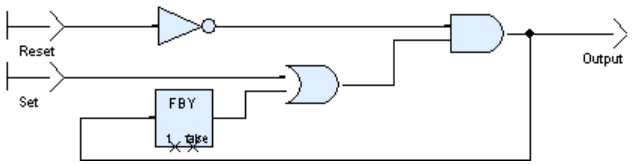


FIGURE 1.9 – La bascule R*S

La figure 1.10 représente le modèle de la fonction d'aérofreinage. L'envoi d'un ordre *airBrakeExtension* active le système d'aérofreinage. Ce dernier est désactivé en envoyant un ordre *airBrakeRetraction*. L'application de l'ordre d'aérofreinage peut être vérifiée en mesurant la position physique de la surface de contrôle utilisée. Dans cet exemple simple, la fonction utilise deux capteurs *airBrakeDown* et *airBrakeUp* qui indiquent respectivement que la surface est rentrée et que la surface est sortie. Les autres entrées *airBrakeLeverOn* et *airBrakeLeverOff* proviennent de deux capteurs positionnés sur le levier d'aérofrein du cockpit. L'entrée *airBrakeLeverOn* est VRAI lorsque le pilote veut sortir les aérofreins et *airBrakeLeverOff* est VRAI lorsque le pilote veut rentrer les aérofreins.

Une exigence de niveau système que l'on peut vérifier est qu'il n'existe pas d'état atteignable où *airBrakeRetraction* et *airBrakeExtension* sont VRAI simultanément. On peut formaliser l'exigence de la manière suivante :

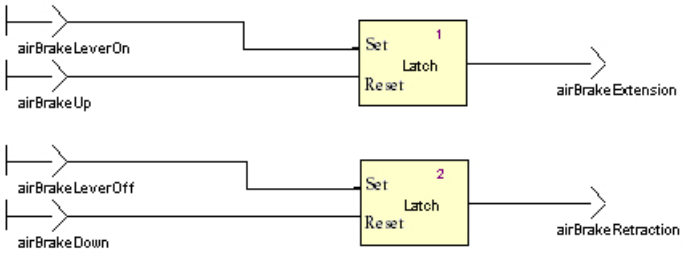


FIGURE 1.10 – Exemple : modèle du système d'aérofreinage

P : Not (*airBrakeExtension* and *airBrakeRetraction*)

D'un point de vue mécanique, *airBrakeLeverOn* et *airBrakeLeverOff* ne peuvent jamais être VRAI au même instant. Ceci est dû au fait que le levier d'aérofrein ne peut être positionné sur on et off au même instant. Nous formulons donc l'hypothèse suivante pour prendre en compte ce phénomène :

H : Not (airBrakeLeverOn and airBrakeLeverOff)

Nous pouvons maintenant définir les observateurs de la figure 1.11 où P est fonction des sorties *airBrakeRetraction* et *airBrakeExtension* du modèle à vérifier. La notation textuelle à gauche de H est une assertion indiquant que la variable H doit être valuée à VRAI tout au long de l'analyse formelle.

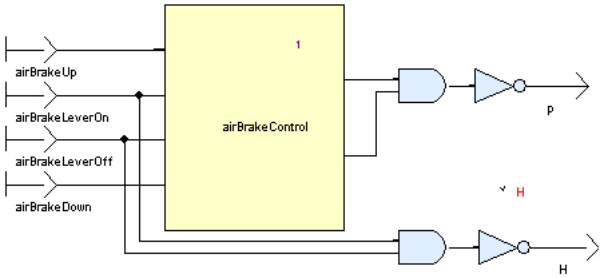


FIGURE 1.11 – Observateurs

Comme le modèle ne satisfait pas la propriété, un contre-exemple est fourni permettant d'identifier la cause de la violation. Au premier cycle d'horloge, le levier d'aérofrein est sur *on* (*airBrakeLeverOn* est VRAI et *airBrakeLeverOff* est FAUX), valuant *airBrakeExtension* à VRAI. Au second cycle d'horloge, le levier d'aérofrein est sur *off* (*airBrakeLeverOn* est FAUX et *airBrakeLeverOff* est VRAI), valuant *airBrakeRetraction* à VRAI. Comme *airBrakeUp* n'est jamais apparu entre ces deux événements, *airBrakeExtension* reste valué à VRAI conduisant le modèle dans un état indésirable violant la propriété. L'étude du contre-exemple nous permet de comprendre l'erreur et de proposer le modèle de la figure 1.12.

En utilisant les mêmes observateurs que précédemment, nous pouvons démontrer que le nouveau modèle satisfait la propriété.

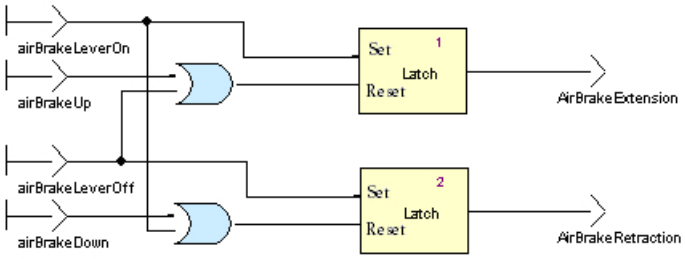


FIGURE 1.12 – Exemple : modèle du système d'aérofreinage corrigé

1.6 Model Checking dans le domaine aéronautique

Nous pouvons déjà citer un certain nombre d'utilisations du model checking à des problématiques aéronautiques.

Le centre de recherche de Langley de la NASA [34] soutient les programmes de recherche en terme de transfert de technologie des méthodes formelles, de la recherche vers l'industrie [35].

Cela fait maintenant plusieurs années que Rockwell-Collins étudie l'intégration des méthodes formelles dans le cycle de développement des systèmes avioniques afin de réduire les coûts en détectant les erreurs au plus tôt.

Rockwell Collins a développé, en partenariat avec le centre de recherche de Langley de la NASA, des passerelles permettant de transformer des modèles simulink ou SCADE dans le langage d'entrée de divers outils d'analyse formelle tels que des model checkers et des démonstrateurs de théorème [36]. Les éventuels contre-exemples générés par les outils sont ensuite transformés dans le langage de départ.

Un exemple d'application [Miller 2009] est l'analyse d'une application d'affichage de cockpit complexe où les analyses formelles ont permis de corriger 98 erreurs. A la fin du projet, 573 propriétés ont pu être démontrées formellement.

Une autre étude [Whalen 2007] a permis la découverte de 12 erreurs en analysant 62 propriétés sur un modèle de système développé par Lockheed Martin. L'analyse formelle a été réalisée en complément des tests sur le modèle. [Whalen 2007] mentionne que le temps total de la vérification était approximativement de 130 heures parmi lesquelles 70 heures ont été consacrées à la préparation du modèle et aux vérifications initiales. La démonstration des 62 propriétés a été réalisée en moins de 8 minutes par le model checker NuSMV. L'étude décompose les travaux en trois étapes :

Préparation du modèle afin que ce dernier soit adapté au model checker.

Vérifications initiales Cette étape regroupe les premières étapes de vérification du modèle.

34. National Aeronautics and Space Administration
35. http ://shemesh.larc.nasa.govfmindex.html
36. theorem provers

Correction du modèle Cette étape regroupe les efforts de vérification et de correction du modèle jusqu'à démontrer les propriétés.

Les conclusions montrent que le temps de préparation est dû au fait que les modèles fournis à Rockwell Collins ne sont pas dédiés à la vérification. Ce coût de préparation peut être réduit en prenant en considération les aspects de vérification dès la réalisation du modèle. La phase de vérification est la plus coûteuse en temps. Cependant, une fois terminée, la correction du modèle et la vérification de non régression sont très rapides et quasi automatiques.

Les techniques de model checking sont aussi étudiées pour automatiser certains traitements liés aux analyses de sécurité [Joshi 2005].

1.7 Conclusion

Ce chapitre a décrit comment est spécifiée la partie logicielle du système de CDV. Il a fait ressortir les efforts importants mis en jeux pour vérifier le bon comportement du logiciel : les tests de modèle, les tests et les vérifications formelles sur le code, les essais labo, au sol et en vol. Le coût de ces vérifications est très élevé. Airbus développe des efforts de recherche afin de trouver des solutions permettant de réduire ces coûts de vérification tout en conservant un degré d'assurance maximal. Les modèles formels, écrit en SCADE, complètement déterministes, ouvrent la porte aux méthodes formelles. Depuis dix ans, Airbus étudie une approche prometteuse permettant de vérifier des propriétés de sûreté sur les modèles formels : le model checking. L'objet du chapitre 2 est de résumer les expérimentations qui ont eu lieu depuis dix ans et au cours de cette thèse sur l'applicabilité du model checking.

Chapitre 2

Expérimentations et motivations

Sommaire

Nous venons de voir, dans le chapitre précédent, à quel niveau du cycle de développement le model checking peut être utilisé. Nous allons maintenant regarder comment s'applique le model checking pour vérifier le modèle du logiciel de CDV. Ce chapitre présente les expérimentations que le département des CDV d'Airbus a mené sur le model checking. Le chapitre se décompose en trois parties. La section 2.1 est une synthèse des études antérieures à la thèse. Nous avons synthétisé les résultats des trois études qui ont eu lieu depuis les années 1999. La section 2.2 décrit une expérimentation plus récente qui entre dans le cadre de la thèse. La section 2.3 conclut sur l'utilisation du model checking comme moyen de vérification de propriétés de CDV et propose des axes de travail pour améliorer l'industrialisation du model checking. Ce chapitre a fait l'objet d'une publication [Bochot 2009].

2.1 Études précédentes

Cette section synthétise l'expérience du département des CDV dans le domaine du model checking avant les travaux de thèse. Le travail de synthèse est le fruit de la première année de thèse. Cette synthèse se base en majorité sur des rapports techniques et des interviews.

Cela fait maintenant 10 ans que ce département étudie les techniques de model checking pour vérifier les propriétés fonctionnelles du système de CDV. Le tableau 2.1 donne un aperçu des trois études pré-thèse. Les abréviations A3456 et SDV signifient respectivement « A340 500-600 » et « Scade Design Verifier ». Parmi ces trois études, seule la première a fait l'objet d'une publication [Laurent 2001].

Dans chacune de ces études, l'objectif est de vérifier que le modèle est conforme à ses spécifications, en utilisant la méthode de l'observateur synchrone décrit dans la section 1.5.2.5. La première étude était une étude de faisabilité. Elle consistait

Etudes	Vepres	CVF	A400M
Année	2000-2002	2002-2003	2005-2006
Cas d'étude	FCSC A3456	FCSC A3456	PRIM A400M
model checker	NP-Tool et Lesar	SDV 4.1 à 4.3	SDV
Type d'exigence	haut niveau	haut niveau	bas niveau système
Nb. d'exp.	3	12	135

TABLE 2.1 – Tableau récapitulatif des expérimentations pré-thèse

à vérifier si les techniques de model checking étaient applicables au système de CDV. La seconde avait pour but d'identifier le domaine d'application du model checking. Trois caractéristiques majeures des systèmes de CDV ont été identifiées. Douze expérimentations ont eu pour objectif d'évaluer les performances du model checking sur des modèles représentatifs de ces caractéristiques. La troisième étude avait pour but de comparer le « retour sur investissement » entre le test de modèle et le model checking dans un contexte opérationnel. L'idée était d'évaluer le model checking pour réduire l'effort et la durée de la vérification du modèle.

Nous allons maintenant présenter ces études plus en détails. La section 2.1.1 décrit la première étude, la section 2.1.2 détaille la deuxième étude, et la section 2.1.3 synthétise les résultats de la troisième étude.

2.1.1 Première étude : VEPRES

VEPRES signifie « Verification par preuves de systèmes critiques embarqués ». Cette première étude [Laurent 2001] a été conduite en collaboration avec l'ONERA.

2.1.1.1 Contexte et objectifs de l'étude

Cette étude avait pour objectif de clarifier deux points :

- les types des propriétés pouvant être vérifiées avec des méthodes formelles,
- les moyens à mettre en œuvre.

Les axes d'études envisagés au début de l'étude étaient le model-checking et la démonstration de théorème. Cependant, pour des raisons techniques, le deuxième axe a été abandonné. Deux outils de model-checking ont été évalués :

NP-TOOLS [Ljung 1999] (commercialisé par Prover Technology) est un outil de vérification pour les circuits combinatoires basé sur la méthode de Stalmarck. (Prover Plug-in, le model checker utilisé aujourd'hui par Scade Design Verifier, est l'évolution de cet outil.)

LESAR [Ratel 1992] est un outil de model checking académique qui analyse les états atteignables. Il permet le choix entre trois stratégies de construction et d'exploration du graphe des états atteignables.

2.1.1.2 Cas d'étude

Le cas d'étude est le calculateur secondaire de CDV du programme A340 500-600 [1]. Les propriétés à vérifier sont issues des SRD. Elles ont été classées en trois catégories :

1. Propriétés liées à un domaine de valeur,
2. Propriétés fonctionnelles de sécurité,
3. Propriétés de reconfiguration système.

Trois expérimentations ont été réalisées, illustrant chacune un des types de propriété.

La **première catégorie** de propriétés concerne les aspects relatifs aux contraintes de précision, à la surveillance et aux calculs d'erreur.

Type de propriété	**Domaine de valeurs**
Exigence informelle	*En supposant que les deux manches ne tombent pas en panne, on a toujours un manche actif,* [...].
Résultat (NP-TOOLS)	L'outil ne parvient pas à conclure
Résultat (LESAR)	L'outil ne parvient pas à conclure

La **deuxième catégorie** de propriétés regroupe principalement les exigences fonctionnelles du système répertoriées dans le SRD.

Type de propriété	**Fonctionnelle de sécurité**
Exigence informelle	*Si une panne est détectée sur le manche pilote, alors le manche est déclaré en panne.*
Résultat (NP-TOOLS)	P_1 : démontrée P_2 : l'outil ne parvient pas à conclure
Résultat (LESAR)	P_1 : démontrée P_2 : l'outil ne parvient pas à conclure

La mise en place de l'observateur a permis d'écrire des exigences plus complètes et moins ambiguës : *il n'existe pas de cas où, soit les deux manches sont en pannes, soit les deux manches ont la priorité, soit le manche qui a la priorité est en panne.* Le premier cas est exclu par hypothèse [2]. Les deuxième et troisième cas sont formalisés par deux propriétés P_1 et P_2. Seule la propriété $P1$ a pu être démontrée avec les deux outils.

La **troisième catégorie** de propriétés prend en compte les propriétés de robustesse et de redondance.

1. Version 500 et 600 de la famille A340
2. Les analyses de sécurité démontrent qu'il est *extrêmement improbable* (probabilité d'occurrence : 10^{-9} par heure de vol) que les deux manches soient en panne au même moment.

Type de propriété	**Reconfiguration système**
Exigence informelle	*Si les calculateurs primaires ne sont pas opérationnels pour l'aileron interne droit, alors un calculateur secondaire opère.*
Résultat (NP-TOOLS)	Propriétés démontrées
Résultat (LESAR)	Propriétés démontrées

Là aussi, la mise en place de l'observateur a permis d'écrire des exigences plus complètes et moins ambiguës : *Si les calculateurs primaires ne sont pas opérationnels pour l'aileron interne droit, alors un calculateur secondaire opère si possible. Sinon le calculateur secondaire signale aux autres calculateurs secondaires qu'il n'est pas opérationnel.* La propriété a pu être démontrée avec les deux outils.

2.1.1.3 Conclusions de l'étude

[Laurent 2001] a classé les conclusions de l'étude en quatre catégories.

La Construction de l'observateur synchrone La technique de l'observateur synchrone doit être transparente à l'utilisateur. Il existe un besoin de construction automatique de l'observateur synchrone.

Le traitement des nombres à virgule flottante Les expérimentations font ressortir que les deux outils parviennent à vérifier les propriétés lorsque les nombres à virgule flottante ont été remplacés par des entiers. Concernant les réels, il y a là une limitation technologique, car d'une part, NP-Tool n'est pas capable de les manipuler et, d'autre part, les expérimentations ont montré l'absence de résultat avec Lesar. Les logiques booléennes n'ont en revanche pas posé de problème.

La bibliothèque des *symboles* On rappelle (voir section 1.4.2) que les *symboles* sont les fonctions primitives qu'Airbus a développé spécialement pour spécifier les CDV. Contrairement à l'ensemble des spécifications de CDV, ces fonctions sont spécifiées textuellement et directement implémentée en C ou en assembleur. Or l'application du model checking nécessite une définition formelle du système. Pour réaliser les expérimentations, il a été nécessaire de définir formellement les symboles utilisés en Scade.

Expressions des propriétés Dans cette étude, les propriétés sont écrites à partir des SRD. Dans deux cas sur trois, la formulation des propriétés a pu être améliorée. Les propriétés sont donc « raffinées » au fil des analyses jusqu'à ce qu'elles soient correctes. Ce travail permet, dans un deuxième temps, de compléter la formulation de l'exigence dans le SRD.

Les auteurs de l'étude font aussi la remarque qu'il est essentiel de s'assurer que les méthodes d'expression des propriétés à démontrer soient compatibles avec la culture des utilisateurs (concepteurs ou valideurs). De ce point de vu, l'écriture des propriétés en langage SCADE est un avantage. De plus, les

auteurs mentionnent le besoin de pouvoir exprimer les propriétés sous la forme d'une table de vérité.

L'étude conclut qu'il existe un intérêt pour les CDV d'utiliser le model checking pour vérifier des propriétés. Cependant, de nouvelles études sont nécessaires afin de mieux comprendre l'applicabilité du model checking. Ceci fait l'objet de l'étude décrite en section 2.1.2.

Suite à cette étude, Esterel Technologies a intégré le model checker *Prover Plug-in* commercialisé par Prover Technology dans son produit *Scade Suite* [Editor 2009].

2.1.2 Seconde étude : CVF

Suite aux résultats de l'étude précédente, Airbus a décidé de poursuivre les recherches sur le model checking. Contrairement à la première, cette étude ne découpe pas les expérimentations selon le type d'exigence mais plutôt selon le type du modèle que l'on souhaite analyser.

2.1.2.1 Contexte et objectifs de l'étude

Cette seconde étude avait pour objectif d'évaluer les capacités de l'outil *Scade Design Verifier* à vérifier des modèles présentant des caractéristiques propres au système de CDV. Un échantillon de modèles avait été sélectionné. Nous proposons de distinguer trois catégories non exclusives de caractéristiques :

Multi processeurs les CDV Airbus utilisent des architectures multiprocesseurs asynchrones, tolérantes aux fautes.

Multi fréquentiel Dans un processeur donné, différentes parties logicielles sont exécutées à différents cycles temporels (selon un séquencement prédéterminé).

Calculs numériques certaines parties des modèles contiennent des calculs numériques complexes sur des entiers ou des flottants (par exemple, modélisation d'un filtre du second ordre).

2.1.2.2 Cas d'étude

Le calculateur secondaire des commandes de vol de l'A340 500-600 a fait l'objet de douze expérimentations. Cette section en décrit quatre d'entre elles. Les autres, pour des raisons de confidentialité, ne sont pas abordées. Une synthèse est proposée en section 2.1.2.3.

Avant de décrire les expérimentations, nous apportons quelques précisions sur le système de CDV. Ce dernier est composé de plusieurs calculateurs. On distingue les calculateurs primaires des calculateurs secondaires. Chaque calculateur est composé de deux unités de calcul :

- une unité de commande, appelée COM, a une fonction de commande,
- une unité d'observation, appelée MON, a une fonction de surveillance.

Ces deux unités permettent de mettre en place un *dispositif de surveillance COM/-MON*. Les deux unités réalisent les mêmes calculs. L'unité COM, compare le résultat de son calcul avec celui de MON afin de le valider. La commande est ensuite appliquée. Si les deux unités sont en désaccord, le calculateur se place en dysfonctionnement et un autre calculateur se met à opérer.

Expérimentation 1 : Non-déclenchement intempestif d'une surveillance.

Type de propriété	Reconfiguration système
Caractéristiques	Booléens, multi-processeurs, mono-fréquentiel
Exigence informelle	Sans problème de transmission entre les calculateurs Com et Mon, la surveillance ne doit pas se déclencher
Résultat	Erreurs détectées dans le modèle et corrigées

Ceci soulève la discussion de la modélisation des communications entre calculateurs. Les planches SCADE modélisent uniquement le comportement logiciel des calculateurs. Lorsque l'on considère un échange d'information entre deux calculateurs, une modélisation des phénomènes liés aux communications est souhaitable. Deux phénomènes physiques interviennent lors d'une communication :

– l'asynchronisme des horloges des deux calculateurs,
– le temps de transmission d'une communication inter-calculateurs.

Pour modéliser ces phénomènes, la notion d'horloge lustre a été utilisée et quatre modélisations ont été testées et sont décrites dans le tableau 2.2.

#	Horloges des calculateurs	Temps de transmission
1	asynchrones variables	variable
2	synchrones	variable
3	asynchrones constantes	Constant
4	synchrones	Constant

Table 2.2 – Modélisations des communications inter-unités

Le premier modèle de communication introduit une telle complexité que le model checker ne parvient pas à analyser la propriété. Le second modèle de communication n'est pas représentatif de la réalité. La troisième modélisation ne représente pas finement la réalité cependant elle présente l'avantage d'être peu complexe. La quatrième modélisation est un cas particulier de la troisième. Elle sera utilisée lorsqu'une analyse avec la troisième modélisation échoue.

D'une part on remarque que la modélisation de phénomène asynchrone n'est pas simple. Cette difficulté de modélisation peut s'expliquer par le langage. Scade se base sur le langage Lustre qui est un langage synchrone. Il n'est donc pas destiné à la modélisation de phénomènes asynchrones.

D'autre part, on peut remarquer qu'il existe un compromis entre précision et complexité du modèle. Parmi les modélisations proposées, les modélisations 3 et 4

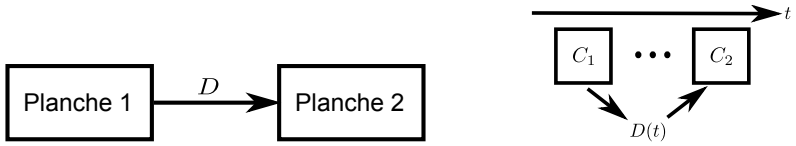
ont été utilisées. Ces modélisations ne sont pas représentatives de la réalité. Cependant, elle ne sont pas moins représentatives que celles utilisées lors des tests de modèle.

Expérimentation 2 : Non-déclenchement intempestif d'une surveillance. La modélisation 3 du tableau 2.2 a été utilisée pour modéliser les communications entre calculateurs (avec un délai de 0ms ou 10ms).

Type de propriété Caractéristiques	Reconfiguration système Booléens, multi-processeurs, multi-fréquentiel
Exigence informelle	Non déclenchement intempestif d'une surveillance de communication inter-calculateurs
Résultat	Propriétés démontrées

Concernant l'aspect *multi-fréquentiel*, la modélisation des différentes horloges a été réalisée grâce à la notion d'*horloge Lustre*. Le séquencement des planches SCADE est statique et défini dans un document que l'on appelle *planche tempo*. Ce séquencement ne respecte pas toujours la règle du producteur/consommateur.

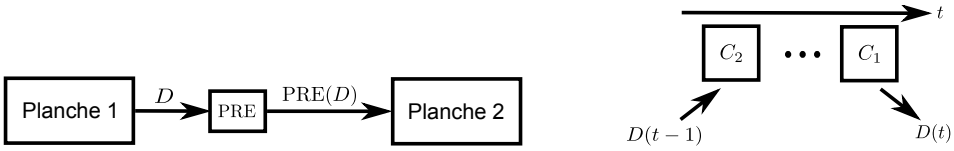
Par exemple, considérons deux planches Scade nommées 1 et 2 séquencées avec la même horloge. La planche 2 utilise une donnée D produite par la planche 1. Le modèle de la figure 2.1(a) suit la règle du producteur/consommateur. Dans ce modèle, le calcul de la planche 1 est réalisé, dans un premier temps, afin de produire D. Dans un deuxième temps, le calcul de la planche 2 est réalisé en utilisant la donnée D qui vient d'être mise à jour, tel que représenté sur la figure 2.1(b).



(a) Modèle (b) séquencement

FIGURE 2.1 – Principe du producteur/consommateur

S'il est défini dans la *planche Tempo* que la planche 2 doit être ordonnancée avant la planche 1, il est nécessaire d'introduire un opérateur PRE entre les deux planches comme représenté sur la figure 2.2(a). Ce modèle spécifie que la planche 2 consomme la variable D produite à l'instant $t-1$ comme représenté sur la figure 2.2(b).



(a) Modèle (b) séquencement

FIGURE 2.2 – Solution proposée

L'effort de modélisation a été important et un outil a été développé afin d'automatiser cette tâche. L'outil prend en entrée l'ensemble des planches Scade qui composent le modèle et la *planche tempo* et génère automatiquement le modèle.

Expérimentation 3 : Engagement d'un calculateur secondaire.

Type de propriété Caractéristiques	Reconfiguration système Booléens, mono-calculateur, mono-fréquentiel
Exigence informelle	Sur l'actionneur de la gouverne de profondeur, lorsqu'il n'y a pas de panne, le calculateur primaire prioritaire est en opération
Résultat	Propriétés démontrées

La modélisation d'un symbole de la bibliothèque Airbus a posé des problèmes au model checker. Une fois le symbole modélisé uniquement avec des opérateurs booléens, la propriété a pu être analysée et validée. La manière dont sont modélisés les symboles de la bibliothèque est un facteur important. Une modélisation inadéquate a pour effet, dans le meilleur des cas, d'augmenter le temps d'analyse du model checker. Dans le pire des cas cela peut conduire à un échec de l'analyse. Pour optimiser la modélisation des symboles, il est plus facile de connaître la manière dont fonctionne le model checker. Dans notre cas, cette exercice est rendu difficile par le fait que les stratégies d'analyse du model checker ne sont pas rendues publiques.

Expérimentation 5 : Engagement d'un calculateur

Type de propriété Caractéristiques	Reconfiguration système Booléens, mono-processeur, multi-fréquentiel
Exigence informelle	Si aucune panne n'est détectée pour le calculateur, alors le calculateur doit s'engager
Résultat	Propriétés démontrées

Le but de cette expérimentation est d'observer le comportement du model checker avec des modèles de taille importante. Le modèle se restreint à de la logique booléenne avec 4 séquencements différents. Cette expérimentation fait intervenir plus de 100 planches SCADE dans lesquelles 633 variables sont produites. Bien que l'aspect graphique du formalisme SCADE soit très apprécié pour des systèmes de petite taille, son utilisation pour des systèmes de taille importante devient un problème.

2.1.2.3 Conclusions de l'étude

Nous allons faire le point sur les expérimentations décrites précédemment. Le tableau 2.3 récapitule le résultat des douze expérimentations.

Lorsqu'aucune des trois colonnes du milieu n'est cochée, cela signifie que l'expérimentation porte sur une fonction booléenne réalisée sur un unique processeur,

#	Multi-processeurs	Multi-fréquentiel	Calculs numériques	Terminaison de l'analyse
1	✓			✓
2	✓	✓		✓
3				✓
4			✓	
5		✓		✓
6	✓	✓	✓	
7	✓			✓
8				✓
9				✓
10	✓	✓		
11			✓	✓
12			✓	

TABLE 2.3 – Résultats de la seconde étude

et toutes les planches sont calculées à une même fréquence. La colonne de droite indique si l'outil de model checking a été capable de terminer son analyse (que la propriété ait été démontrée valide ou invalide).

Les expérimentations sur les aspects multiprocesseurs ont fait ressortir des problèmes de modélisation. Deux phénomènes ont dû être modélisés : les délais de communication et l'asynchronisme des horloges entre les deux processeurs communicants. Ces deux phénomènes augmentent la complexité du modèle de manière significative. Les expérimentations 1, 2, 7 ont réussi car elles utilisaient un modèle simplifié. L'expérimentation 10, en revanche, a été un échec : on observe un échec de l'analyse due à une prise en compte plus fine des phénomènes asynchrones mentionnés plus haut. Il existe donc un compromis entre précision du modèle et faisabilité de la vérification. Il est important de remarquer que SCADE n'est pas dédié à la modélisation de comportements asynchrones.

Les problèmes liés à la modélisation des ordonnancements et des communications inter-calculateurs [3], ont été résolus par le développement d'un outil automatique. La vérification de propriétés de sûreté n'a pas posé de problème particulier. Par exemple, l'expérimentation 5 a pu aboutir à une conclusion pour l'analyse d'un modèle composé de 100 planches SCADE utilisant quatre horloges différentes.

Concernant les problèmes numériques, l'expérimentation 11 a abouti à une conclusion en analysant un problème contenant des nombres à virgule flottante. Ce résultat est plutôt satisfaisant, comparé aux médiocres performances constatées lors de la première étude. Il reflète les améliorations apportées à l'outil de model checking entre ces deux études, pour mieux prendre en compte les traitements numériques. Cependant, l'analyse d'équations non linéaires reste hors de portée, menant les ex-

3. en utilisant la modélisation 3 du tableau 2.2 et en proposant plusieurs délais de communication constants

périmentations 4, 6 et 12 à l'échec.

La conclusion de cette étude est que le model checking peut être utilisé sur beaucoup de fonctions logiciels du système de CDV, pourvu qu'elles n'impliquent pas de calculs numériques trop complexes ni de communication inter-processeurs.

2.1.3 Troisième étude : A400M

Suite aux résultats de l'étude précédente, Airbus a décidé d'étudier une utilisation massive du model checking pour vérifier des exigences système de bas niveau, telles que décrites dans les DTS. Pour évaluer l'approche, il a été décidé de comparer le « retour sur investissement » entre une approche de vérification par test et une approche de vérification par model checking. Un total de 135 expérimentations ont eu lieu portant sur trois fonctions du système de commande de vol primaire de l'A400M. Pour des raisons de confidentialité, nous ne décrirons pas les expérimentations. Cependant, nous énoncerons les conclusions de cette étude.

2.1.3.1 Contexte et objectifs de l'étude

Les objectifs de cette étude étaient de :
- promouvoir les techniques formelles au sein des équipes,
- intégrer les méthodologies dans les processus transverses (V&V, traçabilité des exigences, gestion de configuration, Gestion des modifications...) et,
- de capitaliser l'expérience.

Cette étude avait pour but d'apporter des éléments de réponse à la question : le model checking peut-elle remplacer partiellement ou totalement les *tests de modèle* ?

2.1.3.2 Cas d'étude

Les 135 expérimentations correspondent chacune à une exigence système de bas niveau incluse dans un des trois cas d'études suivant :

1. Fonction Manche des calculateurs PRIM et SEC du programme A400M
2. Fonction Pédale des calculateurs PRIM et SEC du programme A400M
3. Fonction d'équilibrage [4] des calculateurs PRIM et SEC du programme A400M

Au cours de cette étude, une bibliothèque de modèles de capteurs a été développée. Ils modélisent les comportements fonctionnels et disfonctionnels de certains capteurs tels que :
- les potentiomètres mesurant l'angle du mini manche du programme A400M,
- les capteurs RVDT [5] mesurant la position du palonnier,
- les Switchs.

Ces modèles de capteurs permettent :
- d'éliminer les comportements impossibles des entrées du système.

4. ou fonctions de TRIM

5. de l'anglais *Rotary Variable Differential Transformer* est un capteur électrique actif (inductif) de rotation.

– De sélectionner un type de comportement (nominal ou défaillant) d'une entrée.

De plus, pour faciliter le travail d'écriture des propriétés, une bibliothèque d'opérateurs permettant d'exprimer des patrons de propriétés a été implémentée. Ces patrons formalisent des bribes d'exigences systèmes de bas niveau couramment rencontrées.

2.1.3.3 Conclusions de l'étude

L'étude conclut que le taux de détection d'erreur du model checking n'est pas plus élevé que celui du test et que, de plus, son temps de mise en œuvre est deux à trois fois plus élevé. Une raison de ce faible taux de détection d'erreur peut provenir du fait que les exigences vérifiées sont de bas niveau. Les propriétés qui en découlent sont très souvent proches et en quelque sorte similaires à la spécification détaillée. Le pouvoir de détection d'erreur semble décroître à mesure que les propriétés sont syntaxiquement proches du modèle. Il en ressort que, dans le processus de développement Airbus, le model checking n'est pas une approche efficace pour vérifier la conception SCADE vis-à-vis des spécifications détaillées.

2.2 Application du Model checking à la fonction ground spoiler

Nous allons maintenant parler d'une étude plus récente que nous avons réalisée. Elle traite de la fonction ground spoiler du programme A380 [Bochot 2009]. Dans cette section, nous présentons brièvement les expérimentations effectuées, une présentation plus détaillée étant exclue pour des raisons de confidentialité. Les résultats obtenus seront commentés ultérieurement, en section 2.2.1.

La fonction ground spoiler a pour but de garder l'avion au sol à l'atterrissage ou en cas d'abandon du décollage (*rejected take-off*). Les *ailerons* et les *spoilers* (figure 1.1) sont utilisés pour casser la portance de l'aile dans le but d'optimiser le freinage des roues de l'avion. De plus, le braquage de ces surfaces provoque un effet d'aérofreinage. La fonction est réalisée de telle manière qu'une activation intempestive soit *extrêmement improbable* lorsque l'avion est en vol.

Le cas d'étude a été choisi comme donnant un exemple dans lequel une analyse formelle pourrait compléter utilement la validation actuelle de la conception SCADE. La propriété que l'on souhaite vérifier est une propriété haut niveau de sécurité qui stipule que la fonction ground spoiler ne doit pas s'activer lorsque l'avion est en vol. Nous disposons de deux modélisations SCADE de la fonction : une version erronée et une corrigée. La première présente une faute de conception ayant échappé aux tests de niveau modèle. La faute a été révélée durant les *essais labo.* Notre but est de démontrer que cette faute aurait pu être trouvée plus tôt par une approche de model-checking, ce qui aurait été moins coûteux. Nous voulons aussi démontrer que la version corrigée satisfait l'exigence de sécurité.

La fonction ground spoiler est une bonne candidate à l'application du model checking. Elle est en grande partie basée sur du raisonnement booléen. Les calculs

numériques y sont peu nombreux et peu complexes (additions et comparaisons de valeurs réelles). Cependant, la présence de compteurs temporels peut être une source de problème, du fait du nombre de cycles que l'outil a besoin de considérer pour réaliser l'analyse.

Compte tenu des études passées (section 2.1), le contexte de la vérification formelle est délibérément réduit. Nous allons nous focaliser sur un petit nombre de planches SCADE extraites de la conception complète. Nous exclurons les aspects multiprocesseurs et focaliserons l'analyse sur le fonctionnement nominal de la fonction : pas de *reset* et pas de cas de panne.

L'analyse de la version erronée a démarré en incluant une seule planche SCADE, celle qui calcule l'ordre final envoyé aux surfaces de contrôle. L'analyse a nécessité plusieurs étapes de vérification :

1. Un certain nombre d'itérations initiales ont été nécessaires pour obtenir un observateur adapté au problème de vérification. Nous avons expérimenté différentes formulations de la propriété de sûreté et des hypothèses restreignant le contexte de vérification, avant d'en retenir une.
2. Une fois la formalisation établie, nous obtenons un contre-exemple qui n'est pas simple à interpréter. La violation de la propriété a été causée par l'absence de *reset* d'une bascule. La condition de *reset* (*RCond*) est maintenue à FAUX. Pour comprendre si ce comportement est possible, nous avons dû étendre l'analyse pour inclure la planche Scade produisant *RCond*. Nous avons donc modifié l'observateur en conséquence. Le modèle analysé est maintenant composé de deux planches SCADE.
3. L'analyse formelle du modèle étendu a confirmé le contre-exemple précédemment obtenu.
4. Nous avons décidé de poursuivre notre recherche d'erreur. Comme nous ne voulions pas modifier les planches SCADE, nous avons modifié l'expression de la propriété de telle manière à exclure le contre-exemple précédent. L'analyse formelle renvoie un nouveau contre-exemple avec *RCond* maintenu à VRAI.

Les contre-exemples correspondent à des scénarios opérationnels différents. Dans les deux cas, un problème de *reset* de bascule se produit. Mais chacun de ces problèmes résulte d'une trajectoire spécifique (inhabituelle) de l'avion combinée à des actions pilote. Le second scénario est celui qui avait été trouvé tardivement par test. Le premier est une manifestation alternative de la même faute de conception.

La version corrigée de la fonction ground spoiler emploie une logique interne différente qui n'utilise plus *RCond*. Pour vérifier cette logique, nous avons réutilisé le même observateur que celui obtenu à la fin de l'étape 1, avec de légères adaptations pour prendre en compte le changement d'interface de la fonction. Les premières tentatives de vérification ont été des échecs (l'outil de model checking ne parvenait pas à terminer son calcul). Après l'introduction d'hypothèses sur des variables intermédiaires, l'analyse formelle a pu se conclure avec succès. Des vérifications dédiées nous ont ensuite permis de démontrer la validité des hypothèses précédemment introduites, complétant ainsi l'analyse formelle.

2.2.1 Les leçons tirées de cette étude

Après cette présentation factuelle, nous revenons sur ce qui, à notre avis, constitue les principales leçons que l'on peut tirer de l'étude.

2.2.1.1 Efficacité de la vérification formelle

Bien que la fonction Ground Spoiler utilise peu de calculs numériques, elle est suffisamment complexe pour poser problème à l'outil de vérification, notamment à cause de la présence de compteurs temporels. Il a fallu 48 heures pour réaliser l'analyse de la version correcte (sur un Pentium 1.7-GHz avec 256 Mb de RAM). En ce qui concerne la version incorrecte, la génération de contre-exemples a pris de quelques minutes à quelques heures, selon la longueur des contre-exemples et la stratégie d'exploration choisie (SCADE propose deux stratégies). Les contre-exemples produits avaient une longueur comprise entre 50 et 160 cycles.

D'un point de vue de la détection d'erreur, la vérification formelle a été très efficace. Rappelons que les tests du modèle n'avaient pas permis de révéler le problème. Ceci est dû au fait que les aspects très dynamiques ne sont pas la préoccupation principale de ces premiers tests : les trajectoires avions inhabituelles, telles que celles des scénarios qui violent la propriété, ne sont pas considérées à cette étape de validation. La dynamique du vol est traitée de façon approfondie par les *essais labo* (en utilisant le simulateur de vol) et bien sûr par les *essais en vol*. Cependant, plus les fautes de conception sont révélées tôt, moins leur correction coûte cher. Notre étude a montré que la vérification formelle peut offrir une solution pratique pour trouver de telles fautes lors de la conception détaillée. L'étude a également fourni de nombreuses informations sur le comportement erroné, en permettant d'identifier deux scénarios de violation de la propriété (correspondant à deux trajectoires d'avion différentes).

2.2.1.2 Expression des propriétés

Passer d'exigences en langage naturel à des énoncés formels de propriétés est connu comme un problème difficile. Dans notre cas, le problème est exacerbé par le fait que les modèles à vérifier sont exprimés à un niveau unitaire détaillé, tandis que les exigences informelles sont relatives à un niveau avion plus abstrait. Par conséquent, il n'est pas facile d'exprimer ces exigences en utilisant les variables concrètes des planches SCADE concernées.

Ce problème avait déjà été identifié dans la première étude mentionnée en section 2.1.1. Dans cette étude, un exemple de difficulté était l'expression formelle de la notion de manche *actif*. Informellement, le manche pilote ou copilote est actif s'il peut avoir un effet sur les surfaces de l'avion. Formaliser cette notion en termes d'entrées/sorties des planches SCADE s'était avéré délicat et plusieurs formulations alternatives avaient été étudiées. Dans l'étude Ground Spoiler, une difficulté a été de formaliser la notion d'avion *en vol*.

Un écueil serait de paraphraser le modèle SCADE (voir les conclusions de la troisième étude, section 2.1.3). Par exemple, la fonction Ground Spoiler calcule une variable intermédiaire pour savoir si l'avion est au sol ou pas. Si le même calcul est utilisé dans l'observateur, le modèle satisfera la propriété de façon triviale. Il a donc fallu trouver une formalisation indépendante.

Il est intéressant de noter que ce problème d'expression de propriétés est beaucoup moins aigu pour l'analyse des résultats de tests. Pour les tests sur le modèle, comme pour les *essais labo*, l'environnement de simulation fournit une référence externe pour déterminer quelle est la configuration « réelle » de l'avion (par opposition avec la configuration « perçue » par les fonctions logicielles). En l'occurrence, déterminer si l'avion est en vol ne présente pas de difficulté lorsque l'on a accès aux données du simulateur.

2.2.1.3 Détermination des hypothèses

Déterminer un ensemble d'hypothèses qui permettent de restreindre le domaine de la vérification est un problème récurrent quelle que soit la méthode de vérification choisie. Deux types d'hypothèses existent dans notre étude :

- Des hypothèses pour exclure des comportements irréalistes,
- Des hypothèses pour simplifier le problème de vérification et permettre l'analyse automatique.

En ce qui concerne la première catégorie, il n'est pas facile en pratique de différencier finement les comportements réalistes des irréalistes. Il y a une difficulté inhérente due à la prise en compte de la dynamique du vol. Alors que des modèles dynamiques complexes sont facilement intégrés dans les environnements de test, leur prise en compte par la vérification formelle n'est pas concevable : l'analyse automatique ne serait pas possible. De plus, des problèmes de performance nous obligent à nous concentrer sur un (petit) sous-ensemble de planches SCADE, ce qui signifie que les entrées considérées ne sont pas forcément les entrées du système. Par conséquent, nous ne prétendons pas caractériser le domaine des entrées valides de la fonction considérée. Les hypothèses permettent seulement d'éliminer des comportements grossièrement irréalistes (par exemple, nous ajoutons des domaines de valeur pour les données numériques ou nous exprimons des relations évidentes entre certaines entrées).

Au contraire des hypothèses précédentes, les hypothèses de simplification nous amènent à exclure délibérément certains comportements réalistes. Par exemple, nous avons mentionné que l'analyse ne considère qu'une seule unité de calcul dans des conditions nominales de fonctionnement : nous ne prenons pas en compte les erreurs de capteurs ou les *resets* du calculateur. Un autre exemple concerne les trains d'atterrissage. Nous supposons que les roues sont soit toutes arrêtées soit toutes en rotation à la même vitesse. En réalité, les roues peuvent bien sûr avoir des comportements différents, mais nous avons jugé que ces considérations alourdiraient inutilement l'analyse.

Pour résumer, la vérification est faite sous des hypothèses qui concernent un sous-ensemble des comportements possibles sans pour autant exclure tous les comportements irréalistes. Un ensemble d'hypothèses donné peut toujours être remis en question ; un avantage de la formalisation est précisément de rendre ces hypothèses explicites. En règle générale, nous avons évité de mettre des hypothèses sur les actions du pilote : la satisfaction des exigences ne dépend donc pas des procédures opérationnelles de l'équipage.

2.2.1.4 Interprétation des contre-exemples

Si la vérification formelle se termine avec succès, on a démontré que la fonction respecte bien la propriété sous les hypothèses définies. Mais si un contre-exemple est trouvé, une analyse supplémentaire est nécessaire.

Le processus d'analyse d'un contre-exemple peut se décomposer en deux étapes séquentielles :

1. la première étape consiste à comprendre le scénario. A la différence du test ou de la simulation, où l'utilisateur possède un modèle mentale du scénario, les contre-exemples ne sont pas connus de l'utilisateur. Cette étape va permettre à l'utilisateur de juger du degré de réalisme du scénario. Dans nos expérimentations, cette étape a été réalisée à l'aide d'un simulateur.
2. La deuxième étape consiste à comprendre ce qui a causé la violation de la propriété. Même si le contre-exemple a été jugé irréaliste, il est toujours intéressant de chercher ce qui l'a provoqué. Cela peut permettre d'identifier un scénario réaliste, une mauvaise formulation de la propriété, ou encore une hypothèse manquante. Dans nos expérimentations, cette étape de *diagnostic* a été réalisée par une analyse structurelle et temporelle à rebours. Ce parcours à rebours permet de filtrer l'information du contre-exemple inutile et à identifier les événements clés ayant engendrés la violation.

Rappelons que la vérification est effectuée au niveau unitaire : les configurations d'entrée du contre-exemple doivent donc être interprétées au niveau système. De plus, les hypothèses de vérification ne sont pas suffisantes pour exclure tous les comportements irréalistes. Il faut donc déterminer si le contre-exemple correspond à un scénario avion réel, ce qui nécessite un effort significatif.

La situation est différente pour les tests. Toutes les catégories de test, y compris les tests de modèle, reposent sur des scénarios de test au niveau système qui ont une signification physique claire (par exemple un scénario d'atterrissage). L'analyse des résultats de test est donc facilitée. Par exemple, pour analyser les résultats des tests sur le modèle, les testeurs utilisent des chronogrammes qui visualisent les traces de test. L'interprétation de l'évolution des variables est guidée par la compréhension du scénario considéré.

Les chronogrammes ne s'avèrent pas aussi utiles pour analyser les contre-exemples. L'évolution de certaines variables semble incompréhensible d'un point de vue physique (par exemple, l'outil de vérification peut donner des valeurs arbitraires aux

variables qui ne contribuent pas directement à la violation de la propriété). Pourtant, malgré quelques détails irréalistes, le contre-exemple peut indiquer un problème réel dans la fonction considérée.

Comme les chronogrammes n'étaient pas très utiles, une autre approche a été adoptée. Elle a consisté à :

1. identifier les parties du modèle activées par le contre-exemple au cours du temps ;
2. essayer de trouver un scénario de niveau système qui reproduit le schéma d'activation observé.

La deuxième étape est la plus difficile et requiert de l'expertise dans le domaine avionique. La première étape, par contre, pourrait être automatisée à condition de préciser la notion de schéma d'activation.

2.2.1.5 Approche itérative de vérification

Du fait des problèmes évoqués ci-dessus (expression des propriétés et des hypothèses, interprétation des contre-exemples), l'analyse formelle de la conception SCADE est loin de constituer une approche « presse-bouton ». Cela ressemble plutôt à une approche itérative, comme décrit sur la figure 2.3.

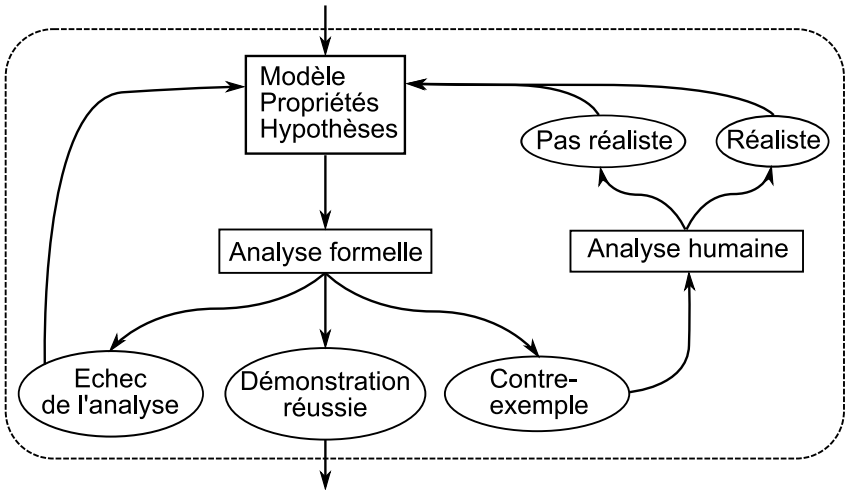


FIGURE 2.3 – Méthodologie de l'analyse

Des itérations sont nécessaires pour explorer des formulations alternatives du problème de vérification (aussi bien de la propriété que des hypothèses). Les étapes de vérification associées fournissent des retours utiles pour améliorer cette formulation. L'échec de l'analyse suggère éventuellement le besoin d'hypothèses supplémentaires. Les contre-exemples irréalistes mettent en cause la pertinence des hypothèses ou de la propriété exprimée. De façon générale, la figure 2.3 montre bien l'importance du jugement humain dans l'analyse des contre-exemples. Il faut remarquer que dans certains cas, la recherche d'une interprétation système du contre-exemple

peut soulever des questions sur d'autres parties du modèle SCADE. L'intégration de ces parties dans le modèle analysé peut nécessiter la formulation de nouvelles hypothèses ou propriétés.

Étant donné un problème de vérification, le premier contre-exemple trouvé par l'outil n'est pas forcément le plus intéressant. Il peut représenter un comportement irréaliste que l'on n'a pas pu exclure par des hypothèses simples. L'analyse doit alors se poursuivre pour savoir si d'autres contre-exemples peuvent être trouvés, qui pourraient révéler une violation de la propriété. Ou bien, dans le cas où un problème réel est révélé, il peut être utile d'identifier différents scénarios dans lesquels ce problème se manifeste. Lors de l'étude, nous avons rencontré ces différents cas justifiant la recherche de contre-exemples supplémentaires. Malheureusement, SCADE Design Verifier (comme la plupart des model checkers) ne fournit pas plusieurs contre-exemples. Répéter la vérification produira toujours le même contre-exemple. Pour pallier ce problème, nous avons dû modifier manuellement l'observateur pour exclure le contre-exemple déjà trouvé. De cette façon, nous avons pu explorer des scénarios alternatifs de violation de la propriété. Nous avons finalement trouvé la faute révélée lors des tests et démontré qu'il existait plusieurs scénarios de violation associés à cette faute.

2.3 Conclusion

Les expérimentations décrites dans ce chapitre ont pour cadre le processus de développement Airbus pour les systèmes de CDV. Ce cadre particularise la mise en œuvre du model checking de la façon suivante :

- La formalisation intervient lors de la phase de conception système. Les modèles analysés sont donc très détaillés, et de niveau concret.
- Les fonctions étudiées s'exécutent dans un environnement multi-processeurs et multi-fréquentiel (y compris au sein d'un même processeur).
- Certaines fonctions contiennent essentiellement de la logique booléenne, d'autres font appel à des calculs numériques qui peuvent être complexes.
- Les propriétés vérifiées peuvent être du même niveau d'abstraction que les modèles (spécifications détaillées) ou de niveau plus élevé (exigences de sécurité).

L'étude de la fonction Ground Spoiler fait suite à une série d'expérimentations qui avaient déjà permis de cerner le champ d'application du model-checking dans ce cadre spécifique. La fonction et la propriété ciblées rentraient a priori dans le champ d'application identifié. Outre la confirmation de la faisabilité de l'analyse formelle, l'objectif était de se confronter à un problème réel, survenu lors du développement de l'A380.

Les résultats obtenus montrent que le model checking peut être très efficace pour trouver des fautes subtiles qui sont autrement révélées plus tard dans le cycle de développement. Nous pensons donc qu'une utilisation industrielle du model checking est possible et intéressante pour Airbus au niveau de la conception système. Pour le

moment, il semble que l'utilisation la plus efficace soit une utilisation *poids plume*[6] [Saiedian 1996], où l'analyse formelle est appliquée à un petit nombre de fonctions critiques et se concentre sur un sous-ensemble de comportements possibles. Le model checking n'est pas encore prêt pour une utilisation plus étendue[7].

Au-delà des retours sur l'applicabilité et l'efficacité du model-checking dans le cadre du système de CDV Airbus, les études menées permettent de pointer sur des besoins industriels en termes d'outillage.

Même dans le cas d'une utilisation poids plume, on peut rencontrer des problèmes de performances. Pour la fonction Ground Spoiler les premières tentatives de preuve ont échoué et la tentative finale a pris 48 heures environ. Il serait intéressant que l'outil soit capable de proposer des métriques pour indiquer la difficulté de la preuve pour un modèle et une propriété donnée. De telles métriques permettraient de guider la décision de tenter ou non une vérification formelle, et dans l'affirmative, de planifier l'attribution de ressources de calcul en adéquation avec la difficulté estimée.

Une autre conclusion importante de l'étude Ground Spoiler est que le model checking ne se résume pas à appuyer sur un bouton pour lancer une analyse automatique. Il s'agit plutôt d'une approche itérative, comme décrite sur la figure 2.3. De plus, ce processus va lui-même être répété plusieurs fois : même si la preuve se termine avec succès, l'utilisateur peut vouloir aller plus loin dans l'analyse en supprimant une hypothèse, en modifiant la propriété ou en élargissant le modèle analysé.

Les outils de vérification ne supportent pas ce processus itératif de façon suffisante. Il existe clairement un besoin de mieux outiller la gestion de configuration de la vérification, pour pouvoir par exemple archiver les expériences successives et leurs résultats, extraire un historique à partir d'une archive, faciliter la comparaison des configurations considérées. Un autre problème est l'impossibilité de générer plusieurs contre-exemples. La modification manuelle de l'observateur pour éviter le contre-exemple précédent n'est pas une solution satisfaisante. Nos perspectives sont de chercher à définir une approche automatisée. Un problème intéressant est de forcer l'outil à produire des contre-exemples qui illustrent des scénarios de violation différents. Lors de l'interprétation d'un contre-exemple, une première étape consistait à chercher à caractériser la violation observée, en identifiant les parties du modèle activées au cours du temps.

Nous allons étudier l'intérêt d'une telle analyse structurelle non seulement :

– pour faciliter la compréhension d'un contre-exemple, mais aussi
– pour guider le model checker vers l'exploration de comportements différents.

Des problèmes similaires concernant l'exploitation de contre-exemples ont déjà été étudiés par d'autres auteurs. Dans le domaine du model checking appliqué au code source, [Groce 2003] cherche plusieurs exécutions exhibant une même erreur afin de localiser plus précisément la faute. Dans le domaine du model checking

6. *lightweight*
7. telle que celle mise en œuvre au niveau du code [Duprat 2006]

appliqué a une spécification, [van den Berg 2007] propose d'utiliser une connaissance spécifique du domaine afin d'extraire l'information pertinente et la présente sous une forme adaptée aux ingénieurs du domaine (signalisation ferroviaire).

Notre approche se situe au niveau d'un modèle Scade et tient compte de sa structure pour expliquer la violation observée. Le chapitre 4 met en place un cadre formel qui nous permettra d'exprimer une notion de *cause*. Cette notion sera ensuite utilisée pour :

- analyser un contre-exemple,
- guider le model checker vers des contre-exemples « différents ».

Chapitre 3

Analyse structurelle de modèles : État de l'art

Sommaire

Nous avons vu dans le chapitre 2 que l'application du model checking donne lieu à un processus itératif. Ce dernier est guidé par l'interprétation d'un *contre-exemple* et par la synthèse de nouveaux *observateurs* dépendants du *contre-exemple*. Nous souhaitons apporter une assistance à l'interprétation d'un contre-exemple en identifiant les parties du modèle activées au cours du temps par ce contre-exemple. Cette assistance permettra, d'une part de faciliter la compréhension du contre-exemple et, d'autre part de générer de nouveaux contre-exemples différents.

Cet objectif met en jeu deux aspects, diagnostic et analyse structurelle, qui ont tous les deux des liens avec le test. Dans ce chapitre, nous allons d'abord présenter brièvement le test dans la section 3.1, puis nous positionnerons notre approche par

rapport à des travaux proches. La section 3.2 présente des travaux dans le domaine du test du matériel. Nous avons été amenés à nous y intéresser, car un modèle SCADE est très proche d'un modèle de circuit. L'objectif des travaux que nous présentons est d'exciter une faute particulière et de la propager pour la rendre observable. Ces travaux ont en commun avec notre approche la nécessité d'effectuer un parcours structurel dans un circuit combinatoire ou séquentiel. Ce parcours est guidé par la valuation de certaines variables. Dans la section 3.3, nous présentons les critères de couverture structurelle définis classiquement dans le domaine du test de logiciels, puis nous présentons des travaux plus récents de définition de critères de couverture structurelle pour le langage Lustre dont nous adapterons certains concepts par la suite. Dans la section 3.4, nous présenterons des outils capables de générer automatiquement des tests à partir d'un modèle Lustre. Finalement, nous présentons dans la section 3.5 des travaux concernant le diagnostic de fautes dans les modèles Lustre.

3.1 L'activité de test

Le test est une technique de vérification dynamique permettant de s'assurer qu'un système est conforme à ses *exigences*. Les techniques de *test* s'appliquent aussi bien dans le domaine du matériel [1] [Wang 2006, Bushnell 2000, Abramovici 1990], que dans le domaine du logiciel [Beizer 1990, Ammann 2008].

L'activité de *test* se déroule généralement en trois étapes :

1. on *sélectionne* les vecteurs de test,
2. on *exécute* les vecteurs de test sur la cible [2],
3. on *décide de l'exactitude des résultats* des tests fournis par le programme en réponse aux vecteurs de test.

L'entité qui décide de l'exactitude des résultats de test est appelée l'**oracle**. L'*oracle* est généralement un opérateur humain. On appelle **vecteurs de test** l'ensemble des valuations des entrées appliqué au système durant le *test*.

De manière générale, la sélection des *vecteurs de test* est problématique. Si le composant est simple, on peut procéder à un *test exhaustif*. Le composant est alors exercé avec toutes les combinaisons possibles d'entrées. Sinon, il faut sélectionner un sous-ensemble de vecteurs. On distingue deux types de test qui se différentient par la manière dont sont sélectionnés les *vecteurs de tests* :

1. le *test fonctionnel*,
2. le *test structurel*.

3.1.1 Le test fonctionnel

Dans le cadre du *test fonctionnel*, les *vecteurs de test* sont construits uniquement à partir des *exigences*, en faisant abstraction de la manière dont est réalisé le système.

1. des composants électroniques.
2. le calculateur

Les exigences peuvent être décrites en langage naturel, ou être formalisées dans des descriptions telles que des tables de vérité, des BDDs (Binary Decision Diagrams), des automates, des spécifications algébriques ou ensemblistes. Les techniques de sélection des vecteurs varient selon le type de description disponible.

Les exigences en langage naturel sont généralement données sous une forme structurée, permettant une décomposition en exigences élémentaires. On doit alors construire un ou plusieurs tests pour chaque exigence élémentaire. Dans certains cas, les exigences sont analysées pour en déduire des **classes d'équivalence** sur les valeurs des variables d'entrée. Chaque *classe d'équivalence* regroupe des valeurs d'entrée qui affectent de façon similaire les fonctionnalités du programme. On choisit alors un élément de la *classe d'équivalence* qui sera testé et on suppose que les autres éléments de la *classe* engendreraient le même comportement. Ces tests sont généralement complétés par des **tests aux limites**. Ils ont pour objectif d'exercer le système en utilisant des valeurs d'entrée qui se situent aux frontières des différentes *classes d'équivalence*.

Les descriptions formelles introduisent la possibilité d'automatiser la sélection des tests. Le problème de la sélection se posera différemment selon le type de description utilisée. Par exemple, il s'agira de sélectionner des parcours d'un automate, ou de sélectionner des instances d'un axiome d'une spécification algébrique. Différents algorithmes pourront alors être mis en œuvre, comme des algorithmes de graphes ou des algorithmes de résolution de contraintes.

Une fois le système exercé avec tous les vecteurs de test, on peut se poser la question suivante : *ai-je suffisamment testé mon système ?* La notion de **couverture** est une métrique permettant de répondre à ce type de question. Une analyse de *couverture* fournit une mesure généralement exprimée en terme de pourcentage. Par exemple, on s'intéresse au pourcentage d'exigences élémentaires testées, ou au pourcentage de transitions d'un automate traversées au cours des tests. Lorsque la **couverture fonctionnelle** souhaitée est atteinte, on décide que le système a suffisamment été testé.

3.1.2 Le test structurel

Le test structurel se base sur un modèle de la structure du système. Des exemples de modèles sont la description d'un circuit au niveau porte, ou le graphe de contrôle d'un programme. Nous verrons ultérieurement des *critères de couverture* associés à ces modèles (sections 3.2 et 3.3). Ces critères peuvent être utilisés pour déterminer des test qui, par construction, donnent la couverture souhaitée. Ils peuvent également être utilisés pour évaluer *a posteriori* la couverture structurelle de tests fonctionnels. Cette approche est adoptée pour le test de logiciels avioniques critiques. Le DO-178B/ED-12B recommande explicitement de ne pas déduire les vecteurs de test de la structure du programme.

Si l'*analyse de couverture* n'atteint pas 100%, cela signifie que certaines parties du système n'ont pas été exercées durant les tests. Cela peut provenir de deux raisons :

– les tests sont insuffisants et doivent être complétés,
– il n'existe pas de vecteurs de test permettant d'exercer les parties non couvertes.

Dans le domaine du test du logiciel, on ne peut pas toujours déterminer s'il existe des vecteurs de test permettant d'exercer un chemin d'exécution, ou même une instruction particulière dans la structure du programme. Dans le cas de logiciels avioniques critiques, la conception doit être telle que toutes les instructions soient exécutables. Le programme doit toujours terminer, et le DO-178B/ED-12B n'autorise pas la présence de *code mort*. Le *code mort* désigne l'ensemble des parties du logiciel qui ne sont pas testables. Une couverture structurelle de 100% des instructions est considérée comme un minimum.

Dans le domaine du matériel, nous allons voir dans la section suivante un critère basé sur la couverture de fautes dans la structure d'un circuit. La couverture de ce critère n'est généralement pas de 100%, car il existe des fautes non testables du fait de redondances logiques. En pratique, une couverture de 98% des fautes est typiquement considérée comme satisfaisante.

3.2 Génération automatique de vecteurs de test dans le domaine du matériel

Le test matériel a pour objectif de vérifier le bon fonctionnement d'un composant matériel. Nous distinguerons deux catégories de tests :
– les tests liés à la conception du composant,
– les tests liés à la production (en série) du composant.

La première catégorie a lieu durant la conception du composant. Nous ne détaillerons pas cette catégorie de tests. La deuxième catégorie a lieu durant la phase de production du composant.

3.2.1 Le test en production

L'imperfection des procédés de fabrication peut introduire des fautes modifiant le comportement du composant. Les tests en production ont pour objectif de détecter les composants défectueux afin d'éviter leur commercialisation. On peut noter qu'il n'existe pas d'équivalence dans le domaine du logiciel pour cette catégorie de tests.

La figure 3.1 montre un schéma de principe très simple permettant de tester un composant. Les sorties du composant sous test B sont comparées avec les sorties que délivreraient un composant A sans défaut, pour les mêmes vecteurs d'entrée. Si le comportement de B diffère de celui de la référence A, on en déduit que B est défectueux.

Les algorithmes de génération automatique de vecteurs de test ou ATPG [3] ont pour objectif de générer les entrées. Les vecteurs d'entrée générés doivent permettre de détecter des fautes répertoriées dans des *bibliothèques de fautes*. Les algorithmes

3. Automatic Test Pattern Generation

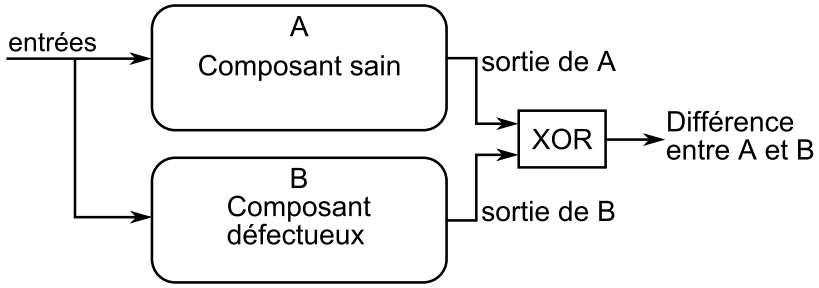


FIGURE 3.1 – Schéma de principe d'un test d'un composant

les plus évolués ont aussi pour objectif de minimiser la taille des vecteurs afin d'augmenter la productivité. Nous allons présenter le principe de fonctionnement de certains de ces algorithmes. En général, les algorithmes dépendent du type de faute que l'on veut détecter. Une des fautes typiques est le *collage de fils*[4]. On considère des collages à la source ou à la masse. Un fil collé à la source voit toujours la valeur de vérité VRAI (ou 1 logique) et un fil collé à la masse voit toujours la valeur de vérité FAUX (ou 0 logique).

On distingue deux types de circuit électrique :

les circuits combinatoires, pour lesquels les signaux de sorties dépendent uniquement des entrées au même instant.

les circuits séquentiels possédant des états internes. Les signaux de sorties dépendent des entrées et des états internes au même instant.

La figure 3.2 donne un exemple de faute unique de collage à 1 sur un circuit combinatoire.

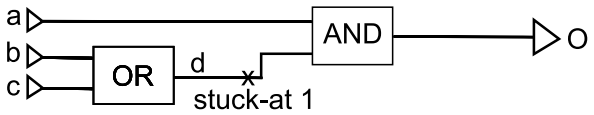


FIGURE 3.2 – Exemple de faute unique de collage à 1

Dans cet exemple, le fil noté d est collé à 1. Afin de révéler cette faute, il est nécessaire de procéder en deux étapes :

Exciter la faute en appliquant la valeur opposée au fil d : un 0 logique. Ceci peut être réalisé en affectant la valeur 0 à b et c ;

Propager la faute jusqu'à la sortie. Ceci peut être réalisé en affectant la valeur 1 à l'entrée a.

Pour des fautes de type collage de fils, on envisage les collages à 0 et à 1 sur tous les fils du composant. L'objectif est de couvrir un fort pourcentage de ces fautes (par exemple, $> 98\%$), et le nombre de vecteurs de test peut être très important.

4. en anglais *stuck-at* fault

Une première solution est la *génération aléatoire de tests* ou RTG [5]. Les vecteurs de tests sont générés aléatoirement et appliqués. Cependant, ce type de technique peut devenir coûteux en nombre de vecteurs sans parvenir à une bonne couverture. Une solution est alors d'améliorer la génération en mettant en œuvre des heuristiques de recherche, utilisant la simulation de fautes.

Il existe des algorithmes déterministes qui permettent de générer des vecteurs de tests pour révéler les fautes de collage dans les circuits combinatoires. Les algorithmes pour les circuits séquentiels peuvent alors s'appuyer sur les algorithmes précédents, le circuit étant vu comme une succession de circuits combinatoire dans le temps. Dans ce qui suit, des algorithmes déterministes utilisés pour les circuits combinatoires sont décrits. Nous allons présenter brièvement le principe de fonctionnement de ce type d'algorithme au travers d'un exemple.

3.2.2 Exemple d'algorithme ATPG

Considérons un algorithme ATPG simple possédant deux sous-fonctions :

Justifier() permettant de valuer un signal à une valeur de vérité,

Propager() permettant de propager la valuation d'un signal.

Considérons le schéma de principe de la figure 3.3 d'un composant C. Imaginons que l'on souhaite détecter un collage à 0 du signal g.

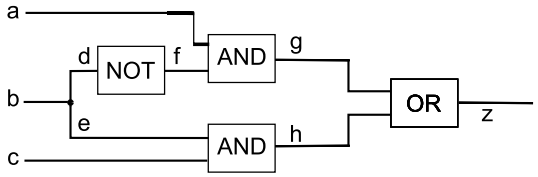


FIGURE 3.3 – Schéma de principe du composant C

La première chose que l'algorithme fait est d'exciter la faute en valuant le signal g à 1. Pour cela l'algorithme utilise la fonction $Justifier(C, g, 1)$. La fonction est appelée récursivement jusqu'aux entrées du composant de la manière suivante : $Justifier(C, a, 1)$, $Justifier(C, f, 1)$, $Justifier(C, d, 0)$, $Justifier(C, b, 0)$.

Le vecteur $abc = 10X$ permet d'exciter la faute. L'algorithme va maintenant propager la faute jusqu'à la sortie afin de la rendre observable. L'algorithme appelle la fonction $Propage(C, g)$ qui appelle $Justifier(C, h, 0)$ qui appelle à son tour $Justifier(C, c, 0)$. Produisant le vecteur de test $abc = 100$.

Maintenant, on souhaite générer un test qui permette de détecter un collage à 1 de z. La fonction $Justifier(C, z, 0)$ va respectivement appeler $Justifier(C, g, 0)$ et $Justifier(C, h, 0)$. Pour justifier $g = 0$, la fonction doit prendre une **décision** entre justifier $a = 0$ ou $f = 0$. Supposons que la fonction décide de choisir f, cela finira par justifier $b = 1$ à la fin de la récursivité. Ensuite, pour justifier $h = 0$, si l'algorithme décide de justifier $e = 0$ cela créera un **conflit** au niveau de la justification de b. Pour

5. Random Test Generation

éviter ce conflit, l'algorithme reviendra sur sa décision précédente et choisira $c = 0$ pour justifier $h = 0$. Dans ces cas, on dit que la fonction procède à un **backtracking**. Parfois il peut arriver qu'une valeur soit injustifiable ou non propageable, on parle alors de faute non testable.

[Wang 2006] et [Bushnell 2000] présentent trois algorithmes ATPG pour des fautes de collage : *D-Algorithm*, *PODEM* et *FAN*. Le premier ne possède pas d'intelligence particulière. Le nombre de décisions qu'il doit prendre est égal au nombre d'opérateurs présents. Les algorithmes suivants sont des améliorations dans lesquelles on cherche à minimiser le nombre de *points de décision* et de *backtracks*. Le second algorithme réduit le nombre de décisions au nombre d'entrées. Le troisième est capable de traiter plusieurs objectifs de test en une seule itération.

3.2.3 Notre positionnement

L'analyse de contre-exemples que nous voulons mener va nécessiter un parcours structurel du modèle SCADE à l'instar de ce qui est fait dans les travaux présentés ici. La fonction de justification, qui part de la faute pour remonter aux entrées du circuit, présente une problématique proche de la nôtre (partir de la violation de propriété pour remonter aux entrées du contre-exemple). Cependant, le problème que nous considérons est différent et le contexte plus restreint : notre point de départ est un contre-exemple, nous possédons donc les valuations de toutes les variables, nous cherchons à identifier parmi toutes les variables celles qui ont un réel impact sur la valeur de la sortie.

Les algorithmes évoqués ci-dessus rencontrent des problèmes pour des circuits contenant des *chemins reconvergents*[6]. Cette notion caractérise deux chemins différents qui divergent au niveau d'un même signal et qui reconvergent au niveau d'un même composant. La présence de tels chemins dans la structure complexifie de manière importante l'analyse de la structure [Abramovici 1990]. De fait, le problème de la génération de test pour les circuits combinatoires est connu comme étant NP complet. Nous verrons dans le chapitre 4 que nous n'échapperons pas à la difficulté des chemins reconvergents.

Nous utiliserons également dans le chapitre 5.2 la notion de *niveau d'un arc*, proposée dans [Abramovici 1990], qui permet de caractériser la distance d'un arc par rapport aux entrées du composant.

3.3 Critères de couverture structurelle dans le domaine du logiciel

3.3.1 Couverture structurelle de programmes impératifs

Nous allons maintenant présenter des critères de couverture structurelle dans le domaine du logiciel. Nous commencerons par introduire les critères classiquement

6. ou *reconvergent fanout*

utilisés pour les programmes écrits dans un langage impératif. Nous remarquerons ensuite que le principe de couverture structurelle peut être appliqué à différents types de langage. Ceci nous amènera aux travaux portant sur le langage Lustre, sur lequel SCADE est basé.

Il existe plusieurs critères de couverture pour les programmes écrits dans un langage impératif. Certains sont basés sur le *flot de contrôle*, d'autres sur le *flot des données*.

Le **graphe de contrôle** représente la structure de contrôle d'un programme impératif, c'est à dire la manière dont sont ordonnées les instructions du programme. Des exemples de critères basés sur ce graphe sont la couverture des *instructions*, des *branches*, ou des *chemins complets* d'exécution.

Le graphe de contrôle peut être enrichi avec des informations sur le **flot de données**. On décrit la dépendance entre les données, leurs définitions (la donnée reçoit une valeur) et leurs utilisations (utilisation de la valeur de la donnée dans un calcul ou un prédicat de branche). Un exemple de critère associé au flot de données est la couverture de *toutes les utilisations*, qui requiert d'activer au moins un chemin entre chaque définition et toutes ses utilisations ultérieures.

Dans le domaine avionique, les critères utilisés sont des critères basés sur le flot de contrôle. Ils servent à mesurer la couverture structurelle fournie par des tests *fonctionnels*. La norme DO-178B/ED-12B impose, en fonction du niveau de criticité (DAL) (introduite en section 1.2) du logiciel, d'utiliser les critères suivants :

niveau de DAL	Critère
A	100% MC/DC
B	100% DC
C	100% SC

Le critère SC (Statement Coverage) est la couverture des instructions. Pour expliquer les autres critères [Hayhurst 2001, Chilenski 1994], il est nécessaire de définir les notions de *condition* et de *décision*. Une **condition** est une expression booléenne élémentaire qui ne contient pas d'opérateur booléens. Une **décision** est une expression booléenne composée de *conditions* et éventuellement d'opérateurs booléens. Par exemple, un prédicat de branche est une *décision*, et une instruction de la forme : $x = a$ OR b contient une *décision*. Si une *condition* apparaît plus d'une fois dans une *décision*, chaque occurrence est une *condition* distincte. Différents critères de couverture des expressions booléennes peuvent alors être définis, dont les suivants :

Decision coverage (DC) Tous les points d'entrée et de sortie du programme ont été utilisés au moins une fois, et toutes les *décisions* ont pris toutes les valeurs possibles au moins une fois.

Multiple-condition coverage (M-CC) DC est couvert, et pour chaque *décision*, toutes les combinaisons des *conditions* ont pris toutes les valeurs possibles au moins une fois. Ce critère est théoriquement très intéressant mais est impraticable du fait du nombre important de tests à réaliser. Pour une *décision* à n *conditions*, 2^n tests sont nécessaires.

Modified condition/decision coverage (MC/DC) Le critère MC/DC apporte une alternative coûteuse par rapport à DC, mais plus réaliste que M-CC. DC est couvert, et pour chaque *décision*, toutes les *conditions* ont pris toutes les valeurs possibles au moins une fois. De plus, chaque *condition* doit affecter indépendamment le résultat de la *décision*. Une condition *affecte indépendamment le résultat de la décision* si une variation de cette *condition*, sans faire varier les autres *conditions*, fait varier le résultat de la *décision*.

Le *critère de couverture* MC/DC, exigé pour les logiciels les plus critiques dans l'avionique, a été développé dans le but de conserver la plupart des avantages de M-CC tout en imposant une croissance linéaire sur le nombre de tests nécessaires.

Il existe une **relation d'inclusion** qui permet de comparer les *critères de couverture*. Un critère A inclut un critère B si et seulement si les *vecteurs de test* qui satisfont A satisfont aussi B. Par exemple, les critères tous-les-chemins et MC/DC sont incomparables selon la relation d'inclusion. Par contre, MC/DC inclut DC, qui lui-même inclut la couverture de toutes les instructions.

3.3.2 A quel niveau de langage mesurer la couverture ?

La couverture structurelle est classiquement mesurée au niveau d'un code source impératif (par exemple, un code C). Mais on peut envisager d'appliquer le principe de la mesure de couverture pour des langages de niveau différent. Considérons une approche basée sur des modèles, où une description dans un langage de haut niveau subit une chaîne de transformations avant de produire un code objet exécutable. Par exemple, un modèle Lustre peut être vu comme le *code source* d'un compilateur KCG qui produira un programme en langage C. Ce dernier sera à son tour compilé pour produire du code exécutable. L'analyse de couverture peut être envisagée à ces différents niveaux : Lustre, C, ou code objet.

Lorsque l'analyse est réalisée à un haut niveau, on peut se demander quelle est la couverture obtenue sur le code qui est réellement exécuté. Selon [Beizer 1990], un test qui couvre 100% des instructions C pourrait couvrir 75% des instructions (ou critère équivalent) ou moins au niveau du *code objet*. Ce chiffre dépend du compilateur utilisé.

3.3.3 Couverture structurelle de modèles Lustre

Les critères de couverture ont été largement étudiés dans le cadre des langages de programmation impératifs. Cependant, avec l'arrivée des processus de développement dirigés par les modèles et de la volonté de tester ces modèles, il existe aujourd'hui une volonté de mesurer la couverture au niveau des modèles. Les critères définis pour les langages impératifs ne sont pas directement transposables aux modèles Lustre. Il existe donc un besoin pour de nouveaux critères permettant de mesurer la *couverture structurelle* de test de modèle. Les travaux de [Lakehal 2009, Lakehal 2007, Lakehal 2005] apportent de nouvelles définitions de la *couverture structurelle* pour le langage Lustre.

Nous allons présenter des critères de *couverture structurelle* dédiés au langage Lustre. Ces critères se basent sur la notion de *chemins* activés par les vecteurs de test.

Les modèles Lustre peuvent se représenter sous forme d'un *réseau d'opérateurs* noté $\mathcal{N}$. Un *réseau d'opérateurs* est un graphe dirigé composé de N opérateurs et d'un ensemble $A \subseteq N \times N$ d'arcs dirigés connectant les opérateurs entre eux. La sortie d'un opérateur est un flot de données qui dépend de la sémantique de l'opérateur et de ses flots d'entrée.

La notion de chemin est définie pour les opérateurs Lustre booléens et temporels : AND, OR, NOT, PRE, ->, et ITE, ainsi que pour les opérateurs relationnels. Un *chemin* est une suite d'arcs. Un opérateur est défini par un ensemble de couples $< e_i, s >$ où e_i est la $i^{\text{ème}}$ entrée de l'opérateur et s est la sortie de l'opérateur. A chaque paire $< e_i, s >$ est associé un prédicat définissant la *condition d'activation* du chemin formé par l'entrée e_i et la sortie s.

Il y a autant d'arcs d'entrée que de variables d'entrée du modèle, et autant d'arcs de sortie que de variables de sortie. Les arcs internes sortent d'un unique opérateur et entrent dans un ou plusieurs opérateurs. L'arc α_2 est un successeur de l'arc α_1 s'il existe un opérateur *op* du graphe tel que e_1 entre dans *op* et e_2 en sort.

3.3.3.1 La notion de chemin

Le *réseau d'opérateurs* définit des chemins dans le modèle Lustre. Un chemin est une séquence finie d'arcs $< e_0, e_1, \ldots, e_n >$ telle que $\forall i \in [1, n-1], < e_i, e_{+1} > \in \mathcal{N}$ c'est-à-dire que e_{i+1} est le *successeur* de e_i.

La *longueur d'un chemin* est définie par le nombre d'arcs qui le forment. Un *chemin complet* est un chemin dont le premier arc est un arc d'entrée et le dernier arc est un arc de sortie. Un *n-chemin* p_n est un chemin dont la longueur est n. On appellera *chemin unitaire* un chemin de longueur 2.

3.3.3.2 Condition d'activation

La *condition d'activation* est la condition pour laquelle un flot de donnée est transféré depuis le premier arc d'un chemin vers le dernier arc d'un chemin.

Soit $\mathcal{N}$ un *réseau d'opérateur*, et soit P l'ensemble de tous les chemins de $\mathcal{N}$. Soit E l'ensemble des expressions Lustre. La *condition d'activation* est une fonction $\mathcal{AC} : P \rightarrow E$. L'évaluation de $\mathcal{AC}$ pour un chemin $p \in P$ dépend de la sémantique des opérateurs traversés par p et de leur ordre. La *condition d'activation* d'un *chemin* est récursivement définie comme suit.

Soit $\mathcal{N}$ un réseau d'opérateur, et soit $p_n =< e_0, \ldots, e_n >$ un *n-chemin* de P_n, la *condition d'activation* de p, notée $\mathcal{AC}(p_n)$, est :

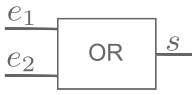
$$\mathcal{AC}(p_n) = \begin{cases} \text{VRAI} & \text{si } n = 1 (\text{cas d'arrêt}) \\ \text{FAUX} -> pre(AC(p_{n-1})) & \text{si } e_n = pre(e_{n-1}) \\ AC(p_{n-1}) \wedge OC(e_{n-1}, e_n) & \text{sinon} \end{cases}$$

Où :

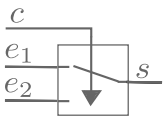
- si $s = NOT(e_1)$ alors :
  - $\mathcal{OC}(e_1, s)$ =VRAI

- si $s = AND(e_1, e_2)$ alors :
  - $\mathcal{OC}(e_1, s) = \neg(e_1) \vee e_2$
  - $\mathcal{OC}(e_2, s) = \neg(e_2) \vee e_1$

- si $s = OR(e_1, e_2)$ alors :
  - $\mathcal{OC}(e_1, s) = e_1 \vee \neg(e_2)$
  - $\mathcal{OC}(e_2, s) = e_2 \vee \neg(e_1)$



- si s=IF c THEN e_2 ELSE e_1 alors :
  - $\mathcal{OC}(c, s)$ =VRAI
  - $\mathcal{OC}(e_2, s) = c$
  - $\mathcal{OC}(e_1, s) = \neg c$



La condition d'activation de l'opérateur NOT est toujours VRAI car quelle que soit la valuation de son arc d'entrée, l'information est propagée de l'entrée vers la sortie de l'opérateur. Il est de même pour tous les opérateurs relationnels.

Les conditions d'activation associées à l'opérateur AND (respectivement OR) sont symétriques.

3.3.3.3 Exemple

Cet exemple a pour but d'illustrer la notion de *condition d'activation*. Considérons l'équation Lustre suivante :

O=NOT(A AND (B OR C))

La figure 3.4 représente le réseau d'opérateurs modélisant l'équation Lustre.

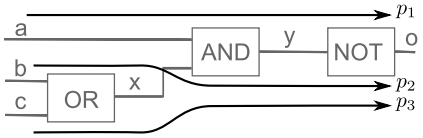


FIGURE 3.4 – Réseau d'opérateurs

Le réseau d'opérateurs est composé de trois opérateurs AND, OR et NOT. Il définit trois chemins p_1, p_2 et p_3 respectivement formés des suites d'arcs $< a, y, o >$, $< b, x, y, o >$, $< c, x, y, o >$. Le tableau 3.1 indique les conditions d'activation de p_1, p_2, et p_3.

Chemins	Condition d'activation
p_1	$\neg a \vee x$
p_2	$(\neg x \vee a) \wedge (b \vee \neg c)$
p_3	$(\neg x \vee a) \wedge (c \vee \neg b)$

TABLE 3.1 – Conditions d'activation des chemins

Le détail du calcul de la condition d'activation de $p_1 =< a, y, o >$ est le suivant :

$$\mathcal{AC}(p_1) = \underbrace{\text{VRAI}}_{\mathcal{AC}(<a>)} \wedge \underbrace{(\neg a \vee x)}_{\mathcal{OC}(a,y)} \wedge \underbrace{\text{VRAI}}_{\mathcal{OC}(y,o)}$$

Le détail du calcul de la condition d'activation de $p_2 =< b, x, y, o >$ est le suivant :

$$\mathcal{AC}(p_2) = \underbrace{\text{VRAI}}_{\mathcal{AC}(<b>)} \wedge \underbrace{(b \vee \neg c)}_{\mathcal{OC}(b,x)} \wedge \underbrace{(\neg x \vee a)}_{\mathcal{OC}(x,y)} \wedge \underbrace{\text{VRAI}}_{\mathcal{OC}(y,o)}$$

Le calcul de p_3 est identique à celui de p_2. L'unique différence est $\mathcal{OC}(b, x)$ est remplacé par $\mathcal{OC}(c, x) = (c \vee \neg b)$ dans le cas de p_3 :

$$\mathcal{AC}(p_3) = \underbrace{\text{VRAI}}_{\mathcal{AC}(<c>)} \wedge \underbrace{(c \vee \neg b)}_{\mathcal{OC}(c,x)} \wedge \underbrace{(\neg x \vee a)}_{\mathcal{OC}(x,y)} \wedge \underbrace{\text{VRAI}}_{\mathcal{OC}(y,o)}$$

3.3.3.4 Critères de couverture

Trois critères de *couverture structurelle* sont définis autour de la notion de condition d'activation de chemins :

- le critère de base,
- les critères de couverture des conditions élémentaires,
- les critères de couverture des conditions multiples.

Le **critère de base** dépend uniquement des conditions d'activation des *n-chemins*. On dira qu'un ensemble de *vecteurs de test* satisfait 100% du *critère de base* si tous les *chemins complets* de longueur inférieure ou égale à K du réseau d'opérateurs ont été activés durant la simulation des séquences. Où K est un paramètre du critère défini par l'utilisateur.

Dans le cadre de l'exemple de la figure 3.4, le vecteur :

cycle	1
a	VRAI
b	VRAI
c	VRAI

satisfait le critère de base pour $K = 4$.

Le but de ce critère est de s'assurer que toutes les dépendances entre les entrées et les sorties ont été exercées au moins une fois. Ce critère ne prend pas en compte la valuation de quelque variable que ce soit. Par exemple, le vecteur :

cycle	1
a	FAUX
b	FAUX
c	FAUX

satisfait aussi le *critère de base* pour $K = 4$.

Le **critère de couverture des conditions élémentaires** a été défini afin de prendre en compte les valuations des variables d'entrées. On dira qu'un ensemble de vecteurs de test satisfait 100% du *critère de couverture des conditions élémentaires* si tous les *chemins* p de longueur inférieure ou égale à n du *réseau d'opérateurs* $\mathcal{N}$ ont été activés tel que :

$$in(p) \wedge \mathcal{AC}(p) = \text{VRAI}$$
$$\neg in(p) \wedge \mathcal{AC}(p) = \text{VRAI}$$

Où $in(p)$ désigne la variable d'entrée (Booléenne) par laquelle le chemin p commence.

Dans le cadre de l'exemple 3.3.3.3, la séquence de vecteurs :

cycle	1	2
a	VRAI	FAUX
b	VRAI	FAUX
c	VRAI	FAUX

satisfait le **critère de couverture des conditions élémentaires** pour $K = 4$.

Le **critère de couverture des conditions multiples** a été défini afin de prendre en compte les valuations de toutes les variables. On dira qu'un ensemble de vecteurs de test satisfait 100% du *critère de couverture des conditions multiples* si tous les *chemins* p de longueur inférieure ou égale à n du *réseau d'opérateurs* $\mathcal{N}$ ont été activés tel que :

$$e_i \wedge \mathcal{AC}(p) = \text{VRAI}$$
$$\neg e_i \wedge \mathcal{AC}(p) = \text{VRAI}$$

Où e_i désigne le $i^{\text{ème}}$ $(i <= longueur(p))$ arc composant le chemin p.

Dans le cadre de l'exemple 3.3.3.3, la séquence de vecteurs :

cycle	1	2
a	VRAI	FAUX
b	VRAI	FAUX
c	VRAI	FAUX

satisfait le **critère de couverture des conditions multiples** pour $K = 4$.

3.3.4 Notre positionnement

Les critères de couverture définis pour Lustre ne nous sont pas directement utiles pour l'analyse de contre-exemples. En revanche, la notion de *condition d'activation* caractérise les conditions pour lesquelles la valeur d'une entrée d'un opérateur a une influence sur la valeur de la sortie de cet opérateur. Cette notion nous sera utile dans le chapitre 4 pour définir le concept de *cause* d'un contre-exemple donné.

3.4 Génération automatique de tests pour les modèles Lustre

Il existe plusieurs outils de génération automatique de test pour les modèles Lustre. On peut citer par exemple Lutess[du Bousquet 1999], Lurette[Raymond 1998] et GATeL[Marre 2000]. Les deux premiers étant des outils de test fonctionnel, nous détaillerons seulement l'outil GATeL qui permet également de générer des tests structurels. Lurette est brièvement décrit dans la section 3.5 car il est intégré dans un environnement de déboguage.

A partir d'un modèle et d'un objectif de test, qui peut être une propriété ou un prédicat de chemin, GATeL génère automatiquement une séquence d'entrées amenant le modèle dans l'état satisfaisant l'objectif. GATeL peut également générer plusieurs séquences d'entrées assurant une couverture structurelle du modèle. Cette fonctionnalité nécessite cependant une interaction avec l'utilisateur pour sélectionner les cas qu'il souhaite couvrir. Le fonctionnement de GATeL est basé sur la résolution d'un système de contraintes construit à partir du modèle et de l'objectif. La résolution consiste à construire un passé respectant les contraintes. Elle combine :

- la propagation / simplification des contraintes et,
- l'instanciation partielle des variables.

Le premier mécanisme vérifie que les contraintes sont toujours respectées alors que le second fixe les valeurs de certaines variables. L'instanciation d'une variable relance le mécanisme de propagation et de vérification de satisfiabilité des contraintes.

3.4.1 Notre positionnement

La génération automatique de test n'est pas le sujet de cette thèse et GATeL n'est pas directement utile pour l'analyse de contre-exemples. En revanche, les stratégies de GATeL, appliquées à des prédicats de chemins, pourraient guider la génération de contre-exemples multiples.

3.5 Déboguage de programmes Lustre

La volonté de comprendre les contre-exemples nous amène à étudier les travaux sur le déboguage de programmes.

Un débogueur est un outil qui apporte un support dans la recherche de fautes dans un programme informatique. Ce type d'outil est généralement utilisé pour

observer (pas à pas) le comportement d'un programme informatique. Nous nous intéressons ici au cas des modèles SCADE. Le formalisme SCADE est basé sur la sémantique de Lustre. Contrairement à un langage impératif, les équations Lustre sont écrites de manière déclarative : sans ordre établi. Alors qu'il est possible de parcourir séquentiellement le code d'un langage impératif, il est impossible d'en faire autant pour le code d'un langage déclaratif. Par exemple, la notion de *points d'arrêts* telle que définie pour les langages impératifs n'est pas directement transposable.

L'approche synchrone des langages déclaratifs permet une simulation étape par étape. Les simulateurs, tels que *Luciole* pour Lustre ou *Scade Simulator* pour Scade, tirent partie de ce phénomène. L'utilisateur peut simuler le modèle en modifiant les entrées et en visualisant la valuation des variables à chaque cycle de l'horloge de base.

3.5.1 Ludic : *fonctionnalités de base*

Des travaux du laboratoire Verimag [Maraninchi 2000, Gaucher 2003] ont pour but d'apporter un support à l'utilisateur dans le déboguage de modèles Lustre. Plusieurs fonctionnalités de simulation et de visualisation sont implémentées dans un outil appelé Ludic. La fonctionnalité de base est la **simulation**. Cette dernière peut être dirigée, soit par un utilisateur, soit au travers d'un fichier. Dans le premier cas, l'utilisateur spécifie, à chaque cycle de l'horloge de base, la valuation des entrées du modèle. Dans le deuxième cas, les valuations des entrées sont spécifiées dans le fichier.

Durant la *simulation*, l'utilisateur peut **visualiser sur un chronogramme** la valuation des variables qu'il souhaite.

A cela s'ajoute une fonctionnalité permettant de définir des **points d'arrêts**. Dans le cas de Ludic, un point d'arrêt n'est pas caractérisé par une ligne de code mais par un état. L'état en question est défini au travers d'un observateur (tel que défini dans la section 1.5.2.5). Lorsque la sortie de l'observateur est valuée à FAUX, la simulation est stoppée.

Ludic propose à l'utilisateur de **marquer des états** d'un label afin de pouvoir y revenir ultérieurement. Les états *labélisés* apparaissent dans un graphe d'état réduit.

L'outil **mémorise les états** du modèle parcourus durant la simulation. Ceci permet à tout moment de la simulation de revenir en arrière, par **simulation inversée**.

3.5.2 Ludic : *atteignabilité d'états*

Ludic propose une fonctionnalité d'atteignabilité automatique d'un état. Cela permet à l'utilisateur, à n'importe quel moment de la simulation (que l'on appellera $etat_i$), de demander à l'outil d'atteindre un état (que l'on appellera $etat_f$). Si $etat_f$ est atteignable depuis $etat_i$, Ludic conduit automatiquement la simulation vers l'$etat_f$. S'il existe plusieurs chemins allant de $etat_i$ à $etat_f$, Ludic tente de choisir le plus court. Cette fonctionnalité nécessite le couplage de Ludic avec deux outils :

– un outil de vérification : Nbac,
– et un outil de génération automatique de séquences de test : Lurette.

Dans un premier temps, nous allons décrire ces deux outils. Puis, dans un deuxième temps, nous expliquerons les fonctionnalités obtenues en les connectant.

Nbac [Jeannet 1999, Jeannet 2003] est un outil de vérification permettant de décider des problèmes d'atteignabilité. Il est capable de décider s'il existe (au moins) un chemin entre deux états. Cet outil prend en entrée :
– une spécification du modèle dans le format de Nbac,
– la spécification de l'état de départ,
– la spécification de l'état d'arrivée.

Nbac se base sur les techniques de l'interprétation abstraite. L'outil construit un automate abstrait qu'il va analyser et raffiner. Deux réponses sont possibles, la première est : *il n'existe pas de chemin allant de l'état de départ vers l'état d'arrivée.* $etat_f$ est inatteignable et a été supprimé de l'automate abstrait lors d'une étape de raffinement. La seconde est : *il peux exister (au moins) un chemin.* L'outil renvoie l'automate abstrait représentant l'espace des états atteignables depuis l'état de départ. Dans ce cas, on ne sait pas si $etat_f$ est atteignable ou pas. Pour conclure, il est nécessaire d'analyser l'automate abstrait afin de tenter d'en extraire un contre-exemple concret.

Lurette [Raymond 1998] est un outil de génération automatique de séquences de test pour les modèles Lustre. Il prend en entrée :
– un modèle Lustre,
– un ensemble de *contraintes*.

Les *contraintes* sont exprimées sous formes d'observateurs (tels que décrit dans la section 1.5.2.5) ou d'expressions régulières. Ces dernières sont très utiles pour décrire des séquences d'états. Lurette est alors capable de générer des scénarios respectant ces *contraintes*. Lors de cette génération, le modèle est simulé car les *contraintes* peuvent dépendre des sorties du modèle aux instants passés. L'outil peut être amené à faire des choix de valuations aléatoires sur un domaine de valeurs respectant les contraintes.

Lurette possède aussi une fonctionnalité d'*oracle*. L'utilisateur peut spécifier un observateur décrivant le comportement correct du modèle. L'*oracle* est alors interprété en parallèle du modèle Lustre et de l'observateur d'environnement.

La figure 3.5 illustre la manière dont les outils sont connectés pour réaliser la fonctionnalité d'*atteignabilité d'états*.

Ludic prends en entrée un modèle Lustre et réalise automatiquement la transformation vers le formalisme d'entrée de Nbac. L'état de départ ($etat_i$) est l'état courant de la simulation. L'état d'arrivée ($etat_f$) est celui spécifié par l'utilisateur. Nbac tente de trouver le chemin le plus court entre $etat_i$ et $etat_f$. Si Nbac démontre qu'il n'en n'existe pas, cela signifie que $etat_f$ est inatteignable depuis $etat_i$. L'utilisateur poursuit la simulation depuis l'état courant $etat_i$. En revanche, si Nbac ne démontre pas qu'il n'en n'existe pas, il renvoie un automate abstrait pouvant contenir, ou pas, un chemin. Cet automate est exploité par Lurette qui va tenter de

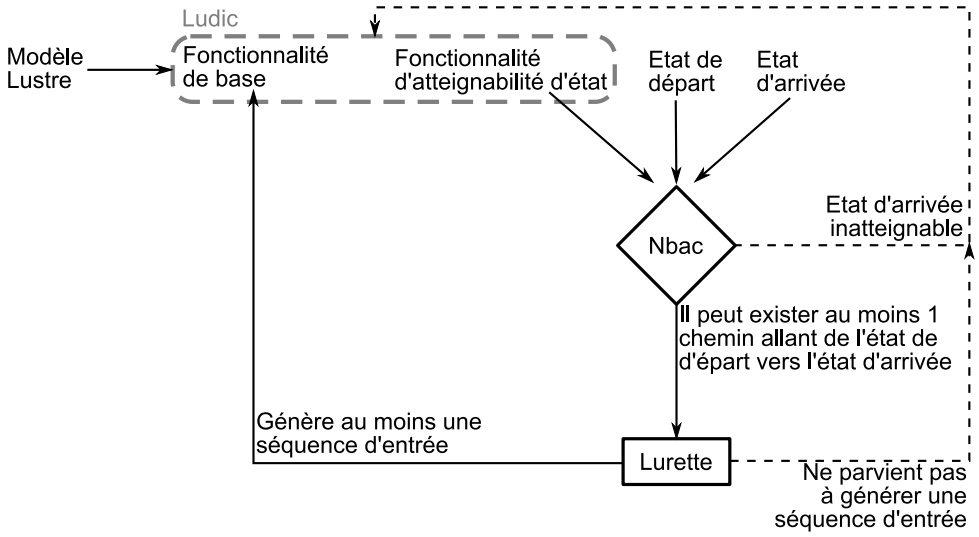


FIGURE 3.5 – Interaction de Ludic, Nbac et Lurette

générer au moins un chemin concret. S'il n'y parvient pas, cela peut soit signifier qu'il n'en n'existe pas, soit que Lurette a échoué dans sa tentative. En revanche, si Lurette génère un chemin, la séquence des valuations des entrées du modèle concret est simulée par Ludic afin d'arriver dans l'état choisi : $etat_f$.

3.5.3 Ludic : *diagnostic*

Ludic propose un algorithme permettant d'inspecter le comportement du modèle pour une simulation donnée. L'algorithme explore le graphe de dépendances en exploitant la définition des variables. Il parcourt les expressions, en interagissant avec l'oracle qui, dans le cas de Ludic, n'est autre que l'utilisateur. L'interaction se fait au travers de questions telles que : *Êtes-vous d'accord avec cette expression ?* Si l'oracle juge l'expression *exp* erronée, alors l'analyse s'arrête. Dans le cas contraire, l'analyse se poursuit sur les termes de *exp*. Dans ce cas, l'interaction se fait au travers de questions telles que : *Êtes-vous d'accord avec cette valuation ?* L'algorithme évolue en s'appelant récursivement selon 5 cas :

- l'analyse porte sur une expression de la forme : $exp = f(e_1, \ldots, e_n)$, l'oracle est sollicité afin de savoir si *exp* est correctement écrit. Si la réponse est *non*, alors l'analyse se termine et l'utilisateur peut corriger l'expression incorrecte. Si la réponse est *oui*, l'oracle est de nouveau sollicité afin de juger de la bonne valuation des termes de l'expression. L'analyse se poursuit récursivement sur les termes n'ayant pas la bonne valuation.
- l'analyse porte sur une condition ITE[7]. L'oracle est sollicité afin de savoir si la *condition* de l'opérateur ITE est correcte. Si la réponse est *non*, alors l'analyse se termine et l'utilisateur peut corriger la *condition* incorrecte. Si la

7. IF THEN ELSE

réponse est *oui*, l'analyse se poursuit récursivement sur l'entrée sélectionnée par la *condition* au moment de l'analyse.
Ce raisonnement est identique pour un opérateur d'initialisation Lustre -> (décrit dans la section 1.4.1). Ce dernier peut se modéliser sous la forme : IF clock = 1 THEN e_1 ELSE e_2.
- L'expression en cours d'analyse est le résultat d'une fonction [8], l'analyse se poursuit dans le corps de la fonction.
- L'expression est un opérateur PRE (décrit dans la section 1.4.1). Dans ce cas, l'analyse se poursuit à l'instant précédent.
- L'expression est une sortie ou une variable locale. Dans ce cas, l'analyse se poursuit sur la définition de la variable sélectionnée. Si l'expression est une variable d'entrée, l'algorithme questionne l'oracle sur la validité de la valuation de la variable d'entrée.

L'analyse est cependant limitée par la taille de l'historique (décrit dans les fonctionnalités de base) des états parcourus durant la simulation.

3.5.4 Notre positionnement

Ces travaux ont pour objectif de fournir un support à l'utilisateur dans une phase de simulation d'un modèle Lustre.

Nos travaux ont pour objectif de fournir un support à l'utilisateur afin de :
- comprendre un contre-exemple fourni par un model checker,
- identifier les *causes* de ce contre-exemple,

dans le cadre d'un modèle SCADE.

Une différence importante est que Ludic a été développé pour répondre à un besoin de simulation. Dans ce contexte, l'utilisateur possède un modèle mental du scénario en cours d'application. Dans le cas d'un *contre-exemple* renvoyé par un model checker, l'utilisateur ne sait pas ce qui se passe. S'il part sur une analyse pas-à-pas, la quantité d'information à traiter peut être très importante. L'utilisateur aura du mal jouer le rôle d'oracle, pour déterminer quels sont les termes erronés.

Notre proposition est d'aider l'utilisateur à construire son modèle mental du contre-exemple. Plutôt que d'interagir avec lui sur les détails de l'exécution d'un scénario, on cherche à lui en donner une vue synthétique. Le principe est de lui indiquer quelles variables d'entrée à quels instants ont joué un rôle dans la violation de propriété, et de lui montrer quels chemins structurels ont été actifs. Ces informations sont calculées de manière non-interactive, en se basant sur la sémantique des opérateurs qui composent le modèle.

3.6 Conclusion

Ce chapitre a permis d'offrir un état de l'art sur les thématiques du diagnostic et de l'analyse structurelle. Le chapitre retrace brièvement la problématique du test

8. d'un noeud Lustre plus précisément

qui est liée à ces deux notions.

Dans un premier temps, le test dans un contexte matériel est abordé. Ce travail nous a permis d'identifier le type de problème que l'on peut rencontrer dans l'analyse de la structure d'un système, et nous a également fourni des notions que nous adapterons dans l'approche proposée dans les chapitres 4 et 5. Nous retiendrons particulièrement les problèmes liés aux *chemins reconvergents* et la notion de *niveau d'arc*.

Dans un deuxième temps, ce chapitre a illustré le test dans le domaine du logiciel en insistant sur la notion de couverture structurelle. Les critères les plus utilisés dans le monde de l'aéronautique ont été abordés. Des critères plus récents, s'appliquant aux modèles Lustre ont été illustrés. Nous retiendrons de ces derniers la notion de *condition d'activation*. Cette dernière sera réutilisée dans le chapitre 4.

Dans un troisième temps, le chapitre a présenté des outils de génération automatique de tests de modèles Lustre. On pourrait s'inspirer des algorithmes mis en oeuvre par ce type d'outils pour définir des stratégies de génération de contre-exemples différents. Nous laissons cette approche en perspective de nos travaux.

Dans un quatrième temps, le chapitre a illustré des travaux sur le diagnostic de fautes des modèles Lustre. La problématique, liée au diagnostic, abordée par ces travaux est proche de nos préoccupations. Cependant, le contexte et les motivations sont différentes. Dans le cas de Ludic, le contexte de la simulation sous-entend que l'utilisateur possède un modèle mental du scénario qu'il analyse. Dans le cadre du model checking, l'utilisateur ne sait absolument pas ce qu'il se passe dans un contre-exemple. Une analyse de ce dernier sera nécessaire afin de comprendre le scénario produit. Au niveau du diagnostic de faute, nous souhaitons avoir une approche totalement autonome qui ne sollicite pas l'utilisateur. Au delà, du diagnostic, nous souhaitons générer des contre-exemples différents sans solliciter l'utilisateur.

Chapitre 4

Formalisation du problème

Sommaire

Les chapitres précédents ont montré qu'un model checker peut nous fournir un contre-exemple sous la forme d'une suite de valuations des variables d'entrée du modèle en cours de vérification. Notre objectif est d'apporter un moyen d'assister l'analyse de ce contre-exemple. Il s'agit d'extraire automatiquement des informations pertinentes pour d'une part comprendre la violation de propriété observée, et d'autre part permettre la génération de contre-exemples différents.

Pour cela, notre proposition est de caractériser un contre-exemple par les parties du modèle SCADE qu'il active. Ainsi, ce chapitre définit la notion de *cause* d'un contre-exemple en se référant à des ensembles de chemins actifs dans la structure du modèle. Nous montrons que cette notion de cause répond bien aux objectifs énoncés ci-dessus.

Les définitions sont graduellement introduites et illustrées sur un petit exemple, une bascule RS à reset prioritaire. La section 4.1 formalise la notion de modèle

SCADE. Des formalisations existent déjà dans la littérature [Liu 2005], mais nous avons préféré définir un cadre plus réduit qui cible nos besoins. La section 4.2 introduit la notion de scénarios et d'exécution d'un modèle avec un scénario. La section 4.3 introduit la notion de *chemin* qui nous servira à définir la notion de *cause* dans la section 4.4. Enfin, la section 4.5 revient sur les choix de formalisation retenus, en discutant leurs limites et leurs extensions possibles.

4.1 Modèle

Dans ce chapitre, les modèles SCADE sont considérés comme des graphes connectant des *opérateurs*.

Définition 1 (Modèle) *Un modèle est composé d'**opérateurs**, d'**arcs** et de **variables**. Dans la suite, lorsqu'on parlera de modèle, on fera référence à l'ensemble {modèle à vérifier, observateur des hypothèses, observateur de la propriété}. Un modèle a une variable de sortie unique, notée O, qui est la sortie de l'observateur de la propriété.*

Pour simplifier le problème, nous limitons l'étude aux modèles possédant une horloge unique et des variables booléennes. Nous reviendrons sur ces limitations lors d'une discussion des choix de formalisation, dans la section 4.5. Dans ce qui suit, nous introduisons successivement chaque élément du modèle : opérateurs, arcs et variables.

4.1.1 Opérateurs

Définition 2 (Opérateur) *Un opérateur est composé d'une ou plusieurs entrées et d'une ou plusieurs sorties. On distingue quatre types d'opérateurs élémentaires :*

- *les opérateurs d'**interface** : opérateur d'entrée noté IN, opérateur de sortie noté OUT, constantes $faux$ et $vrai$,*
- *les opérateurs **booléens** : NOT(E), AND(E_1,E_2), OR(E_1,E_2),*
- *l'opérateur **temporel** : FBY(E,1,Init)[1],*
- *l'opérateur **conditionnel** : ITE(C,E_1,E_2)[2].*

Ces opérateurs SCADE sont analogues aux opérateurs considérés dans les travaux [Lakehal 2005, Lakehal 2007, Lakehal 2009] sur le test structurel des modèles Lustre. Il n'y a cependant pas de correspondance un à un pour les opérateurs temporels. Les travaux dans le cadre de Lustre considèrent les deux opérateurs temporels élémentaires : *pre* (précédent) et $->$ (initialisation) introduit dans la section 1.4.1. Dans nos définitions, nous pouvons sans perte de généralité nous limiter à l'opérateur FBY avec 1 cycle de retard. Les retards plus longs peuvent être modélisés en connectant plusieurs opérateurs en série.

1. l'opérateur FBY est celui défini dans le cadre du formalisme SCADE (voir section 1.4.1).
2. If Then Else.

Les constantes $faux$ et $vrai$ ne sont pas à proprement parler des opérateurs d'*interface*. Cependant, il est commode pour notre analyse de les considérer comme des entrées (constantes) du modèle.

Un modèle peut contenir plusieurs instances d'un type d'*opérateur*. Notamment, chaque entrée du modèle correspond à une intance d'opérateur IN, $faux$ ou $vrai$. De même, il y a autant d'opérateurs OUT que de sorties du modèle c'est-à-dire, par définition, 1.

Notation 1 (Op, $\mathbf{Op_{in}}$, $\mathbf{Op_{out}}$) *On notera* ***Op*** *l'ensemble des opérateurs du modèle que l'on considère. On notera* $\mathbf{Op_{in}} \subset Op$ *l'ensemble des opérateurs d'entrée (IN) et des constantes* $faux$, $vrai$ *du modèle. On notera* $\mathbf{Op_{out}} \subset Op$ *l'ensemble des opérateurs de sortie (OUT) du modèle qui n'est finalement qu'un singleton correspondant à la sortie O de l'observateur de la popriété.*

4.1.2 Arcs

Définition 3 *Un arc a pour origine la sortie d'un opérateur, et pour destination l'entrée d'un opérateur. Plusieurs arcs peuvent avoir la même origine*[3]*. En revanche, plusieurs arcs ne peuvent pas avoir la même destination.*

Notation 2 (Arc) *On notera* ***Arc*** *l'ensemble des arcs du modèle que l'on considère. On dit qu'un arc* ***entre*** *dans un opérateur op lorsqu'il est connecté à une entrée de op. De façon similaire, on dit qu'un arc* ***sort*** *d'un opérateur op lorsqu'il est connecté à une sortie de op.*

Nous allons illustrer nos propos au travers d'un exemple. Considérons le modèle M de la figure 4.1. Les arcs sont annotés α_1 à α_8.

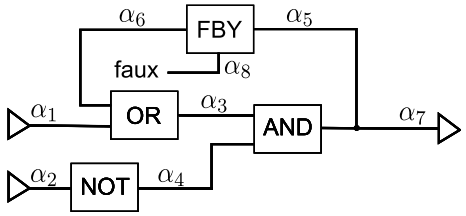


FIGURE 4.1 – Illustration d'arcs et d'opérateurs

Ce modèle comporte deux opérateurs d'entrée IN (dont les arcs sortants sont α_1 et α_2), une constante $faux$, trois opérateurs booléens (OR, NOT, AND), un opérateur temporel (FBY) et un unique opérateur de sortie (dont l'arc entrant est α_7). Ces opérateurs sont connectés entre eux par les arcs α_1 à α_8. On peut remarquer que α_5 et α_7 sont deux arcs différents bien qu'ils sortent tous deux de l'opérateur AND.

On définit **arcIn** et **arcOut**, deux fonctions renvoyant respectivement l'ensemble des arcs *entrants* et *sortants* de l'opérateur passé en paramètre.

3. nommés *fanouts* dans le domaine du matériel.

$$\text{arcIn} : \begin{cases} Op & \rightarrow & \mathcal{P}(Arc) \\ op & \mapsto & \text{arcIn}(op) \end{cases}$$

$$\text{arcOut} : \begin{cases} Op & \rightarrow & \mathcal{P}(Arc) \\ op & \mapsto & \text{arcOut}(op) \end{cases}$$

Définition 4 (Arc Prédécesseur) *Un arc α_1 est le prédécesseur d'un arc α_2 s'il existe un opérateur $op \in Op$ tel que α_1 est connecté à une des entrées de op et tel que α_2 est connecté à une des sorties de op.*

On définit la fonction **prédécesseur** telle que :

$$\text{Prédécesseur} : \begin{cases} Arcs \times Arcs & \rightarrow & \mathbb{B} \\ (\alpha_1, \alpha_2) & \mapsto & \begin{cases} \text{VRAI} & \text{si } \exists op \in Op / \alpha_1 \in arcIn(op) \\ & \wedge\ \alpha_2 \in arcOut(op) \\ \text{FAUX} & sinon \end{cases} \end{cases}$$

Par exemple, les arcs α_3 et α_4 sont *prédécesseurs* de α_7 et α_5 ; α_1 et α_6 sont *prédécesseurs* de α_3 ; α_2 est *prédécesseur* de α_4 ; α_8 et α_5 sont *prédécesseurs* de α_6 (noter le rebouclage).

4.1.3 Variables

Avant d'introduire la notion de variable, il est nécessaire d'introduire la notion de flot de données. Notre objectif étant d'analyser des contre-exemples, nous ne nous intéresserons qu'à des flots finis.

Définition 5 (Flot de données) *Un flot de données est une séquence finie de valeurs.*

De plus, nous considérerons uniquement des flots de données **booléens**.

Notation 3 (Flot) *On note **Flot** l'ensemble de tous les flots de données booléens possibles.*

On définit la fonction size(f) qui renvoie la longueur du flot f.

$$\text{size} : \begin{cases} Flot & \rightarrow & \mathbb{N}^* \\ f & \mapsto & size(f) \end{cases}$$

On définit la fonction partielle val(f, n) qui renvoie la n^{ieme} valeur du flot f pour $0 < n \leq size(f)$ (f n'est pas définie pour les autres valeurs de n).

$$\text{val} : \begin{cases} Flot \times \mathbb{N}^* & \rightarrow & \mathbb{B} \\ (f, n) & \mapsto & val(f, n) \end{cases}$$

Notation 4 (Var) *On notera* ***Var*** *l'ensemble des variables du modèle que l'on considère.*

Définition 6 (Variable) *A chaque arc est associé une unique variable* $\in Var$*. Cependant, une variable peut être associée à plusieurs arcs si et seulement si ces derniers sortent de la même sortie d'un opérateur.*

On définit **arc2var** la fonction qui à un arc associe l'unique variable qui lui est rattachée.

$$\text{arc2var} : \left\{ \begin{array}{rcl} Arc & \rightarrow & Var \\ \alpha & \mapsto & arc2var(\alpha) \end{array} \right.$$

On introduit la notion d'opérateur **producteur** et **consommateur** de variable. On dira qu'un opérateur *op produit* la variable A si A est associée à un *arc sortant* de *op*. On dira qu'un opérateur *op consomme* la variable A si A est associée à un *arc entrant* dans *op*. On qualifiera les variables produites par un *opérateur d'entrée* comme étant des *variables d'entrée*.

Notation 5 ($\mathbf{Var_{in}}$) *On notera* $Var_{in} \subseteq Var$ *l'ensemble des variables d'entrée du modèle.*

Il n'est pas nécessaire de caractériser l'ensemble des variables de sortie dans la mesure où il existe une unique variable de sortie notée O. La figure 4.2 montre un exemple d'affectation de variables sur le modèle de la figure 4.1.

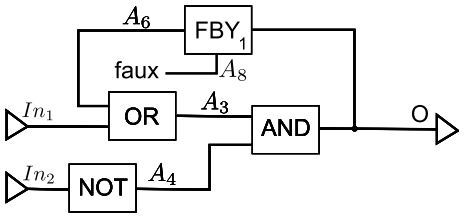


FIGURE 4.2 – Illustration des variables

In_1, In_2, A_3, A_4, A_6 et A_8 sont respectivement associées aux arcs α_1, α_2, α_3, α_4, α_6 et α_8. On remarque que la variable O est associée aux arcs α_5 et α_7. In_1, In_2 et A_8 sont les trois variables d'entrée de ce modèle [4].

Définition 7 (Valuation) *Une valuation est une fonction partielle* [5] *qui à une variable associe un flot de données.*

$$\sigma : \left\{ \begin{array}{rcl} Var & \rightarrow & Flot \\ v & \mapsto & \sigma(v) \end{array} \right.$$

4. La variable A_8 est une constante. Nous considérons que c'est une variable d'entrée dont le flot booléen associé est toujours FAUX quelque soit le scénario considéré.

5. car toutes les variables ne sont pas nécessairement valuées.

Notation 6 (Σ) *On note Σ l'ensemble de toutes les valuations possibles. On considère deux sous-ensembles remarquables de Σ :*
- $\Sigma_{in} = \{\sigma \in \Sigma / dom(\sigma) = Var_{in} \wedge \exists N \in \mathbb{N}^*, \forall v \in Var_{in}, size(\sigma(v)) = N\}$,
- $\Sigma_{all} = \{\sigma \in \Sigma / dom(\sigma) = Var \wedge \exists N \in \mathbb{N}^*, \forall v \in Var, size(\sigma(v)) = N\}$.

Où $dom(\sigma)$ désigne le domaine de la fonction partielle σ.

Σ_{in} représente les valuations de toutes les variables d'entrée par des flots de données de même longueur N, sans prendre en compte les autres variables du modèle. Σ_{all} représente la valuation de toutes les variables du modèle par des flots de données de longueur N. Dans nos analyses, N correspondra typiquement à la longueur du contre-exemple.

4.2 Scénario et exécution

Nous allons introduire la notion d'*exécution* et de *scénario*. Un *contre-exemple* sera alors défini comme un scénario dont l'exécution value à FAUX la sortie du modèle au dernier instant.

Définition 8 (Exécution) *Une exécution du modèle M, notée $Exec_M(\Sigma_{in})$, est une fonction qui à un élément de Σ_{in} associe un élément de Σ_{all} tel que les valuations des variables intermédiaires et de sortie respectent la sémantique du modèle.*

Dans notre cas, la fonction $Exec_M$ est donnée par un outil de simulation du modèle. Remarquons qu'on ne considère que des flots finis, et que les opérateurs du modèle ont des sorties définies pour toutes leurs valeurs d'entrée : une simulation termine donc toujours en valuant toutes les variables.

Définition 9 (Scénario) *Un scénario σ est un élément de Σ_{in}. Soit $N \in \mathbb{N}^*$, la longueur des flots de donnée d'entrée (cohérente pour toutes les variables). On dit que N est la longueur du scénario σ.*

Notation 7 ($\mathbf{V(n)_\sigma}$) *On note $V(n)_\sigma$, la valuation de la variable V d'un modèle M à l'instant n, lors de l'exécution du scénario σ :*

$$V(n)_\sigma = val(Exec_M(\sigma)(V), n)$$

Définition 10 (Contre-exemple) *Un scénario σ est un contre-exemple si l'exécution du modèle avec σ provoque la violation de la propriété uniquement au dernier instant. Plus formellement, un scénario σ de longueur N est un contre-exemple si :*
- $\forall n \in [1, N-1], O(n)_\sigma =$VRAI,
- $O(N)_\sigma =$FAUX.

Pour illustrer nos propos, nous considérons le scénario d'entrée σ_{in} de longueur $N = 3$ dans le modèle M de la figure 4.2 :

cycle	1	2	3
In_1	1	0	0
In_2	0	0	1
A_8	0	0	0

Pour alléger les notations dans les tableaux, les 0 (respectivement 1) désignent la valeur booléenne FAUX (respectivement VRAI). On peut remarquer que le modèle de la figure 4.2 est celui de la bascule RS à reset prioritaire, où In_1 et In_2 correspondent respectivement aux entrées Set et Reset de la bascule. La fonction $Exec_M(\sigma_{in})$ renvoie la valuation totale suivante :

cycle	1	2	3
In_1	1	0	0
In_2	0	0	1
A_8	0	0	0
A_3	1	1	1
A_4	1	1	0
A_6	0	1	1
O	1	1	0

On peut remarquer, en observant les valeurs de O, que σ_{in} est un *contre-exemple.*

4.3 Notion de chemin

Dans cette section nous allons détailler la notion de *chemins* en nous inspirant de celle proposée dans [Lakehal 2005, Lakehal 2007, Lakehal 2009] (voir section 3.3.3 dans le chapitre précédent). Nous considérerons uniquement des chemins finis, mais dont la longueur peut être arbitrairement grande.

4.3.1 Définitions de base

Définition 11 (Chemin) *Un chemin p de longueur $K \in \mathbb{N}^*$ est une suite finie d'arcs ($< \alpha_1, \ldots, \alpha_K >$) du modèle telle que :*

$$\forall k \in [\![1, K-1]\!],\ \textit{prédécesseur}(\alpha_k, \alpha_{k+1})$$

Notation 8 (P) *On note P l'ensemble de tous les chemins finis du modèle M. Si le modèle possède une boucle, P est infini.*

On introduit la notion de **naissance** et de **mort** d'un chemin ($< \alpha_1, \ldots, \alpha_K >$). La *naissance* d'un chemin désigne le premier arc α_1 du chemin. La *mort* d'un chemin désigne le dernier arc α_K.

Définition 12 (Chemin complet) *Un chemin p est **complet** si son premier arc sort d'un opérateur d'entrée et son dernier arc entre dans un opérateur de sortie.*

$$\exists op_{in} \in Op_{in}, \exists op_{out} \in Op_{out} / \\ (naissance(p) = arcOut(op_{in})) \wedge (mort(p) \in arcIn(op_{out}))$$

Dans l'exemple de la figure 4.1, il existe une infinité de chemins complets. Par exemple, les chemins débutant par α_1 et se terminant par α_7 sont de la forme : $<\alpha_1, \alpha_3><\alpha_5, \alpha_6, \alpha_3>^*<\alpha_7>$, où $*$ indique un nombre arbitraire d'itérations (y compris 0) de la boucle. De même, il y a une infinité de chemins complets débutant par α_2 et α_8.

Définition 13 (Ordre d'un chemin) *L'**ordre** du chemin p est le nombre n_i des opérateurs FBY(E,1,Init) traversés, de leur entrée retardée vers leur sortie, par le chemin p.*

L'ordre d'un chemin mesure donc le retard total subi lors de la propagation du signal porté par le premier arc du chemin, jusqu'au dernier arc du chemin. Par exemple, le chemin $< \alpha_1, \alpha_3, \alpha_5, \alpha_6, \alpha_3, \alpha_7 >$ de la figure 4.1 est d'ordre 1 car il traverse un opérateur FBY(E,1,Init) en passant par son entrée retardée E. Par contre, $< \alpha_8, \alpha_6, \alpha_3, \alpha_7 >$ est d'ordre 0, car la traversée de FBY s'effectue par l'entrée d'initialisation.

Définition 14 (Origine) *La fonction partielle **origine** associe à un chemin $p \in P$ et un instant $n > ordre(p)$, une variable v et un instant $n' \in [1, n]$ tels que :*

$$origine : \begin{cases} P \times \mathbb{N}^* & \rightarrow & Var \times \mathbb{N}^* \\ p, n & \mapsto & (v, n') = \text{origine}(p, n) / \begin{cases} v & = & arc2var(naissance(p)) \\ n' & = & n - ordre(p) \end{cases} \end{cases}$$

Considérons le modèle de la figure 4.2 et le chemin p formé par la suite d'arcs $< \alpha_1, \alpha_3, \alpha_5, \alpha_6, \alpha_3, \alpha_7 >$. Intuitivement, la fonction *origine* indique que, via ce chemin, la valeur de O à l'instant n est potentiellement dépendante de la valeur de In_1 à l'instant $n' = n - ordre(p)$. C'est à dire à l'instant $n' = n - 1$ dans ce cas. Cette influence n'est cependant effective que si le chemin p est actif à l'instant n. Le paragraphe suivant formalise cette notion de chemin actif.

4.3.2 Condition d'activation d'un chemin

Pour définir la condition d'activation d'un chemin p, nous reprenons essentiellement l'approche proposée dans [Lakehal 2005, Lakehal 2007, Lakehal 2009]. La condition d'activation est une expression Lustre, construite récursivement en parcourant p de son dernier arc vers le premier. Par rapport à [Lakehal 2005, Lakehal 2007, Lakehal 2009], nous introduisons simplement quelques modifications mineures, pour

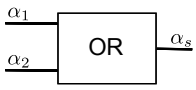
- prendre en compte l'opérateur temporel FBY de SCADE (plutôt que les opérateurs *pre* et -> dans la définition initiale des auteurs),
- adapter la construction de l'expression Lustre aux détails de notre modélisation (pour nous, la variable portée par un arc est donnée par la fonction *arc2var*).

Soit $K \in \mathbb{N}^*$, soit $p =< \alpha_1, \ldots, \alpha_{K-1}, \alpha_K >$ un chemin de longueur K, $p' =< \alpha_1, \ldots, \alpha_{K-1} >$ son *préfixe* de longueur $K-1$. Si $K = 1$, le préfixe est vide. Si $K > 1$, le préfixe est non vide et il existe un opérateur op tel que $\alpha_{K-1} \in in(op)$ et $\alpha_K \in out(op)$.

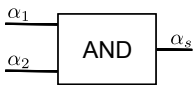
Définition 15 (Condition d'activation) *La condition d'activation du chemin p est une expression booléenne notée $\mathcal{AC}(p)$, définie comme suit :*

1. *Si* ***K=1*** *alors* $\mathcal{AC}(p)$=VRAI,
2. *sinon*
   (a) *si op est un* ***opérateur booléen*** *ou* ***conditionnel*** *alors :*
      i. $\mathcal{AC}(p) = \mathcal{AC}(p') \wedge \mathcal{OC}(\alpha_{k-1}, \alpha_k)$ *où* $\mathcal{OC}(\alpha_{k-1}, \alpha_k)$ *est une expression booléenne qui dépend de la sémantique de l'opérateur op.*
   (b) *Si op est un* ***opérateur FBY****(E,1,Init) alors :*
      i. *si* α_{k-1} *entre dans l'entrée retardée E,* $\mathcal{AC}(p)$ = FAUX -> $pre(\mathcal{AC}(p'))$
      ii. *si* α_{k-1} *entre dans l'entrée d'initialisation Init,* $\mathcal{AC}(p) = \mathcal{AC}(p')$ -> FAUX

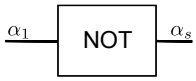
Voici la définition de la fonction $\mathcal{OC}$ pour les opérateurs OR, AND, NOT et ITE.



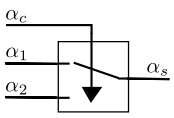
$$\begin{aligned} \mathcal{OC}(\alpha_1, \alpha_s) &= \text{arc2var}(\alpha_1) \vee \neg\text{arc2var}(\alpha_2) \\ \mathcal{OC}(\alpha_2, \alpha_s) &= \neg\text{arc2var}(\alpha_1) \vee \text{arc2var}(\alpha_2) \end{aligned}$$



$$\begin{aligned} \mathcal{OC}(\alpha_1, \alpha_s) &= \neg\text{arc2var}(\alpha_1) \vee \text{arc2var}(\alpha_2) \\ \mathcal{OC}(\alpha_2, \alpha_s) &= \text{arc2var}(\alpha_1) \vee \neg\text{arc2var}(\alpha_2) \end{aligned}$$



$$\mathcal{OC}(\alpha_1, \alpha_s) = \text{VRAI}$$



$$\begin{aligned} \mathcal{OC}(\alpha_1, \alpha_s) &= \neg\text{arc2var}(\alpha_c) \\ \mathcal{OC}(\alpha_2, \alpha_s) &= \text{arc2var}(\alpha_c) \\ \mathcal{OC}(\alpha_c, \alpha_s) &= \text{VRAI} \end{aligned}$$

Dans le cadre de nos travaux, nous nous intéressons aux chemins activés par l'exécution d'un contre-exemple (ou plus généralement, d'un scénario) au cours du temps.

Définition 16 (Activation d'un chemin à un instant) *On dira qu'un scénario σ active un chemin p au cycle n si la condition d'activation de p est évaluée à* VRAI *au cycle n lors de l'exécution de σ, ce que l'on note :*

$$\mathcal{A}(p, \sigma, n) = \text{VRAI}$$

Certains chemins sont trivialement actifs à chaque cycle d'exécution du scénario : ce sont par exemple les chemins constitués d'un arc unique.

Parmi les chemins actifs à l'instant n, nous distinguons plus particulièrement ceux dont le premier arc porte une variable d'entrée. Ces chemins permettent de propager une valeur d'entrée du scénario pour affecter la valeur d'une variable du modèle. Par exemple, si le premier arc d'un chemin p porte la variable d'entrée E, et le dernier arc porte une variable V, alors $\mathcal{A}(p, \sigma, n) = \text{VRAI}$ indique que le calcul de $V(n)_\sigma$ dépend de l'entrée $E(n - ordre(p))_\sigma$.

Définition 17 (Ensemble des chemins de propagation) *L'ensemble des chemins de propagation* ***des entrées à l'instant*** **n** *dans le scénario* σ *est :*

$$P_e(\sigma, n) = \{p \in P / \mathcal{A}(p, \sigma, n) = \text{VRAI} \wedge arc2var(naissance(p)) \in V_{in}\}$$

4.3.3 Exemple

Pour illustrer ces notions, reprenons l'exemple de la bascule RS à reset prioritaire. La figure 4.3 visualise l'exécution d'un scénario de longueur 1. Les valeurs des variables sont notées sur les arcs (pour alléger la figure, on note 0, 1 plutôt que FAUX, VRAI). La figure visualise également les trois chemins complets d'ordre 0 du modèle, que l'on note p_1, p_2 et p_3. Ils sont respectivement définis par les suites d'arcs $< \alpha_1, \alpha_3, \alpha_7 >$, $< \alpha_2, \alpha_4, \alpha_7 >$ et $< \alpha_8, \alpha_6, \alpha_3, \alpha_7 >$ (voir les références des arcs sur la figure 4.1). Nous allons montrer que ces trois chemins sont activés par le scénario, à l'instant 1.

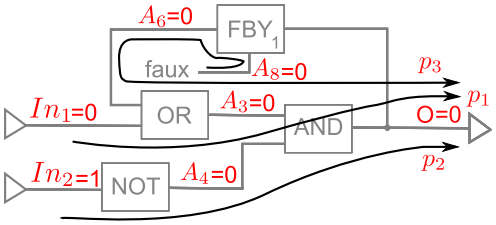


FIGURE 4.3 – Illustration des chemins actifs

Commençons par exprimer la *condition d'activation* de p_2 qui est la plus simple :

$$\mathcal{AC}(p_2) = \underbrace{\text{VRAI}}_{\mathcal{AC}(<\alpha_2>)} \wedge \underbrace{\text{VRAI}}_{\mathcal{OC}(\alpha_2,\alpha_4)} \wedge \underbrace{(\neg A_4 \vee A_3)}_{\mathcal{OC}(\alpha_4,\alpha_7)}$$

La *condition d'activation* de p_1 est légèrement plus compliquée :

$$\mathcal{AC}(p_1) = \underbrace{\text{VRAI}}_{\mathcal{AC}(<\alpha_1>)} \wedge \underbrace{(In_1 \vee \neg A_6)}_{\mathcal{OC}(\alpha_1,\alpha_3)} \wedge \underbrace{(\neg A_3 \vee A_4)}_{\mathcal{OC}(\alpha_3,\alpha_7)}$$

La *condition d'activation* de p_3 est la suivante :

$$\mathcal{AC}(p_3) = \underbrace{(\text{VRAI} -> \text{FAUX})}_{\mathcal{AC}(<\alpha_8,\alpha_6>)} \wedge \underbrace{(A_6 \vee \neg In_1)}_{\mathcal{OC}(\alpha_6,\alpha_3)} \wedge \underbrace{(\neg A_3 \vee A_4)}_{\mathcal{OC}(\alpha_3,\alpha_7)}$$

On rappelle que l'opérateur Lustre d'initialisation (cf. section 1.4.1) : A->B fait référence au flot de données $(A(1), B(2), B(3), \ldots)$. $\mathcal{AC}(p_3)$ peut se simplifier de la manière suivante :

$$\mathcal{AC}(p_3) = [(A_6 \vee \neg In_1) \wedge (\neg A_3 \vee A_4)] \text{ -> FAUX}$$

C'est-à-dire que ce chemin ne pourra jamais être activé à des cycles > 1.

Considérons maintenant le scénario montré en Figure 4.3. Il s'agit d'un scénario σ de longueur 1 tel que In_1 =FAUX, In_2 =VRAI. En évaluant les conditions ci-dessus, on établit que σ active p_1, p_2 et p_3 au cycle 1 :

$$\mathcal{A}(p_1, \sigma, 1) = \text{VRAI et } \mathcal{A}(p_2, \sigma, 1) = \text{VRAI et } \mathcal{A}(p_3, \sigma, 1) = \text{VRAI}$$

Aucun autre chemin complet n'est activé par ce scénario. En effet, tous les chemins complets d'ordre ≥ 1 (passant au moins une fois par l'entrée retardée du FBY) ont une condition d'activation qui peut se mettre sous la forme : FAUX -> $(\ldots)$. Ces chemins ne sont jamais activés au premier cycle d'exécution. Plus généralement, un chemin d'ordre n n'est jamais actif lors des n premiers cycle.

L'ensemble des chemins de propagation des entrées $P_e(\sigma, 1)$ contient donc :
- p_1, p_2 et p_3,
- tous les chemins préfixes de p_1, p_2 et p_3,
- les trois chemins formés à partir de p_1, p_2 et p_3 en remplaçant le dernier arc α_7 par α_5.

4.4 Notion de *cause*

Nous allons maintenant caractériser la notion de *cause* pour une valeur de variable à un cycle d'exécution. Dans l'analyse d'un contre-exemple σ de longueur N, on s'intéresse typiquement à la cause de la valeur $O(N)_\sigma$ = FAUX, correspondant à la violation de la propriété observée au cycle N. Plus généralement, on peut s'intéresser à une variable portée par n'importe quel arc du modèle, et dont on cherche à expliquer la valeur à un instant $n \in [1, N]$.

4.4.1 Définition de base

Les causes de la valeur de $V = arc2var(\alpha)$ à l'instant n sont recherchées parmi les chemins de $P_e(\sigma, n)$ ayant propagé des valeurs d'entrées jusqu'à l'arc α. Une cause $\mathcal{C}$ va alors être définie comme un sous-ensemble de $P_e(\sigma, n)$, ne contenant que des chemins dont le dernier arc est α. Les origines de ces chemins déterminent des valeurs d'entrée $E_i(n_i)_\sigma$ ayant eu une influence sur $V(n)_\sigma$. Pour que $\mathcal{C}$ soit une cause, il faut que ces valeurs d'entrées soient suffisantes pour :
- garantir l'activation des chemins de $\mathcal{C}$ au cycle n,
- garantir que V a la valeur observée au cycle n.

Cette propriété de suffisance peut s'exprimer en se référant à des variantes σ' de σ, conservant la même valuation des entrées déterminées par la cause, mais pouvant valuer différemment les autres entrées.

Définition 18 (Cause) *Pour un arc α portant une variable V, une cause de la valeur $V(n)_\sigma$ observée lors de l'exécution du scénario σ est un ensemble de chemins $\mathcal{C}$ vérifiant :*

1. $\mathcal{C} \subseteq P_e(\sigma, n)$
2. $\forall p \in \mathcal{C}, mort(p) = \alpha$,
3. *Soit* $Origines = \{(V_i, n_i) \in Var_{in} \times \mathbb{N}^* / \exists p \in \mathcal{C}, (V_i, n_i) = origine(p, n)\}$.
   $\forall \sigma' \in \Sigma_{in}$, *tel que* $size(\sigma') = size(\sigma)$ *et* $V_i(n_i)_{\sigma'} = V_i(n_i)_\sigma$ *pour tout* $(V_i, n_i) \in Origines$, *on a :*
   (a) $\forall p \in \mathcal{C}, \mathcal{A}(p, \sigma', n) =$VRAI
   (b) $V(n)_{\sigma'} = V(n)_\sigma$

Notation 9 ($\mathcal{C}(\alpha, \mathbf{n}, \sigma)$) *On note $\mathcal{C}(\alpha, n, \sigma)$ l'ensemble des causes de la valuation de $V = arc2var(\alpha)$ à l'instant n dans le scénario σ.*

Lorsque σ est un contre-exemple de longueur N et α l'arc de sortie de l'observateur de propriété, on dira que $\mathcal{C}(\alpha, N, \sigma)$ désigne l'ensemble des **causes du contre-exemple**. Remarquons que c'est alors un ensemble d'ensembles de chemins complets.

4.4.2 Exemple

Reprenons l'exemple de la bascule RS, et du scénario σ de longueur 1, précédemment proposé (figure 4.3) : $In_1 =$ FAUX et $In_2 =$ VRAI. Nous cherchons l'ensemble de causes $\mathcal{C}(\alpha_7, 1, \sigma)$. L'arc α_7 porte la variable O et est consommé par l'opérateur de sortie. Chaque cause doit être un ensemble de chemins complets, actifs à l'instant 1. Nous avons vu que p_1, p_2 et p_3 sont tous actifs à l'instant 1, donc une cause est un sous-ensemble de $\{p_1, p_2, p_3\}$.

Nous allons maintenant montrer que les ensembles suivants :

1. $c_1 = \{p_1, p_2, p_3\}$,
2. $c_2 = \{p_1, p_3\}$,
3. $c_3 = \{p_2\}$,

sont des *causes*.

Considérons le **troisième cas** : $c_3 = \{p_2\}$, on peut écrire :

1. $mort(p_2) = \alpha_7$;
2. $p_2 \in P_e(\sigma, 1)$;
3. $Origines = \{(In_2, 1)\}$.
   $\forall \sigma'$ tel que In_2=VRAI alors : $A_4(1)_{\sigma'}$=FAUX et $O(1)_{\sigma'}$=FAUX, donc :
   (a) $\mathcal{A}(p_2, \sigma', 1)$=VRAI,
   (b) $O(1)_{\sigma'} = O(1)_\sigma$=FAUX.

Donc $\{p_2\}$ est une **cause** de la valuation de O=$arc2var(\alpha_7)$ à l'instant $N = 1$.

Le raisonnement est identique pour le **deuxième cas** : $c_2 = \{p_1, p_3\}$. On peut écrire :

1. $mort(p_1) = \alpha_7$ et $mort(p_3) = \alpha_7$;
2. p_1 et p_3 appartiennent à $P_e(\sigma, 1)$;
3. $Origines = \{(In_1, 1), (A_8, 1)\}$.
   $\forall \sigma'$ qui vérifie $In_1(1)_{\sigma'}$=FAUX et $A_8(1)_{\sigma'}$ =FAUX alors : $A_3(1)_{\sigma'}$=FAUX et $O(1)_{\sigma'}$=FAUX, donc :
   (a) $\mathcal{A}(p_1, \sigma', 1)$=VRAI et $\mathcal{A}(p_3, \sigma', 1)$=VRAI,
   (b) $O(1)_{\sigma'} = O(1)_{\sigma}$=FAUX.

Donc, $\{p_1, p_3\}$ est une **cause** de la valuation de O=$arc2var(\alpha_7)$ à l'instant $N = 1$. Il en est de même pour le **premier cas** : $c_1 = \{p_1, p_2, p_3\}$.

Le lecteur pourra vérifier qu'il n'existe pas d'autre sous-ensemble de $\{p_1, p_2, p_3\}$ formant une cause. On a donc :

$$\mathcal{C}(\alpha_7, 1, \sigma) = \{\{p_1, p_2, p_3\}, \{p_1, p_3\}, \{p_2\}\}$$

Il apparaît clairement sur cet exemple que les *causes* c_2 et c_3 expliquent chacune, de manière indépendante, le résultat de la sortie du modèle. Cependant, on peut observer que c_2 et c_3 sont incluses dans c_1, et que cette dernière n'apporte pas d'information supplémentaire. Pour prendre en compte ce phénomène, nous allons introduire la notion de *minimalité*.

4.4.3 Minimalité d'une cause

Définition 19 (Cause minimale) *On dira qu'une cause $c_1 \in \mathcal{C}(\alpha, n, \sigma)$ est minimale si :*

$$\forall c_2 \in \mathcal{C}(\alpha, n, \sigma), (c_2 \neq c_1) \Rightarrow (c_1 \not\supseteq c_2)$$

On souhaite déterminer les *causes minimales* car elles ne contiennent pas de chemins superflus. La minimalité répond au besoin de synthèse de l'analyse d'un contre-exemple. Nous verrons cependant, dans notre discussion des choix de formalisation, que la minimalité se révèle une notion difficile à prendre en compte sur certains types de modèles.

Notation 10 ($\mathcal{C}_{min}$) *On note $\mathcal{C}_{min}(\alpha, n, \sigma)$ l'ensemble des causes minimales responsables de la valuation de $arc2var(\alpha)$ à l'instant n dans le scénario σ.*

Ainsi, si l'on reprend l'exemple de la bascule RS, la cause c_1 n'est pas minimale car : $c_1 \supset c_2$ et $c_1 \supset c_3$. En revanche, c_2 est une *cause minimale* car $c_2 \not\supseteq c_1$ et $c_2 \not\supseteq c_3$. Il est de même pour c_3. Donc, dans ce cas :

$$\mathcal{C}_{min}(\alpha_7, n, \sigma) = \{\{p_1, p_3\}, \{p_2\}\}$$

4.4.4 Comparaison de deux contre-exemples

La notion de cause minimale permet de comparer deux contre-exemples obtenus à partir de la vérification d'un même modèle. Notons que ces deux contre-exemples peuvent être de longueur différente.

Définition 20 (Contre-exemples différents) *Deux contre-exemples cex_1 de longueur N_1 et cex_2 de longueur N_2 sont différents si :*

$$\mathcal{C}_{min}(O, N_1, cex_1) \neq \mathcal{C}_{min}(O, N_2, cex_2)$$

En pratique, l'ordre dans lequel les contre-exemples sont obtenus est important. Connaissant cex_1, on souhaitera produire un contre-exemple cex_2 qui non seulement soit différent du premier, mais aussi apporte de nouvelles informations.

Définition 21 (Contre-exemple nouveau) *Soit cex_1 un contre-exemple déjà connu de longueur N_1. Un contre-exemple cex_2 de longueur N_2 est nouveau par rapport à cex_1 si :*

$$\mathcal{C}_{min}(O, N_2, cex_2) \nsubseteq \mathcal{C}_{min}(O, N_1, cex_1)$$

Prenons des exemples de scénarios valuant la sortie de la bascule RS à FAUX. Nous les interprétons comme des contre-exemples de la propriété : $\forall n \in \mathbb{N}^*, O(n) =$VRAI.

Les contre-exemples sont représentés dans le tableau 4.1. Le trois premiers sont des contre-exemples de longueur 1. Le dernier est un contre-exemple de longueur 2.

	In_1	In_2	O	$\mathcal{C}_{min}$
cex_1	0	0	0	$\{\{p_1, p_3\}\}$
cex_2	0	1	0	$\{\{p_1, p_3\}, \{p_2\}\}$
cex_3	1	1	0	$\{\{p_2\}\}$
cex_4	11	01	10	$\{\{p_2\}\}$

TABLE 4.1 – Quatre contre-exemples

On constate que cex_3 et cex_4 ne sont pas différents : ils apportent strictement la même information, c'est-à-dire que l'activation de l'entrée Reset de la bascule (In_2) met systématiquement la sortie à FAUX. Décaler le Reset d'un cycle ne change pas le type de violation observé.

Si cex_3 est le premier contre-exemple observé, il serait intéressant de générer un nouveau contre-exemple, mettant en évidence un deuxième type de violation : l'absence de Set (In_1) au premier cycle d'exécution, la bascule étant initialisée à FAUX à ce cycle. Les contre-exemples cex_1 ou cex_2 permettent d'identifier ce nouveau type de violation, en exhibant la cause minimale $\{p_1, p_3\}$.

Pour guider la production d'un nouveau contre-exemple, on pourra ainsi indiquer au model checker de :

– chercher un contre-exemple dans lequel certains chemins des causes minimales connues sont inactifs (par exemple, p_2 inactif),

– chercher un contre-exemple dans lequel de nouveaux chemins sont actifs (par exemple, p_3 actif).

Les détails des stratégies correspondantes restent à définir. Cependant, nous avons un critère pour décider du succès de l'application d'une stratégie donnée : la comparaison des causes minimales, qui permet de déterminer si un contre-exemple obtenu est réellement nouveau.

4.5 Remarques sur le choix de la formalisation

Nous allons expliquer les motivations des choix réalisés et les limites de la formalisation précédemment proposée.

4.5.1 Discussion sur la notion de *cause*

Nous revenons sur la définition d'une *cause* afin de justifier notre choix. La notion de *cause* d'une valuation d'une variable $V = arc2var(\alpha)$ à l'instant n aurait pu être définie par un ensemble de couples (V_i, n_i) tel que :

$$\mathcal{C} = \{(V_i, n_i) \in Var_{in} \times \mathbb{N}^* / \forall \sigma' \in \Sigma_{in}, (V_i(n_i)_{\sigma'} = V_i(n_i)_\sigma) \Rightarrow (V(n)_{\sigma'} = V(n)_\sigma)\}$$

La différence par rapport à notre définition est que l'on focalise sur des valeurs d'entrée à certains cycles d'un scénario, sans prendre en compte explicitement les chemins actifs dans le modèle.

Pour certains modèles simples, les deux définitions donnent des résultats équivalents. Considérons par exemple le modèle de la figure 4.4.

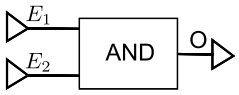


FIGURE 4.4 – Exemple d'un modèle unique composé d'un opérateur AND

Considérons la propriété : $\forall n \in \mathbb{N}^*, O(n) =$VRAI et le contre-exemple : ($E_1(1)$, $E_2(1)$)=(FAUX, FAUX). En utilisant la définition focalisée sur les entrées, l'ensemble $\{(E_1, 1)\}$ est une *cause* car quelle que soit la valeur de E_2 à l'instant 1, la propriété est violée à l'instant 1. De même, les ensembles $\{(E_2, 1)\}$ et $\{(E_1, 1), (E_2, 1)\}$ sont des causes. Selon notre définition, les causes sont $\{p_1\}$, $\{p_2\}$ et $\{p_1, p_2\}$, où p_1 (respectivement p_2) est le chemin connectant l'entrée E_1 (respectivement E_2) à la sortie O. Les deux définitions donnent ici des résultats équivalents, car on peut retrouver les chemins actifs à partir des valeurs d'entrée, et inversement les origines des chemins déterminent les valeurs d'entrée.

Cependant, nous allons montrer que les deux définitions ne sont pas équivalentes dans le cas général. Considérons le modèle de la figure 4.5 composé d'un opérateur ITE et de deux sous-modèles combinatoires M_1 et M_2. On suppose que le contre-exemple obtenu est de longueur 1, et que les variables internes X et Y prennent la même valeur FAUX lors de l'exécution du contre-exemple.

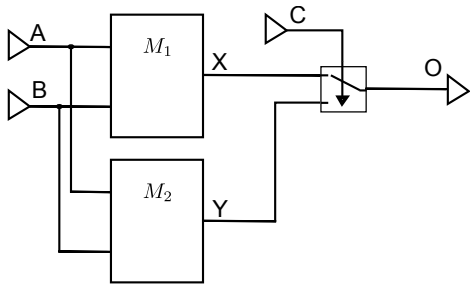


FIGURE 4.5 – Sélection entre deux modèles identiques ayant les mêmes entrées

Dans la définition focalisée sur les entrées, une cause est $\{(A, 1), (B, 1)\}$. En effet, les valeurs d'entrée de A et B déterminent X=Y=FAUX, et la sortie O sera à FAUX quel que soit C. Notons que la connaissance des seules valeurs de A et B ne permet pas de déterminer si la propagation de la valeur FAUX s'effectuera via l'arc associé à X ou l'arc associé à Y. Néanmoins, en raisonnant de la sémantique du modèle, on voit que la sortie O sera nécessairement valuée à FAUX dans tous les cas.

Dans notre définition, au contraire, on veut savoir quels chemins de propagation ont permis de valuer la sortie à FAUX. En particulier, toute cause devra contenir le chemin connectant l'entrée C à O. Les contre-exemples où l'opérateur ITE sélectionne la valeur de X seront considérés comme différents des contre-exemples sélectionnant Y. Un chemin inactif ne fera jamais partie d'une cause.

Considérer explicitement les chemins de propagation est classique quand on fait du déboguage et du diagnostic, et c'est aussi le choix que nous avons pris. Pour nous comparer à l'existant, prenons l'exemple de l'outil Ludic, présenté dans la section 3.5. Cet outil offre une fonctionnalité de diagnostic, permettant d'inspecter le comportement d'un modèle Lustre pour une simulation donnée. Le principe est d'explorer la structure d'un *réseau d'opérateurs* par un parcours en arrière. En particulier, le traitement d'un ITE consiste à interagir avec l'utilisateur pour déterminer si la sortie de l'opérateur (jugée erronée) est due à une erreur dans la valeur de la condition, ou à une erreur dans la valeur de l'entrée de l'ITE sélectionnée. La définition de la valeur erronée correspondante sera alors la seule explorée. Dans notre cas, il n'y a pas d'interaction avec l'utilisateur lors la construction des causes, et donc pas de choix du chemin à explorer. Mais le principe général est le même que dans Ludic : la cause de l'erreur est cherchée dans la condition et l'entrée sélectionnée. Les chemins inactifs passant par l'autre entrée ne sont pas explorés.

4.5.2 Minimalité et chemins reconvergents

Nous revenons sur la notion de *minimalité*, et montrons que le calcul des causes minimales peut s'avérer complexe, car il faut considérer les dépendances entre chemins du modèle.

Nous avons évoqué dans la section 3.2 les difficultés rencontrées dans le domaine du test du matériel en présence de **chemins reconvergents**. Deux chemins re-

convergents partent d'un même signal, et possèdent des sous-chemins qui divergent puis convergent plus loin dans la structure. Les dépendances introduites par ces chemins rendent plus complexe la recherche de vecteurs de test. Par exemple, pour un circuit combinatoire sans chemins reconvergents, la recherche peut s'effectuer en temps linéaire par rapport à la taille du circuit. Dans le cas général avec chemins reconvergents, les pires cas sont de complexité exponentielle.

Un modèle SCADE est très similaire à un circuit combinatoire ou séquentiel, et il peut posséder des chemins reconvergents. Dans nos notations, deux chemins p_1, p_2 seront dits reconvergents si : $arc2var(naissance(p_1)) = arc2var(naissance(p_2))$, $mort(p_1) = mort(p_2)$, et $p_1 \neq p_2$. Nous nous intéresserons plus particulièrement aux chemins reconvergents **de même ordre**, qui possèdent donc la propriété d'avoir la même origine, au sens de la définition 14. Les figures 4.5 et 4.6 sont deux exemples de modèles avec des chemins reconvergents d'ordre 0. Le premier (figure 4.5) ne pose pas de problème vis-à-vis du calcul des causes, car les conditions d'activation des chemins reconvergents sont exclusives. Cependant, dans le deuxième exemple, on peut avoir à analyser un scénario qui active les deux chemins reconvergents : les dépendances existantes doivent donc être prises en compte pour déterminer si une cause est minimale.

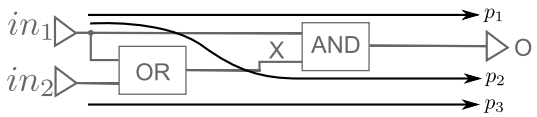


FIGURE 4.6 – Cas particulier

Considérons le modèle de la figure 4.6, activé avec un scénario σ de longueur 1 : in_1, in_2=VRAI,VRAI. Les trois chemins complets indiqués sur la figure, p_1, p_2 et p_3, sont actifs. Soit o l'arc portant la sortie O. L'ensemble des causes $\mathcal{C}(o, 1, \sigma)$ est :

$$\{\{p_1\}, \{p_2\}, \{p_1, p_2\}, \{p_1, p_3\}, \{p_2, p_3\}, \{p_1, p_2, p_3\}\}$$

et seuls $\{p_1\}$ et $\{p_2\}$ sont des *causes minimales*. Le chemin p_3 n'est pas inclus dans ces *causes minimales*. On pourra vérifier que, du fait des chemins reconvergents, la sémantique du modèle de la figure 4.6 est en fait $O = in_1$.

Si le modèle de la figure 4.6 ne comportait pas de chemins reconvergents, par exemple en introduisant une entrée supplémentaire in'_1 à l'extrémité du chemin p_2, l'ensemble des causes serait :

$$\{\{p_1, p_2\}, \{p_1, p_3\}, \{p_1, p_2, p_3\}\}$$

pour un scénario valuant toutes les entrées à VRAI. Les calculs des causes minimales donnerait alors $\{p_1, p_2\}$ et $\{p_1, p_3\}$. Nous verrons dans le chapitre 5 qu'il est possible d'effectuer un tel calcul par un simple parcours en arrière des opérateurs du modèle, avec un traitement local approprié à chaque type d'opérateur. Néanmoins, ce calcul suppose que les origines (v_i, n_i) des chemins explorés sont indépendantes, ce qui est inexact avec des chemins reconvergents de même ordre.

Le but du calcul des causes minimales est d'offrir à l'utilisateur une assistance dans l'analyse des contre-exemples. Cette assistance n'a d'intérêt que si elle lui permet d'avoir rapidement des résultats exploitables. L'analyse par model-checking étant déjà très coûteuse en temps, il ne nous a pas paru souhaitable d'implémenter des traitements additionnels complexes. Ainsi, nous avons fait le choix de ne pas traiter le problème des chemins reconvergents de façon exacte. Pour un modèle comme celui de la figure 4.6, nous tolérerons donc que le résultat délivré ne soit pas minimal, c'est à dire : $\{\{p_1, p_2\}, \{p_1, p_3\}\}$, plutôt que $\{\{p_1\}, \{p_2\}\}$.

4.5.3 Cas des hypothèses du modèle

Le modèle analysé peut contenir des hypothèses, permettant de contraindre les entrées. Elles apparaissent sous la forme d'un *observateur synchrone* que l'on appelle *observateur d'hypothèses*. Le model checker tient compte de ces hypothèses pour démontrer les propriétés. Si un *contre-exemple* est généré, il respectera les hypothèses. C'est-à-dire qu'à chaque instant du *contre-exemple*, la sortie de l'observateur d'hypothèses est valuée à VRAI.

Notre calcul des causes minimales porte sur le même modèle que celui analysé par le model checker. Il inclut donc l'observateur d'hypothèses. Cependant, en pratique, l'information contenue dans cet observateur n'est jamais exploitée.

Remarquons tout d'abord que, dans un modèle SCADE, les observateurs de propriété et d'hypothèse sont des composants indépendants. Un chemin complet entre une entrée du modèle et la sortie O de l'observateur de propriété ne traversera jamais la structure de l'observateur d'hypothèses.

Remarquons ensuite que notre définition de *cause* ne prend pas en compte les contraintes sur les entrées introduites par les hypothèses. La définition 18 fait référence à des variantes σ' d'un scénario σ de référence, variantes pour lesquelles certaines entrées peuvent prendre des valeurs quelconques - y compris des valeurs hors hypothèse. L'exécution de σ' est toujours bien définie, car nous considérons des flots d'entrée finis, et des opérateurs ayant des sorties définies pour toutes leurs entrées possibles. Par exemple, considérons le modèle M', créé à partir de la figure 4.6 en ajoutant une entrée in'_1 à l'extrémité du chemin p_2, et en introduisant une hypothèse $in_1 = in'_1$. Considérons également le scénario σ qui value les trois entrées de M' à VRAI. Selon notre définition, $\{p_1\}$ n'est pas une cause de la valuation de la sortie à VRAI, car on peut exhiber un scénario σ' avec $in_1, in'_1, in_2 =$ VRAI, FAUX, FAUX dont l'exécution valuerait O à FAUX. Ce scénario σ' ne respecte pas l'hypothèse $in_1 = in'_1$, mais la définition 18 ne le contraint pas à le faire. Si nous changions la définition pour écarter les σ' hors hypothèse, alors $\{p_1\}$ serait une cause (et même, une cause minimale).

Le choix de ne pas considérer les hypothèses est délibéré. Il résulte de notre volonté de réaliser une analyse légère, pouvant donner des retours rapides à l'utilisateur. Les traitements complexes sont réservés au model checker, le calcul des causes minimales n'intervenant qu'en support à l'exploitation des contre-exemples produits.

4.5.4 Limitations sur le type de modèle considéré

4.5.4.1 Remarques sur le type des variables

Dans la formalisation proposée, seul le type de flot de donnée *booléen* est abordé. Nous avons repris les conditions d'activations de chemins définies dans [Lakehal 2005, Lakehal 2007, Lakehal 2009], qui ne considéraient pas les opérateurs de calcul numérique. Notre définition de *cause* est basée sur cette notion de *condition d'activation* et hérite donc de la même limitation. Dans les perspectives de nos travaux, il serait souhaitable de généraliser la formalisation des causes pour traiter les autres types de variable impliquées dans un système de commande de vol, les flots entiers et réels.

Une solution immédiate serait d'étendre la notion de *condition d'activation* à ces types. Par exemple, on peut considérer que tous les sous-chemins entre les entrées d'un additionneur et sa sortie sont actifs à chaque instant. Les causes de la sortie de l'additionneur intégreraient alors ces sous-chemins.

Cette solution nous paraît cependant très insuffisante. Supposons qu'une planche contienne une cascade d'opérateurs arithmétiques. L'analyse des causes minimales extrairait alors tous les chemins complets de cette planche, ce qui aurait une faible valeur informative. Une solution plus ambitieuse serait d'intégrer des informations symboliques dans la notion de *cause*. Ceci nécessiterait cependant des investigations plus poussées, que nous n'avons pas menées dans le cadre de cette thèse. Un souci serait d'adopter un compromis entre la richesse de l'information symbolique fournie à l'utilisateur, et la volonté de garder un calcul des causes simple et performant.

4.5.4.2 Limitation liée à l'horloge unique

Les notions théoriques introduites ne sont définies que dans le cas où le modèle possède une horloge unique. D'après notre expérience de fonctions extraites de systèmes de commande de vol, ceci permet déjà de traiter un bon nombre de propriétés intéressantes. Néanmoins, dans certains cas, il faut pouvoir analyser des modèles possédant plusieurs horloges. Dans ces modèles, les horloges multiples sont gérées à l'aide des opérateurs Lustre *when* et *current*.

Les travaux de [Lakehal 2005, Lakehal 2007, Lakehal 2009], sur lesquels nous nous sommes basés, ne considéraient pas ces opérateurs. Cependant, une extension a récemment été proposée [Papailiopoulou 2008] pour permettre le test structurel des modèles multi-horloges. Cette extension serait un point de départ adéquat pour les perspectives de nos travaux.

4.6 Conclusion

Ce chapitre a reformulé un problème général : *assister l'analyse de contre-exemples*, en un problème plus précis : *calculer les causes minimales d'un contre-exemple*.

La notion de *cause minimale* a été formellement introduite. Les *causes minimales* d'un contre-exemple sont recherchées parmi les *chemins actifs* au moment de la

violation. Intuitivement, on s'intéresse aux chemins propageant certaines valeurs d'entrée du contre-exemple à la valeur de sortie FAUX de l'*observateur de la propriété*, observée au dernier cycle. La notion de *chemin actif* est directement reprise des travaux de [Lakehal 2005, Lakehal 2007, Lakehal 2009]. Être une *cause*, et être une *cause minimale*, sont alors définis comme des propriétés permettant d'identifier des ensembles de *chemins actifs*. Le cadre de notre formalisation est pour l'instant celui des modèles avec des flots booléens et une unique horloge. L'extension aux flots numériques et aux horloges multiples constitue une des perspectives de nos travaux.

Une exigence forte, qui a guidé nos choix de formalisation, est que le temps de calcul des *causes minimales* doit être négligeable par rapport au temps passé par le model checker pour analyser le modèle. Nous avons donc cherché à éviter, autant que possible, d'avoir à réaliser des raisonnements complexes sur le comportement du modèle. Ainsi, nous n'avons pas retenu une formulation alternative de la notion de cause, exclusivement focalisée sur les entrées du contre-exemple. Une telle formulation obligerait à analyser les conséquences de sous-ensembles de valeurs d'entrée, quels que soient les chemins de propagation actifs. De plus, l'extraction des causes minimales ne prend pas en compte les hypothèses du modèle, pour ne pas avoir à raisonner sur les conséquences sémantiques des contraintes posées sur les entrées.

Il reste cependant des difficultés liées à la présence de *chemins reconvergents* de même ordre. Ces chemins introduisent des voies de propagation multiples entre une valeur d'entrée $E_i(n_i)_\sigma$ et la sortie $O(N)_\sigma$. Les dépendances sémantiques ainsi introduites rendent plus complexe la détermination des causes minimales. Notre discussion de ce problème a conclu qu'il était préférable de ne pas chercher à le traiter de façon exacte. On peut alors implémenter un algorithme performant, mais produisant des *causes* non *minimales* dans certains cas.

Le chapitre suivant présente un tel algorithme.

Chapitre 5

Algorithme d'analyse d'une violation

Sommaire

Dans le chapitre précédent, nous avons introduit la notion de *causes* d'une valuation d'une variable à un instant. Ce chapitre présente un algorithme que nous appellerons *algorithme d'analyse d'une violation* permettant de calculer des *causes* de la violation d'une propriété. Lorsque le modèle analysé sera sans chemins reconvergents de même ordre, les causes calculées seront *minimales*. L'algorithme intervient typiquement lors de l'analyse du contre-exemple renvoyé par un model checker. Cependant, il peut tout aussi bien être utilisé pour analyser les causes de la valuation d'une variable quelconque à un instant donné.

La section 5.1 détaille l'*algorithme d'analyse d'une valuation*. La section 5.2 démontre la terminaison de l'algorithme dans tous les cas. La section 5.3 établit un lien entre l'algorithme et la formalisation du chapitre 4, préparant ainsi les démonstrations que : (1) l'algorithme produit bien des *causes*, (2) ces causes sont *minimales* en l'absence de chemins reconvergents de même ordre. Seule la première démonstration sera donnée dans le corps du chapitre, dans la section 5.4. La deuxième démonstration, qui considère un cadre plus restreint d'utilisation de l'algorithme, pourra être trouvée en Annexe C. La section 5.5 conclut.

5.1 Algorithme d'analyse d'une valuation

Soit un modèle M et une propriété ψ tel que $M \not\models \psi$. Un model checker renvoie un contre-exemple σ de longueur N. L'algorithme a pour but d'identifier des *causes* de la violation de la propriété ψ. L'algorithme a deux paramètres d'entrée :

- $\alpha \in Arc$, l'arc par lequel l'analyse débute,
- $\mathbf{n} \in \mathbb{N}^*$, l'instant auquel l'analyse débute,

Le contre-exemple $\sigma \in \Sigma_{in}$ à analyser et le modèle $\mathbf{M}$ ne sont pas explicitement passés en paramètre. Ils sont stockés dans des structures de données globales. L'algorithme y accède au travers de fonctions utilitaires qui seront décrites dans la section 5.1.1.

Pour que l'*algorithme d'analyse d'une violation* identifie des *causes* de la violation de la propriété ψ, il est nécessaire d'initialiser l'algorithme avec l'arc de sortie de l'observateur de la propriété ψ à l'instant de violation N

L'algorithme est récursif. Il réalise un parcours de type *backward structurel* et *temporel*. Le parcours *structurel*, réalisé sur la structure du modèle, va de l'arc de départ α, vers les arcs d'entrée du modèle. Le parcours *temporel*, réalisé sur l'exécution du contre-exemple σ_{all} va de l'instant de départ n vers l'instant initial 1. Pour l'analyse d'un contre-exemple, α est l'arc de sortie du modèle [1] et n est l'instant de violation N [2].

L'algorithme est capable d'analyser des modèles contenant des opérateurs de base : IN, OR, AND, ITE, FBY, et des opérateurs de la *bibliothèque de symboles* Airbus : PULSE1, R*S, CONF1 et MTRIG1.

Il renvoie un ensemble E_c d'ensembles de *chemins datés*. Un *chemin daté* est une suite ordonnée de couples $(arc, instant)$, où $arc \in Arc$ et $instant \in \mathbb{N}^*$. Nous verrons ultérieurement que la datation des arcs des chemins est nécessaire pour traiter les symboles Airbus. Nous ferons le lien avec les chemins (non datés) considérés dans la formalisation du chapitre 4, et démontrerons (section 5.4) que le E_c renvoyé correspond à un ensemble de *causes*.

1. C'est à dire l'arc de sortie de l'observateur
2. C'est à dire le dernier instant du contre-exemple

5.1.1 Les fonctions utilitaires

L'algorithme s'appuie sur un ensemble d'utilitaires, récapitulés dans le tableau 5.1. Ces utilitaires relèvent de deux catégories. La première catégorie permet d'interroger les structures de données globales qui encodent le modèle M et les résultats de l'exécution du contre-exemple σ. La deuxième catégorie concerne la construction d'ensembles d'ensembles de chemins datés.

Certains utilitaires de la première catégorie correspondent à des fonctions déjà présentées dans la formalisation du chapitre 4 : *arcIn*, *arc2var* et $V(n)_\sigma$. Leur sens est rappelé dans le tableau 5.1. D'autres sont spécifiquement introduits pour l'algorithme : *from* et *getType*, *getParameter*. Nous les présentons ci-dessous.

Fonction	Résultat de la fonction
arcIn	renvoie l'ensemble des arcs d'entrée de l'opérateur passé en paramètre
arc2var	renvoie la variable associée à l'arc passé en paramètre
$V(n)_\sigma$	renvoie la valuation de la variable V à l'instant n dans le scénario σ
from	renvoie l'instance de l'opérateur d'où l'arc passé en paramètre sort
getType	renvoie le type de l'opérateur passé en paramètre
getParameter	renvoie l'ensemble des paramètres internes de l'opérateur passé en paramètre
concat	ajoute un arc daté à la fin d'un chemin daté
concatPath	ajoute un arc daté à la fin de chaque élément d'un ensemble de chemins datés
completeAllPath	ajoute un arc daté à la fin de chaque chemin daté d'un ensemble d'ensembles de chemins datés

TABLE 5.1 – Fonctions utilitaires de base

La fonction **from** renvoie l'instance de l'*opérateur* d'où l'*arc* passé en paramètre sort.

$$\text{from} : \begin{cases} Arc & \rightarrow & OP \\ \alpha & \mapsto & from(\alpha) \end{cases}$$

La fonction **getType** renvoie le type de l'*opérateur* passé en paramètre.

$$\text{getType} : \begin{cases} OP & \rightarrow & operatorType \\ operator & \mapsto & getType(operator) \end{cases}$$

Où *operatorType* est un ensemble contenant le type des *opérateurs* : $\{IN, OR, AND, ITE, FBY, PULSE1, R^*S, CONF1$ *et* $MTRIG1\}$.

La fonction **getParameter** renvoie l'ensemble des *paramètres internes* de l'opérateur passé en paramètre. Cette fonction est utilisée pour des opérateurs possédant des *paramètres internes* tels que les opérateurs de la *bibliothèque de symboles* Airbus. Dans tous les cas un *paramètre interne* est un entier.

$$\text{getParameter} : \begin{cases} OP & \rightarrow & \mathbb{N} \\ operator & \mapsto & arcIn(operator) \end{cases}$$

Par exemple, cette fonction permet de récupérer le temps de confirmation d'un opérateur CONF1 : $n_{conf} \in \mathbb{N}$.

Pour manipuler des (ensembles d'ensembles de) chemins datés, la fonction la plus simple est **concat** (voir l'algorithme ci-dessous). Cette fonction a trois paramètres d'entrée :

- Un *chemin daté* : p,
- un arc : $\alpha \in Arc$,
- un instant : $n \in \mathbb{N}^*$.

La fonction *concat* ajoute l'arc daté (α, n) au chemin daté p.

Algorithm 1 concat(p, α, n)

$p \leftarrow p$ avec (α, n) concaténé à la fin
return p

Par exemple, si $p =< (\alpha_1, n_1), (\alpha_2, n_2) >$ alors, $concat(p, \alpha_3, n_3)$ renvoie le chemin daté p telle que $p =< (\alpha_1, n_1), (\alpha_2, n_2), (\alpha_3, n_3) >$.

La fonction *concatPath* prend 3 paramètres :

- c un ensemble de chemins datés,
- $\alpha \in Arc$ un arc,
- $n \in \mathbb{N}^*$ un instant.

La fonction *concatPath* appelle la fonction **concat** sur tous les éléments de c (voir l'algorithme ci-après). Ainsi, elle concatène l'arc daté (α, n) à tous les chemins datés x présents dans l'ensemble c.

Algorithm 2 concatPath(c,α,n)

for all $x \in c$ **do**
  $x \leftarrow concat(x, \alpha, n)$
end for
return c

Par exemple, $concatPath(\{< (a,1),(b,1),(c,1) >, < (f,2) >\}, e, 2)$ renvoie l'ensemble $\{< (a,1),(b,1),(c,1),(e,2) >, < (f,2),(e,2) >\}$.

La fonction **completeAllPath**(E_c, α, n) s'applique à un ensemble d'ensembles de chemins datés. Elle complète tous les chemins datés de tous les ensembles c présents dans E_c avec l'arc daté (α, n).

Algorithm 3 completeAllPath(E_c, α, n)

Require: E_c ensemble d'ensembles de structures tel que renvoyé par analyzeIN.
  for all $c \in E_c$ **do**
    $c \leftarrow concatPath(c, \alpha, n)$
  end for
  return E_c

Par exemple completeAllPath($\{\{< (a,1),(b,1),(c,1) >, < (f,2) >\}, \{< (d,1) >\}\}, (e,2)$) renvoit l'ensemble : $\{\{< (a,1),(b,1),(c,1),(e,2) >, < (f,2),(e,2) >\}, \{< (d,1),(e,2) >\}\}$.

D'autres fonctions plus complexes ont été définies pour combiner deux ensembles d'ensembles de chemins datés. Nous les présenterons au moment de leur utilisation dans l'algorithme.

5.1.2 La fonction *analyze*

La fonction *analyze* est la fonction principale de l'algorithme. Cette fonction a les paramètres d'entrée (α, n), où α correspond à l'identifiant de l'arc dans le modèle et n est un instant dans l'exécution du scénario. La fonction *analyze* implémente l'algorithme 4.

Algorithm 4 analyze(α, n)

  $operator = from(\alpha)$
  $operatorType = getType(operator)$
  if operatorType = IN **then**
    $E_c \leftarrow$ analyzeIN(α, n) {Cas d'arrêt}
    ⋮
  else if operatorType = OR **then**
    $E_c \leftarrow$ analyzeOR(α, n)
  else if operatorType = AND **then**
    $E_c \leftarrow$ analyzeAND(α, n)
    ⋮
  else if operatorType = CONF1 **then**
    $E_c \leftarrow$ analyzeCONF1(α, n)
    ⋮
  end if
  return E_c

Dans un premier temps, la fonction identifie l'opérateur d'où l'arc provient en utilisant la fonction *from*. Dans un deuxième temps, la fonction identifie le type de l'opérateur en utilisant la fonction *getType*. Finalement, la fonction *analyze* appelle la *fonction adaptée* correspondant au type de l'opérateur. A l'exception de *analyseIN*, qui est le cas d'arrêt de l'algorithme, les *fonctions adaptées* appellent récursivement

la fonction *analyze* sur tout ou partie des arcs d'entrée de l'opérateur en cours d'analyse.

On distingue 9 fonctions adaptées aux 9 types d'opérateur : *analyzeIN*, *analyzeOR*, *analyzeAND*, *analyzeITE*, *analyzeFBY*, *analyzeCONF1*, *analyzePulse1*, *analyzeR*S*, *analyzeMTRIG1*. On appellera *analyzeOP* l'ensemble de ces 9 fonctions.

Nous allons maintenant décrire les six premières fonctions, pour montrer le traitement de tous les opérateurs de base ainsi qu'un exemple de traitement de symbole ($analyzeCONF1$). La description des fonctions restantes $analyzePulse1$, $analyzeR*S$ et $analyzeMTRIG1$ est donnée en annexe B.

5.1.2.1 AnalyzeIN

La fonction *analyzeIN* est appelée lorsque l'arc α sort d'un opérateur IN. Elle est la seule de l'ensemble $analyzeOP$ qui ne soit pas récursive. Cette fonction représente le **cas d'arrêt** de la récursion. Comme on peut le voir sur l'algorithme 5, elle renvoie l'ensemble d'un ensemble contenant le chemin daté $< (\alpha, n) >$, où n correspond à l'instant d'analyse courant. Cette structure basique $\{\{< (\alpha, n) >\}\}$ sera complétée lors des retours des appels récursifs, pour former des structures contenant des chemins de la forme $< (\alpha, n), \ldots >$.

Algorithm 5 analyzeIN(α, n)

return $\{\{< (\alpha, n) >\}\}$

5.1.2.2 AnalyzeNOT

La fonction $analyzeNOT$ implémente l'algorithme 6. Elle est invoquée lorsque l'arc α sort d'un opérateur NOT. Cette fonction concatène l'arc daté (α, n) à tous les chemins datés contenus dans tous les éléments du résultat de l'appel récursif à *analyze*. L'appel récursif a induit l'analyse de l'arc d'entrée α_{in} de l'opérateur, au même instant n que la sortie.

Algorithm 6 analyzeNOT(α, n)

$operator = from(\alpha)$
$\alpha_{in} \leftarrow arcIn(operator)$
$E_c \leftarrow completeAllPath(analyze(\alpha_{in}, n), \alpha, n)$
return E_c

5.1.2.3 AnalyzeFBY

La fonction $analyzeFBY$ implémente l'algorithme 7. Elle est invoquée lorsque l'arc α sort d'un opérateur FBY. L'analyse de l'opérateur FBY est légèrement plus compliquée que celle du NOT. On distingue le cas où l'analyse s'effectue à l'instant initial $n = 1$, du cas général où $n > 1$.

Dans le cas où $n = 1$, $analyzeFBY$ lance l'analyse de l'arc de l'entrée d'initialisation α_{init} à l'instant courant n. Dans le cas où $n > 1$, $analyzeFBY$ lance l'analyse de l'arc de l'entrée retardée α_{in} à l'instant précédent $n - 1$. Ensuite, la fonction concatène l'arc de sortie daté de l'opérateur (α, n) à tous les chemins contenus dans la structure de retour de l'appel récursif.

Intuitivement, on voit que la datation des arcs est liée à la notion d'*ordre* de chemin qui avait été définie dans le chapitre 4. La date $n - 1$ pour l'arc α_{in} indique un retard d'ordre 1 par rapport à la sortie. Plus généralement, un chemin daté de la forme $< (\alpha_1, n_1), \ldots, (\alpha_k, n_k) >$ correspond à un chemin d'ordre $n_k - n_1$.

Algorithm 7 analyzeFBY(α, n)

$operator = from(\alpha)$
$\alpha_{in}, \alpha_{init} \leftarrow arcIn(operator)$
if $n = 1$ **then**
  $E_c \leftarrow analyze(\alpha_{init}, n)$
else
  $E_c \leftarrow analyze(\alpha_{in}, n - 1)$
end if
$E_c \leftarrow completeAllPath(E_c, \alpha, n)$
return E_c

5.1.2.4 Combinaison de causes *dépendantes* et *indépendantes*

Comme nous l'avons vu, le traitement des opérateurs NOT et FBY lance une analyse récursive sur un unique arc d'entrée. Cependant, le traitement des autres opérateurs peut considérer des appels récursifs sur *plusieurs* arcs d'entrée. Nous avons donc besoin d'utilitaires pour combiner les structures résultantes, afin de construire un unique ensemble E_c. Ces utilitaires sont *combineCausesIndep* et *combineCausesDep*, correspondant respectivement à la combinaison de causes indépendantes et dépendantes.

Considérons par exemple l'opérateur OR représenté sur la figure 5.1. L'arc de sortie est noté α et les deux arcs d'entrées sont notés α_1 et α_2.

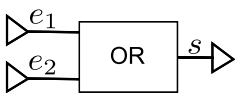


FIGURE 5.1 – L'opérateur OR

Pour illustrer la notion de causes *indépendantes* ou *dépendantes*, nous allons considérer que cet opérateur est noyé dans un modèle. Nous souhaitons comprendre la raison de la valuation de la sortie de l'opérateur à l'instant n. Dans le cas du OR, nous devons dissocier le raisonnement lorsque la sortie du OR est VRAI du raisonnement lorsque la sortie du OR à FAUX.

Causes indépendantes La valuation de la sortie du OR à **vrai** s'explique par la valuation d'une ou plusieurs entrées du OR à VRAI. L'algorithme récursif fournit l'analyse des *causes* des entrées du OR à VRAI. Dans le cas le plus général (deux entrées à VRAI), l'algorithme récursif fournit deux ensembles de *causes* notés E_1 et E_2. La *cause* de la valuation de la sortie du OR à VRAI est liée, soit à une *cause* de E_1, soit à une *cause* de E_2. L'algorithme va réunir des ensembles de causes **indépendantes**. La fonction *combineCausesIndep* implémente l'union de E_1 et E_2.

Algorithm 8 combineCausesIndep(E_1, E_2)

$E \leftarrow E_1 \cup E_2$
return E

Par exemple, si :

$$E_1 = \{\{(A,1),(B,1)\},\{(C,1),(D,1)\}\} \text{ et } E_2 = \{\{(E,1)\},\{(G,1)\}\}$$

alors $combineCausesIndep(E_1, E_2)$ renvoie :

$$\{\{(A,1),(B,1)\},\{(C,1),(D,1)\},\{(E,1)\},\{(G,1)\}\}$$

Causes dépendantes La valuation de la sortie du OR à **faux** ne saurait s'expliquer que par la valuation de toutes les entrées à FAUX. L'algorithme récursif fournit l'analyse des *causes* des entrées du OR à FAUX. On note les deux ensembles de *causes* des deux entrées : E_1 et E_2. La *cause* de la valuation de la sortie du OR à FAUX est liée à une *cause* de E_1 et à une *cause* de E_2, elles sont **dépendantes** l'une de l'autre. L'algorithme va effectuer la combinaison de toutes les *causes* de E_1 avec toutes les *causes* de E_2. La fonction **combineCausesDep** implémente cette *union combinatoire*.

Algorithm 9 combineCausesDep(E_1, E_2)

$E \leftarrow \{x / \exists e_1 \in E_1, \exists e_2 \in E_2, x = e_1 \cup e_2\}$
return E

Par exemple, si :

$$E_1 = \{\{(A,1),(B,1)\},\{(C,1),(D,1)\}\} \text{ et } E_2 = \{\{(E,1)\},\{(G,1)\}\}$$

alors $combineCausesDep(E_1, E_2)$ renverra :

$$\{\{(A,1),(B,1),(E,1)\},\{(C,1),(D,1),\\(E,1)\},\{(A,1),(B,1),(G,1)\},\{(C,1),(D,1),(G,1)\}\}$$

L'algorithme décidera du type de composition à appliquer en fonction de l'opérateur traité et de la valuation de la variable de sortie. Ce principe est mis en œuvre dans les fonctions d'analyse qui suivent.

5.1.2.5 AnalyzeOR

La fonction *analyzeOR* implémente l'algorithme 10. Elle est invoquée lorsque l'arc α sort d'un opérateur OR. La variable V_{out} est définie avec le flot associé à l'arc de sortie. La variable V_{in} est définie plusieurs fois avec les flots associés à tous les arcs d'entrée successivement. Le raisonnement de la fonction se décompose en deux parties :

si $V_{out}(n)_\sigma =$**FAUX** alors la valuation de la sortie s'explique par le fait que toutes les entrées de l'opérateur sont valuées à FAUX.

si $V_{out}(n)_\sigma =$**VRAI** alors la valuation de la sortie s'explique (de manière indépendante) par les entrées qui sont valuées à VRAI.

Algorithm 10 analyzeOR(α, n)

```
operator = from(α)
E_c ← ∅
V_out ← arc2var(α)
if V_out(n)_σ =FAUX then
  for all α' ∈ arcIn(operator) do
    E_c ← combineCausesDep(E_c, analyze(α', n))
  end for
else
  for all α' ∈ arcIn(operator) do
    V_in = arc2var(α')
    if V_in(n)_σ =VRAI) then
      E_c ← combineCausesIndep(E_c, analyze(α', n))
    end if
  end for
end if
E_c ← completeAllPath(E_c, α, n)
return E_c
```

5.1.2.6 AnalyzeAND

La fonction *analyzeAND* implémente l'algorithme 11. Elle est invoquée lorsque l'arc α sort d'un opérateur AND. La variable V_{out} est définie avec le flot associé à l'arc de sortie. La variable V_{in} est définie plusieurs fois avec les flots associés à tous les arcs d'entrée. Le raisonnement de la fonction se décompose en deux parties :

si $V_{out}(n)_\sigma =$**FAUX** alors la valuation de la sortie s'explique (de manière indépendante) par les entrées qui sont valuées à FAUX.

si $V_{out}(n)_\sigma =$**VRAI** alors la valuation de la sortie s'explique par le fait que toutes les entrées de l'opérateur sont valuées à VRAI.

Algorithm 11 analyzeAND(α, n)

```
operator = from(α)
E_c ← ∅
V_out ← arc2var(α)
if V_out(n)_σ =VRAI then
  for all α' ∈ arcIn(operator) do
    E_c ← combineCausesDep(E_c, analyze(α', n))
  end for
else
  for all α' ∈ arcIn(operator) do
    V_in = arc2var(α')
    if V_in(n)_σ =FAUX then
      E_c ← combineCausesindep(E_c, analyze(α', n))
    end if
  end for
end if
E_c ← completeAllPath(E_c, α, n)
return E_c
```

5.1.2.7 AnalyzeITE

La fonction *analyzeITE* implémente l'algorithme 12. Elle est invoquée lorsque l'arc α sort d'un opérateur ITE[3]. La variable V_{cond} est définie avec le flot associé à l'arc conditionnel. Le raisonnement de la fonction se décompose en deux parties. Lorsque la condition $V_{cond}(n)_\sigma$ est valuée à FAUX, l'analyse doit se poursuivre à l'instant n sur l'arc d'entrée α_1, sinon elle se poursuit sur α_2. Mais, dans tous les cas l'analyse doit aussi se poursuivre sur α_{cond} car la source de l'erreur peut également provenir du calcul de α_{cond}. Les résultats des analyses sur une des entrées classiques et sur l'entrée conditionnelle sont composés de manière dépendante.

Algorithm 12 analyzeITE(α, n)

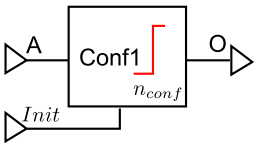
```
operator = from(α)
[α_1, α_2, α_cond] ← arcIn(operator)
V_cond ← arc2var(α_cond)
if V_cond(n)_σ=FAUX then
  E_c ← combineCausesDep(analyze(α_1, n), analyze(α_cond, n))
else
  E_c ← combineCausesDep(analyze(α_2, n), analyze(α_cond, n))
end if
E_c ← completeAllPath(E_c, α, n)
return E_c
```

3. If Then Else

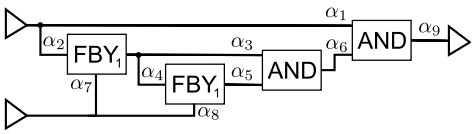
5.1.2.8 L'opérateur CONF1

L'opérateur CONF1 est, à la différence des opérateurs présentés précédemment, un opérateur de la *bibliothèque des symboles* Airbus. Dans le cadre de la thèse, nous avons considéré quatre opérateurs de la bibliothèque des symboles : CONF1, PULSE1, R*S et MTRIG1. Le détail de tous ces opérateurs et de leur fonction d'analyse associée serait fastidieux. Nous ne présentons ici que le cas du CONF1. Le détail des autres opérateurs est disponible dans l'annexe B.

Considérons le modèle constitué d'un opérateur CONF1, dont la représentation graphique est visible sur la figure 5.2(a). La figure 5.2(b) correspond à la description de référence de l'opérateur CONF1 lorsque le paramètre n_{conf} est 2. Cette description de référence est un modèle SCADE modélisant le comportement de l'opérateur CONF1 avec des opérateurs de base. On peut noter que la description de référence ne contient pas de chemins reconvergents de même ordre. Les arcs sont annotés de α_1 à α_9. Les arc α_1 et α_2 sont associés à la variable A et α_7 et α_8 sont associés à $Init$. L'arc α_9 est associé à la variable O.



(a) Représentation graphique de CONF1



(b) Modèle de référence du CONF1 avec n_{conf}=2

FIGURE 5.2 – Opérateur CONF1

Le paramètre $n_{conf} \in \mathbb{N}^+$ est le temps de confirmation. Par définition, la variable d'entrée A doit rester VRAI durant $n_{conf}+1$ cycles consécutifs pour que la variable d'entrée soit dit confirmée sur n_{conf} cycles. A partir de cet instant, la sortie O est VRAI tant que A est VRAI. La sortie du confirmateur repasse à FAUX dès que A repasse à FAUX. L'entrée A perd alors la confirmation. L'exécution suivante permet d'illustrer le comportement de l'opérateur CONF1 avec n_{conf}=2 :

n	1	2	3	4	5	6	7	8	9
$A(n)$	0	1	0	1	1	1	1	0	1
$O(n)$	0	0	0	0	0	1	1	0	0

On observe que le signal d'entrée A est VRAI 3 cycles consécutifs aux instants 4, 5 et 6. Le signal de sortie O, traduisant la confirmation de A, est donc valué à VRAI à l'instant 6. Il reste VRAI tant que A reste VRAI, c'est à dire jusqu'à l'instant 7.

5.1.2.9 analyzeCONF1

Nous allons maintenant présenter la fonction $analyzeCONF1$, qui implémente l'algorithme 13. Les variables V_{in}, V_{out} et V_{init} sont respectivement définies avec les flots associés aux arcs d'entrée, de sortie et d'initialisation. Le paramètre n_{conf} est récupéré grâce à la fonction *getParameter*.

Le raisonnement de l'analyse peut se décomposer en deux parties :

1. l'instant d'analyse n est inférieur au temps de confirmation n_{conf},
2. l'instant d'analyse n est strictement supérieur au temps de confirmation n_{conf}.

Dans le premier cas, il sera nécessaire d'analyser l'entrée d'initialisation : la fenêtre temporelle d'analyse sera $n' = [1, n]$ sur l'entrée α_{in} et $n' = 1$ sur l'entrée α_{init}. Dans le deuxième cas, l'entrée d'initialisation ne sera pas analysée, la fenêtre temporelle sera $n' = [n - n_{conf}, n]$.

Dans chacune des deux parties, l'analyse se décompose de nouveau en deux sous-parties :

– la sortie du confirmateur est VRAI : $V_{out}(n)_\sigma$=VRAI,
– la sortie du confirmateur est fausse : $V_{out}(n)_\sigma$=FAUX.

Dans le premier cas, la fonction appelle récursivement $analyze(\alpha_{in}, n')$ sur tous les instants n' et compose les résultats de chaque appel de manière **dépendante**. Ce choix s'explique par le fait que toutes les valuations d'entrée présentent dans la fenêtre temporelle $n' = [1, n]$ justifient la confirmation de l'entrée à l'instant n.

Dans le deuxième cas, la fonction appelle récursivement $analyze(\alpha_{in}, n')$ sur tous les instant n' pour lesquels $V_{in}(n')_\sigma$=FAUX et compose les résultats de chaque appel de manière **indépendante**. Ce choix s'explique par le fait que chaque valuation d'entrée à FAUX présente dans la fenêtre temporelle $n' = [1, n]$ suffit à justifier la non confirmation de l'entrée à l'instant n.

Une fois l'ensemble E_c construit, la fonction appelle *completeAllPath* afin d'ajouter l'arc (α, n) à la fin de chaque chemin daté de E_c.

5.1.3 Exemple combinatoire

Après cette présentation détaillée des différentes fonctions d'analyse des opérateurs, nous allons illustrer le fonctionnement de l'algorithme complet sur un exemple simple. Considérons l'exemple de la figure 5.3. Soit le contre-exemple : <A,B,C>=<VRAI, VRAI, VRAI>.

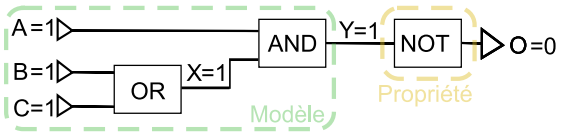


FIGURE 5.3 – Un exemple combinatoire

La figure 5.4 illustre l'arbre d'exécution de l'algorithme pour cet exemple. Nous adoptons la convention que la référence d'un arc est le nom de la variable qu'il porte, écrit en minuscule. Par exemple, o est la référence de l'arc portant O. L'algorithme est initialisé avec l'arc o à l'instant 1.

Les feuilles de l'arbre d'exécution sont toutes des appels à *analyzeIN*, produisant les structures $\{\{< (a, 1) >\}\}$, $\{\{< (b, 1) >\}\}$ et $\{\{< (c, 1) >\}\}$.

Au niveau de $AnalyzeOR(x, 1)$, les causes sont indépendantes, donnant un ensemble de deux ensembles, chacun contenant un unique chemin daté :

Algorithm 13 analyzeCONF1(α, n)

$operator = from(\alpha)$
$n_{conf} \leftarrow getParameter(operator)$
$[\alpha_{in}, \alpha_{init}] \leftarrow arcIn(operator)$
$E_c \leftarrow \emptyset$
$V_{out} = arc2var(\alpha)$
$V_{in} = arc2var(\alpha_{in})$
$V_{init} = arc2var(\alpha_{init})$
if $n \leq n_{conf}$ **then**
  if $V_{out}(n)_\sigma$ =VRAI **then**
    for all $n' \in [1, n]$ **do**
      $E_c \leftarrow$combineCausesDep($E_c, analyze(\alpha_{in}, n')$)
    end for
    $E_c \leftarrow$combineCausesDep($E_c, analyze(\alpha_{init}, 1)$)
  else
    $\{V_{out}(n)_\sigma$ =FAUX$\}$
    for all $n' \in [1, n]$ **do**
      if $V_{in}(n')_\sigma$ =FAUX **then**
        $E_c \leftarrow$combineCausesIndep($E_c, analyze(\alpha_{in}, n')$)
      end if
    end for
    if $V_{init}(1)_\sigma$ =FAUX **then**
      $E_c \leftarrow$combineCausesIndep($E_c, analyze(\alpha_{init}, 1)$)
    end if
  end if
else
  if $V_{out}(n)_\sigma$=VRAI **then**
    for all $n' \in [n - n_{conf}, n]$ **do**
      $E_c \leftarrow$combineCausesDep($E_c, analyze(\alpha_{in}, n')$)
    end for
  else
    $\{V_{out}(n)_\sigma$=FAUX$\}$
    for all $n' \in [n - n_{conf}, n]$ **do**
      if $V_{in}(n')_\sigma$ =FAUX **then**
        $E_c \leftarrow$combineCausesIndep($E_c, analyze(\alpha_{in}, n')$)
      end if
    end for
  end if
end if
$E_c \leftarrow completeAllPath(E_c, \alpha, n)$
return E_c

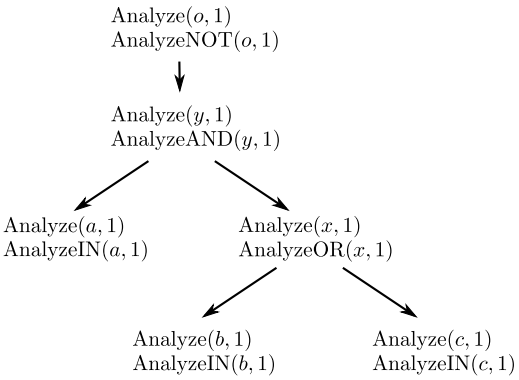


FIGURE 5.4 – Arbre d'exécution de l'algorithme pour l'exemple 1

$$\{\{< (b,1),(x,1) >\},\{< (c,1),(x,1) >\}\}$$

Au niveau de $AnalyzeAND(y,1)$, les causes sont dépendantes, donnant un ensemble de deux ensembles, chacun contenant deux chemins datés :

$$\{\{< (b,1),(x,1),(y,1) >,< (a,1),(y,1) >\},\{< (c,1),(x,1),(y,1) >,< (a,1),(y,1) >\}\}$$

Au niveau de $AnalyzeNOT(o,1)$, on ne fait que concaténer l'arc $(o,1)$ à chaque chemin de la structure produite par $AnalyzeAND(y,1)$:

$$\{\{< (b,1),(x,1),(y,1),(o,1) >,< (a,1),(y,1),(o,1) >\},\{< (c,1),(x,1),(y,1),(o,1) >,< (a,1),(y,1),(o,1) >\}\}$$

Au final, on a trouvé deux causes :
- une cause contenant le chemin de A vers O, et le chemin de B vers O,
- une cause contenant le chemin de A vers O, et le chemin de C vers O.

Ce sont d'ailleurs des causes minimales, car il n'y a pas de chemins reconvergents de même ordre dans le modèle.

5.2 Terminaison de l'algorithme

Dans cette section, nous allons démontrer que l'algorithme se termine dans tous les cas. Pour cela, nous poserons deux hypothèses réalistes :
- le nombre d'arcs et d'opérateurs d'un modèle est fini,
- les contre-exemples que nous considérons sont de longueur finie.

Notre méthode de démonstration utilise un ordre strict bien fondé. Nous allons associer une mesure entière ≥ 0, ou *variant*, aux paramètres de chaque appel récursif. Soit $variant(\alpha, n)$ cette mesure. Nous montrerons que chaque appel récursif depuis une fonction de $AnalyzeOP$ diminue strictement le variant [4]. Comme il n'existe

4. On peut ignorer les appels depuis *Analyze*. Cette fonction ne fait que transmettre ses paramètres d'entrée par un unique appel à une fonction de $AnalyzeOP$

pas de suite infinie d'entiers strictement décroissante, nous aurons alors montré qu'il n'existe pas non plus de suite infinie d'appels récursifs. L'arbre d'exécution est de profondeur finie, l'algorithme termine.

Le variant choisi est basé à la fois sur la date n, et sur le *niveau* de l'arc α. Le calcul du niveau d'un arc est défini en introduisant la notion de *modèle aplati*.

5.2.1 Notion de modèle aplati M_{flat}

Soit M un modèle composé d'opérateurs de base et d'*opérateurs de la bibliothèque* Airbus. Chaque opérateur de la bibliothèque Airbus possède une *description de référence*.

Définition 22 (description de référence) *Les descriptions de référence modélisent en SCADE le comportement des symboles de la bibliothèque Airbus. Ces modélisations sont composées uniquement d'opérateurs de base et ne contiennent pas de chemins reconvergents de même ordre.*

La description de référence de l'opérateur CONF1 est disponible dans la section 5.1.2.8. Les descriptions de référence des opérateurs PULSE1, R*S, et MTRIG1 sont disponibles en annexe B.

A partir de M et des descriptions de référence des symboles, on peut alors construire un modèle M_{flat} qui ne contient que des opérateurs de base.

Définition 23 (Modèle aplati) *Le modèle aplati M_{flat} du modèle M est un modèle composé uniquement d'opérateurs de base tel que :*

- *les opérateurs de la bibliothèque Airbus sont substitués par leur description de référence.*
- *Les opérateurs de base demeurent inchangés.*

Ces deux notions sont illustrées sur l'exemple qui suit. Considérons le modèle M de la figure 5.5. Ce modèle contient un opérateur de la *bibliothèque de symbole* Airbus : CONF1, et deux opérateurs de base : AND et OR.

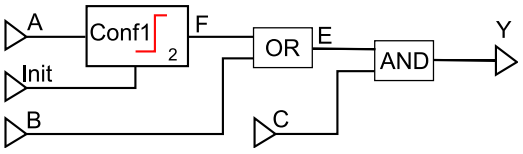


FIGURE 5.5 – Exemple de modèle M

Considérons le modèle M_{flat} de la figure 5.6. Ce modèle correspond au modèle M dans lequel l'opérateur CONF1 a été substitué par sa description de référence. Les deux modèles sont sémantiquement équivalents.

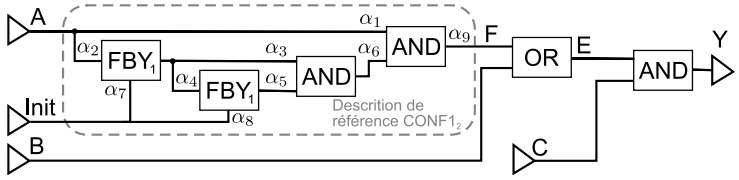


FIGURE 5.6 – Exemple de *modèle aplati* M_{flat} de M

5.2.2 Notion de niveau d'un arc

Cette notion a été introduite dans le domaine du test matériel [Abramovici 1990] afin de calculer la distance d'un arc par rapport aux entrées. Pour déterminer le niveau des arcs, on réalise un parcours en largeur du modèle M_{flat}, en allant des entrées vers les sorties. Pour chaque arc, on applique la fonction *niveau* qui à un *arc* associe un entier représentant son niveau :

$$\text{niveau} : \begin{cases} Arc & \rightarrow & \mathbb{N} \\ \alpha & \mapsto & \begin{cases} 0 \text{ si } \alpha \text{ sort d'un opérateur d'entrée} \\ niveau(\text{entrée d'init}) + 1 \text{ si } \alpha \text{ sort d'un opérateur FBY} \\ (\underset{\alpha' \in \text{prédécesseur}(\alpha)}{MAX} \text{niveau}(\alpha')) + 1 \text{ sinon} \end{cases} \end{cases}$$

Les niveaux d'arcs du modèle M sont les mêmes que ceux du modèle aplati M_{flat}. Les niveaux des arcs internes aux descriptions de référence ne sont pas considérés.

Pour illustrer ce calcul, considérons sur la figure 5.7 le modèle aplati M_{flat} correspondant au modèle M de la figure 5.9 dont le niveau des arcs a été annoté, en rouge. A partir de ce calcul sur M_{flat}, on peut déduire le niveau des arcs de M. Par exemple, les entrées du confirmateur ont le niveau 0, la sortie du confirmateur a le niveau 3, et la sortie finale de M a le niveau 5.

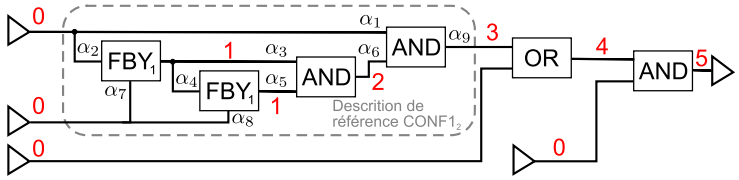


FIGURE 5.7 – Illustration des niveaux d'arcs

Comme on peut le voir sur la figure 5.8, la notion de niveau d'arc est aussi définie en présence de rebouclages dans le modèle. Le principe est de couper la boucle au niveau de l'entrée retardée (ici, l'arc α_5), et de ne considérer que le niveau de l'entrée d'initialisation : $niveau(\alpha_6) = niveau(\alpha_8) + 1$.

Pour un modèle fini, le niveau des arcs est borné.

5.2.3 Variant associé aux paramètres des appels récursifs

Nous allons maintenant définir la fonction *variant*.

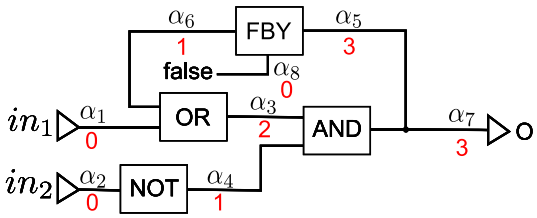


FIGURE 5.8 – Illustration des niveaux d'arcs avec un rebouclage

$$\text{variant} : \begin{cases} Arc \times \mathbb{N}^* & \rightarrow & \mathbb{N} \\ (\alpha, n) & \mapsto & \text{variant}(\alpha, n) \end{cases}$$

Soit α et n les paramètres d'appel transmis par la fonction *analyze*, soit

$$niveauMax = \underset{\alpha' \in Arc}{MAX}(\text{niveau}(\alpha'))$$

le niveau maximum d'un arc dans le modèle. Le variant est le suivant :

$$\text{variant}(\alpha, n) = [\text{niveauMax} \times (n-1)] + \text{niveau}(\alpha)$$

Remarquons que $0 \leq niveau(\alpha) \leq niveauMax$, et $1 \leq n \leq N$ pour un scénario de longueur N. On a donc toujours $0 \leq variant(\alpha, n) \leq niveauMax \times (N)$. On a plusieurs manières de faire décroître strictement le variant :

- **Diminuer strictement la date d'analyse.** Même si le niveau de l'arc augmente, le variant diminue. Considérons le pire cas. L'opérateur en cours de traitement a un arc de sortie de niveau 1 (c'est la possibilité la plus basse, car si l'arc était de niveau 0, il n'y aurait pas d'appel récursif). L'instant courant d'analyse est n. On fait alors un appel récursif avec les paramètres $(\alpha', n-1)$, tels que $niveau(\alpha') = niveauMax$.

$$\begin{array}{rcl} niveauMax \times (n-1) & < & niveauMax \times (n-1) + 1 \\ \text{variant}(\alpha', n-1) & < & \text{variant}(\alpha, n) \end{array}$$

- **Diminuer strictement le niveau de l'arc, sans augmenter la date d'analyse**. Considérons le pire cas : le temps reste inchangé, le niveau de l'arc décroît de 1.

$$\begin{array}{rcl} niveauMax \times (n-1) + (niveau(\alpha) - 1) & < & niveauMax \times (n-1) + niveau(\alpha) \\ \text{variant}(\alpha', n) & < & \text{variant}(\alpha, n) \end{array}$$

Dans notre algorithme, tous les appels récursifs depuis une fonction d'analyse d'opérateur diminuent strictement le variant. Le traitement des opérateurs de base atemporels (AND, OR, NOT, ITE) génère un ou deux appels qui diminuent le niveau d'arc sans changer la date d'analyse. Le traitement du FBY fait un appel qui soit décrémente la date de 1, soit garde la même date mais décrémente le niveau d'arc

de 1. Les appels depuis la fonction d'analyse du CONF1 sont en nombre fini, et décrémentent le niveau d'arc de $n_{conf} + 1$, soit en laissant la date inchangée soit en diminuant la date. On pourra vérifier que la décroissance stricte du variant se produit aussi depuis les autres symboles de la bibliothèque Airbus que nous traitons (voir annexe B).

L'analyse termine donc toujours.

5.3 Liens entre l'algorithme et la formalisation

Il est important de bien comprendre la différence entre la formalisation élaborée dans le chapitre 4 et l'algorithme introduit dans ce chapitre :

- D'un côté, l'algorithme considère des chemins datés dans un modèle, que l'on notera M, pouvant contenir des symboles de la bibliothèque Airbus.
- D'un autre côté, la formalisation du chapitre 4 traite des chemins non datés dans un modèle ne comportant que des opérateurs de base.

Pour faire le lien entre ces deux contextes, nous allons utiliser le modèle aplati M_{flat} défini précédemment, dans lequel les symboles de la bibliothèque Airbus sont remplacés par leur description de référence.

Le modèle M et son modèle aplati M_{flat} sont sémantiquement équivalents. Toutes les opérations de construction de chemins datés de l'algorithme sur M ont une interprétation déterministe en termes d'opérations de construction de chemins sur M_{flat}.

Prouver que ce que produit l'algorithme dans M correspond à une cause de la formalisation du chapitre 4 revient alors à prouver que les opérations équivalentes sur M_{flat} produisent des causes.

Caractéristiques de M	Caractéristiques de M_{flat}
Composé de tous types d'opérateur	Composé uniquement d'opérateurs de base
L'algorithme calcule un ensemble d'ensembles de chemins datés	L'algorithme calcule un ensemble d'ensembles de chemins
Les fonctions utilisées par l'algorithme sont décrites dans la section 5.1	Les fonctions sur M sont transposées sur M_{flat}. Par convention, leur nom est identique et accompagné du mot clé $_{flat}$.

Nous allons maintenant présenter les différentes étapes réalisées par l'algorithme, et montrer les étapes équivalentes sur le modèle M_{flat}.

5.3.1 Étape 1 : *identification des chemins locaux actifs*

Une fonction d'analyse appropriée est appelée, selon l'opérateur *op* qui produit la variable associée à l'arc α passé en paramètre de la fonction *analyze*. La première étape d'analyse met alors en jeu un raisonnement local au niveau de *op*. L'explication de cette étape nécessite d'introduire la notion de *modèle local*.

Définition 24 (Modèle local) *Le modèle local $ML(\alpha, n)$ est une restriction du modèle M au seul opérateur op d'où α sort. Tous les arcs d'entrée sont conservés et connectés à des opérateurs d'entrée. L'arc α est connecté à un opérateur de sortie. Les arcs et leur variable associée sont identiques à ceux du modèle M. Les valuations des variables d'entrée respectent l'exécution σ.*

Le modèle local ML_{flat} correspond à la description de référence de *op*. Si *op* est une opérateur de base alors $ML_{flat} = ML$.

Opérations de l'algorithme sur ML	Opérations équivalentes sur ML_{flat}
Identification de couples (α_i, n_i) qui sont des origines des chemins locaux $< (\alpha_i, n_i), (\alpha, n) >$ au niveau de l'opérateur en cours d'analyse ML	Identification de chemins p_i locaux au niveau de l'opérateur en cours d'analyse ML_{flat} tels que $naissance(p_i) = \alpha_i$ et $ordre(p_i) = n - n_i$

Dans cette étape, l'idée est d'identifier des chemins de propagation locaux, des entrées de l'opérateur vers sa sortie. Ces chemins vont permettre d'expliquer la valuation de la variable associé à l'arc de sortie α à l'instant n. Dans ML_{flat}, les éventuels retards subis lors de la propagation d'une entrée vers la sortie apparaissent explicitement dans la structure interne (par exemple, opérateurs FBY dans la description du CONF1). Dans ML, il n'y a pas de structure interne, et les retards sont représentés par les dates n_i associées aux entrées. Comme la description de référence ne comporte pas de chemins reconvergents de même ordre, chaque couple (α_i, n_i) calculé par l'algorithme sur ML correspond de manière unique à un chemin p_i, d'ordre $n - n_i$, reliant α_i à α dans ML_{flat}.

5.3.2 Étape 2 : *choix du type de composition des* $analyze(\alpha_i, n_i)$

Cette étape décide du type de *composition* à appliquer : *dépendante* ou *indépendante*. Par exemple, dans l'analyse d'un OR, si l'étape précédente avait identifié deux couples (α_i, n_i) correspondant à des entrées à VRAI, la composition sera indépendante. Nous démontrerons dans la section 5.4.4 que chaque élément de l'ensemble calculé sur ML est une *cause* dans ML_{flat}.

Opérations de l'algorithme sur ML	Opérations équivalentes sur ML_{flat}
Choix du type de composition pour produire un ensemble d'ensembles de chemins datés sur ML	Choix du type de composition pour produire un ensemble d'ensembles de chemins sur ML_{flat}

La composition choisie sera appliquée à l'étape 4 aux résultats des analyses récursives de l'étape 3

5.3.3 Étape 3 : *analyse récursive des chemins actifs*

Cette étape consiste à appeler récursivement la fonction *analyze* sur les différents couples (α_i, n_i) identifiés à l'étape 1.

Opérations de l'algorithme sur M	Opérations équivalentes sur M_{flat}
Appel récursif de la fonction *analyze* avec chaque couple (α_i, n_i) identifié à l'étape 1 Noté : $E_c = analyze(\alpha_i, n_i)$ Le résultat E_c est un ensemble d'ensembles de chemins datés sur M Tous les chemins se terminent par l'arc daté (α_i, n_i)	Analyse récursive qui construit l'analogue de E_c avec des chemins dans M_{flat} à l'instant n_i Noté : $E_c = analyze_{flat}(\alpha_i, n_i)$ Le résultat E_c est un ensemble d'ensembles de chemins sur M_{flat} Tous les chemins se terminent par l'arc α_i

Nous démontrerons dans la section 4.4 que l'ensemble d'ensembles de chemins datés issus de la fonction $analyze_{flat}$ sont des *causes* de la variable associée à α_i à l'instant n dans M_{flat}.

5.3.4 Étape 4 : *composition des résultats de tous les* $analyze(\alpha_i, n_i)$

Cette étape compose les ensembles d'ensembles résultant des appels récursifs de l'étape 3 selon le type de composition décidé à l'étape 2.

Opérations de l'algorithme sur M	Opérations équivalentes sur M_{flat}
Composition des ensembles d'ensembles de chemins datés telle que définie à l'étape 2 Noté : $CombineCausesDep(E_1, E_2)$ ou $CombineCausesIndep(E_1, E_2)$	Composition des ensembles d'ensembles de chemins telle que définie à l'étape 2 Noté : $CombineCausesDep_{flat}(E_1, E_2)$ ou $CombineCausesIndep_{flat}(E_1, E_2)$

Nous rappelons que $CombineCausesDep(E_1, E_2) = \{x / \exists e_1 \in E_1, \exists e_2 \in E_2, x = e_1 \cup e_2\}$ et $CombineCausesIndep(E_1, E_2) = E_1 \cup E_2$ sont deux fonctions qui travaillent sur des ensembles d'ensembles sans tenir compte du type d'élément. Les fonctions $CombineCausesDep_{flat}$ et $CombineCausesIndep_{flat}$ peuvent donc être décrites de la même manière.

5.3.5 Étape 5 : *concaténation des chemins avec le suffixe local*

Cette étape termine en concaténant tous les chemins de chaque ensemble issus de l'étape 4 par le dernier morceau de chemin calculé à l'étape 1.

Opérations de l'algorithme sur M	Opérations équivalentes sur M_{flat}
Concaténation de l'arc daté (α, n) à chaque ensemble d'ensembles de chemins datés se terminant par un arc daté (α_i, n_i)	Concaténation des chemins locaux p_i de ML_{flat} avec les chemins calculés récursivement sur M_{flat} .
Noté : $completeAllPath(E_c, \alpha, n)$ qui fait appel à $concatPath(\ldots, \alpha, n)$ qui fait appel à $concat(\ldots, \alpha, n)$	Noté : $completeAllPath_{flat}(E_c, p_i)$ qui fait appel à $concatPath_{flat}(\ldots, p_i)$ qui fait appel à $concat_{flat}(\ldots, p_i)$
Tous les *chemins datés* de tous les ensembles se terminent par l'arc daté (α, n)	Tous les *chemins* de tous les ensembles se terminent par l'arc α

5.3.6 Remarques

La fonction *concat* de l'algorithme qui concatène des chemins datés est modélisée par une fonction $concat_{flat}$ qui concatène des chemins. La fonction $concat_{flat}(p_1, p_2)$ n'est définie que si $mort(p_1) = naissance(p_2)$.

Les fonctions $analyzeNOT$ et $analyzeFBY$ sont des cas particuliers où les étapes 2 et 4 n'ont pas lieu d'être dans la mesure où il n'y a qu'un seul appel récursif. La fonction $analyzeIN$, le cas d'arrêt de l'algorithme, est un cas particulier où seule l'étape 1 est utilisée.

Une dernière remarque concerne les étapes 4 et 5 qui peuvent être permutées sans changer la sémantique de l'algorithme.

$$CompleteAllPath(CombineCausesDep(E_1, E_2), \alpha, n)$$
$$\equiv$$
$$CombineCausesDep(CompleteAllPath(E_1, \alpha, n), CompleteAllPath(E_2, \alpha, n))$$

Cette remarque s'applique aussi pour la fonction $CombineCausesIndep$ et pour les fonctions $_{flat}$. Les démonstrations de la section 5.4 s'appuient sur cette remarque afin d'être simplifiées. Nous allons maintenant illustrer ces cinq étapes sur un exemple. Pour chaque étape, nous détaillerons en parallèle les opérations sur M et les opérations équivalentes sur M_flat.

5.3.7 Exemple détaillé

Considérons le *modèle à analyser* M de la figure 5.9 dont la variable de sortie est notée Y. Les entrées du modèles sont notées A, B, C et $Init$. E et F sont deux variables intermédiaires. Les arcs portent le nom des variables en minuscule. Soit la propriété : $\forall n \in \mathbb{N}^*, Y(n) =$ FAUX. Pour simplifier l'exemple, nous n'avons pas représenté l'*observateur de la propriété* $NOT(Y)$ qui ne présente pas d'intérêt pédagogique. Dans les tableaux et les figures, les valeurs de vérité VRAI et FAUX sont représentées par des 1 et des 0. Imaginons le contre-exemple σ, de longueur 3 cycles, suivant :

n	1	2	3
A	1	1	1
B	0	0	1
C	0	0	1
$Init$	0	0	0

σ est un contre-exemple car la valeur de Y est VRAI uniquement à l'instant 3, la sortie O de l'observateur est donc fausse uniquement à l'instant 3. La valuation des variables est annotée sur les arcs du modèle M qui leur sont associés.

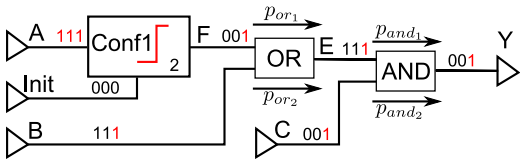


FIGURE 5.9 – Exemple illustratif

La définition de l'opérateur CONF1 avec $n_{conf} = 2$ est disponible dans la section 5.1.2.8. La figure 5.10 représente le modèle aplati M_{flat} du modèle M.

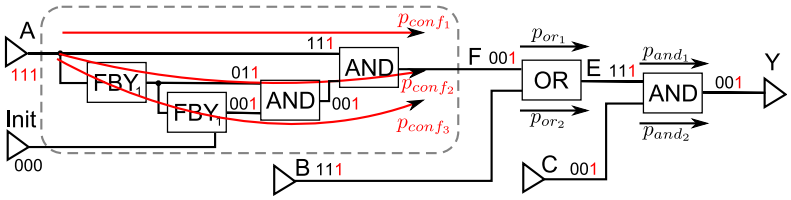


FIGURE 5.10 – Calculs à l'intérieur de l'opérateur CONF1

Considérons, dans l'opérateur CONF1 de M_{flat}, trois chemins : p_{conf_1}, p_{conf_2}, p_{conf_3}. Nous ne considérons pas les chemins partant de l'entrée $Init$ des opérateurs FBY car ils n'ont pas d'intérêt dans le scénario étudié.

Ces chemins sont respectivement modélisés par les suites d'arcs datés : <(a,1),(f,3)>, <(a,2),(f,3)>, <(a,3),(f,3)> dans M. Cette correspondance est unique si et seulement si la description de références des *opérateurs de la bibliothèque* Airbus ne comportent pas de *chemins reconvergents de même ordre*. Dans le cas contraire, une suite d'arcs datés aurait pu être associée à deux chemins internes.

Ainsi, le modèle M_{flat} de la figure 5.10 possède cinq *chemins complets* que l'on nommera $p_1 =< p_{conf_1}, p_{or_1}, p_{and_1} >$, $p_2 =< p_{conf_2}, p_{or_1}, p_{and_1} >$, $p_3 =< p_{conf_3}, p_{or_1}, p_{and_1} >$, $p_4 =< p_{or_2}, p_{and_1} >$ et $p_5 =< p_{and_2} >$.

La fonction *analyze*, initialisé avec $analyze(y, 3)$, parcourt la structure en appelant récursivement la fonction *analyze*. Ceci a pour effet de déplier un arbre d'exécution qui est représenté sur la figure 5.11. Les appels récursifs s'arrêtent aux entrées du modèle. Le résultat est construit à la fin de l'exécution des fonctions. Il est établi dans l'ordre inverse de l'ordre d'appel : des entrées du modèle vers les sorties.

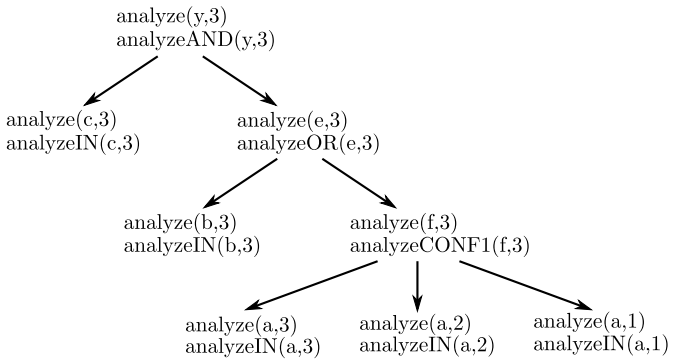


FIGURE 5.11 – Arbre d'appels récursifs des fonctions *analyze*

Nous allons décrire les objets construits par les fonction *analyze* et $analyze_{flat}$ à chaque étape de l'arbre d'exécution.

5.3.7.1 Les appels à *analyzeIN*

Dans un premier temps, nous allons présenter le résultat des fonctions $analyzeIN$ puis nous remonterons l'arbre d'appel.

Chaque appel de la fonction $analyzeIN$ renvoie un ensemble contenant un ensemble, lui-même contenant un chemin daté composé d'un unique arc daté (α, n). Regardons le cas de $analyzeIN(b, 3)$. Le résultat de la fonction est $\{\{< (b, 3) >\}\}$. Le chemin correspondant dans M_{flat}, renvoyé par $analyzeIN_{flat}$, est le chemin formé par l'unique arc $a : \{\{a\}\}$, qui est toujours actif quelque soit la valeur de sa variable associée. Ceci est dû à la définition de la *condition d'activation*, pour les chemins constitués d'un seul arc. Il en est de même pour chaque appel de la fonction $analyzeIN$.

5.3.7.2 L'appel à *analyzeCONF1*

Le tableau ci-dessous décrit le comportement de la fonction $analyzeCONF1(f, 3)$, dans M pour la partie de gauche, $analyzeCONF1_{flat}(f, 3)$ dans M_{flat} pour la partie de droite. Les étapes 1 et 2 sont réalisées avant l'appel récursif des trois fonctions $analyze(a, 1)$, $analyze(a, 2)$, $analyze(a, 3)$. Les étapes 3, 4 et 5 utilisent les résultats de ces fonctions pour construire le résultat de la fonction $analyzeCONF1$.

Étape	Opérations dans M	Opérations dans M_{flat}
1	$\{(a,1),(a,2),(a,3)\}$	$\{p_{conf_1}, p_{conf_2}, p_{conf_3}\}$
2	CombineCausesDep	$\{\{p_{conf_1}, p_{conf_2}, p_{conf_3}\}\}$
3	$\{\{< (a,1) >\}\}$	$\{\{< a >\}\}$
	$\{\{< (a,2) >\}\}$	$\{\{< a >\}\}$
	$\{\{< (a,3) >\}\}$	$\{\{< a >\}\}$
4	$\{\{< (a,1) >, < (a,2) >, < (a,3) >\}\}$	$\{\{< a >, < a >, < a >\}\}$
5	$\{\{< (a,1),(f,3) >, < (a,2),(f,3) >,$ $< (a,3),(f,3) >\}\}$	$\{\{p_{conf_1}, p_{conf_2}, p_{conf_3}\}\}$

On observe à l'étape 3 le résultat des appels récursifs à $analyzeIN_{flat}$: le chemin constitué de l'unique arc a et l'arc daté $(a, 1)$ pour $analyze$. On observe à l'étape 4 le résultat de la fonction $combineCausesDep$ avec en paramètre les résultats de l'étape 3. On précise que, dans le modèle M, p_{conf_1} correspond au résultat de $concat_{flat}(< a >, p_{conf_1})$. Il en est de même pour p_{conf_2} et p_{conf_3}.

5.3.7.3 L'appel à *analyzeOR*

Le tableau ci-dessous décrit le comportement de la fonction $analyzeOR(e, 3)$, dans M pour la partie de gauche, $analyzeOR_{flat}(e, 3)$ dans M_{flat} pour la partie de droite. Les étapes 1 et 2 sont réalisées avant l'appel récursif des deux fonctions $analyze(f, 3)$, $analyze(b, 3)$. Les étapes 3, 4 et 5 utilisent les résultats de ces fonctions pour construire le résultat de la fonction $analyzeOR$.

Étape	Opérations dans M	Opérations dans M_{flat}
1	$\{(f, 3), (b, 3)\}$	$\{p_{or_1}, p_{or_2}\}$
2	CombineCausesIndep	$\{\{p_{or_1}, p_{or_2}\}\}$
3	$\{\{< (a, 1), (f, 3) >, < (a, 2), (f, 3) >$ $, < (a, 3), (f, 3) >\}\}$	$\{\{p_{conf_1}, p_{conf_2}, p_{conf_3}\}\}$
	$\{\{< (b, 3) >\}\}$	$\{\{< b >\}\}$
4	$\{\{< (a, 1), (f, 3) >,$ $< (a, 2), (f, 3) >, < (a, 3), (f, 3) >\},$ $\{< (b, 3) >\}\}$	$\{\{p_{conf_1}, p_{conf_2}, p_{conf_3}\},$ $\{< b >\}\}$
5	$\{\{< (a, 1), (f, 3), (e, 3) >,$ $< (a, 2), (f, 3), (e, 3) >,$ $< (a, 3), (f, 3), (e, 3) >\},$ $\{< (b, 3), (e, 3) >\}\}$	$\{\{concat_{flat}(p_{conf_1}, p_{or_1}),$ $concat_{flat}(p_{conf_2}, p_{or_1}),$ $concat_{flat}(p_{conf_3}, p_{or_1})\},$ $\{p_{or_2}\}\}$

On observe à l'étape 3 le résultat des appels récursifs à :

- $analyzeCONF1$
- $analyzeIN_{flat}$: le chemin constitué de l'unique arc b.

On observe à l'étape 4 le résultat de la fonction $combineCausesIndep$ avec en paramètre les résultats de l'étape 3.

5.3.7.4 L'appel à *analyzeAND*

Le tableau ci-dessous décrit le comportement de la fonction $analyzeAND(y, 3)$, dans M pour la partie de gauche, $analyzeAND_{flat}(y, 3)$ dans M_{flat} pour la partie de droite. Les étapes 1 et 2 sont réalisées avant l'appel récursif des deux fonctions $analyze(c, 3)$, $analyze(e, 3)$. Les étapes 3, 4 et 5 utilisent les résultats de ces fonctions pour construire le résultat de la fonction $analyzeAND$.

Étape	Opérations dans M	Opérations dans M_{flat}
1	$\{(e,3),(c,3)\}$	$\{p_{and_1}, p_{and_2}\}$
2	CombineCausesDep	$\{\{p_{and_1}, p_{and_2}\}\}$
3	résultat de l'étape 5 de $analyzeOR(e,3)$	$\{\{< p_{conf_1}, p_{or_1} >,$ $< p_{conf_2}, p_{or_1} >,$ $< p_{conf_3}, p_{or_1} >\}, \{p_{or_2}\}\}$
	$\{\{< (c,3) >\}\}$	$\{\{< c >\}\}$
4	$\{\{< (a,1),(f,3),(e,3) >,$ $< (a,2),(f,3),(e,3) >,$ $< (a,3),(f,3),(e,3) >,$ $< (c,3) >\},$ $\{< (b,3),(e,3) >, < (c,3) >\}\}$	$\{\{< p_{conf_1}, p_{or_1} >,$ $< p_{conf_2}, p_{or_1} >,$ $< p_{conf_3}, p_{or_1} >,$ $< c >\},$ $\{p_{or_2}, < c >\}\}$
5	$\{\{< (a,1),(f,3),(e,3),(y,3) >,$ $< (a,2),(f,3),(e,3),(y,3) >,$ $< (a,3),(f,3),(e,3),(y,3) >,$ $< (c,3),(y,3) >\},$ $\{< (b,3),(e,3),(y,3) >, < (c,3),(y,3) >\}\}$	$\{\{p_1, p_2, p_3, p_5\},$ $\{p_4, p_5\}\}$

On observe à l'étape 4 le résultat de la fonction *combineCausesDep* avec en paramètre les résultats de l'étape 3. L'algorithme renvoie le résultat de l'étape 4 de la fonction $analyzeAND(y,3)$. On rappelle que, dans le modèle M_{flat}, p_1 est le résultat de l'opération $concat_{flat}(< p_{conf_1}, p_{or_1} >, < p_{and_1} >)$, p_2 est le résultat de l'opération $concat_{flat}(< p_{conf_2}, p_{or_1} >, < p_{and_1} >)$, p_3 est le résultat de l'opération $concat_{flat}(< p_{conf_3}, p_{or_1} >, < p_{and_1} >)$, p_4 est le résultat de l'opération $concat_{flat}(< p_{or_2} >, < p_{and_1} >)$, et p_5 est le résultat de l'opération $concat_{flat}(< c >, < p_{and_2} >)$.

5.4 Correction partielle : Génération de causes

Cette section a pour but de démontrer que la fonction *analyze* calcule des causes. Dans la mesure où *analyze* considère des *chemins datés* et des *opérateurs de la bibliothèque* Airbus et que la notion de *cause* est définie avec des *chemins* sur des modèles ne contenant que des opérateurs de base, nous démontrerons que *analyze* calcule l'équivalent de *causes*.

Soit un modèle M et un scénario σ de longueur $N \in \mathbb{N}^*$. Soit α un arc du modèle, V sa variable associée et $n \in \mathbb{N}^*$ tel que $n \leq N$.

Théorème 1 *La fonction $analyze(\alpha, n)$ calcule l'équivalent de causes de la valuation de V à l'instant n, où $V = arc2var(\alpha)$.*

Il est important de noter que le raisonnement que nous avons choisi pour démontrer ce théorème se base sur le modèle aplati M_{flat}. Cette transposition permet de faire le lien avec les définitions de la formalisation du chapitre 4. Ainsi, le raisonnement portera sur des *chemins* et non sur des *chemins datés*. Nous démontrerons donc que la fonction $analyze_{flat}$ calcule des *causes* sur M_{flat} et nous déduirons ensuite que la fonction *analyze* calcule l'équivalent de *causes* sur le modèle M.

5.4.1 Méthode de démonstration

La démonstration du théorème 1 est une démonstration par induction sur l'arbre d'exécution de la fonction *analyze* :

Cas de base Les appels à $analyzeIN$ renvoient des *causes* ;

Cas d'induction Les appels aux fonctions de *analyzeOP* renvoient des *causes* si les appels récursifs à $analyze_{flat}(\alpha_i, n_i)$ renvoient des *causes* de la valuation de V_i à l'instant n_i, où $V_i = arc2var(\alpha_i)$.

La démonstration se base sur l'*hypothèse locale* que l'étape 1 : le *calcul des causes locales* p_i et l'étape 2 : *choix du mode de composition* calculent des *causes* du modèle local $ML_{flat}(\alpha)$. Nous démontrerons dans la section 5.4.4 que, pour chaque opérateur, cette hypothèse est juste. Le cas de base est alors trivialement satisfait. Le cas d'induction est un peu plus complexe. Considérons un élément e de $analyze_{flat}(\alpha, n)$. Le raisonnement peut se décomposer en deux parties :

1. dans le cas où e est le résultat d'une *composition indépendante* des appels récursifs (section 5.4.2),
2. dans le cas où e est le résultat d'une *composition dépendante* (section 5.4.3).

Pour chacune des deux parties, la démonstration se décompose en trois étapes. Il faut démontrer que, **conformément à la définition 18 :**

1. $e \subseteq P_e(\sigma, n)$;
2. $\forall p \in e, mort(p) = \alpha$;
3. soit $Origines = \{(V_i, n_i) \in Var_{in} \times \mathbb{N}^* / \exists p \in e, (V_i, n_i) = origine(p, n)\}$.
   $\forall \sigma' \in \Sigma_{in}$ tel que σ' est de même longueur que σ. $(V_i(n_i)_{\sigma'} = V_i(n_i)_\sigma)$ pour tout $(V_i, n_i) \in Origines$, alors :
   (a) $\forall p \in e, \mathcal{A}(p, \sigma', N) =$VRAI
   (b) $V(n)_{\sigma'} = V(n)_\sigma$

5.4.2 Correction partielle dans le cas *indépendant*

Dans cette section, on considère que l'étape 2 a décidé que les causes locales étaient *indépendantes*. Soit $\mathcal{C}_{min}$ l'ensemble des causes locales minimales (α, n) de ML_{flat} identifiées aux étapes 1 et 2.

On suppose que $\mathcal{C}_{min}$ est indexé sur $[1, J]$, où l'indice $j \in [1, J]$ détermine de manière unique une cause locale $\{p_j\}$. On notera $\alpha_j = naissance(p_j)$ et $n_j = n - ordre(p_j)$. Dans le cas où $J = 2$, les étapes 3, 4 et 5 peuvent se synthétiser de la manière suivante :

$$\begin{aligned} analyze_{flat}(\alpha, n) = combineCausesIndep_{flat}(& \\ & completeAllPath_{flat}(analyze_{flat}(\alpha_1, n_1), p_1), \\ & completeAllPath_{flat}(analyze_{flat}(\alpha_2, n_2), p_2)) \end{aligned}$$

avec $\{p_1\}$ et $\{p_2\}$ deux *causes* de la valuation de V ($V = arc2var(\alpha)$) à l'instant n dans $ML_{flat}(\alpha)$ pour l'exécution du scénario σ. Et $naissance(p_1) = \alpha_1$,

$naissance(p_2) = \alpha_2$, $n_1 = n - ordre(p_1)$ et $n_2 = n - ordre(p_2)$. Nous rappelons que les étapes 4 et 5 sont permutables.

Soit e un élément de $analyze_{flat}(\alpha, n)$. Démontrons que e est une *cause* de la valuation V à l'instant n dans le scénario σ. La première remarque que l'on peut faire, par définition de la fonction $combineCausesIndep_{flat}$, est :

$$\exists j \in [1, J]/e \in completeAllPath_{flat}(analyze_{flat}(\alpha_j, n_j), p_j)$$

Soit le chemin $p \in e$, alors $\exists e' \in analyze_{flat}(\alpha_j, n_j)$ et $\exists p' \in e'$ tels que :

$$p = concat_{flat}(p', p_j)$$

1. Par *hypothèse d'induction*, e' est une *cause* de (α_j, n_j) et donc $p' \in P_e(\sigma, n_j)$. Par définition, p' est activé par σ à l'instant n_j :

   $$\mathcal{A}(p', \sigma, n_j) = \text{VRAI}$$

   D'après l'*hypothèse locale*, p_j est activé par σ à l'instant n :

   $$\mathcal{A}(p_j, \sigma, n) = \text{VRAI}$$

   La *condition d'activation* de $p = concat_{flat}(p', p_j)$ est égale à :

   $$\mathcal{AC}(p) = \mathcal{AC}(p') \wedge \mathcal{AC}(p_j)$$

   D'où :

   $$\mathcal{A}(p, \sigma, n) = \text{VRAI}$$

   Donc p est activé par σ à l'instant n : $p \in P_e(\sigma, n)$

2. On rappelle que $p = concat_{flat}(p', p_j)$ et $mort(p_j) = \alpha$ car p_j est un *chemin complet* de $ML_{flat}(\alpha)$ donc $\mathbf{mort(p)} = \alpha$.

3. Soit $Origines = \{(V_i, n_i) \in Var_{in} \times \mathbb{N}^* / \exists p \in e, (V_i, n_i) = origine(p, n)\}$
   $\forall \sigma' \in \Sigma_{in}$ tel que $size(\sigma') = size(\sigma)$ et $(V_i(n_i)_{\sigma'} = V_i(n_i)_\sigma)$ pour tout $(V_i, n_i) \in Origines$ alors :
   par *hypothèse d'induction*, e' est une *cause* de $V_j = arc2var(\alpha_j)$ à l'instant n_j.
   Soit $Origines' = \{(V_i, n_i) \in Var_{in} \times \mathbb{N}^* / \exists p' \in e', (V_i, n_i) = origine(p', n)\}$.
   On remarquera que dans le cas *indépendant*, $Origines = Origines'$ ce qui implique que σ' garantit que :

   $$A(p', \sigma', n') = \text{VRAI et } V_j(n_j)_{\sigma'} = V_j(n_j)_\sigma$$

   Par *hypothèse locale*, dans le cas *indépendant*, le fait que $V_j(n_j)_{\sigma'} = V_j(n_j)_\sigma$ implique que :

   (a) $A(p_j, \sigma', n)$=VRAI, ainsi la *condition d'activation* de $p = concat_{flat}(p', p_j)$ donne :

   $$\mathcal{A}(p, \sigma', n) = \text{VRAI}$$

   Alors p est activé par σ' à l'instant n.

(b) $V_j(n_j)_{\sigma'} = V_j(n_j)_\sigma$ implique que :

$$\mathbf{V}(\mathbf{n})_{\sigma'} = \mathbf{V}(\mathbf{n})_\sigma$$

En conclusion de quoi, chaque élément e de $analyze_{flat}(\alpha, n)$ est une *cause* de V à l'instant n sur le scénario σ pour le modèle M_{flat}. Ainsi, nous en déduisons que $analyze_{flat}$ calcule des *causes* et par similarité, *analyze* calcule l'équivalent de *causes*.

5.4.3 Correction partielle dans le cas *dépendant*

Dans cette section, on considère que l'étape 2 a décidé que les causes locales étaient *dépendantes*. Considérons $\{p_1, \ldots, p_j\}$, la *cause locale* de $ML_{flat}(\alpha)$ identifiées à l'étape 1 et 2. Dans le cas où $J = 2$, les étapes 3, 4 et 5 peuvent se synthétiser de la manière suivante :

$$\begin{aligned} analyze_{flat}(\alpha, n) = {} & combineCausesDep_{flat}(\\ & completeAllPath_{flat}(analyze_{flat}(\alpha_1, n_1), p_1), \\ & completeAllPath_{flat}(analyze_{flat}(\alpha_2, n_2), p_2)) \end{aligned}$$

avec $\{p_1, p_2\}$ est une *cause* de la valuation de V ($V = arc2var(\alpha)$) à l'instant n dans $ML_{flat}(\alpha)$ pour l'exécution du scénario σ. Et $naissance(p_1) = \alpha_1$, $naissance(p_2) = \alpha_2$, $n_1 = n - ordre(p_1)$ et $n_2 = n - ordre(p_2)$. Nous rappelons que les étapes 4 et 5 sont permutables.

Soit $e =$ un élément de $analyze_{flat}(\alpha, n)$. Démontrons que e est une *cause* de la valuation V à l'instant n dans le scénario σ. Tous chemin $p \in e$ a été construit à partir d'un chemin local p_j et d'une cause $e'_j \in analyze_{flat}(\alpha_n, n_j)$. Ce chemin est de la forme :

$$p = concat(p'_j, p_j)/p'_j \in e'_j$$

e'_j est une cause des valuations de (α_j, n_j) donc par *hypothèse d'induction* :

1. $e'_j \subseteq P_e(\sigma, n_j)$ donc :

$$\forall p'_j \in e'_j, \mathcal{A}(p'_j, \sigma, n_j) = \text{VRAI}$$

Par *hypothèse locale*, p_j est activé par σ à l'instant n :

$$\mathcal{A}(p_j, \sigma, n) = \text{VRAI}$$

$\forall p \in e, p = concat_{flat}(p'_j, p_j)$, la condition d'activation de p est égale à :

$$\mathcal{AC}(p) = \mathcal{AC}(p'_j) \wedge \mathcal{AC}(p_j)$$

d'où :

$$\mathcal{A}(p, \sigma, n) = \text{VRAI}$$

2. $\forall p \in e, p = concat_{flat}(p'_j, p_j)$ et $mort(p_j) = \alpha$ car p_j est un chemin complet de $ML_{flat}(\alpha)$ donc $mort(p) = \alpha$.

3. Soit $Origines = \{(V_i, n_i) \in Var_{in} \times \mathbb{N}^* / \exists p \in e, (V_i, n_i) = origine(p, n)\}$.
   $\forall \sigma' \in \Sigma_{in}$ tel que $size(\sigma') = size(\sigma)$ et $(V_i(n_i)_{\sigma'} = V_i(n_i)_\sigma)$ pour tout $(V_i, n_i) \in Origines$.

   Par *hypothèse d'induction*, e'_j est une *cause* de α_j, n_j.
   Soit $Origines'_j = \{(V_i, n_i) \in Var_{in} \times \mathbb{N}^* / \exists p'_j \in e'_j, (V_i, n_i) = origine(p'_j, n_j)\}$.

   On remarquera que dans le cas *dépendant*, $Origines'_j \subseteq Origines$ d'où σ' garantit que :

   $$[(A(p'_j, \sigma', n_j) = \text{VRAI}) \Rightarrow (A(p'_j, \sigma, n_j) = \text{VRAI})] \text{ et } [V_j(n_j)_{\sigma'} = V_j(n_j)_\sigma]$$

   (a) $A(p'_j, \sigma', n_j)$=VRAI, ainsi la *condition d'activation* de $p = concat_{flat}(p', p_j)$ donne :

   $$\mathcal{A}(p, \sigma', n) = \text{VRAI}$$

   Alors p est activé par σ' à l'instant n.

   (b) $V_j(n_j)_{\sigma'} = V_j(n_j)_\sigma$ d'où :

   $$\mathbf{V(n)}_{\sigma'} = \mathbf{V(n)}_\sigma$$

En conclusion de quoi, chaque élément e de $analyze_{flat}(\alpha, n)$ est une *cause* de V à l'instant n sur le scénario σ pour le modèle M_{flat}. Ainsi, nous en déduisons que $analyze_{flat}$ calcule des *causes* et par similarité, *analyze* calcule l'équivalent de *causes*.

5.4.4 Correction partielle au niveau des opérateurs

Dans cette section, nous allons démontrer que la fonction $analyze_{flat}$ calcule **toutes** les **causes minimales** d'un modèle local ML_{flat} quelque soit le scénario σ considéré.

5.4.4.1 Analyse d'un opérateur IN

Considérons la fonction $analyze_{flat}(\alpha, n)$ lorsque ML_{flat} est composé d'un opérateur IN comme représenté sur la figure 5.12. L'arc reliant l'opérateur d'entrée à l'opérateur de sortie est α et la variable associée est V.

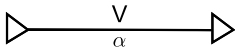


FIGURE 5.12 – ML_{flat} composé d'un opérateur IN

Dans ce cas, il existe est un unique chemin p de longueur $K = 1$ tel que $p =< \alpha >$. $analyze_{flat}$ renvoie l'ensemble $e = \{\{p\}\}$ qui contient la totalité des **causes minimales** car :

1. $p \in P_e(\sigma, n)$ est toujours actif par définition : si $K = 1$ alors $\mathcal{AC}(p)$=VRAI ;
2. $mort(p) = \alpha$ car $p =< \alpha >$;
3. Soit $Origines = \{(V, n) \in Var_{in} \times \mathbb{N}^* / \exists p \in e, (V, n) = origine(p, n)\}$.
   $\forall \sigma' \in \Sigma_{in}$ tel que $size(\sigma') = size(\sigma)$ et $(V(n)_{\sigma'} = V(n)_\sigma)$ pour tout $(V, n) \in Origines$ alors :
   (a) $(A(p, \sigma', n)$=VRAI) par définition de $\mathcal{AC}(p)$;
   (b) $V(n)_{\sigma'} = V(n)_\sigma$ car (V, n) est unique donc $\sigma' = \sigma$.

L'ensemble $\{\{p\}\}$ contient l'unique **cause minimale** car : $\{p\}$ est unique et car $\emptyset$ n'est pas une cause. La démonstration est identique pour une constante. $analyze(\alpha, n)$ renvoie **toutes les causes minimales** lorsqu'il analyse un opérateur d'entrée.

5.4.4.2 Analyse d'un opérateur NOT

Considérons la fonction $analyze_{flat}(\alpha, n)$ lorsque ML_{flat} est composé d'un opérateur NOT comme représenté sur la figure 5.13. L'arc entrant est α_i, sa variable associée est V_i, et l'arc sortant est *alpha* et sa variable associée est V.

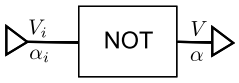


FIGURE 5.13 – ML_{flat} composé d'un opérateur NOT

Dans ce cas, il existe un unique chemin p de longueur $K = 2$ tel que $p =< \alpha_i, \alpha >$. L'algorithme renvoie l'ensemble $\{\{p\}\}$ qui contient la totalité des **causes** car :

1. $p \in P_e(\sigma, n)$ est toujours actif par définition de la condition d'activation de l'opérateur NOT :

$$\mathcal{AC}(p) = \mathcal{OC}(\alpha_i, \alpha) = \mathcal{AC}(< \alpha_i >) = \text{VRAI}$$

2. $mort(p) = \alpha$ car $p =< \alpha_i, \alpha >$;
3. Soit $Origines = \{(V, n) \in Var_{in} \times \mathbb{N}^* / \exists p \in e, (V, n) = origine(p, n)\}$.
   $\forall \sigma' \in \Sigma_{in}$ tel que $size(\sigma') = size(\sigma)$ et $(V(n)_{\sigma'} = V(n)_\sigma)$ pour tout $(V, n) \in Origines$ alors :
   (a) $(A(p, \sigma', n)$=VRAI) par définition de $\mathcal{AC}(p)$;
   (b) $V(n)_{\sigma'} = V(n)_\sigma$ car (V, n) est unique donc $\sigma' = \sigma$

L'ensemble $\{\{p\}\}$ contient l'unique **cause minimale** car : $\{p\}$ est unique et car $\emptyset$ n'est pas une *cause*. Donc, la fonction $analyze(\alpha, n)$ renvoie **toutes les causes minimales** lorsqu'elle analyse un opérateur NOT.

5.4.4.3 Analyse d'un opérateur FBY

Considérons la fonction $analyze_{flat}(\alpha, n)$ lorsque ML_{flat} est composé d'un opérateur FBY comme représenté sur la figure 5.14. L'arc de l'entrée retardée est α_i, sa variable associée est V_i, l'arc de l'entrée d'initialisation est α_{init} et sa variable associée V_{init}. L'arc sortant est *alpha* et sa variable associée est V.

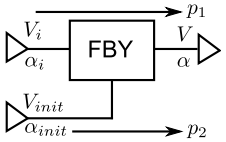


FIGURE 5.14 – ML_{flat} composé d'un opérateur FBY

Il existe deux chemins $p_1 =< \alpha_i, \alpha >$ et $p_2 =< \alpha_{init}, \alpha >$ de longueur $K = 2$. Il y a deux cas de figure, en fonction de l'instant d'analyse n :

– si $n = 1$ alors $analyze_{flat}(\alpha, n)$ renvoie l'ensemble $\{\{p_2\}\}$,

– si $n > 1$ alors $analyze_{flat}(\alpha, n)$ renvoie l'ensemble $\{\{p_1\}\}$.

Le principe de fonctionnement de l'opérateur est décrit au travers du tableau 5.2. La ligne n représente le cycle. La ligne V représente la valuation que prend la variable de sortie $V(n)$. Finalement, R représente le résultat de la fonction $analyze_{flat}(\alpha, n)$.

n	1	2	3	...
V	Init(1)	A(1)	A(2)	...
R	$\{\{p_2\}\}$	$\{\{p_1\}\}$	$\{\{p_1\}\}$	...

TABLE 5.2 – Principe de fonctionnement de l'opérateur FBY

Par définition (15), les conditions d'activation de p_1 et p_2 sont :

$$\mathcal{AC}(p_1) = FBY(\mathcal{AC}(< \alpha_i >), 1, \text{FAUX})$$
$$\mathcal{AC}(p_2) = FBY(\text{FAUX}, 1, \mathcal{AC}(< \alpha_{init} >))$$

Or α_i et α_{init} sont des arcs d'entrée donc $\mathcal{AC}(< \alpha_i >) = \mathcal{AC}(< \alpha_{init} >) =$VRAI. Il est alors possible d'écrire :

$$\mathcal{AC}(p_1) = FBY(\text{VRAI}, 1, \text{FAUX})$$
$$\mathcal{AC}(p_2) = FBY(\text{FAUX}, 1, \text{VRAI})$$

En d'autres termes :

– si $n = 1$, $\mathcal{AC}(p_1) =$FAUX et $\mathcal{AC}(p_2)=$VRAI

– si $n > 1$, $\mathcal{AC}(p_1) =$VRAI et $\mathcal{AC}(p_2)=$FAUX

La démonstration se décompose en deux parties.

Dans le cas où n = 1 , on démontre que $\{p_2\}$ est l'unique cause minimale :

1. $p_2 \in P_e(\sigma, 1) = \{p_2\}$

2. $mort(p_2) = \alpha$
3. Soit $Origines = \{(V, n) \in Var_{in} \times \mathbb{N}^* / \exists p \in e, (V, n) = origine(p, n)\}$. $\forall \sigma' \in \Sigma_{in}$ tel que $size(\sigma') = size(\sigma)$ et $(V(n)_{\sigma'} = V(n)_{\sigma})$ pour tout $(V, n) \in Origines$ alors :
   (a) $\forall \sigma', A(p_2, \sigma', 1) =$VRAI ;
   (b) $\forall \sigma', V(1)_{\sigma} = V_{init}(1)_{\sigma} = V(1)_{sigma'} = V_{init}(1)_{sigma'}$.

Donc $\{p_2\}$ est une *cause*. De plus, $\forall \sigma, AC(p_1, \sigma, 1) =$FAUX donc p_2 est l'unique chemin activable lorsque $n = 1$, impliquant que $\{p_2\}$ l'unique *cause minimale* lorsque $n = 1$.

Dans le cas où n > 1

1. $p_1 \in P_e(\sigma, 1) = \{p_1\}$
2. $mort(p_1) = \alpha$
3. Soit $Origines = \{(V, n) \in Var_{in} \times \mathbb{N}^* / \exists p \in e, (V, n) = origine(p, n)\}$. $\forall \sigma' \in \Sigma_{in}$ tel que $size(\sigma') = size(\sigma)$ et $(V(n)_{\sigma'} = V(n)_{\sigma})$ pour tout $(V, n) \in Origines$ alors :
   (a) $\forall \sigma', A(p_1, \sigma', n > 1) =$VRAI ;
   (b) $\forall \sigma', V(n)_{\sigma} = V_i(n)_{\sigma} = V(n)_{sigma'} = V_i(n)_{sigma'}$.

Donc $\{p_1\}$ est une *cause*. De plus, $\forall \sigma, AC(p_2, \sigma, n) =$FAUX donc p_1 est l'unique chemin activable lorsque $n > 1$, impliquant que $\{p_1\}$ l'unique *cause minimale* lorsque $n > 1$.

Donc, la fonction $analyze(\alpha, n)$ renvoie **toutes les causes minimales** lorsqu'elle analyse un opérateur FBY.

Remarque : pour les fonctions *analyze* des opérateurs IN, NOT et FBY, nous avons énoncé ce que renvoyé les fonctions $analyze_{flat}$ et nous avons démontrer que ce résultat contenait toutes les *causes minimales*. Pour les opérateurs qui suivent, nous procéderons d'une manière plus légère. Nous énoncerons les *causes minimales* et nous démontrerons que la fonction *analyze* les calcules.

5.4.4.4 Analyse d'un opérateur OU

Considérons la fonction $analyze_{flat}(\alpha, n)$ lorsque ML_{flat} est composé d'un opérateur OR comme représenté sur la figure 5.14. Les arcs d'entrée sont α_1 et α_2, et leur variable associée sont respectivement V_1 et V_2. L'arc sortant est *alpha* et sa variable associée est V.

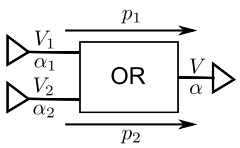


FIGURE 5.15 – ML_{flat} composé d'un opérateur OR

Soient les chemins $p_1 =< \alpha_1, \alpha >$ et $p_2 =< \alpha_2, \alpha >$ et leur condition d'activation respective : $\mathcal{AC}(p_1) = V_1 \vee \neg V_2$ et $\mathcal{AC}(p_2) = V_2 \vee \neg V_1$.

Pour démontrer que $analyzeOR(\alpha, n)$ renvoie toutes les *causes minimales*, nous allons considérer tous les scénarios possibles de longueur 1 : s_1 à s_4 comme représenté dans le tableau 5.3. Le scénario s_1 est un contre-exemple de la propriété $\forall n, V(n)$=VRAI. Les scénarios s_2 à s_4 sont des contre-exemples de la propriété $\forall n, V(n)$=FAUX. Pour chaque scénario, nous avons représenté, tel que défini dans la formalisation du chapitre 4, $P_e(s_i, n)$ ($i \in [1, 4]$) l'ensemble des chemins de propagation, C l'ensemble des *causes*, et C_{min} l'ensemble des *causes minimales*.

Scénario	V_1	V_2	V	$P_e(s_i, n)$	C	C_{min}
s_1	FAUX	FAUX	FAUX	$\{p_1, p_2\}$	$\{p_1, p_2\}$	$\{p_1, p_2\}$
s_2	FAUX	VRAI	VRAI	$\{p_2\}$	$\{p_2\}$	$\{p_2\}$
s_3	VRAI	FAUX	VRAI	$\{p_1\}$	$\{p_1\}$	$\{p_1\}$
s_4	VRAI	VRAI	VRAI	$\{p_1, p_2\}$	$\{p_1\}, \{p_2\}, \{p_1, p_2\}$	$\{p_1\}, \{p_2\}$

TABLE 5.3 – Scénario

On peut vérifier que dans tous ces cas, la fonction $analyzeOR$ renvoie toutes les *causes minimales*.

Trace de l'algorithme pour le scénario s_1 La sortie de l'opérateur étant FAUX, $analyzeOR(\alpha, n)$ passe dans la première branche de la condition et combine les éléments de manière **dépendante**. Ci-dessous, on représente à gauche l'opération effectuée par $analyzeOR(\alpha, n)$ et à droite son résultat.

$E_c \leftarrow \{\}$	$E_c = \{\}$
$E_c \leftarrow combineCausesDep(E_c, \{\{(\alpha_1, n)\}\})$	$E_c = \{\{(\alpha_1, n)\}\}$
$E_c \leftarrow combineCausesDep(E_c, \{\{(\alpha_2, n)\}\})$	$E_c = \{\{(\alpha_1, n), (\alpha_2, n)\}\}$

Dans ce cas, le résultat de $analyzeOR(\alpha, n)$ modélise l'ensemble $\{\{p_1, p_2\}\}$ qui est l'unique **cause minimale** telle que définie dans la formalisation du chapitre 4.

Trace de l'algorithme pour le scénario s_2 et s_3 La sortie de l'opérateur étant FAUX, $analyzeOR(\alpha, n)$ passe dans la deuxième branche de la condition et combine les éléments de manière **indépendante**. Ci-dessous, on représente à gauche l'opération effectuée par $analyzeOR(\alpha, n)$ et à droite son résultat. Dans ce cas, le

$E_c \leftarrow \{\}$	$E_c = \{\}$
$E_c \leftarrow combineCausesIndep(E_c, \{\{(\alpha_2, n)\}\})$	$E_c = \{\{(\alpha_2, n)\}\}$

résultat de $analyzeOR(\alpha, n)$ modélise l'ensemble $\{\{p_2\}\}$ qui est l'unique **cause minimale** telle que définie dans la formalisation du chapitre 4. La démonstration dans le cas du scénario s_3 est identique, $analyzeOR(\alpha, n)$ renvoie $E_c = \{\{(\alpha_1, n)\}\}$

Trace de l'algorithme pour le scénario s_4 La sortie de l'opérateur étant FAUX, $analyzeOR(\alpha, n)$ passe dans la deuxième branche de la condition et combine les éléments de manière **indépendante**. Ci-dessous, on représente à gauche l'opération effectuée par l'algorithme et à droite son résultat. Dans ce cas, le résultat de

$E_c \leftarrow \{\}$	$E_c = \{\}$
$E_c \leftarrow combineCausesIndep(E_c, \{\{(\alpha_1, n)\}\})$	$E_c = \{\{(\alpha_1, n)\}\}$
$E_c \leftarrow combineCausesIndep(E_c, \{\{(\alpha_2, n)\}\})$	$E_c = \{\{(\alpha_1, n)\}, \{(\alpha_2, n)\}\}$

$analyzeOR(\alpha, n)$ modélise l'ensemble $\{\{p_1\}, \{p_2\}\}$ renvoyé par $analyzeOR_{flat}(\alpha, n)$ qui est l'unique **cause minimale** telle que définie dans la formalisation du chapitre 4.

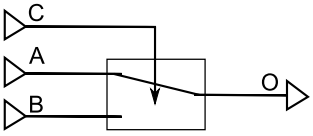
Nous avons démontré que pour tous les scénarios de longueur $n = 1$, $analyzeOR$ calcule **toutes les causes minimales**. Nous en déduisons que ce résultat est VRAI $\forall n \in \mathbb{N}^*$ dans la mesure où le temps n'intervient pas dans l'analyse.

5.4.4.5 Analyse d'un opérateur ET

La démonstration, lorsque le ML_{flat} est composé d'un opérateur AND, est similaire à la démonstration lorsque le modèle est composé d'un opérateur OR.

5.4.4.6 Analyse d'un opérateur ITE

Considérons le modèle réduit à un opérateur ITE tel que représenté sur la figure ci-dessous. Ce dernier possède trois entrées A et B et C et une sortie O. Si l'entrée conditionnelle C est FAUX, O prends la valeur de A, de B sinon. On note les arcs comme les variables en minuscule. Soit les chemins $p_1 =< a, o >$, $p_2 =< b, o >$ et $p_3 =< c, o >$. Leurs conditions d'activations respectives sont : $\mathcal{AC}(p_1) = \neg C$, $\mathcal{AC}(p_2) = C$ et $\mathcal{AC}(p_3) =$VRAI. Donc si $C =$FAUX, les chemins actif sont p_1 et p_3, sinon p_2 et $p3$.



$\forall n \in \mathbb{N}^*, \forall \sigma \in \Sigma_{in}$

$C(n)_\sigma$	$V(n)_\sigma$	$P_e(\sigma, n)$	C	C_{min}
FAUX	$A(n)_\sigma$	$\{p_1, p_3\}$	$\{p_1, p_3\}$	$\{p_1, p_3\}$
VRAI	$B(n)_\sigma$	$\{p_2, p_3\}$	$\{p_2, p_3\}$	$\{p_2, p_3\}$

Pour démontrer que $analyzeITE(O, n)$ renvoie toutes les *causes minimales*, nous allons considérer les deux valuations possibles de la variable C. Pour ces deux valuations, nous avons représenté, tel que défini par la théorie, P_e l'ensemble des chemins de propagation, C l'ensemble des causes, et C_{min} l'ensemble des causes minimales.

Trace de l'algorithme pour $C(n)_\sigma =$FAUX La trace de $analyzeITE(O, n)$ passe dans la première branche de la condition et combine les éléments de manière **dépendante**. Ci-dessous, on représente à gauche l'opération effectuée par l'algorithme et à droite son résultat. Dans ce cas, le résultat de l'algorithme modélise la notion

$E_c \leftarrow combineCausesIndep(\{\{(a,n)\}\},\{\{(c,n)\}\}) \quad E_c = \{\{(a,n),(c,n)\}\}$

de **cause minimale** fournie par la théorie. La démonstration dans le cas où l'on considère $C(n)_\sigma$=VRAI est identique, l'algorithme renvoie $E_c = \{\{(b,n),(c,n)\}\}$.

La fonction $analyzeITE(O,n)$ renvoie toutes les *causes minimales* telles que définie dans la formalisation du chapitre 4.

5.4.4.7 Analyse d'un opérateur CONF1

Considérons le modèle ML_{flat} (la description de référence) de l'opérateur CONF1 tel que représenté sur la figure 5.16.

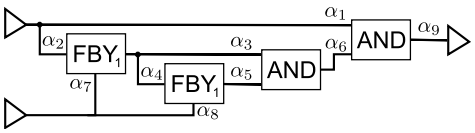


FIGURE 5.16 – Modèle de référence de l'opérateur CONF1 avec n_{conf}=2

La fonction $\forall n \in \mathbb{N}^*, analyzeCONF1(y,n)$ renvoie exactement le même résultat que la fonction $analyze_{flat}(y,n)$.

$\mathbf{n > n_{conf}}$

Il n'existe qu'une *cause minimale* de $\forall\sigma, Y(n)_\sigma$ =VRAI. Pour que la sortie soit VRAI, il faut que toutes les entrées des opérateurs AND soient VRAI, impliquant que l'entrée aux instants $n-n_{conf}, \ldots, n-1$ et n doivent être VRAI. La fonction $analyzeCONF1(\alpha_9,n)$ renvoie l'ensemble d'ensemble $\{\{(\alpha_1,n),(\alpha_2,n-1),(\alpha_2,n-2)\}\}$ représentant l'unique *cause minimale* : $\{p_1,p_2,p_3\}$ où $p_1 =< \alpha_1,\alpha_9 >$, $p_2 =< \alpha_2,\alpha_3,\alpha_6,\alpha_9 >$ et $p_3 =< \alpha_2,\alpha_4,\alpha_5,\alpha_6,\alpha_9 >$.

En revanche, si la sortie du confirmateur est FAUX, il peut exister plusieurs causes minimales. La sortie de l'opérateur AND à faux s'explique par la valuation d'une de ses entrés à faux. La fonction $analyzeCONF1(\alpha_9,n)$ renvoie de manière *indépendantes* toutes valuation à FAUX de l'entrée A dans la fenêtre temporelle $[n, n-n_{conf}]$.

$\mathbf{n \leq n_{conf}}$

L'unique différence par rapport au raisonnement précédent est que tous les arcs datés (A,n) de l'ensemble d'ensemble résultant sont remplacés par $(Init,1)$ dés lors que l'on que l'entrée retardée d'un FBY pointe vers un instant inférieur à 1.

La fonction $analyzeCONF1$ renvoie toutes les *causes minimales.*

A l'exception de $analyzeR^*S$, les fonctions *analyze* pour les opérateurs de la *bibliothèque de symbole* Airbus renvoie le même résultat que la fonction *analyze* appliqué sur la description de référence de l'opérateur. Dans le cas de la fonction $analyzeR^*S$, le traitement a été simplifié. Ces opérateurs sont décrits en annexe B.

5.5 Conclusion

Ce chapitre a présenté un algorithme appelé *algorithme d'analyse d'une violation*. Nous avons démontré que l'algorithme calcule des *causes* de la violation d'une propriété. De plus, si le modèle à analyser ne contient pas de chemin reconvergent de même ordre, nous avons démontré (annexe C) que les causes calculées sont *minimales*. Il est possible de faire évoluer l'algorithme afin qu'il renvoie toutes les causes minimales de la valuation d'une variable à un instant. Cependant, l'état de l'art sur le domaine du test matériel à démontrer que la complexité d'un tel algorithme peut être importante. Une de nos exigences est de réaliser un algorithme qui soit peu coûteux en temps de calcul qui soit négligeable devant le temps de calcul du model checker.

L'algorithme est typiquement utilisé pour l'analyse du contre-exemple renvoyé par un model checker. Cependant, il peut tout aussi bien être utilisé pour analyser les causes de la valuation d'une variable quelconque à un instant donné.

Nous allons maintenant présenter un prototype qui permettra de comprendre l'utilité d'un tel algorithme. Ce prototype a deux fonctionnalités dont la première implémente l'algorithme présenté dans ce chapitre. La seconde implémente un algorithme permettant de générer des contre-exemples nouveaux.

Chapitre 6

Prototype

Sommaire

Ce chapitre a deux objectifs. D'une part, il illustre l'application du model-checking sur un cas d'étude. Une attention particulière est portée à la description de l'analyse d'un contre-exemple. D'autre part, il présente le prototype permettant d'assister l'analyse de contre-exemples et d'en générer de nouveaux. La fonctionnalité d'analyse implémente les algorithmes présentés dans le chapitre précédent. Cette fonctionnalité a pour objectif d'extraire les chemins actifs des causes et les informations pertinentes permettant d'expliquer la violation de la propriété. Les développements s'effectueront dans l'environnement Matlab/Simulink.

La section 6.1 introduit cet environnement et détaille notamment le model checker Simulink Design Verifier (SLDV). La section 6.2 présente le cas d'étude qui permet d'illustrer les deux fonctionnalités du prototype : assistance à l'analyse de contre-exemples et génération de contre-exemples nouveaux. La première fonctionnalité fait l'objet de la section 6.3 ; la seconde fait l'objet de la section 6.4. Finalement, la section 6.5 conclut sur ces deux fonctionnalités et propose des pistes d'améliorations.

6.1 Vérification de modèle sous l'environnement Matlab/Simulink

Simulink et Matlab sont commercialisés par The MathWorks [1]. Simulink [2] est un environnement pour la simulation multi-domaine et le développement basé sur les modèles pour les systèmes embarqués dynamiques. Il fournit un environnement graphique interactif et des opérateurs de bases qui permettent de modéliser, simuler et tester plusieurs types de systèmes, discrets ou continus.

Simulink est intégré à Matlab [3], fournissant un accès à un éventail de fonctionnalités. L'extension SLDV permet l'utilisation de techniques de model cheking sur des modèles Simulink.

Le choix de cet environnement s'explique par le fait qu'il :

1. offre des capacités de modélisation similaires à celles de Scade,
2. intègre un outil de model checking : Simulink Design Verifier (SLDV),
3. offre des solutions de prototypage rapide,
4. offre une interface de programmation adéquate pour raisonner sur la structure interne des modèles Simulink.

De plus, le département des CDV Airbus développe une passerelle qui permet de modéliser automatiquement les *planches Scade* en un modèle simulink.

6.1.1 Modélisation d'un système discret sous Matlab/Simulink

Cette section présente comment un système discret peut se modéliser sous l'environnement Simulink. Comme dans le cas de Scade, Simulink propose des opérateurs de base permettant de modéliser le comportement du système. Leur représentation graphique est visible sur la figure 6.1.

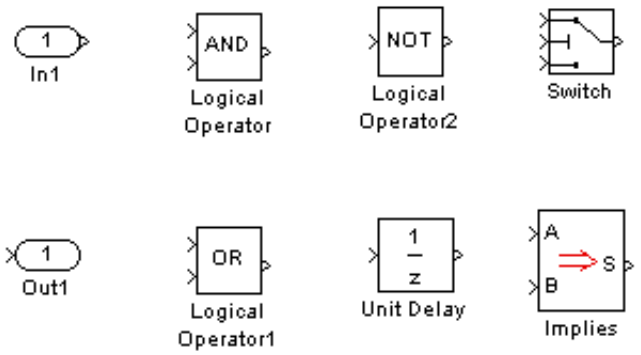


FIGURE 6.1 – Représentation graphique des opérateurs de base sous Simulink

1. http ://www.mathworks.com
2. http ://www.mathworks.com/products/simulink/
3. http ://www.mathworks.com/products/matlab/

L'opérateur nommé *In1* (respectivement Out1) est un opérateur d'entrée (respectivement de sortie). Il permet de spécifier les entrées (respectivement les sorties) d'un modèle. La sémantique de l'opérateur *Unit Delay* est équivalente à celle de l'opérateur FBY de SCADE. L'unique différence est que l'entrée d'initialisation de cet opérateur est un paramètre. De manière générale, sur les opérateurs de base Simulink, les paramètres n'apparaissent pas explicitement sur la représentation graphique comme cela est le cas pour les opérateurs de base en SCADE.

La sémantique du bloc *Switch* est équivalente à celle de l'opérateur ITE [4] de SCADE. La sémantique du bloc *Implies* est équivalente à l'implication logique [5].

Les opérateurs de la *bibliothèque de symboles* Airbus, introduits en section 1.4.2, ont été modélisés sous la forme d'opérateurs Simulink. Ces opérateurs sont décrits en annexe B. Leur représentation graphique est visible sur la figure 6.2.

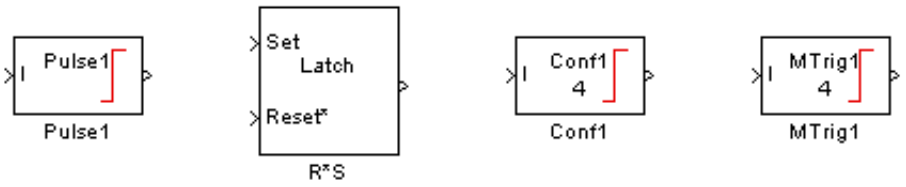


FIGURE 6.2 – Représentation graphique des opérateurs de la *bibliothèque de symboles* Airbus

Il est rappelé que le système de CDV est modélisé sous la forme d'un ensemble de planches SCADE. Un outil de transformation des planches Scade vers des modèles Simulink est en cours de développement au sein d'Airbus. Cet outil a pour but d'offrir aux ingénieurs la possibilité de simuler les planches SCADE sous l'environnement Matlab/Simulink. L'extension SLDV permet de faire de la vérification de modèle en faisant appel au model checker Prover Plug-In. Nous détaillons maintenant cette fonctionnalité.

6.1.2 Principe de fonctionnement de Simulink Design Verifier

L'outil SLDV permet de démontrer des propriétés de sûreté sur des modèles Simulink. Comme représenté sur la figure 6.3, SLDV utilise le model checker Prover Plug-In. Le modèle Simulink est automatiquement transformé en un modèle TECLA interprétable par Prover Plug-In.

Comme expliqué en section 1.5.2, Prover Plug-In, comme la plupart des model checkers peut renvoyer trois types de réponses. Si l'analyse formelle ne termine pas, un message est renvoyé à l'utilisateur. Si l'analyse termine, alors

- si la propriété est valide, un message est renvoyé à l'utilisateur,
- si la propriété est violée, Prover Plug-In génère un contre-exemple.

Il est rappelé qu'un contre-exemple est une séquence d'instanciation des variables d'entrée du modèle. Dans une utilisation classique de SLDV, comme dans la plu-

4. If Then Else
5. Implies(a,b)$\equiv \neg a \vee b$

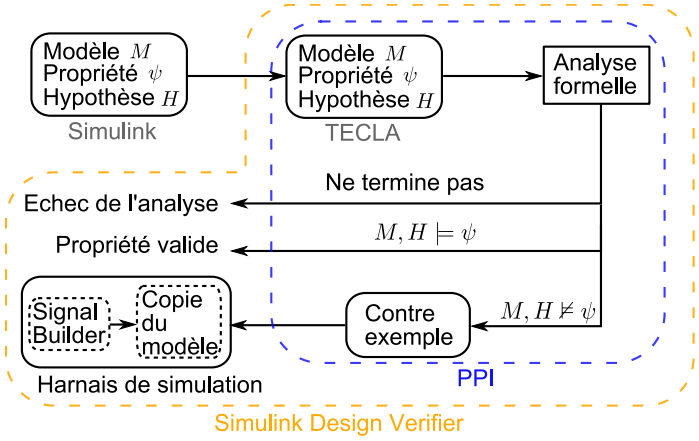


FIGURE 6.3 – Principe de fonctionnement de Simulink Design Verifier

part des model checkers, l'utilisateur doit utiliser le contre-exemple tel quel afin de comprendre les raisons de la violation. La plupart du temps un outil de simulation se révèle être nécessaire. SLDV anticipe ce besoin et génère automatiquement un *harnais de simulation*. La figure 6.4 montre le harnais généré suite à l'analyse du modèle de la figure 6.5.

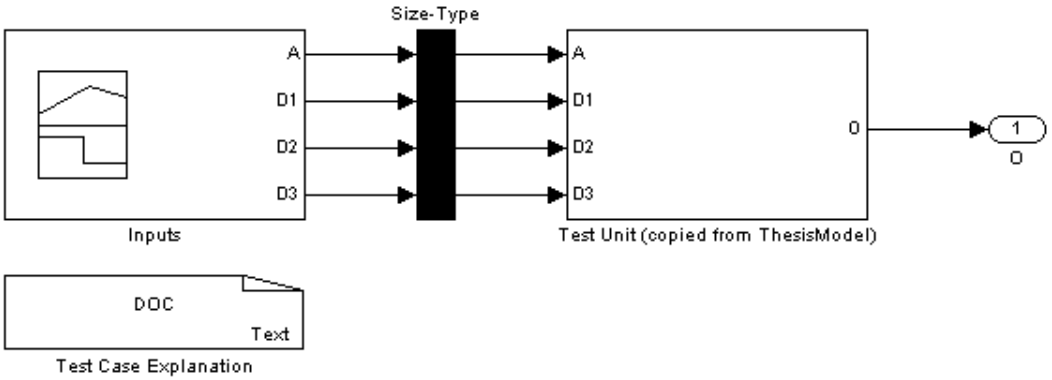


FIGURE 6.4 – Harnais généré après la détection d'un contre-exemple

Un harnais est composé de trois éléments :

1. d'un opérateur *Signal Builder* appelé *Inputs* contenant le contre-exemple,
2. d'une copie du *modèle global*, appelé *Test Unit* à laquelle est connecté l'opérateur *Signal Builder*,
3. d'un descriptif textuel [6] du contre-exemple appelé *Test Case Explanation*.

Le *harnais* permet à l'utilisateur de simuler le contre-exemple sur la copie du modèle initial. Avant de lancer la simulation, l'utilisateur doit néanmoins associer des opérateurs *Scope* aux variables dont il souhaite visualiser la valuation. Dans la

6. sous la forme d'un document html

mesure où l'on ne sait pas ce qui se passe, toutes les variables sont généralement observées. Dans le cas du modèle de la figure 6.7, plus de 30 variables peuvent être observées.

Simulink, dans sa version R2007b ne possède pas de solution permettant d'observer proprement les états des systèmes discrets. Les fonctionnalités d'observation sont plutôt adaptés aux systèmes continus tels que les boucles de contrôle. Néanmoins, des améliorations devraient être apportées dans la nouvelle version.

6.2 Cas d'étude

Un cas d'étude est proposé afin d'illustrer la vérification de modèle. Pour des raisons de confidentialité, le cas d'étude n'est pas un modèle du système de CDV. Cependant, la complexité du cas d'étude proposé est représentative d'une fonction de CDV.

6.2.1 Modèle

Considérons la fonction *processDecision* qui calcule une variable Booléenne : *Decision*. Un modèle M Simulink de cette fonction est proposé en figure 6.5. Le modèle M est composé d'opérateurs de base et d'opérateurs de la *bibliothèque de symboles* Airbus.

M possède quatre entrées A, $D1$, $D2$, $D3$ et une sortie *Decision*. La description de la fonction *processDecision* est la suivante.

La variable *Decision* est VRAI lorsque l'*autorisation* A est VRAI et que la *condition* C est VRAI. L'*autorisation* A est une variable d'entrée du modèle. En revanche, la *condition* est le résultat de l'expression $C_1 \vee C_2$. C_1 et C_2 sont deux *sous-conditions*. Leurs définitions sont respectivement encadrées en bleu et vert. La *condition 1* utilise les entrées $D1$, $D2$ et A pour calculer $C1$. La *condition 2* utilise $D3$, A et la décision à l'instant précédent pour calculer $C2$. Il est possible de considérer ces deux conditions de manière indépendante. Ces deux conditions sont décrites ci-dessous.

6.2.1.1 Description de la condition C_1

La condition C_1 est VRAI dès lors que l'entrée $D1$ est VRAI. Lorsque l'information $D1$ reste VRAI plus de 75 cycles, la bascule R*S s'active et *mémorise* C_1. Alors la condition C_1 reste VRAI même si l'information $D1$ est perdue. Il y a alors deux moyens de désactiver la bascule R*S :

- un *front montant* sur l'*information* $D2$ ou,
- un *front descendant* sur l'*autorisation* A.

Si l'un ou l'autre de ces deux événements apparaît, et que $D1$ est FAUX, alors $C1$ devient FAUX.

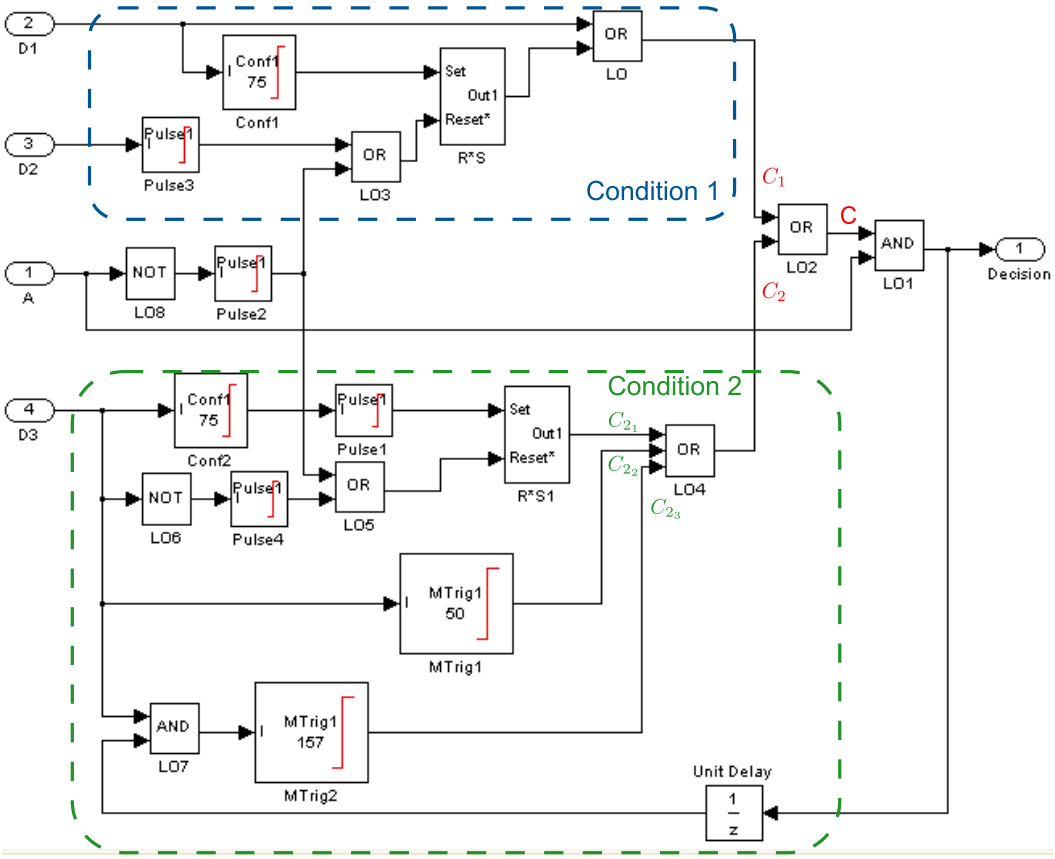


FIGURE 6.5 – *Modèle à analyser* du cas d'étude

6.2.1.2 Description de la condition C_2

La condition C_2 est VRAI dès lors que C_{2_1}, C_{2_2} ou C_{2_3} est VRAI. Ces trois variables sont maintenant décrites.

C_{2_1} devient VRAI dès que la bascule R*S1 s'active. Cette dernière s'active au moment où l'information $D3$ est confirmée 75 cycles et seulement à ce moment là. La condition C_{2_1} devient alors FAUX si l'un des deux événements suivants survient :

– un *front descendant* sur l'*information* $D3$ [7],

– un *front descendant* sur l'*autorisation* A.

C_{2_2} devient VRAI dès que $D3$ est VRAI. L'opérateur nommé MTRIG1 s'active et *maintient* l'information $D3$ durant 50 cycles. Au delà de ce temps, l'opérateur MTRIG1 se désactive, l'information $D3$ ne passe plus. C'est-à-dire que la sortie de l'opérateur MTRIG1 est FAUX même si $D3$ est VRAI.

C_{2_3} devient VRAI dès que $D3$ est VRAI et que *Decision* était VRAI à l'instant précédent. L'opérateur nommé MTRIG2 s'active et *maintient* C_{2_3} durant 157 cycles. Au delà de ce temps, l'opérateur MTRIG2 se désactive. C'est-à-dire que la sortie de l'opérateur MTRIG2 est FAUX même si son entrée est VRAI. L'opérateur *Unit Delay*

7. équivalent à un front montant de la négation de $D3$

est initialisé de telle manière que C_{2_3} soit FAUX au premier instant.

6.2.2 Propriété et hypothèse

Avant de procéder à la vérification, il est nécessaire d'exprimer la propriété à vérifier dans le formalisme adéquat. Dans le cas de Simulink, comme dans le cas de Scade, les propriétés doivent être formellement exprimées en utilisant le même langage que celui utilisé pour la modélisation.

Vérifions que le modèle M décrit dans la section 6.2.1 satisfait une **propriété de sûreté** ψ dont l'expression en langage naturel est :

> *Si la donnée D*3 *est confirmée à* FAUX *pendant plus de 50 cycles alors Decision ne peut plus faire de transition de* FAUX *à* VRAI.

Avant de débuter la vérification de modèle, il peut être nécessaire d'exprimer des hypothèses afin de contraindre l'environnement du modèle. Tout comme les propriétés, les hypothèses doivent être exprimées avec le langage de modélisation. Considérons l'**hypothèse d'environnement** H dont l'expression en langage naturel est :

> *Si la donnée D*3 *est confirmée à* FAUX *pendant plus de 50 cycles alors la données D*1 *reste à* FAUX.

Comme décrit dans le chapitre 1.5.2.5, la modélisation de la propriété ψ (respectivement de l'hypothèse H) est appelée *observateur de la propriété* (respectivement *de l'hypothèse*). Les observateurs sont, comme le modèle M, implémentés avec des opérateurs Simulink. Cependant, deux opérateurs supplémentaires sont nécessaires afin de spécifier quelle variable représente la propriété (respectivement l'hypothèse). Ces deux opérateurs, appelés *proof objective* et *Assumption* sont accessibles via la bibliothèque SLDV. Leur représentation graphique est visible sur la figure 6.6. L'opérateur *proof objective* (respectivement *Assumption*) est identifiable par la lettre P (respectivement A).

FIGURE 6.6 – Opérateurs de la bibliothèque Simulink Design Verifier

Le *modèle global* de la figure 6.7 contient le modèle M du système à analyser (encadré en vert), l'observateur de la propriété ψ (encadré en orange) et l'observateur de l'hypothèse H (encadré en violet). Le terme *modèle global* est utilisé pour distinguer le *modèle à analyser* de la figure 6.5 de celui de la figure 6.7.

L'opérateur appelé *Terminator*, représenté sur la figure 6.8, utilisé dans l'*observateur d'hypothèse*, est un opérateur sans sémantique.

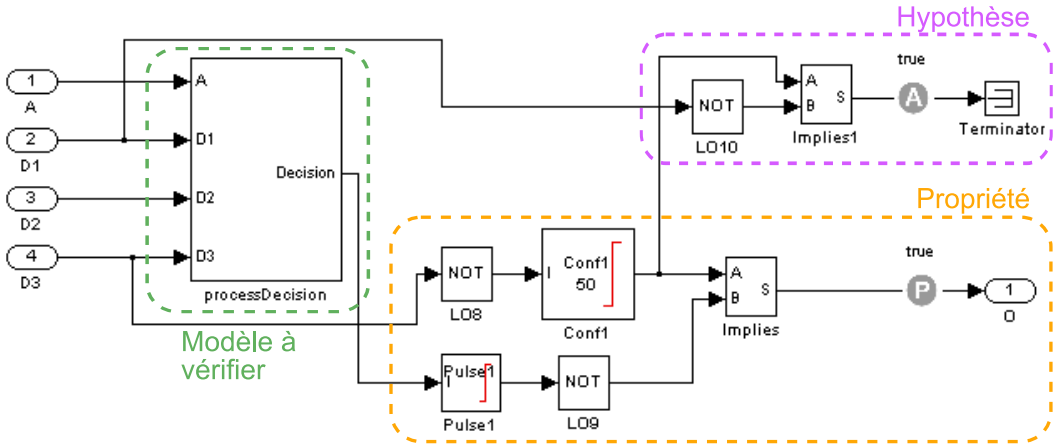


FIGURE 6.7 – *Modèle global* du cas d'étude

FIGURE 6.8 – Opérateur *terminator*

Une fois les observateurs implémentés, il est possible de procéder à une analyse formelle. L'outil SLDV permet de vérifier si le modèle M satisfait la propriété ψ sous l'hypothèse H.

6.2.3 Analyse du modèle

L'analyse du modèle de la figure 6.7 renvoie un contre-exemple. SLDV génère automatiquement un *harnais* de simulation. L'opérateur *Signal Builder*, contenu dans le *harnais*, permet de visualiser le contre-exemple sous la forme d'un chronogramme comme représenté sur la figure 6.9.

Le contre-exemple contient toutes les valuations des variables A, $D1$, $D2$ et $D3$ sur 53 cycles. Le harnais va permettre de simuler le contre-exemple sur le modèle.

Pour diagnostiquer les causes de la violation, un utilisateur procède en deux temps :

- il cherche à comprendre le contre-exemple. Cette étape se fait généralement par une simulation du contre-exemple. Globalement cette étape permet à l'utilisateur de se faire une représentation mentale de ce qui se passe. L'utilisateur cherche à donner un sens opérationnel au scénario en cours de simulation, afin de déterminer si le contre-exemple trouvé est réaliste ou non. Cependant, il est préférable de ne pas juger le contre-exemple non-réaliste avant d'avoir procédé à la seconde étape qui est souvent nécessaire pour pouvoir juger du caractère réaliste du contre-exemple.
- il cherche à identifier la liste d'événements qui a engendré la violation de la propriété. La manière la plus simple de procéder est de réaliser une simulation à rebours en partant de l'instant de violation. Cette simulation a pour but

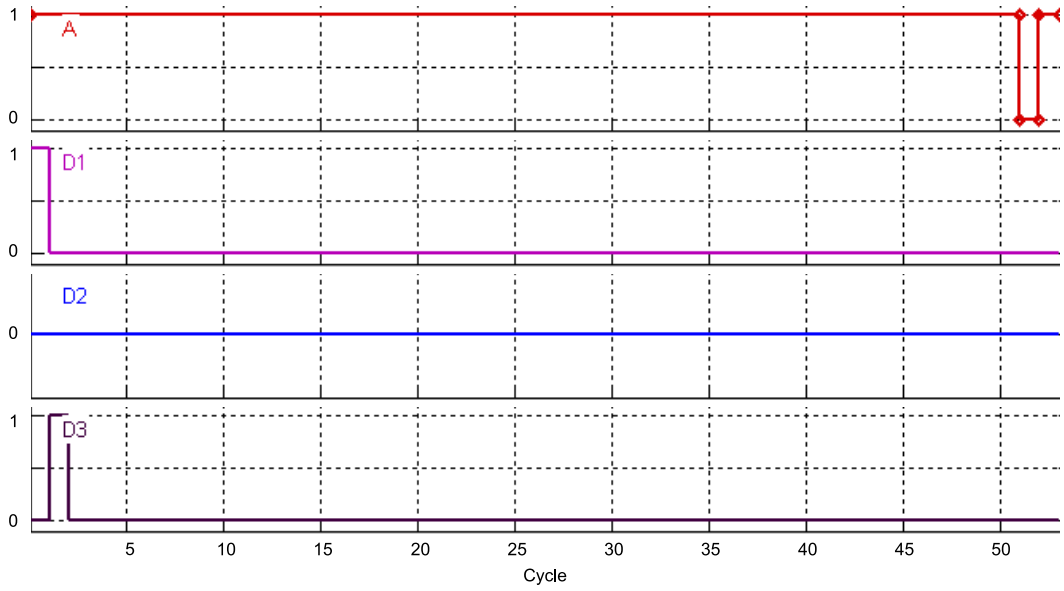


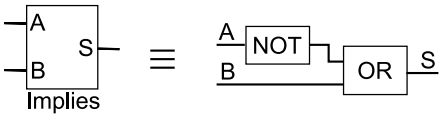
FIGURE 6.9 – Contre-exemple renvoyé par le model checker

d'identifier les couples (variable, instants) impliqués dans la violation de la propriété. Cette étape s'apparente à une étape de filtrage de l'information utile, celle qui permet de comprendre les raisons de la violation et de trouver la faute. Cette tâche est facilitée lorsque l'on dispose d'un outil de simulation permettant de faire cela. Simulink ne propose pas cette fonctionnalité.

Durant ces deux étapes, l'utilisateur cherche à faire le lien entre le scénario observé et un comportement opérationnel de niveau système ou avion. Cependant, il est nécessaire de comprendre la violation avant de pouvoir juger de la nature (réaliste ou pas) du contre-exemple.

Pour comprendre la cause de la violation de la propriété, il est nécessaire d'observer la structure de l'observateur de la propriété (figure 6.7). La variable désignant la propriété est le résultat d'un opérateur Implies. La figure ci-dessous contient le modèle équivalent de cet opérateur ainsi que sa table de vérité. Dans ce tableau les valeurs de vérité FAUX et VRAI sont remplacées par 0 et 1.

A	B	S
0	0	1
0	1	1
1	0	0
1	1	1



L'unique moyen de violer la propriété, c'est-à-dire de placer la sortie de l'opérateur *Implies* à FAUX, est de valuer son entrée A à VRAI et son entrée B à FAUX.

D'après l'observateur de la propriété, l'entrée A de l'opérateur *Implies* devient VRAI après que la variable $D3$ est confirmée à FAUX sur 50 cycles. Dans le contre-exemple, $D3$ est confirmé à FAUX au cycle 53.

Concernant l'entrée B de l'opérateur Implies, l'analyse est plus complexe. Si la propriété est violée au cycle 53, l'unique raison est qu'il y a un front montant de la variable *Decision* du cycle 52 à 53. La simulation du harnais confirmera cette hypothèse. Il est maintenant nécessaire de comprendre ce qui a provoqué ce front montant. L'idéal est de procéder par une analyse à rebours dans le modèle (figure 6.5) :

- Au cycle 53, la condition 2 est active et un front montant de la variable A provoque le front montant de la variable *Decision*.
- La condition 2 est active au cycle 53 car le MTRIG2 a été activé au cycle 2 par la valuation à VRAI de la variable $D3$ au cycle 2 et *Decision* au cycle 1.
- *Decision* est VRAI au cycle 1 car A et D1 sont VRAI.

Cette suite d'événements produit un front montant sur *Decision* au cycle 53, qui engendre la valuation de l'entrée B à FAUX, alors que l'entrée A de l'opérateur Implies est VRAI. La propriété est alors violée.

6.2.4 Conclusion

Cette section a présenté un cas d'étude et son analyse. Le cas d'étude choisi est de complexité représentative d'une fonction de CDV, autant au niveau du modèle qu'au niveau de la propriété et de l'hypothèse.

L'analyse du cas d'étude a permis d'illustrer la modélisation et la vérification du modèle sous l'environnement Matlab/Simulink. Il a notamment permis de souligner le caractère délicat de l'analyse d'un contre-exemple. 212 valuations de variables ont du être observées (4 variable d'entrée sur 53 cycles), sans compter les varaibles intermédiaires. L'effort d'analyse a permis d'identifier la cause de la violation de la propriété, au terme de simulations, en avant et en arrière. La première permettant surtout à l'utilisateur de se faire une représentation mentale des états traversés. La seconde ayant pour but d'identifier les informations pertinentes expliquant la violation. Il est à noter que l'analyse arrière n'est actuellement pas du tout supportée par l'environnement simulink, elle est donc entièrement manuelle.

Le prototype présenté dans la section suivante a justement pour objectif d'automatiser cette étape de compréhension du contre-exemple.

6.3 Prototype : *analyse automatisée de contre-exemples*

Cette section a pour but d'illustrer la fonctionnalité d'analyse automatisée de contre-exemples. Cette fonctionnalité permet d'apporter à l'utilisateur des informations sur les raisons de la violation de la propriété. Une remarque est que cette fonctionnalité se généralise à la justification de la valuation de n'importe quelle variable du modèle à un instant dans un scénario.

Pour cela, le prototype propose à l'utilisateur de visualiser, non pas le contre-exemple, mais directement :

- les causes de la violation de la propriété en colorant sur le modèle les opérateurs traversés par les chemins actifs ;

– la suite d'événements qui ont conduit à la violation. Ces informations proviennent des *symboles de la bibliothèque* Airbus. L'utilisateur, possédant *a priori* un modèle mental du comportement des symboles, peut interpréter ces informations sans observer la définition des symboles.

6.3.1 Illustration sur le cas d'étude

Afin d'illustrer la fonctionnalité d'assistance à la compréhension d'un contre-exemple, le prototype est utilisé avec le modèle de la figure 6.7.

Le contre-exemple exhibe une situation où la *Decision* a été prise alors que la donnée $D3$ est FAUX depuis 50 cycles. Le principe du diagnostic est de comprendre pourquoi il y a eu un front montant de *Decision* après la confirmation de $D3$ à FAUX.

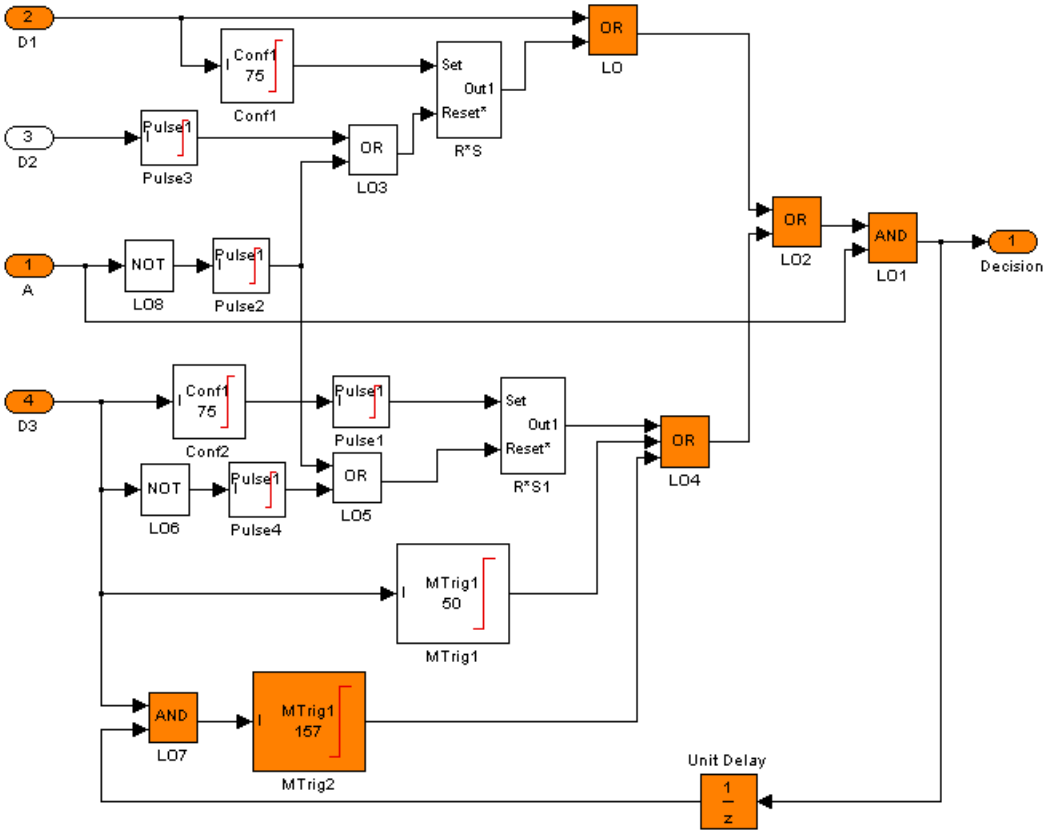


FIGURE 6.10 – Modèles avec chemins actifs à l'instant de violation

Comme représenté sur la figure 6.10 Le prototype colorie en orange les chemins actifs à l'instant de violation de la propriété. Cette information a pour objectif de montrer quelles sont les parties de la structure du modèle qui sont impliquées dans la violation de la propriété. Sous Simulink, il n'est pas possible de colorier uniquement des arcs, il faut colorier des variables (et donc tous les arcs associés

à cette variable). Cette solution ne nous convenait pas, nous avons donc choisi de colorier les opérateurs.

Le prototype renvoie aussi des informations temporelles qui sont liées aux informations structurelles, les instants considérés sur le chemin colorié. Cela donne sur l'exemple :

MTrig2(53)	:	true since 2
A(53)	:	true
A(52)	:	false
D3(2)	:	true
D1(1)	:	true

La première ligne informe l'utilisateur que la sortie du symbole de la bibliothèque Airbus appelé MTRIG2 est valuée à VRAI et que cet opérateur a été activé à l'instant 2. Les quatre dernières lignes indiquent les valeurs d'entrées du modèle responsables de la violation. N'importe quel autre scénario conservant précisément ces valuations provoquera la violation de la propriété à l'instant 53.

6.3.2 Principe de fonctionnement

Le prototype prend en entrée le *modèle global* et renvoie :

- un message à l'utilisateur si la propriété est valide ou si le model checker ne termine pas ;
- l'analyse d'un contre-exemple si un contre-exemple est trouvé par le model checker.

Comme représenté sur la figure 6.11, le prototype (encadré de pointillés verts) se positionne en interface entre le modèle à analyser (grisé) et l'outil Simulink Design Verifier (encadré de pointillés oranges) présenté en section 6.1.2. Sur la figure 6.11, seul le cas où le model checker renvoie un contre-exemple est représenté.

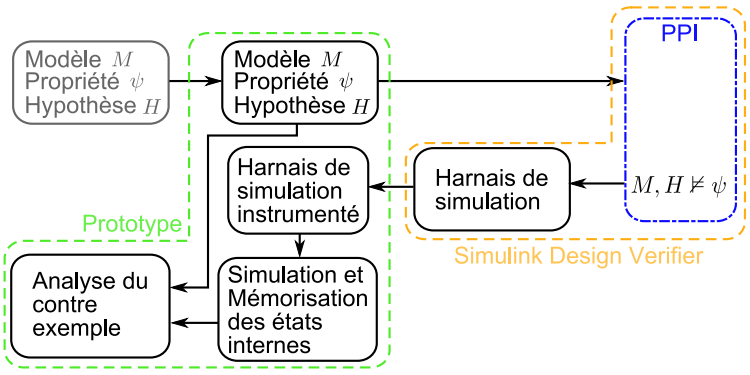


FIGURE 6.11 – Principe de fonctionnement du prototype

Le prototype procèdera en quatre étapes :

1. le prototype travaille sur une copie du *modèle global*. Cette étape est détaillée en section 6.3.2.1.

2. le prototype initialise et appelle ensuite l'outil SLDV en lui passant en paramètre la copie du *modèle global*. Cette étape est détaillée en section 6.3.2.2.
3. Si la propriété est valide ou si le model checker ne parvient pas à achever son calcul, le prototype s'arrête en renvoyant le message adéquat. En revanche, s'il existe un contre-exemple, SLDV renvoie le *harnais* permettant la simulation du contre-exemple. Le *harnais* est utilisé pour la **simulation et reconstruction des états** internes et des sorties du modèle. Cette étape est détaillée en section 6.3.2.3,
4. Finalement, le prototype **analyse le contre-exemple** pour identifier **les causes de la violation**. Cette étape est détaillée en section 6.3.2.4.

6.3.2.1 Copie du modèle

Le prototype travaille sur une copie du *modèle global* afin de ne pas prendre le risque de modifier ce dernier. La fonction *save_system* permet de faire la copie.

```
1 %work will be performed on a copy of the observer
2 [PATHSTR,observer,EXT,VERSN] = fileparts(observer);
3 observer = save_system(observer, [observer 'Analyzed']);
```

Le nouveau modèle porte le nom de l'original avec, à la fin, le mot clé *Analyzed*.

6.3.2.2 Analyse du modèle de travail

Avant d'analyser le modèle de travail, le prototype initialise les paramètres de simulation de Simulink [8] :

```
1 %Solver must be 'FixedStepDiscrete' before launching SLDV
2 set_param(observer,'solver','FixedStepDiscrete')
```

Ensuite, il initialise les paramètres de SLDV. Dans sa fonctionnalité *PropertProving*, SLDV possède trois stratégies d'utilisation du model checker :

Find violation le model checker cherche à violer la propriété en un nombre d'itérations fini spécifié. Cette stratégie utilise l'algorithme *Bounded model checking* de Prover Plug-In.

Prove le model checker cherche à démontrer la propriété. Cette stratégie utilise l'algorithme *Induction over time* de Prover Plug-In.

Prove with violation detection Dans un premier temps, le model checker cherche à violer la propriété en un nombre d'itérations fini en utilisant l'algorithme *Bounded model checking*, s'il n'y parvient pas, il cherche à démontrer la propriété en utilisant l'algorithme *Induction over time*.

SLDV est paramétré de la manière suivante :

8. nécessaires au bon fonctionnement de Simulink Design Verifier

```
%create a type "sldv options" variable
mySldvOptions = sldvoptions;
%Use assertions specified in the observer
mySldvOptions.Assertions = 'EnableAll';
%Do not display the report
mySldvOptions.DisplayReport = 'off';
%proving a property rather than performing test generation
mySldvOptions.Mode = 'PropertyProving';
%Use the property specified in the observer
mySldvOptions.proofAssumptions = 'EnableAll';
%Specify the strategy to use (rather than FindViolation or
%ProveWithViolationDetection)
mySldvOptions.provingStrategy = 'Prove';
%Save files
mySldvOptions.SaveDataFile = 'on';
mySldvOptions.SaveHarnessModel = 'on';
mySldvOptions.SaveReport = 'on';
%If the internal timer exceed this value, analysis is stopped
mySldvOptions.MaxProcessTime = 600000000;
```

Le paramètre $mySldvOptions.Assertions =' EnableAll'$ spécifie que toutes les propriétés du modèle doivent être prises en compte. De même, le paramètre $mySldvOptions.proofAssumptions =' EnableAll'$ spécifie que toutes les hypothèses doivent être prises en compte.

Le paramètre $mySldvOptions.Mode =' PropertyProving'$ permet de sélectionner la fonctionnalité de vérification plutôt que celle de génération automatique de séquences de test.

Le paramètre $mySldvOptions.provingStrategy =' Prove'$ permet de sélectionner l'algorithme *Induction over time*.

Le paramètre $mySldvOptions.MaxProcessTime = 600000000$ permet de spécifier que si l'analyse n'a pas terminé avant, elle s'arrêtera au bout de 600000000 secondes en renvoyant un message à l'utilisateur.

Finalement, le prototype utilise SLDV afin d'analyser la propriété en invoquant la commande suivante :

```
%Starting the model checker
[status, filenames] = sldvrun(observer, mySldvOptions);
```

Si l'analyse ne termine pas, le prototype renverra un message d'erreur à l'utilisateur. Si l'analyse termine et que la propriété est valide, le prototype affichera un message à l'utilisateur. Dans le cas où la propriété est violée, SLDV renvoie un contre-exemple sous la forme d'un *harnais*. Ce dernier est récupéré par le prototype qui le simulera afin de reconstruire les états traversés par le contre-exemple.

6.3.2.3 Simulation et reconstruction des états internes

Le prototype installe des *points de piquage* sur toutes les variables de *TestUnit* présentes dans le harnais en utilisant la fonctionnalité *DataLogging*. Pour chaque

opérateur du *modèle global*, la commande suivante est invoquée :

```
1 set_param((BlockPortHandles.Outport),'DataLogging','on');
```

La figure 6.12 montre le modèle à analyser agrémenté des points de piquage.

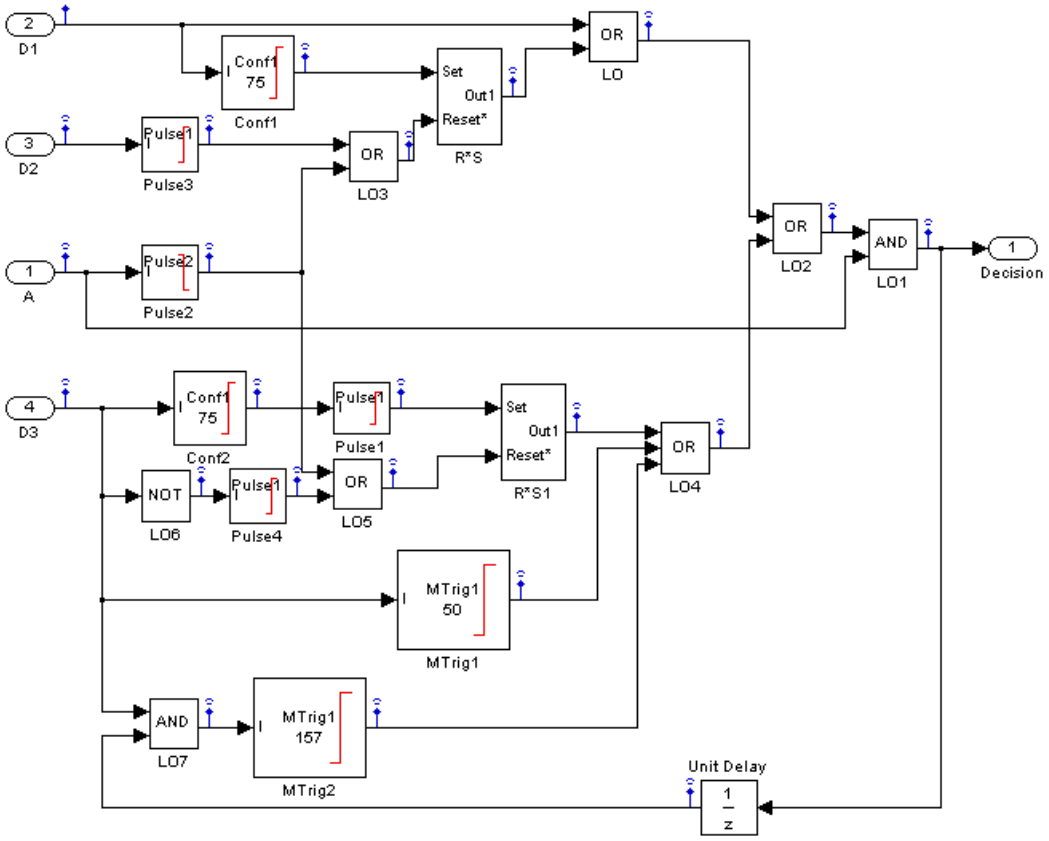


FIGURE 6.12 – modèle à analyser agrémenté de points de piquage

Les *points de piquage* permettent de mémoriser la valeur de chaque variable à chaque instant lors de la simulation du contre-exemple.

La simulation du *harnais* est lancée en invoquant la commande :

```
1 sim(harness);
```

Une fois la simulation terminée, les valeurs des variables sont stockées dans une structure de donnée. La fonction $storeDataLog(blocks, logsout)$ va stocker chaque valuation de variable dans chaque instance de l'opérateur qui l'a produite.

```
1 blocks = findBlocks(observer);
2 storeDataLog(blocks, logsout);
```

6.3.2.4 Analyse du contre-exemple

L'analyse du contre-exemple peut alors commencer. Le prototype initialise l'algorithme d'analyse des contre-exemples avec :

- la variable de sortie du modèle,
- le dernier instant du contre-exemple qui correspond à l'instant de violation.

```
%PreviousBlockHandle – pointe l'opérateur dont la sortie est Oobs.
%portsNumber – number of output Oobs in the previous operator.
%blockChart – is set to empty cell because the analysis start at the
%higher level.
%stepAnalysis – analysis start at last step of counterexample.
%infoBackup – will contain relevant events of the cex.
%branchToRemove – will contains infos to refine property in order
%to search for others cex.
branchToRemove = 0; %tableau de handle
[infoBackup, branchToRemove] = analyze2(previousBlockHandle, ...
... portsNumber, {}, stepAnalysis, branchToRemove);
```

L'algorithme d'analyse procède comme indiqué dans la section 5.1. Cependant, la notion d'arc n'existe pas dans Simulink. Elle a été remplacée par un couple $< op, n >$ où op est l'instance de l'opérateur et n est le numéro de la sortie d'où l'arc sort.

Une fois l'analyse terminée, les chemins actifs sont colorés sur le modèle afin de repérer la zone impliquée dans la violation de la propriété. L'intérêt de la coloration des chemins de violation est d'autant plus grand que le modèle est gros.

6.3.3 Interface Graphique

Une interface graphique permet de sélectionner l'observateur à analyser. La figure 6.13 montre l'interface graphique une fois l'observateur *ThesisModel* sélectionné.

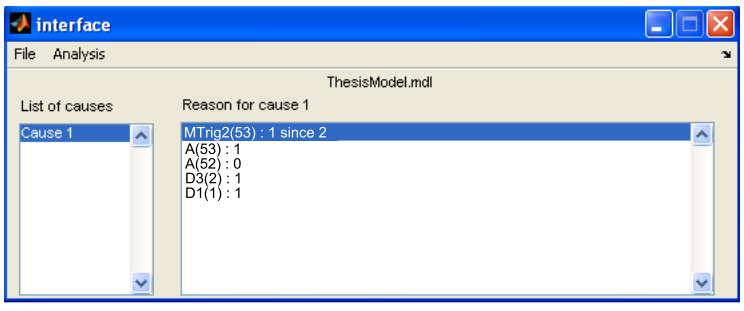


FIGURE 6.13 – Interface graphique

L'interface graphique est composée d'un menu et d'une zone d'information. Cette dernière contient une zone de texte et deux *Listbox*. La *Listbox* intitulée *List of causes* est destinée à afficher la liste des causes. La *Listbox* intitulée *Reason for cause* est dédiée, pour une cause donnée, à l'affichage des événements clés.

Les *Listbox* intitulées *List of causes* et *Reason for cause* seront remplies une fois l'analyse du contre-exemple terminée. Un appui sur le bouton *Launch* du menu *Analysis* demande au prototype d'analyser le modèle sélectionné. Le prototype va suivre alors la procédure décrite sur la figure 6.11.

L'interface est particulièrement utile lorsque l'on traite un contre-exemple possédant plusieurs *causes*. La *Listbox* intitulée *List of causes* affiche la liste des *causes* identifiées. Pour chaque *cause*, un clic sur l'occurrence dans *List of causes* permet :

- de colorier les chemins actifs de cette cause à l'instant de violation.
- d'afficher dans la *Listbox* intitulée *Reason for cause* la liste des événements clés ayant provoqué la violation de la propriété.

L'utilisateur peut donc analyser les causes une à une.

6.3.4 Conclusion

Cette section a, d'une part, illustré la fonctionnalité d'analyse automatisée d'un contre-exemple ; d'autre part, elle a présenté les détails d'implémentation du prototype mettant en œuvre l'analyse. En pratique, l'information apportée à l'utilisateur à l'origine - un contre-exemple - est remplacée par des informations pertinentes permettant de comprendre plus facilement la violation de la propriété. Ce type de réponse, que nous pensons être plus adapté aux besoins d'un utilisateur, facilite l'utilisation industrielle des outils de model checking.

L'analyse de la violation donne généralement lieu à une modification des observateurs de la propriété et des hypothèses, puis à une nouvelle analyse qui peut de nouveau générer un contre-exemple. Plutôt que de renvoyer les contre-exemples un à un, chacun de ces contre-exemples provoquant des modifications, il peut être plus efficace de renvoyer plusieurs contre-exemples pour un modèle donné et de prendre ainsi en compte en une seule fois les modifications nécessaires à la correction du modèle ou des observateurs.

6.4 Prototype : *génération de contre-exemples différents*

Le prototype est doté d'une fonctionnalité permettant la génération automatique de contre-exemples « différents ». La motivation de cette fonctionnalité a été explicitée dans le chapitre 2.3. Le terme « différent » signifie que la génération a pour objectif de faire apparaître de nouvelles *causes* qui n'appartiennent pas au premier contre-exemple trouvé. En d'autres termes, le prototype utilise le model checker pour générer un contre-exemple. Il calcule les causes de ce contre-exemple puis, à partir de ces dernières, il cherche à utiliser le model checker pour générer un contre-exemple dont les causes sont différentes.

6.4.1 Principe de fonctionnement

Les travaux sur la génération automatique de vecteurs de test [Ammann 1998, Gargantini 1999, Heimdahl 2004, Heimdahl 2003, Hamon 2004] utilisent des outils

de model checking pour générer des vecteurs de test. Ces travaux se basent sur la capacité des outils de model checking à générer des contre-exemples. Plusieurs stratégies peuvent être utilisées pour manipuler un outil de model checking. Il est possible de :

- modifier la stratégie de vérification du model checker ;
- modifier le modèle à analyser ;
- modifier l'observateur des hypothèses ;
- modifier l'observateur des propriétés.

Le model checker utilisé par SLDV, Prover Plug-In, ne permet pas à l'utilisateur de contrôler sa stratégie de fonctionnement. En revanche, les trois autres alternatives, qui ne dépendent pas du model checker, sont techniquement réalisables. Une première solution très simple est de modifier l'observateur des hypothèses en ajoutant comme hypothèse que le contre-exemple précédemment obtenu ne peut pas se produire. Cependant cette solution n'est pas très satisfaisante dans la mesure où le model checker renvoie des contre-exemples très proches dont la cause est identique au précédent.

Nous avons décidé d'étudier la dernière possibilité : l'observateur des propriétés va être modifié afin d'orienter le model checker pour générer des *contre-exemples différents*. Nous présentons d'abord le principe de modification de l'observateur des propriétés puis l'utilisation de l'analyse précédente pour choisir la modification adéquate.

Modification de la propriété. Prenons le cas d'un opérateur OR dont la sortie, au moment de l'analyse est VRAI. Cette valuation est provoquée par l'entrée A qui est VRAI. Il peut exister un contre-exemple dans lequel la valuation de la sortie de l'opérateur OR à VRAI est provoquée par la valuation de l'entrée B à VRAI. L'idée est de guider le model checker pour savoir si un tel contre-exemple existe.

Dans ce cas, un moyen de forcer le model checker à explorer la branche du OR portant la variable B est de modifier la propriété originelle ψ en une propriété mutante ψ' telle que :

$$\psi' = (B \Rightarrow \psi)$$

Cette expression oblige le model checker à valuer la variable B à VRAI au moment de la violation. Si le model checker génère un contre-exemple, l'arc associé à B appartiendra à un chemin d'une cause. De plus, le contre-exemple trouvé est bien également un contre-exemple de la propriété de départ ($B \Rightarrow \psi = \psi \vee \neg B$ donc le contre-exemple satisfera la négation de cette propriété, c'est-à-dire $\neg\psi \wedge B$ donc satisfera $\neg\psi$).

Cette modification de la propriété est plus fine que l'approche qui consisterait à ajouter une hypothèse sur la valeur de B, car cette hypothèse oblige B à rester VRAI à tous les instants.

Choix de la variable à contraindre. Il reste à déterminer quelle variable contraindre pour générer un contre-exemple différent. Nous allons pour cela utiliser

l'analyse du contre-exemple présentée précédemment. Lors de l'analyse arrière du contre-exemple, le prototype stocke à chaque étape les arcs qui ne sont pas actifs et les valeurs de ces arcs qui pourraient amener à une violation de la propriété. Pour l'instant, la fonctionnalité de génération de contre-exemples différents du prototype utilise l'arc le plus proche de la sortie pour générer un contre-exemple différent (en utilisant la modification de la propriété présentée ci-dessus) mais on pourrait envisager d'utiliser n'importe lequel des arcs (ou chacun des arcs en séquence). Plusieurs stratégies sont envisageables, leur étude fait partie des perspectives de cette thèse.

Remarque : il faudrait aussi stocker le niveau des arcs pour pouvoir traiter les opérateurs temporels.

6.4.2 Illustration sur le cas d'étude

Considérons le modèle de la figure 6.7. Une première analyse a renvoyé le contre-exemple de la section 6.3.1. Nous utilisons la fonctionnalité de génération de *contre-exemples différents* qui produit le contre-exemple de la figure 6.14.

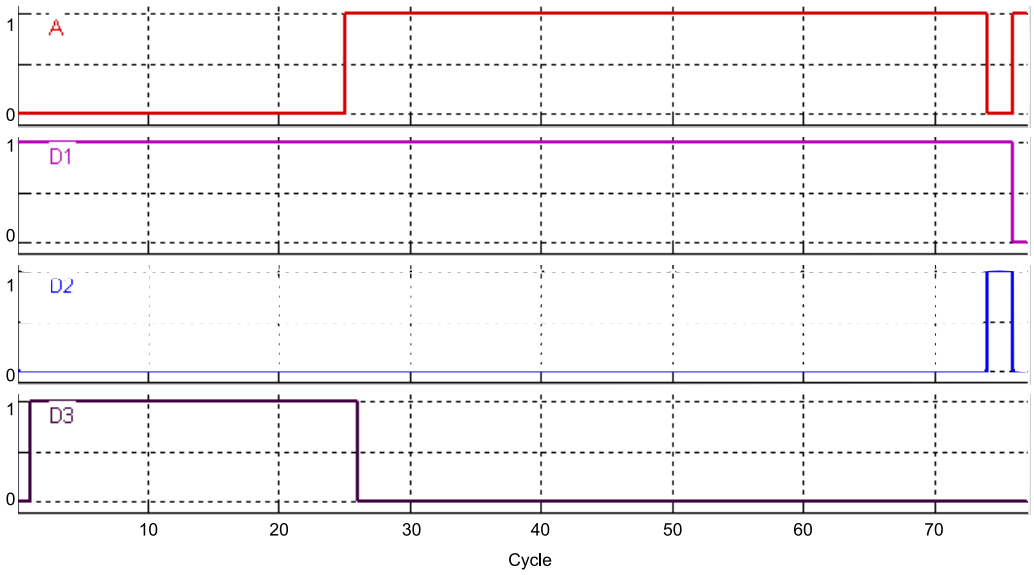


FIGURE 6.14 – Second contre-exemple généré par le model checker

Le contre-exemple tel quel est donné à titre informatif. Le contre-exemple ne permet pas facilement de voir en quoi ce nouveau contre-exemple est différent du précédent. De plus, il n'est pas possible de se rendre compte que ce contre-exemple contient deux causes.

L'interface graphique du prototype indique que le contre-exemple contient deux causes (figure 6.15).

La *cause* numérotée 2 est identique à celle de la section 6.3.1. Un clic sur la *cause* numérotée 1 provoque :

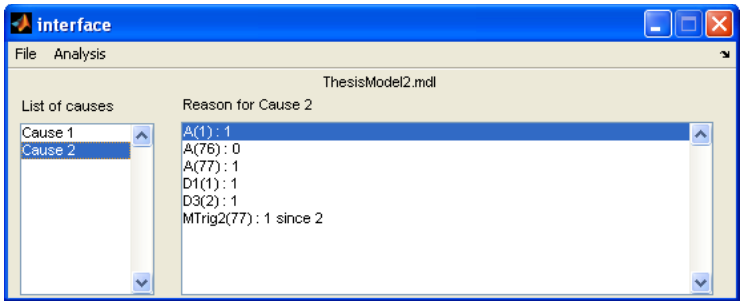


FIGURE 6.15 – Interface graphique

– la coloration des chemins actifs au cycle de la violation sur le modèle (figure 6.16) ;
– l'affichage de la liste d'événements causant la violation dans la *Listbox* intitulée *Reason for cause*.

La liste d'événements causant la violation est la suivante :

R*S(77)	:	true, last set at 76 and, no reset applied from 76 to 77
A(77)	:	true
D2(77)	:	false
A(76)	:	false
A(75)	:	false
Conf1 (76)	:	true since 76 because input true since 1
D2(76)	:	true
D2(75)	:	true
D1(2 to 76)	:	true

La lecture du modèle coloré et de la liste d'événements doit se faire en parallèle. L'utilisateur cherche à comprendre pourquoi la variable *Decision* a subi un front montant sur les cycles 76-77. Pour expliquer cela, il est nécessaire de comprendre pourquoi *Decision* est :

– VRAI au cycle 77
– FAUX au cycle 76

Le prototype indique que *Decision* est VRAI au cycle 77 car :

– la sortie de la bascule nommée R*S est VRAI au cycle 77.
– et que l'autorisation *A* est VRAI au cycle 77.

Le prototype indique que *Decision* est FAUX au cycle 76 car l'autorisation *A* est FAUX au cycle 76.

La sortie de la bascule R*S est VRAI au cycle 77 pour 2 raisons :

1. L'entrée set de la bascule R*S a été VRAI au cycle 76 ;
2. L'entrée reset de la bascule R*S est restée à FAUX aux cycle 76 et 77.

Le premier point est dû au passage à VRAI de la sortie de l'opérateur CONF1 au cycle 76 : l'entrée du confirmateur est VRAI depuis le cycle 1.

Le second point s'explique par deux éléments :

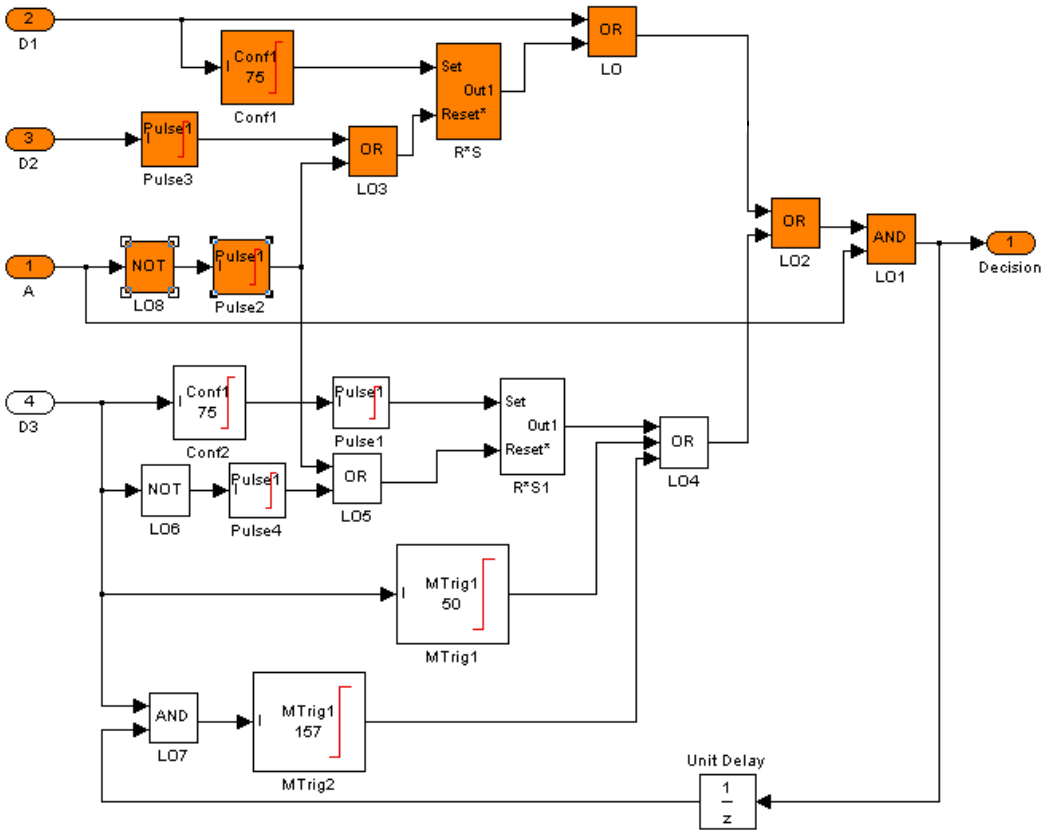


FIGURE 6.16 – *Cause* numérotée 1

– l'absence de front montant de la donnée $D2$ aux cycles 76 et 77 : $D2$ est VRAI aux cycles 75 et 76.
– l'absence de front descendant de l'autorisation A aux cycles 76 et 77 : A est FAUX aux cycles 75 et 76.

Remarque : on voit de nouveau sur ce contre-exemple tout l'intérêt de disposer du modèle coloré et de la liste des événements causant la violation. Sans ces éléments, il aurait été très difficile de comprendre ce contre-exemple qui nécessite 77 cycles.

6.5 Conclusions et évolutions

Ce chapitre a présenté une application du model-checking sur un cas d'étude de complexité industrielle. Ce dernier a permis d'illustrer la difficulté liée à la compréhension d'un contre-exemple qui peut contenir beaucoup d'information superficielle.

A cette problématique, nous tentons d'apporter une solution : un prototype renvoyant directement les informations utiles à l'explication de la violation. Nous pensons que ce type de fonctionnalité facilite le travail d'un utilisateur et lui permet un gain de temps considérable dépendant de la taille du modèle et de la longueur du contre-exemple analysé.

De plus, nous pensons que renvoyer plusieurs contre-exemples différents peut permettre une convergence plus rapide vers des propriétés et un modèle corrects. Cette conviction est liée au temps de calcul important rencontré lors des expérimentations.

Il est important de noter que le prototype ne dépend pas du model checker et peut être utilisé avec n'importe quel model checker sous réserve d'adapter l'interface.

En termes de perspectives, concernant la fonctionnalité d'analyse automatisée de contre-exemples, l'ergonomie de l'interface graphique peut encore être améliorée en profitant de toutes les possibilités offertes à l'heure actuelle. On pourrait notamment envisager un affichage pas à pas des chemins dans le modèle qui rendrait visibles les différents instants intervenant dans le contre-exemple.

Concernant la fonctionnalité de génération de contre-exemples nouveaux, la fonctionnalité offerte par le prototype n'est clairement qu'une première tentative pour explorer la faisabilité et l'intérêt d'une telle génération. Cette tentative a permis de trouver un contre-exemple différent sur le cas d'étude considéré. De nombreuses perspectives restent encore à explorer dans ce domaine.

Pour l'approche implémentée actuellement, il est nécessaire de démontrer que les contre-exemples générés avec la propriété mutante sont des contre-exemples nouveaux par rapport au contre-exemple précédemment trouvé (c'est-à-dire que leurs causes sont différentes).

Ensuite, la version actuelle du prototype ne prend pas en compte les niveaux des arcs considérés, il faudra donc ajouter cette information dans la structure qui stocke les informations utilisées pour la génération de contre-exemples nouveaux. Cette version n'utilise actuellement que l'information concernant le point de choix le plus proche de la sortie du modèle. Il faut étudier les meilleures stratégies pour utiliser les autres informations. Plus généralement, il faut explorer, expérimenter et comparer les différentes stratégies de génération offertes par le cadre que nous avons défini.

Conclusion

Cette thèse CIFRE apporte des contributions et définit des perspectives que nous avons choisi de présenter selon deux points de vue : *industriel* et *académique*.

D'un point de vue *industriel*, la première contribution a été la synthèse des expérimentations sur le model checking menées par Airbus depuis une dizaine d'années. Ces expérimentations avaient été menées par des équipes différentes sur des applications différentes et avec des objectifs différents. Il était donc intéressant de prendre le temps de synthétiser les résultats obtenus et les problèmes rencontrés. Cette synthèse a été consolidée par une expérimentation menée pendant la thèse sur la fonction Ground Spoiler. Les contributions liées à ces expérimentations sont :

- un positionnement du model checking dans le cycle de développement Airbus : en l'état actuel, le model checking ne peut pas se substituer aux tests de V&V ; on privilégie une utilisation amont à des fins de debug, pour des propriétés critiques où l'exhaustivité de la technique est très intéressante.
- la définition d'une méthodologie d'utilisation adaptée pour faciliter le transfert vers les concepteurs.

Un recensement des propriétés les plus critiques a également été entrepris pendant la thèse en lien avec l'analyse de sûreté de fonctionnement. Cette contribution n'a pas été développée dans le manuscrit pour des raisons de confidentialité.

La deuxième contribution importante d'un point de vue *industriel* est le développement du prototype d'aide à l'interprétation d'un contre-exemple. L'expérimentation Ground Spoiler a montré la difficulté de comprendre les contre-exemples fournis par l'outil de model checking. Or cette étape d'interprétation des contre-exemples est une étape essentielle de l'approche et elle est répétée à chacune des itérations décrites dans le chapitre 2. Il est donc très intéressant d'avoir un prototype d'analyse automatisée dont l'efficacité et la pertinence ont été démontrées dans le chapitre 6. On peut remarquer que ce prototype a été développé dans le contexte spécifique des applications de commande de vol, mais qu'il est applicable plus généralement à d'autres modèles SCADE, la seule partie spécifique étant le traitement des symboles de la bibliothèque Airbus.

Les *perspectives industrielles* de ces travaux se placent sur trois axes :

- intégration de trois nouveaux outils dans l'environnement de développement du concepteur ;
- élargissement éventuel de l'utilisation du model checking ;
- évolution dans la façon de concevoir pour faciliter la vérification formelle.

Les trois outils envisagés dans le premier axe sont un outil d'analyse automatisée des contre-exemples, un outil de génération multiple de contre-exemples et un outil de gestion de configuration. Les deux premiers outils peuvent être obtenus en améliorant et en industrialisant le prototype. L'ergonomie du prototype peut encore être améliorée, comme évoqué dans le chapitre 6, on peut envisager un affichage

temporel des événements pertinents du contre-exemple pour faciliter encore la compréhension de ce dernier. L'industrialisation du prototype permettra son intégration dans l'environnement de travail des concepteurs. Un outil de gestion de configuration spécifique pour la vérification serait également très utile pour supporter l'approche itérative de vérification présentée dans le chapitre 2.

Un élargissement de l'utilisation du model checking (deuxième axe) ne peut être envisagé qu'à condition que les outils évoluent et soient capables de traiter de façon plus efficace des modèles industriels.

Le troisième axe mentionné consiste à prendre en compte dès la conception les propriétés à vérifier, il s'agirait de proposer une conception modulaire qui encapsule pour chaque module les propriétés à vérifier sur ce module. Cette méthode permettrait d'une part de découper la conception de façon à disposer de modules dont les propriétés pourraient être vérifiées par model checking. Elle aurait également l'avantage de faciliter la réutilisation : on pourrait réutiliser un module avec ses propriétés et ainsi diminuer l'effort de vérification pour les modèles futurs.

D'un point de vue *académique*, les leçons tirées de la synthèse des expérimentations sont également intéressantes car elles permettent de mettre en lumière certains points faibles de l'approche de vérification formelle du point de vue de son utilisation par des ingénieurs. Cette synthèse fournit des pistes d'amélioration pour les équipes qui travaillent à l'élaboration de ces techniques et des outils associés. En particulier, on rappelle le temps d'analyse important du model checker lorsque ce dernier traite des modèles composés uniquement d'opérateurs booléens et temporels.

La définition de la notion de cause d'un contre-exemple donné est une deuxième contribution intéressante. Cette notion est plus complexe qu'il n'y paraît, sa définition a demandé plusieurs raffinements successifs, comme le montre la discussion sur ce sujet proposée dans le chapitre 4. L'analyse automatisée de contre-exemples basée sur cette notion de cause propose un parcours en arrière de la structure du modèle qui sélectionne les arcs participant à la valeur de sortie du modèle. Cette analyse utilise la notion de condition d'activation définie pour des critères de couverture structurelle, mais elle est originale dans son objectif et dans le fait qu'elle s'appuie sur un scénario existant qu'il s'agit de filtrer.

Notre analyse est pour l'instant limitée aux modèles booléens à horloge unique. Les premières perspectives sont donc d'étendre la notion de cause aux opérateurs numériques et aux modèles multi-horloges. Les travaux [Papailiopoulou 2008] nous fournissent une première piste dans ce domaine.

La notion de cause permet également de définir la notion de contre-exemples différents comme étant des contre-exemples exhibant des causes différentes. La première étape d'analyse du contre-exemple ouvre de nombreuses perspectives concernant la génération de contre-exemples différents. Une première approche a été implémentée dans le prototype et présentée au chapitre 6, mais il s'agit seulement d'une expérimentation permettant de confirmer la faisabilité et l'intérêt de la génération de contre-exemples multiples. Plusieurs stratégies différentes pourront être proposées, étudiées et comparées. La définition de ces stratégies pourra s'inspirer des travaux

dans le domaine de la génération de tests structurels. On pourra notamment considérer les travaux autour de GATeL[Marre 2000] et les approches concoliques mises en oeuvre dans PathCrawler [Mouy 2008, Williams 2005] et CUTE [Sen 2005].

Annexe A

Précisions sur le model checker Prover Plug-in

Prover Plug-in permet de vérifier des propriétés de sûreté sur des *systèmes de transition*.

Un *système de transition* est un triplet (S, S_0, T) tel que :

- S est un ensemble d'états,
- $S_0 \subseteq S$ est l'ensemble des états initiaux,
- $T \subseteq S \times S$ est la *relation de transition*.

Une propriété de sûreté P est un ensemble d'états représentant les états correctes. Soit $Reach_T(S)$ l'ensemble des états atteignables depuis S en utilisant la *relation de transition* T.

Considérons un *système de transitions* $M = (S, S_0, T)$ et une propriété de sûreté P, le model checker a pour objectif de décider si $Reach_T(S_0) \subseteq P$.

Les modèles SCADE sont des *systèmes de transitions*. L'état d'un modèle SCADE est représenté par la valeur courante de tous ses flots de données (toutes ses variables). L'ensemble des états initiaux est spécifié en utilisant l'entrée *Init* de l'opérateur FBY et la *relation de transition* est spécifiée par l'*entrée retardée* du FBY. L'ensemble des états est l'ensemble de toutes les valuations de flots de données possibles du modèle. Si le modèle utilise des variables à types non bornés (entiers ou réels), cet ensemble est infini. Malgré le fait que SCADE peut spécifier des expressions arithmétiques complexes, le model checker est limité à :

- l'arithmétique linéaire sur $\mathbb{Q}$, c'est à dire des expressions de la forme :
  $$a_0 \times C_0 + \ldots + a_n \times C_n \diamond C$$
  où $a0, \ldots, a_n$ sont des variables, $C, C_0, \ldots, C_n$ sont des constantes et $\diamond \in \{=, \neq, >, <, \geq, \leq\}$.
- l'arithmétique non linéaire sur des domaines finis.

Prover Plug-in représente l'ensemble des états atteignables d'un modèle symboliquement en utilisant des prédicats. Ainsi, un problème de *non atteignabilité* d'états indésirables devient un problème de *non satisfiabilité* de formules logiques et arithmétiques linéaires. Cette technique est connue sous le nom de *SAT-based model checking* étendu à l'arithmétique.

Pour un ensemble d'états S, $S(s)$ est un prédicat tel que $s \in S \Leftrightarrow S(s)$. Pour une séquence d'états $s_0 \ldots s_n$, $path(s_0 \ldots s_n)$ est un prédicat signifiant que la séquence correspond à un chemin dans le graphe des états atteignables.

$$path(s_0 \ldots s_n) = \forall i \in [0, n-1], T(s_i, s_i + 1)$$

Le problème d'atteignabilité pour un *système de transitions* (S, S_0, T) et une propriété de sûreté P peut se reformuler de la manière suivante :

$$\forall n \geq 0, \forall s_0 \dots s_n, path(s_0 \dots s_n) \wedge S_0(s_0) \Rightarrow P(s_n)$$

Il existe deux techniques permettant de résoudre ce problème :
- *Bounded model checking* [Clarke 2001]
- *Induction over time* [Sheeran 2000]

La première technique est très pratique pour le debuggage, c'est à dire pour trouver des erreurs dans des modèles erronés.

$$bmc_n(s_0 \dots s_n) = path(s_0 \dots s_n) \wedge S_0(s_0) \Rightarrow P(s_n)$$

Cette technique procède de façon itérative en incrémentant n jusqu'à obtenir $bmc_n(s_0 \dots s_n)$ valué à FAUX. Auquel cas, cela signifie que $s_0 \dots s_n$ est un des plus court chemin vers un *état indésirable*. Cependant, cette technique n'a pas de terminaison pour les systèmes corrects.

La seconde méthode tente de démontrer par k-induction que le système est correct
- Cas de base : $bmc_n(s_0 \dots s_n)$ est une tautologie.
- Hypothèse d'induction : $ih_n(s_k \dots s_{k+n}) = path(s_k \dots s_{k+n}) \wedge \forall i \in [0, n], P(s_{k+i})$
- étape d'induction : $is_n(s_{k+n}) = \forall s_{k+n+1}, T(s_{k+n}, s_{k+n+1}) \Rightarrow P(s_{k+n+1})$

Cette technique incrémente n depuis 0 jusqu'à :

$$(\forall s_0 \dots s_n, bmc_n(s_0 \dots s_n)) \wedge (\forall s_k \dots s_{k+n}, ih_n(s_k \dots s_{k+n}) \Rightarrow is_n(s_{k+n}))$$

Si tous états des chemins de longueur n partant d'état initiaux respectent P et si tous les états de n'importe quel chemin de longueur n respectent P alors tous les états atteignables respectent P

Si la technique termine, elle démontre que le système est correcte, c'est à dire qu'il ne contient pas d'état violant P. Dans le cas contraire, l'étape utilisant la technique du *Bounded model checking* du cas de base détectera l'état violant P. Cette procédure est cependant incomplète. Considérons l'exemple de la figure A.1.

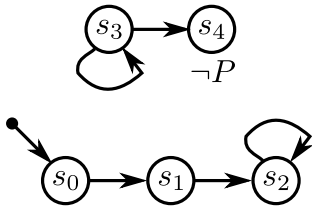


FIGURE A.1 – Exemple d'un *système de transitions*

L'état initial est s_0. s_1 et s_2 sont atteignables à partir de s_0. L'état inatteignable s_3 possède une transition vers lui même et une transition vers un état indésirable s_4. Pour cet exemple, l'étape d'induction ne terminera pas même si ce système est correcte. Ce problème est résolu en remplaçant le prédicat *path* par *loop-free path* :

$$path(s_0 \dots s_n) = \forall i \in [0, n-1], T(s_i, s_i + 1) \wedge \forall j \in \{0, \dots, n\}, (i \neq j) \Rightarrow (s_i \neq s_j)$$

Le model checker Prover Plug-in implémente un solveur [Andersson 2002] qui combine des techniques SAT [1] telles que la méthode de Stalmarck [Sheeran 1998], Davis-Putnam-Loveland-Logemann [Davis 1962], Reduced Ordered Binary Decision Diagrams [Bryant 1986], des techniques de programmation linéaire et de propagation de contraintes. En pratique, le moteur de preuve peut décider un sur-ensemble stricte de formule de type math-formulas [Audemard 2002]. Même si une formule de satisfiabilité contient des prédicats non linéaires, le moteur de preuve arrive souvent à la décider. Les entiers bornés sont transformés en vecteur de bits afin de pouvoir appliquer les techniques d'arithmétique binaire. Cette méthode est capable de manipuler l'arithmétique non linéaire bornée.

1. techniques permettant de résoudre des problèmes de satisfiabilité

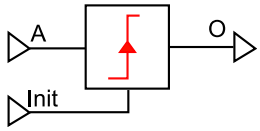
Annexe B

Les opérateurs complexes

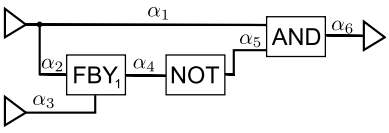
Cette annexe a pour objectif de décrire la sémantique des opérateurs de la *bibliothèque des symboles* Airbus : PULSE1, R*S, MTRIG1. Elle contient les algorithmes implémentés par les fonctions $analyzePULSE1$, $analyzeR*S$ et $analyzeMTRIG1$.

B.1 L'opérateur PULSE1

Considérons le modèle constitué d'un opérateur PULSE1, dont la représentation graphique est visible sur la figure B.1(a) et dont le comportement est défini par le modèle de référence de la figure B.1(b). Les arcs sont annotés de α_1 à α_6. Les arc α_1 et α_2 sont associés à la variable A et α_3 est associé à $Init$.



(a) Représentation graphique de PULSE1



(b) Modèle équivalent du PULSE1

Figure B.1 – Opérateur PULSE1

Cet opérateur est un détecteur de fronts montants. La sortie $O(n)$ est VRAI lorsqu'un front montant sur l'entrée A se produit entre les instant $n-1$ et n. A l'instant initial $n=1$, $A(n-1)$ est remplacé par $Init(n)$. La sortie est fausse dans tous les autres cas. L'exécution suivante permet d'illustrer le comportement de la bascule R*S :

n	1	2	3	4	5	6	7	8	9
$A(n)$	0	1	1	1	0	1	0	1	1
$O(n)$	0	1	0	0	0	1	0	1	0

On observe trois fronts montants aux instants 2, 6 et 8.

B.1.0.1 AnalyzePulse1

L'opérateur PULSE1 a pour objectif de détecter les fronts montants. Lorsqu'un front apparaît sur l'entrée, la sortie est VRAI. La sémantique de cet opérateur est détaillée en annexe B.1.

La fonction $analyzePULSE1$ est invoquée lorsque l'arc α sort d'un opérateur PULSE1. Les variables V_{in}, V_{out} et V_{init} sont respectivement définies avec les flots

associés aux arcs d'entrée, de sortie et d'initialisation. Le raisonnement de la fonction se décompose en deux parties : (1) Elle instancie potentiellement les ensembles E_1 et E_2, puis (2) elle compose ces ensembles avec le type de composition choisi. La première partie est la suivante :

- quelque soit l'instant d'analyse n, si $V_{in}(n)_\sigma = V_{out}(n)_\sigma$ alors la fonction affecte à E_1 le résultat de l'appel récursif à analyse(α_{in}, n) ;
- si $n > 1$ alors, si $V_{in}(n-1)_\sigma \neq V_{out}(n)_\sigma$, la fonction affecte à E_2 le résultat de l'appel récursif à analyse$(\alpha_{in}, n-1)$;
- si $n = 1$ alors, si $V_{init}(n)_\sigma \neq V_{out}(n)_\sigma$, la fonction affecte à E_2 le résultat de l'appel récursif à analyse(α_{init}, n).

La deuxième partie est la suivante :

- si $V_{out}(n)_\sigma$=VRAI alors, la fonction compose les ensembles E_1 et E_2 de manière *dépendante*.
- sinon, elle les compose de manière *indépendamment* pour former l'ensemble résultat E_c.

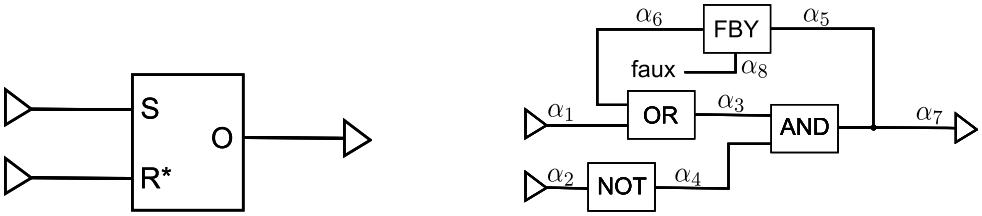
La fonction $analyzePULSE1$ implémente l'algorithme 14. Nous ne démontrerons pas que cette fonction produit toutes les causes minimales sur le modèle local ML composé d'un opérateur PULSE1.

Algorithm 14 analyzePULSE1(α, n)

$operator = from(\alpha)$
$\alpha_{in}, \alpha_{init} \leftarrow arcIn(operator)$
$E_1, E_2 \leftarrow \emptyset$
$V_{out} = arc2var(\alpha)$
$V_{in} = arc2var(\alpha_{in})$
$V_{init} = arc2var(\alpha_{init})$
if $V_{in}(n)_\sigma = V_{out}(n)_\sigma$ **then**
  $E_1 \leftarrow analyze(\alpha_{in}, n)$
end if
if $n > 1$ **then**
  if $V_{in}(n-1)_\sigma \neq V_{out}(n)_\sigma$ **then**
    $E_2 \leftarrow analyze(\alpha_{in}, n-1)$
  end if
else
  $\{n = 1\}$
  if $V_{init}(n)_\sigma \neq V_{out}(n)_\sigma$ **then**
    $E_c \leftarrow analyze(\alpha_{init}, n)$
  end if
end if
{Choix du mode de composition des ensembles}
if $V_{out}(n)_\sigma =$VRAI **then**
  $E_c \leftarrow combineCausesDep(E_1, E_2)$
else
  $E_c \leftarrow combineCausesIndep(E_1, E_2)$
end if
$E_c \leftarrow completeAllPath(E_c, \alpha, n)$
return E_c

B.2 L'opérateur bascule R*S

Considérons le modèle constitué d'un opérateur R*S dont la représentation graphique est visible sur la figure B.2 et nous donnons sur la figure B.2 son modèle de référence. Les arcs sont annotés de α_1 à α_8. L'arc α_1 est associé à la variable S. L'arc α_2 est associé à la variable R. L'arc α_8 est associé à la constante FAUX. L'arc α_7 est associé à la variable O.



(a) Représentation graphique de R*S

(b) Modèle équivalent de R*S

FIGURE B.2 – Opérateur bascule R*S

La bascule R*S se comporte comme un interrupteur. L'entrée S (set) permet d'activer la sortie O, cette dernière est alors valuée à VRAI. La sortie reste VRAI tant que R est FAUX. L'entrée R (reset) permet de désactiver la sortie O, cette dernière est alors valuée à FAUX. Dans le cas de la bascule R*S, le signal R a la priorité, cela signifie que si S et R sont VRAI au même instant, la sortie est valuée à FAUX. L'exécution suivante permet d'illustrer le comportement de la bascule R*S :

n	1	2	3	4	5	6	7	8	9
$S(n)$	0	1	0	0	0	1	1	1	1
$R(n)$	0	0	0	1	0	1	0	0	1
$O(n)$	0	1	1	0	0	0	1	1	0

On observe, à l'instant 2, une activation de la bascule. La sortie O est maintenue à VRAI à l'instant 3. A l'instant 4, on observe une désactivation. A l'instant 6, les signaux S et R sont VRAI simultanément, le signal R étant prioritaire, O reste FAUX.

B.2.0.2 AnalyzeR*S

La fonction $analyzeR*S$ est invoquée lorsque l'arc α sort d'un opérateur R*S. Les variables V_{out}, V_s et V_r sont respectivement définies avec les flots associés aux arcs de sortie, d'activation et de désactivation de la bascule. Le raisonnement de la fonction se décompose en deux parties :

1. si $V_{out}(n)_\sigma$=VRAI, la fonction recherche le *dernier* moment où la bascule a été activée : n_{set}. Elle compose ensuite de manière *dépendante* le résultat de l'analyse de l'entrée *set* à l'instant n_{set} et les analyses de l'entrée *reset* à tous les instants $n' \in [n_{set}, n]$.

2. si $V_{out}(n)_\sigma$=FAUX, la fonction recherche le *dernier* instant où l'entrée reset a été VRAI : n_{reset}. Elle compose ensuite de manière *dépendante* le résultat de l'analyse de l'entrée *reset* à l'instant n_{reset} et les analyses de l'entrée *set* à tous les instants $n' \in [n_{reset}+1, n]$.

La fonction $analyzeR*S$ implémente l'algorithme 15. Les *causes* renvoyés sont *minimales*. Dans cette fonction, nous avons fait le choix de n'explorer que le dernier *set* ou *reset* observé. L'analyse est ainsi simplifiée, mais incomplète au détriment d'être capable de calculer toutes les *causes minimales*.

Algorithm 15 analyzeR*S(α, n)

```
operator = from(α)
[α_s, α_r] ← arcIn(operator)
V_out = arc2var(α)
V_s = arc2var(α_s)
V_r = arc2var(α_r)
if V_out(n)_σ=VRAI then
  {la sortie de la bascule est VRAI}
  n_set ← dernier instant où V_s=VRAI, non défini si ∄ de reset
  if n_set défini then
    E_c ← analyze(α_s, n_set)
  else
    E_c ← analyze(α_init, 1)
    n_set ← 1
  end if
  for all n' ∈ [n_set, n] do
    E_c ←combineCausesDep(E_c, analyze(α_r, n'))
  end for
else
  {la sortie de la bascule est fausse}
  n_reset ←dernier instant où V_r=VRAI ou non défini si ∄ de reset
  if n_reset défini then
    E_c ← analyze(α_r, n_reset)
  else
    E_c ← analyze(α_init, 1)
    n_reset ← 0
  end if
  for all n' ∈ [n_reset + 1, n] do
    E_c ←combineCausesDep(E_c, analyze(α_s, n'))
  end for
end if
E_c ← completeAllPath(E_c, α, n)
return E_c
```

B.3 L'opérateur MTRIG1

Considérons le modèle constitué d'un opérateur MTRIG1, dont la représentation graphique est visible sur la figure B.3.

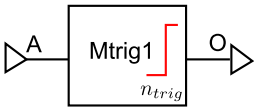


FIGURE B.3 – Représentation graphique de l'opérateur MTRIG1

Le paramètre $n_{trig} \in N^+$ est le nombre de cycles consécutifs de **maintien** du signal de sortie O à VRAI a lorsque le Mtrig se **déclenche**. Le **déclenchement** est réalisé lorsqu'il y a un *front montant* sur le signal d'entrée ($A(n-1)$ =FAUX et $A(n)$ =VRAI) et que la sortie $O(n-1)$ est FAUX. A l'instant 1 la valeur précédente de $O(n-1)$ est considérée fausse. La figure B.3 représente le modèle de référence de l'opérateur Mtrig1 dont le paramètre n_{trig} est 3.

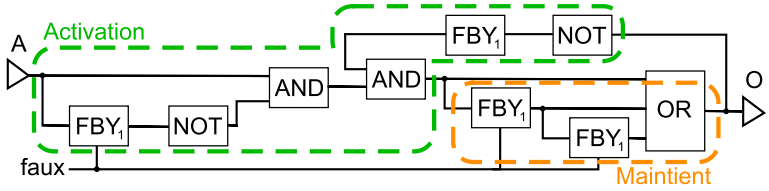


FIGURE B.4 – Modèle de référence du MTRIG1 avec n_{trig}=3

En vert est encadrée la partie du modèle responsable de l'**activation** du MTRIG. En orange est encadrée la partie du modèle responsable du **maintien** de la sortie à VRAI durant les deux cycles qui suivent l'activation. L'exécution suivante permet d'illustrer le comportement de l'opérateur MTRIG1 avec n_{trig}=3 :

n	1	2	3	4	5	6	7	8	9	10	11
$A(n)$	0	1	0	0	0	1	0	1	1	0	1
$O(n)$	0	1	1	1	0	1	1	1	0	0	1

On observe au cycle 2 un *déclenchement*. La sortie O est VRAI au cycle 2 et est *maintenue* trois cycles, c'est à dire jusqu'au cycle 4 inclus. Il en est de même aux cycles 6 et 11.

B.3.0.3 analyzeMTRIG1

Nous allons maintenant présenter la fonction *analyzeMTRIG*1. Le raisonnement de la fonction se décompose en deux parties : (1) l'instant d'analyse $n > 1$ et (2) $n = 1$.

n > 1

- Dans le cas où la sortie $V_{out}(n)_\sigma$=VRAI, l'instant qui nous intéresse est l'instant de déclenchement du MTRIG1. Il est possible de déterminer cet instant en observant le dernier front montant de la sortie. On note n' le dernier instant où $V_{out}(n')_\sigma$=FAUX et $V_{out}(n'+1)_\sigma$=VRAI. A cet instant, on dit que l'opérateur MTRIG1 s'active. Les *causes* de l'activation sont les valuations de $V_{in}(n')$ qui est valué à FAUX, $V_{in}(n'+1)$ qui est valué à VRAI et V_{out} qui est valué à FAUX. La fonction appelle récursivement *analyze* de manière *dépendante* sur l'arc d'entrée α_{in} aux instants n' et $n'+1$ et sur l'arc de sortie du MTRIG1 à l'instant n'.
  Il est important de noter, pour la terminaison de l'algorithme, que lors de l'appel récursif de $analyze(\alpha, n')$, le niveau d'arc de α est supérieur à celui de α_{in} mais que n' est strictement inférieur à n impliquant que le variant diminue. La terminaison de l'algorithme n'est pas compromise.
- ceux où la sortie du MTRIG1 est FAUX cela implique que $\forall n' \in [n - n_{trig}, n], A(n')$=FAUX. Tous les valuations de l'entrée A à ces instants sont des causes indépendantes de la valuation de la sortie O à l'instant n. La fonction compose de manière *indépendante* tous les résultats des appels récursifs de $analyze(\alpha_i n, n')$.

n = 1 Dans le cas particulier où $n = 1$, la valeur de sortie du MTRIG1 est égale à sa valeur d'entrée. La fonction appel donc récursivement $analyze(\alpha_{in}, n)$.

La fonction *analyzeMTRIG1* implémente l'algorithme 16.

Algorithm 16 analyzeMTRIG1(α, n)

$operator = from(\alpha)$
$n_{trig} \leftarrow getParameter(operator)$
$\alpha_{in} \leftarrow arcIn(operator)$
$E_c \leftarrow \emptyset$
if $(n > 1)$ **then**
  $V_{out} = arc2var(\alpha)$
  if $V_{out}(n)_\sigma$=VRAI **then**
    {la sortie du MTRIG1 est VRAI}
    $n' \leftarrow$ le dernier instant où V_{out} est FAUX.
    $E_c \leftarrow$combineCausesDep($E_c, analyze(\alpha_{in}, n' + 1)$)
    $E_c \leftarrow$combineCausesDep($E_c, analyze(\alpha_{in}, n')$)
    $E_c \leftarrow$combineCausesDep($E_c, analyze(\alpha, n')$)
  else
    {la sortie du MTRIG1 est fausse}
    for all $n' \in [n - n_{trig}, n]$ **do**
      $E_c \leftarrow combineCausesIndep(E_c, analyze(\alpha_{in}, n'))$
    end for
  end if
else
  $\{n = 1\}$
  $E_c \leftarrow analyze(\alpha_{in}, n)$
end if
$E_c \leftarrow completeAllPath(E_c, \alpha, n)$
return E_c

ANNEXE C

Preuve de minimalité

Contrairement à « être une cause », « être une cause *minimale* » n'est pas une propriété inductive dans le cas général. Lorsque notre algorithme compose des causes minimales pour les entrées d'un opérateur, cela ne produit pas nécessairement des causes minimales pour la sortie de l'opérateur.

Notre preuve de minimalité va donc s'effectuer dans un contexte restreint : nous supposons que le modèle aplati M_{flat} ne comporte pas de chemins reconvergents de même ordre. Cette hypothèse entraîne un certain nombre de bonnes propriétés (voir §C.1, préliminaires), sur lesquelles nous nous appuierons pour exhiber une preuve par induction sur l'arbre d'exécution.

C.1 Préliminaires

C.1.1 Notion d'indépendance vis-à-vis d'une valeur d'entrée

Formalisons d'abord la notion de valeur d'une variable intermédiaire V(n) *indépendante* d'une valeur d'entrée $E(n_i)$, quel que soit le scénario retenu.

Définition 1. On dira que V(n) est *indépendante* de $E(n_i)$ si et seulement si :
$\forall \sigma \in \Sigma_{in}$ de longueur $\geq \max(n, n_i)$, $\forall \sigma' \in \Sigma_{in}$ construit à partir de σ en changeant la seule valeur $E(n_i)$, on a $V(n_i)_{\sigma} = V(n_i)_{\sigma'}$.

Une condition suffisante pour que V(n) soit indépendante de $E(n_i)$ est qu'il n'y ait aucun chemin de propagation possible entre les deux.

Propriété 1 . Si le modèle M_{flat} ne comporte aucun chemin p d'ordre $(n\text{-}n_i)$, tel que arc2var(naissance(p))=E et arc2var(mort(p))=V, alors V(n) est indépendante de $E(n_i)$.

De même, on peut formaliser la notion de condition d'activation d'un chemin p à l'instant n *indépendante* d'une valeur d'entrée $E(n_i)$, quel que soit le scénario retenu.

Définition 2. On dira que la condition d'activation de p à l'instant n est *indépendante* de $E(n_i)$ si et seulement si :
$\forall \sigma \in \Sigma_{in}$ de longueur $\geq \max(n, n_i)$, $\forall \sigma' \in \Sigma_{in}$ construit à partir de σ en changeant la seule valeur $E(n_i)$, on a $A(p, \sigma, n) = \text{VRAI} \Leftrightarrow A(p,\sigma',n) = \text{VRAI}$.

La condition d'activation d'un chemin est définie récursivement, en parcourant les opérateurs traversés par ce chemin. La dépendance vis-à-vis d'une entrée est introduite soit directement à l'origine du chemin, soit plus tard au moment de la traversée d'un opérateur. Une condition suffisante d'indépendance est alors la suivante.

Propriété 2. Si :
- Origine $(p,n) \neq (E,n_i)$, et
- Il n'existe pas d'opérateur OP traversé par p via les arcs α_{e_op}, α_{s_op} tel que :
  - Le chemin $<\alpha_{s\_op}, \ldots, mort(p)>$ suffixe de p est d'ordre n',
  - Il existe un chemin p_i, avec $mort(p_i) = \alpha_{s\_op}$ et $origine(p_i, n\text{-}n') = (E, n_i)$.

alors la condition d'activation de p à l'instant n est indépendante de $E(n_i)$.

Les propriétés 1 et 2 vont nous servir à énoncer certaines relations d'indépendance existant dans un modèle sans chemins reconvergents de même ordre.

C.1.2 Relations d'indépendance dans un modèle sans chemins reconvergents de même ordre

Une propriété évidente des modèles sans chemins reconvergents de même ordre est la suivante.

Propriété 3. Si M_{flat} est un modèle sans chemins reconvergents de même ordre, alors il existe au plus un chemin d'ordre n entre une entrée E et un arc α.

La propriété 4 particularise la propriété 3. Elle exprime le fait que deux chemins d'une cause ne peuvent avoir une même origine (puisqu'ils ont le même arc final).

Propriété 4. Soit M_{flat} un modèle sans chemins reconvergents de même ordre. Toute cause *c* de la valuation d'un arc de M_{flat} à un instant n est telle que :
$\forall p_1, p_2 \in c, p_1 \neq p_2 \Rightarrow origine\ (p_2, n) \neq origine\ (p_2, n)$.

La propriété 5 s'intéresse aux arcs entrants dans un opérateur de base, pour construire une valeur de sortie à l'instant n. Elle énonce plusieurs relations d'indépendance.

Propriété 5. Soit M_{flat} un modèle sans chemins reconvergents de même ordre. Soient α_1 et α_2 deux arcs d'entrée d'un opérateur de M_{flat}, et α un arc de sortie de cet opérateur. On s'intéresse aux valeurs des variables V_1 et V_2 portées par α_1 et α_2 aux instant n_1 et n_2, avec :

$$n_1 + ordre\ (<\alpha_1, \alpha>) = n_2 + ordre\ (<\alpha_2, \alpha>) = n.$$

Soit P_1 l'ensemble des chemins d'ordre $< n_1$, connectant une entrée de M_{flat} à l'arc α_1. De même, soit P_2 l'ensemble des chemins d'ordre $< n_2$, connectant une entrée de M_{flat} à l'arc α_2. On a alors :

5.a Pour toute paire de chemin $p_1 \in P_1$ et $p_2 \in P_2$, $origine\ (p_1, n_1) \neq origine\ (p_2, n_2)$.

5.b $V_1(n_1)$ est indépendant de toute entrée $E_2(n'_2)$, telle que $(E_2, n'_2) = origine(p_2, n_2)$, où $p_2 \in P_2$.

5.c Tout chemin $p_1 \in P_1$ a une condition d'activation à l'instant n_1 qui est indépendante de $E_2(n'_2)$, avec $(E_2, n'_2) = origine(p_2, n_2)$, et p_2 est un chemin de P_2.

Preuve.

La composante 5.a découle de la propriété 3. On ne peut pas avoir deux chemins reconvergents de même ordre entre l'entrée et l'arc α, l'un via α_1 et l'autre via α_2.

La composante 5.b découle des propriétés 1 & 3. Il n'y a pas de chemin de propagation possible entre $E_2(n'_2)$ et $V_1(n_1)$, car cela conduirait à deux chemins reconvergents d'ordre (n-n'$_2$) entre l'entrée et l'arc α, l'un via α_1 et l'autre via α_2. Les prémisses de la propriété 1 s'appliquent donc.

Pour la composante 5.c, on se sert de 5.a pour établir origine(p_1, n_1) $\neq$ (E_2, n'$_2$). Pour pouvoir satisfaire les prémices de la propriété 2, il faut alors montrer qu'il n'existe pas d'opérateur OP traversé par p_1 via les arcs α_{e_op}, α_{s_op} tel que :

- Le chemin $<\alpha_{s\_op}, \ldots, mort(p_1)>$ suffixe de p_1 est d'ordre n',
- Il existe un chemin p_i, avec $mort(p_i) = \alpha_{s\_op}$ et origine(p_i, n_1- n') = (E_2, n'$_2$).

Si un tel opérateur existait, on pourrait construire un chemin de propagation d'ordre n-n'$_2$ entre l'entrée E_2 et l'arc α, en concaténant : p_i, le suffixe de p_1 après l'opérateur, $<\alpha>$. Ce chemin serait différent de p_2, et violerait la propriété 3. L'opérateur OP n'existe donc pas.

Dans la propriété 5, α_1 et α_2 sont deux arcs entrant dans un opérateur de base. Nous allons maintenant voir qu'on a les mêmes relations d'indépendance lorsque les arcs α_1 et α_2 de M_{flat} sont les analogues d'arcs de M entrant dans un opérateur quelconque (opérateur de base, ou symbole plus complexe comme un confirmateur).

Propriété 6. Toutes les relations d'indépendance exprimées dans la propriété 5 s'appliquent également au cas où :

- α_1 et α_2 sont deux arcs de M_{flat}, qui sont les analogues d'arcs de M entrant dans un opérateur de M,
- α est l'analogue dans M_{flat} de l'arc de sortie de cet opérateur dans M.
- n_1 + ordre (p_{L1}) = n_2 + ordre (p_{L2}) = n, où p_{L1} et p_{L2} sont deux chemins de M_{flat} de la forme $< \alpha_1, \ldots, \alpha >$ et $< \alpha_2, \ldots, \alpha >$,

On notera 6.a, 6.b et 6.c les relations correspondantes.

Preuve. On procède comme dans la preuve de la propriété 5, en s'appuyant sur l'impossibilité d'exhiber plusieurs chemins de même ordre entre une entrée et l'arc α. Dans la preuve de la composante 6.c, le chemin impossible est maintenant formé par concaténation de : p_i, le suffixe de p_1 après l'opérateur, p_{L1}.

Les propriétés 5 et 6 considèrent un ensemble P_i des chemins connectant des entrées à un arc α_i. Or, une cause de la valuation d'un arc à un instant n'est autre qu'un sous ensemble de P_i, contenant certains chemins activés par un scénario σ à cet instant. Toutes les relations d'indépendance que nous avons exprimées s'appliquent donc, en remarquant que :

$C(\alpha_1, n_1, \sigma) \subseteq P_1$ et $C(\alpha_2, n_2, \sigma) \subseteq P_2$.

La propriété 4 (qui concerne les chemins d'une même cause), et la propriété 6 (qui concerne notamment les chemins provenant de deux causes pour les entrées d'un opérateur quelconque) vont être très utiles pour la preuve de minimalité. Nous les utiliserons lorsque nous considèrerons le traitement d'un opérateur, avec des appels récursifs sur les entrées de cet opérateur.

C.1.3 Commandabilité

Pour les modèles sans chemins reconvergents de même ordre, une autre propriété très utile pour nous est que toute variable d'arc est commandable, c'est à dire qu'on peut toujours forcer une valeur particulière à un instant n quelconque.

Propriété 7. Si M_{flat} est un modèle sans chemins reconvergents de même ordre, et α est un arc de M_{flat} portant la variable V, alors :

- $\forall n \in N^*, \exists \sigma \in \Sigma_{in}, V(n)_\sigma = \text{VRAI}$
- $\forall n \in N^*, \exists \sigma \in \Sigma_{in}, V(n)_\sigma = \text{FAUX}$

Preuve. La preuve s'effectue par induction sur la structure de M_{flat}.

- Une variable d'entrée est commandable à tout instant.
- D'après la sémantique des opérateurs booléens, la variable de sortie d'un opérateur AND, OR, NOT est commandable à l'instant n, si ses entrées sont commandables à l'instant n de façon indépendante. La propriété 5 garantit l'indépendance. Il suffit donc d'avoir la commandabilité des entrées à l'instant n.
- De même, la sortie d'un opérateur ITE est commandable à l'instant n si ses trois entrées sont commandables à l'instant n.
- La sortie d'un opérateur FBY est commandable à l'instant 1 si son entrée d'initialisation est commandable à l'instant 1. La sortie est commandable à l'instant n>1 si l'entrée retardée est commandable à l'instant n-1.

Une conséquence de la propriété 7 est qu'un ensemble vide de chemins n'est jamais une cause.

Propriété 8. Si M_{flat} est un modèle sans chemins reconvergents de même ordre, et α est un arc de M_{flat} portant la variable V, alors :

- $\forall \sigma \in \Sigma_{in}, \forall n \in [1, \text{longueur } \sigma], \{\} \notin C(\alpha, n, \sigma)$

Preuve. Si {} était une cause, on aurait : $\forall \sigma' \in \Sigma_{in}, V(n)_{\sigma'} = V(n)_\sigma$. La variable V ne serait pas commandable, ce qui contredit la propriété 7.

C.2 Preuve inductive de la minimalité

Grâce aux bonnes propriétés des modèles sans chemins reconvergents de même ordre, la preuve de minimalité peut suivre la même démarche inductive que la preuve sur la production de causes. On se base sur l'arbre d'exécution :

– Le cas terminal $analyzeIN_{flat}$ fournit une cause *minimale* (la preuve est triviale).
– Le point dur est donc de montrer que le traitement des autres opérateurs fournit des causes *minimales* si les appels récursifs fournissent des causes minimales.

Au niveau du traitement d'un opérateur, l'analyse du modèle local détermine les appels récursifs à effectuer, et la façon de combiner les résultats de ces appels. Nous avons déjà montré (chapitre 5) que notre analyse du modèle local identifie des causes locales minimales. Il reste à montrer que, connaissant des causes minimales dans le modèle local, et connaissant des causes minimales provenant d'appels récursifs, on produit bien des causes minimales

pour la sortie de l'opérateur. La preuve distingue les deux cas présents dans notre algorithme : (1) combinaison indépendante et (2) combinaison dépendante de causes.

2.1 Minimalité dans le cas *indépendant*

Dans ce cas, l'analyse du modèle local détermine un ensemble de chemins p_i tel que $\{p_i\}$ est une cause minimale de (α,n) dans le modèle ML_{flat}. Pour chaque chemin p_i, soit α_i le premier arc, et n_i = n-ordre (p_i). Toute cause c construite par l'algorithme est alors de la forme :

$$\text{concatPath}_{flat}(c', p_i), \qquad \text{où } c' \in \text{analyze}_{flat}(\alpha_i, n_i).$$

Montrons que c ainsi construite à partir de c' et p_i est minimale. Si elle n'est pas minimale, alors il existe un chemin $p \in c$ tel que $c - \{ p \}$ est une cause. Comme les origines des chemins de c sont toutes disjointes (propriété 4), cela revient à ne plus contraindre la valeur d'entrée $e_p(n_p)_\sigma$, où (e_p, n_p) = origine(p,n).
Par construction, le chemin p enlevé est de la forme $\text{concat}_{flat}(p', p_i)$, où $p' \in c'$.
De plus, $c - \{ p \} = \bigcup_{p'' \in c' - \{p'\}} concat_{flat}(p'', p_i)$.
Les origines des chemins sont donc les mêmes. Les ensembles $c - \{ p \}$ et $c' - \{ p' \}$ définissent les mêmes contraintes sur les entrées à valuer.

Nous savons que $c'-\{p'\}$ n'est pas une cause de (α_i, n_i) (hypothèse d'induction). Il existe donc un un scénario $\sigma' \in \Sigma_{in}$, satisfaisant :

- σ' est de même longueur que σ.
- $e_i(n'_i)_{\sigma'} = e_i(n'_i)_\sigma$ pour tout $(e_i, n'_i) \in$ Origines',
  où Origines' = $\{(e_i, n'_i) \in Var_{in} \times N^* \mid \exists\, p'' \in c'-\{p'\}, (e_i, n'_i) = \text{origine}(p'', n_i)\}$
- au moins un des deux cas (a) et (b) se produit,
  (a) $\exists\, p'' \in c'-\{p'\}$, $A(p'', \sigma', n_i)$ = FAUX
  (b) $V_i(n_i)_{\sigma'} \neq V_i(n_i)_\sigma$ avec V_i = arc2var(α_i).

Supposons que le cas (a) se produise. Le chemin $\text{concat}_{flat}(p'', p_i)$ n'est pas actif à l'instant n, car sa condition d'activation est équivalente à $\mathcal{AC}(p'') \wedge \mathcal{AC}(p_i)$. Ce chemin appartient à $c-\{p\}$, donc l'existence de σ' montre que $c-\{p\}$ n'est pas une cause.

Supposons que le cas (b) se produise. Soit V_s = arc2var(α) la variable de sortie de l'opérateur. Si l'exécution de σ' value V_s différemment de σ à l'instant n, alors l'existence de σ' montre que $c-\{p\}$ n'est pas une cause. Sinon, il faut trouver un autre scénario σ'', permettant de conclure sur ce point. Pour cela, nous allons nous appuyer sur σ' et sur un scénario défini au niveau du modèle local ML_{flat}.

Au niveau du modèle local ML_{flat}, $\{\}$ n'est pas une cause (minimalité de $\{p_i\}$, et aussi propriété 8). Il existe donc un scénario local σ_L, dont l'exécution value la variable de sortie V_s différemment de σ : $V_s(n)_{\sigma L} \neq V_s(n)_\sigma$. De plus, ce scénario value la variable V_i[1] à l'instant

[1] Pour le scénario local, V_i est une entrée.

n_i différemment de σ, car sinon la variable de sortie aurait la même valeur qu'avec σ ($\{p_i\}$ est une cause). Les variables étant booléennes, il n'y a que deux possibilités de valuation, d'où $V_i\ (n_i)_{\sigma L} = V_i\ (n_i)_{\sigma'}$. Les autres valeurs d'entrée locales de σ_L peuvent, ou non, correspondre à celles données par l'exécution de σ ou σ'. Nous allons maintenant montrer l'existence d'un scénario global, $\sigma'' \in \Sigma_{in}$, qui reproduit certaines caractéristiques du scénario local σ_L :

- il force une valuation de la variable de sortie à l'instant n différente de $V_s(n)_\sigma$

tout en gardant certaines caractéristique de σ' :

- il force une valuation de la variable V_i à l'instant n_i différente de $V_i\ (n_i)_\sigma$
- il respecte les contraintes de valuation associées à c-{p}.

Pour reproduire le scénario σ_L, remarquons tout d'abord que seul un sous-ensemble des valeurs d'entrée de ML_{flat} peut avoir une influence sur la valeur de V_s à l'instant n : ce sont les valeurs déterminées par les origines de chemins complets locaux d'ordre <n (propriété 1). Soit $Origine_L$ l'ensemble des origines de tels chemins. Par exemple, $Origine_L$ contient le couple (V_i, n_i), correspondant à la valeur d'entrée locale $V_i(n_i)_{\sigma L}$. Pour forcer la valeur de $V_s(n)_{\sigma L}$ depuis les entrées de M_{flat}, il suffit donc de reproduire les valeurs intermédiaires correspondant à $Origine_L$.

D'après les propriétés 6 et 7, ces valeurs sont commandables de façon indépendante. Pour chaque couple (α_j, n_j), soit E_j l'ensemble des valeurs d'entrée $e_{jk}(n_{jk})$ tel qu'il existe un chemin d'ordre n_j-n_{jk} connectant l'entrée e_{jk} à l'arc α_j. Tous les ensembles E_j sont disjoints (propriété 6.a). Quel que soit le scénario, la valeur de arc2var(α_j) (n_j) ne dépend que des valeurs d'entrée dans E_j (propriété 5.b). On peut choisir ces valeurs d'entrée pour donner arc2var(α_j)$(n_j)_{\sigma L}$ (propriété 6). Pour j≠i, on choisira n'importe quelle valuation de E_j donnant arc2var(α_j)$(n_j)_{\sigma L}$. Pour commander $V_i\ (n_i)$, on retiendra la même valuation de E_i que celle donnée par σ'. Les autres entrées sont valuées de manière arbitraire.

On a ainsi construit un scénario σ'' qui respecte les contraintes de valuation associées à c-{p}, tout en produisant une sortie $V_s\ (n)_{\sigma''} \neq V_s\ (n)_\sigma$. Donc c-{p} n'est pas une cause.

Après examen des cas (a) et (b), nous pouvons conclure que c est une cause minimale.

C.2.2 Minimalité dans le cas *dépendant*

Dans le cas dépendant, les chemins p_1, …, p_k sélectionnés sont tels que $c_L = \{p_1, \ldots, p_k\}$ est une cause minimale de (α,n) au niveau du modèle local. Après appel récursif pour construire les causes des origines de chaque chemin de c_L, la production de l'ensemble final Ec est moins simple que l'union réalisée dans le cas indépendant. Mais au final, la même stratégie de preuve peut être appliquée. Nous la rappelons brièvement ci-dessous.

On prend une cause c, et on enlève un chemin p. Ce chemin a pour suffixe un des chemins locaux p_i sélectionnés. On remarque alors que l'information restante est insuffisante pour être une cause du (α_i, n_i) correspondant. On a donc un scénario $\sigma' \in \Sigma_{in}$ tel que :

(a) un des chemins de propagation restants devient inactif
(b) la valeur $V_i\ (n_i)$ est modifiée.

Dans le cas (a), on montre facilement que c –{ p} n'est plus une cause.

Dans le cas (b), la démonstration est également facile si :

- la sortie de l'opérateur $V_s(n)_{\sigma'} \neq V_s(n)_{\sigma}$
- un des chemins de propagation locaux, p_j, devient inactif.

Si aucun de ces sous-cas ne se produit, il faut faire appel à un scénario local, σ_L, qui reprend les mêmes valeurs que σ' pour les origines de chemins dans c_L, et qui change la valeur de la sortie ou la condition d'activation d'un chemin de $c_L - \{ p_i \}$. L'existence de σ_L est garantie par le fait que $c_L - \{ p_i \}$ n'est pas une cause au niveau du modèle local. Ce scénario local peut être reproduit par un scénario global σ'' (commandabilité des valeurs locales, et indépendance des valeurs d'entrée utilisées pour commander les valeurs locales), montrant ainsi que c-{p} n'est pas une cause.

Bibliographie

[Abramovici 1990] Miron Abramovici, Melvin A. Breuer et Arthur D. Friedman. Digital systems testing and testable design. Computer Science Press, New York, 1990.

[Alur 1994] Rajeev Alur et David L. Dill. A Theory of Timed Automata. Theoretical Computer Science, vol. 126, pages 183–235, 1994.

[Alur 1996] Rajeev Alur, Thomas A. Henzinger et Pei-Hsin Ho. Automatic Symbolic Verification of Embedded Systems. IEEE Transactions on Software Engineering, vol. 22, pages 181–201, 1996.

[Alur 1997] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer et S. K. Rajamani. Partial-Order Reduction in Symbolic State Space Exploration. pages 340–351. Springer, 1997.

[Ammann 1998] Paul E. Ammann, Paul E. Black et William Majurski. Using Model Checking to Generate Tests from Specifications. In ICFEM '98 : Proceedings of the Second IEEE International Conference on Formal Engineering Methods, page 46, Washington, DC, USA, 1998. IEEE Computer Society.

[Ammann 2008] Paul Ammann et Jeff Offutt. Introduction to software testing. Cambridge Univ Pr, 1 édition, January 2008.

[Andersson 2002] Gunnar Andersson, Per Bjesse, Byron Cook et Ziyad Hanna. A proof engine approach to solving combinational design automation problems. In DAC '02 : Proceedings of the 39th annual Design Automation Conference, pages 725–730, New York, NY, USA, 2002. ACM.

[Audemard 2002] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz et Roberto Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In CADE-18 : Proceedings of the 18th International Conference on Automated Deduction, pages 195–210, London, UK, 2002. Springer-Verlag.

[Audureau 1990] E. Audureau, P. Enjalbert et L. Farinas Del Cerro. Logique temporelle, sémantique et validation de programmes parallèles. Masson, Paris, France, 1990.

[Baudin 2002] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen et N. Williams. Caveat : a tool for software validation. In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pages 537+, 2002.

[Beizer 1990] B. Beizer. Software testing techniques (2nd ed.). Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[Ben-Ari 1981] Mordechai Ben-Ari, Zohar Manna et Amir Pnueli. The temporal logic of branching time. In POPL '81 : Proceedings of the 8th ACM

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 164–176, New York, NY, USA, 1981. ACM.

[Bensalem 2000] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman et Ashish Tiwari. An Overview of SAL, 2000.

[Berezin 1998] Sergey Berezin, Sérgio Vale Aguiar Campos et Edmund M. Clarke. Compositional Reasoning in Model Checking. In COMPOS'97 : Revised Lectures from the International Symposium on Compositionality : The Significant Difference, pages 81–102, London, UK, 1998. Springer-Verlag.

[Bertin 2001] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil et S. Yovine. Taxys = Esterel + Kronos - A tool for verifying real-time properties of embedded systems. Conference on Decision and Control, CDC'01, 2001.

[Biere 1999] Armin Biere, Alessandro Cimatti, Edmund M. Clarke et Yunshan Zhu. Symbolic Model Checking without BDDs. In TACAS, pages 193–207, 1999.

[Bochot 2009] Thomas Bochot, Pierre Virelizier, Hélène Waeselynck et Virginie Wiels. Model checking flight control systems : The Airbus experience. In ICSE Companion, pages 18–27, 2009.

[Bozga 1998] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis et Sergio Yovine. Kronos : a model-checking tool for real-time systems. In Moshe Y. Hu Alan J. ; Vardi, editeur, Computer Aided Verification 10th International Conference Proceedings Computer Aided Verification 10th International Conference, CAV'98, volume 1427 of Lecture Notes in Computer Science, pages 546–549, Vancouver, BC Canada, 06 1998. Springer.

[Bérard 1999] B. Bérard, M. Bidoit, F. Laroussinie, A. Petit et P. Schnoebelen. Vérification de logiciels : techniques et outils du model-checking. Vuibert, 1999.

[Bryant 1986] R. E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, vol. C-35, no. 8, pages 677–691, 1986.

[Bryant 1992] R. E. Bryant. Symbolic boolean manipulations with ordered binary decision diagrams. ACM Computing Surveys, vol. 24, no. 3, pages 293–317, 1992.

[Bushnell 2000] M.L Bushnell et V.D Agrawal. Essential of electronic testing for digital, memory and mixed-signal vlsi circuits. Kluwer Academic Publishers, Boston, MA, USA, 2000.

[Chilenski 1994] J. J. Chilenski et S. P. Miller. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, vol. 9, no. 5, pages 193–200, 1994.

[Cimatti 1999] A. Cimatti, E.M. Clarke, F. Giunchiglia et M. Roveri. NuSMV : a new Symbolic Model Verifier. In N. Halbwachs et D. Peled, editeurs, Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99), numéro 1633 de Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[Clarke 1994] Edmund M. Clarke, Orna Grumberg et David E. Long. Model Checking and Abstraction. ACM Transactions on Programming Languages and Systems, vol. 16, no. 5, pages 1512–1542, September 1994.

[Clarke 1998] Edmund M. Clarke, E. Allen Emerson, Somesh Jha et A. Prasad Sistla. Symmetry Reductions in Model Checking. In CAV, pages 147–158, 1998.

[Clarke 2000] E. Clarke et Yuan Lu. Counterexample-guided abstraction refinement. In In Computer Aided Verification, pages 154–169. Springer, 2000.

[Clarke 2001] Edmund Clarke, Armin Biere, Richard Raimi et Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. Form. Methods Syst. Des., vol. 19, no. 1, pages 7–34, 2001.

[Cousot 1977] Patrick Cousot et Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL '77 : Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252. ACM Press, 1977.

[Cousot 1979] Patrick Cousot et Radhia Cousot. Systematic design of program analysis frameworks. In POPL '79 : Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 269–282, New York, NY, USA, 1979. ACM Press.

[Cousot 2005] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux et Xavier Rival. The ASTRÉE Analyzer. In ESOP, numéro 3444 de Lecture Notes in Computer Science, pages 21–30, 2005.

[Cousot 2007] P. Cousot. The Verification Grand Challenge and Abstract Interpretation. In B. Meyer et J. Woodcock, editeurs, Verified Software : Theories, Tools, Experiments, volume 4171 of Lecture Notes in Computer Science, pages 227–240. Springer, Berlin, Germany, Décembre 2007.

[Davis 1962] Martin Davis, George Logemann et Donald Loveland. A machine program for theorem-proving. Commun. ACM, vol. 5, no. 7, pages 394–397, 1962.

[de Moura 2004] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea et Ashish Tiwari. SAL 2. In Rajeev Alur et Doron Peled, editeurs, Computer-Aided Verification, CAV 2004, volume 3114 of Lecture Notes in Computer Science, pages 496–500, Boston, MA, Juillet 2004. Springer-Verlag.

[Dijkstra 1976] Edsger W. Dijkstra. A discipline of programming. Prentice Hall, Inc., October 1976.

[du Bousquet 1999] Lydie du Bousquet, Farid Ouabdesselam, Jean-Luc Richier et Nicolas Zuanon. Lutess : A Specification-Driven Testing Environment for Synchronous Software. In ICSE, pages 267–276, 1999.

[Duprat 2006] S. Duprat, J. Souyris et D. Favre-Félix. Formal verification workbench for Airbus avionics software. Embedded Real-Time Software (ERTS), 2006.

[Dutertre 2006] Bruno Dutertre et Leonardo De Moura. The yices smt solver. Rapport technique, 2006.

[Editor 2009] web page : Scade Suite Editor, 2009. `http://www.esterel-technologies.com/products/scade-suite/editor`.

[Ferdinand 2001] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing et Reinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In EMSOFT '01 : Proceedings of the First International Workshop on Embedded Software, pages 469–485, London, UK, 2001. Springer-Verlag.

[Gargantini 1999] Angelo Gargantini et Constance L. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In ESEC / SIGSOFT FSE, pages 146–162, 1999.

[Gaucher 2003] F. Gaucher, E. Jahier, B. Jeannet et F. Maraninchi. Automatic State Reaching for Debugging Reactive Programs. In Int. Workshop on Automated and Algorithmic Debugging, AADEBUG'03, September 2003.

[Goubault 2001] Eric Goubault. Static Analyses of the Precision of Floating-Point Operations. In SAS, pages 234–259, 2001.

[Groce 2003] Alex Groce et Willem Visser. What Went Wrong : Explaining Counterexamples. In SPIN, pages 121–135, 2003.

[Grumberg 1991] Orna Grumberg et David E. Long. Model Checking and Modular Verification. ACM Transactions on Programming Languages and Systems, vol. 16, 1991.

[Halbwachs 1991] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud. The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE, vol. 79, no. 9, pages 1305–1320, September 1991.

[Halbwachs 1992] N. Halbwachs, F. Lagnier et C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, September 1992.

[Halbwachs 1993a] N. Halbwachs, J.-C. Fernandez et A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language Lustre. In Sixth International Symp. on Lucid and Intensional Programming, ISLIP'93, Quebec, April 1993.

[Halbwachs 1993b] Nicolas Halbwachs, Fabienne Lagnier et Pascal Raymond. Synchronous observers and the verification of reactive systems. In Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente. Springer Verlag, 1993.

[Halbwachs 2002] Nicolas Halbwachs et Pascal Raymond. A Tutorial Of Lustre, janvier 2002.

[Hamon 2004] Grégoire Hamon, Leonardo Mendonça de Moura et John M. Rushby. Generating Efficient Test Sets with a Model Checker. In SEFM, pages 261–270, 2004.

[Hayhurst 2001] Kelly Hayhurst, Dan S. Veerhusen, John J. Chilenski et Leanna K. Rierson. A Practical Tutorial on Modified Condition/Decision Coverage, 2001.

[Heimdahl 2003] Mats Per Erik Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj et Jimin Gao. Auto-generating Test Sequences Using Model Checkers : A Case Study. In FATES, pages 42–59, 2003.

[Heimdahl 2004] Mats Per Erik Heimdahl, George Devaraj et Robert Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria ? In HASE, pages 178–186, 2004.

[Henzinger 1997] Thomas A. Henzinger, Pei-Hsin Ho et Howard Wong-toi. HyTech : A Model Checker for Hybrid Systems. Software Tools for Technology Transfer, vol. 1, pages 460–463, 1997.

[Hoare 1969] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, vol. 12, pages 576–580, 1969.

[Holzmann 1997] Gerard J. Holzmann. The Model Checker SPIN. Software Engineering, vol. 23, no. 5, pages 279–295, 1997.

[Jeannet 1999] Bertrand Jeannet, Nicolas Halbwachs et Pascal Raymond. Dynamic Partitioning in Analyses of Numerical Properties. In In SAS, pages 39–50. Springer-Verlag, 1999.

[Jeannet 2003] B. Jeannet. Dynamic Partitioning in Linear Relation Analysis : Application to the Verification of Reactive Systems. Form. Methods Syst. Des., vol. 23, no. 1, pages 5–37, 2003.

[Joshi 2005] Anjali Joshi et Mats Per Erik Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In SAFECOMP, pages 122–135, 2005.

[Lakehal 2005] A. Lakehal et I. Parissis. Lustructu : A Tool for the Automatic Coverage Assessment of Lustre Programs. In 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005), Chicago, USA, November 2005.

[Lakehal 2007] A. Lakehal et I. Parissis. Automated Measure of Structural Coverage for Lustre/Scade Programs : a Case Study. In Workshop on Automated

Software Testing (AST 2007) - satellite event of ICSE 2007, Minnesota, USA, May 2007.

[Lakehal 2009] Abdesselam Lakehal et Ioannis Parissis. Structural coverage criteria for LUSTRE/SCADE programs. Softw. Test., Verif. Reliab., vol. 19, no. 2, pages 133–154, 2009.

[Larsen 1997] Kim G. Larsen, Paul Pettersson et Wang Yi. UPPAAL in a Nutshell, 1997.

[Laurent 2001] Laurent, P. Michel et V. Wiels. Using formal verification techniques to reduce simulation and test effort. In Formal Methods for increasing software productivity, FME2001. Springer, 2001.

[Liu 2005] Xiaojun Liu. Semantic foundation of the tagged signal model. PhD thesis, Berkeley, CA, USA, 2005. Adviser-Lee, Edward A.

[Ljung 1999] M. Ljung. Formal modelling and automatic verification of Lustre programs using NP-Tools. PhD thesis, Prover Technology AB and dept. of Teleinformatics, KTH, Stockholm, 1999.

[Maraninchi 2000] Florence Maraninchi et Fabien Gaucher. Step-wise + Algorithmic debugging for Reactive Programs : LuDiC, a debugger for Lustre. In in : AADEBUG ?2000 ? Fourth International Workshop on Automated Debugging, 2000.

[Marre 2000] B. Marre et A. Arnould. Génération automatique de séquences de test à partir de descriptions LUSTRE : GATeL. Janvier 2000. Journées Francophones des Langages Applicatifs (JFLA00).

[Miller 2006] S. Miller, A. Tribble, M. Whalen et M. Heimdahl. Proving the shalls - early validation of requirements through formal methods. In Journal of Software Tools for Technology Transfer, pages 303–319, August 2006.

[Miller 2009] Steven P. Miller. Bridging the Gap Between Model-Based Development and Model Checking. In TACAS '09 : Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 443–453, Berlin, Heidelberg, 2009. Springer-Verlag.

[Mouy 2008] Patricia Mouy, Bruno Marre, Nicky Williams et Pascale Le Gall. Generation of All-Paths Unit Test with Function Calls. In ICST, pages 32–41, 2008.

[Owre 1997] Sam Owre et Natarajan Shankar. The Formal Semantics of PVS, 1997.

[Papailiopoulou 2008] Virginia Papailiopoulou, Laya Madani, Lydie du Bousquet et Ioannis Parissis. Extending Structural Test Coverage Criteria for Lustre Programs with Multi-clock Operators. In FMICS, pages 23–36, 2008.

[Pnueli 1979] Amir Pnueli. The Temporal Semantics of Concurrent Programs. In Proceedings of the International Sympoisum on Semantics of Concurrent Computation, pages 1–20, London, UK, 1979. Springer-Verlag.

[Randimbivololona 1999] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau et Dominique Schoen. Applying Formal Proof Techniques to Avionics Software : A Pragmatic Approach. In FM '99 : Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II, pages 1798–1815, London, UK, 1999. Springer-Verlag.

[Ratel 1992] C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes Lustre : le système LESAR. PhD thesis, INPG, July 1992.

[Raymond 1998] P. Raymond, X. Nicollin, N. Halbwachs et D. Weber. Automatic Testing of Reactive Systems. In RTSS '98 : Proceedings of the IEEE Real-Time Systems Symposium, page 200, Washington, DC, USA, 1998. IEEE Computer Society.

[Ruys 2004] Theo C. Ruys et Gerard J. Holzmann. Advanced SPIN Tutorial. In In SPIN, LNCS 2989, pages 304–305. Springer, 2004.

[SAE 1996] SAE. Arp4754. certification considerations for highly-integrated or complex systems, November 1996.

[Saiedian 1996] H. Saiedian. An Invitation to Formal Methods. IEEE Computer, vol. 29, no. 4, pages 16–17, 1996.

[Sen 2005] Koushik Sen, Darko Marinov et Gul Agha. CUTE : a concolic unit testing engine for C. In ESEC/FSE-13 : Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 263–272, New York, NY, USA, 2005. ACM.

[Sheeran 1998] Mary Sheeran et Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In Formal Methods in System Design, pages 82–99. Springer-Verlag, 1998.

[Sheeran 2000] Mary Sheeran, Satnam Singh et Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In FMCAD, pages 108–125, 2000.

[Simulator 2009] web page : Scade Suite Simulator, 2009. `http://www.esterel-technologies.com/products/scade-suite/simulation`.

[SMV 2009] web page : SMV, 2009. `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[Tip 1995] Frank Tip. A Survey of Program Slicing Techniques. Journal of Programming Languages, vol. 3, pages 121–189, 1995.

[van den Berg 2007] Lionel van den Berg, Paul A. Strooper et Wendy Johnston. An Automated Approach for the Interpretation of Counter-Examples. Electr. Notes Theor. Comput. Sci., vol. 174, no. 4, pages 19–35, 2007.

[Verifier 2009] web page : Scade Design Verifier, 2009. `http://www.esterel-technologies.com/products/scade-suite/design-verifier`.

[Wang 2006] Laung-Terng Wang, Cheng-Wen Wu et Xiaoqing Wen. Vlsi test principles and architectures : Design for testability (systems on silicon). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[webpage Caveat 2009] webpage Caveat, 2009. `http://www-list.cea.fr/labos/fr/LSL/caveat/index.html`.

[webpage frama c 2009] webpage frama c, 2009. `http://frama-c.cea.fr`.

[Whalen 2007] Michael W. Whalen, Darren D. Cofer, Steven P. Miller, Bruce H. Krogh et Walter Storm. Integration of Formal Analysis into a Model-Based Software Development Process. In FMICS, pages 68–84, 2007.

[Williams 2005] Nicky Williams, Bruno Marre, Patricia Mouy et Muriel Roger. PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In EDCC, pages 281–292, 2005.

Vérification par Model Checking des commandes de vol :
applicabilité industrielle et analyse de contre-exemples

Le système de commande de vol (CDV) est un des systèmes les plus critiques à bord d'un avion. Les fonctions logicielles de ce système sont donc soumises à un effort de vérification important. Chez Airbus, le développement des fonctions critiques suit une approche basée sur des modèles formels, à partir desquels la majeure partie du code embarqué est générée. Certaines vérifications peuvent ainsi s'effectuer dès le niveau de la modélisation formelle, et sont aujourd'hui réalisées par test des modèles dans un environnement de simulation. L'objet de cette thèse est d'étudier comment une technique formelle, le model-checking, s'insère dans ces vérifications amonts.
La contribution comporte trois parties. La première partie tire le bilan des études passées d'Airbus sur l'application du Model Checking au système de CDV. Nous analysons notamment les caractéristiques des fonctions de CDV, et leur impact sur l'applicabilité de la technologie. Le deuxième partie complète la précédente par une nouvelle étude, expérimentant le Model Checking sur la fonction Ground Spoiler de l'A380. Les expérimentations ont permis de consolider notre analyse du positionnement du Model Checking dans le processus Airbus. Un des problèmes pratiques identifiés concerne l'exploitation des contre-exemples retournés par le model-checker, en phase de mise au point d'un modèle. La troisième partie propose une solution à ce problème, basée sur l'analyse structurelle des parties d'un modèle activées par le contre-exemple. Il s'agit, d'une part d'extraire l'information pertinente pour expliquer la violation de la propriété cible et, d'autre part de guider le model-checker vers l'exploration de comportements différents, activant d'autres parties du modèle. Un algorithme d'analyse structurelle est défini, et implémenté dans un prototype afin d'en démontrer le concept.

Mots clés : Model Checking, Méthodes formelles, Analyse de contre-exemples, Analyse structurel, Commandes de vol, V&V

Model Checking the Flinght Control System :
Industrial Applicability and Counterexample Analysis

The Flight Control System (FCS) is one of the most critical system in an aircraft. Software parts of this system are thus subject to massive vérification efforts. At Airbus, critical software function are implemented through a model based development process. Some verification can then be performed earlier on the formal model using simulation techniques. The aim of this PhD is to study how a formal method: model checking can be integrated into such a development process.
PhD work is composed of three parts. Firstly we present insights from past studies about how to use model checking to verify FCS properties. We analyze FCS functions characteristics and their impact over model checking applicability. Secondly, we experiment a use of model checking on a case study: the A380 ground spoiler function. Results reinforce our proposition on the use of model checking in the FCS development process. One the technical problem deals with counterexample analysis during model verification. Finally, the third part propose a solution to that problem. It is based on a structural analysis of parts of the model activated by a counterexample. On the one hand, we extract relevant information to explain the violation of the property and, on the other hand, we guide the model checker to explore different behavior, activating different structural parts of the model. An algorithm is defined and implemented to demonstrate the functionality.

Key words : Model Checking, Formel Methods, Counterexample Analysis, Structural Analysis, Flight Control System, V&V