Synthèse automatique d'architectures tolérantes aux fautes

Delmas Kévin

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 (ISAE)*

Présentée et soutenue le *19/12/2017* **par :**

Kévin DELMAS

Synthèse automatique d'architectures tolérantes aux fautes

JURY

Sylvain CONCHON	Professeur d'Université	Président du Jury
Yves LEDRU	Professeur d'Université	Examinateur
Karama KANOUN	Professeur d'Université	Examinatrice
Pierre BIEBER	Maître de Conférence	Membre du Jury
Mireille BAYART	Professeur d'Université	Rapporteur
Daniel LE BERRE	Professeur d'Université	Rapporteur

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture

Unité de Recherche :
Office national d'études et de recherches aérospatiales (ONERA)

Directeur(s) de Thèse :
Claire PAGETTI et *Rémi DELMAS*

Rapporteurs :
Mireille BAYART et *Daniel LE BERRE*

Table des matières

Liste des tableaux

Table des figures

Abréviations

BDD	Binary Decision Diagram Diagramme de Décision Binaire
MSFOL	Many Sorted First Order Logic Logique multi-sortée du premier ordre
Sat	Satisfiability Satisfiabilité pour la logique propositionnelle
SMT	Satisifability Modulo Theory Satisifabilité Modulo Théorie
STMDD	System Trace MDD MDD des traces système
AdD	Fault Tree Arbres de Défaillances
ARP	Aerospatial Recommended Practices Pratiques Aéronautiques Recommandées
DSE	Design Space Exploration Exploration de l'Espace des Architectures
FHA	Functional Hazard Analysis Analyse des risques fonctionnels
FMEA	Failure Mode Effect and Criticity Analysis Analyse des Modes de Défaillances, de leur Effets et Criticité
MBSA	Model-Based Safety Assessment Évaluation de la sûreté basée sur les modèles
MDD	Mutli-valued Decision Diagramme Diagramme de Décision Multi-valués
PSSA	Preliminary System Safety Assessment Analyse Préliminaire de la Sureté des Systèmes
UFBV	Uninterpreted Function and BitVectors Fonctions non interprétées et vecteurs de bits

Remerciements

Tout d'abord, je tiens à remercier mes encadrants de thèse, Claire Pagetti et Rémi Delmas, pour leur encadrement bienveillant durant ces trois années de thèse. Ils m'ont offert, chacun à leur manière, un soutien tant sur le plan technique qu'humain qui m'a permis de m'épanouir dans ce travail d'apprentissage de la recherche par la recherche.

Je voudrais aussi remercier mes rapporteurs, Mireille Bayart et Daniel Le Berre, pour leurs analyses attentives et leurs remarques constructives.

Je souhaiterais aussi remercier les membres du jury : Pierre Bieber, Sylvain Conchon, Yves Ledru et Karama Kanoun pour leurs questions et leur intérêt pour ce travail. Vos remarques ont été d'une grande aide pour approfondir ma connaissance et ma compréhension du sujet. Je tenais à remercier spécialement Pierre Bieber pour les nombreuses discussions sur la certification des systèmes critiques et plus simplement pour m'avoir donné goût à la sûreté de fonctionnement (et à son enseignement).

Un grand merci aux doctorants de l'(ex) DTIM avec qui nous avons passé de bons moments autour de discussions enflammées lors des repas ou des afterworks (plus ou moins bien organisés). Cette ambiance a été d'une grande aide dans les moments difficiles.

Je tiens à remercier mes amis : Théo, Thomas, Sophie, Lolita, Pierre, Élie, Nono, Benouch, Momo, Susanna et David pour leur soutien qui m'a permis de prendre du recul, leur intérêt (plus ou moins prononcé) pour ce travail et tout simplement pour les bons moments passés ensemble.

Finalement, je remercie ma famille et en particulier mon frère et ma mère pour leur soutien et leurs conseils sans lesquels tout ce travail n'aurait pas été possible.

Première partie

Contexte

Chapitre 1

Introduction

1.1 Systèmes critiques et sûreté de fonctionnement

Les systèmes critiques sont des systèmes dont le dysfonctionnement peut avoir des conséquences dramatiques comme la perte de vies humaines, des dégâts matériels lourds ou encore de graves atteintes à l'environnement. Par conséquent, la sûreté de fonctionnement occupe une place prépondérante dans leur processus de conception.

Afin d'intégrer, au plus tôt, la sûreté de fonctionnement au processus de développement, les concepteurs suivent des recommandations comme l'ARP4754 [88] dans le domaine aéronautique. Selon ces recommandations les concepteurs doivent identifier, dès la conception, les risques d'utilisation et assurer que ces risques sont *acceptables*. Cette notion d'*acceptabilité* d'un risque repose sur la gravité de ses conséquences et sur sa vraisemblance. Cette vraisemblance est évaluée quantitativement et qualitativement par des *indicateurs de sûreté* comme la *fiabilité* et *l'ordre*. Ainsi, en fonction de la gravité du risque, l'acceptabilité est assurée par la satisfaction d'un ensemble d'*exigences de sûreté* qui imposent des bornes sur la valeur des indicateurs.

Après avoir identifié les risques et formulé les exigences, les concepteurs définissent une architecture du système. Puis, les concepteurs mènent une Analyse Préliminaire de la Sûreté du Système (PSSA en anglais) où les comportements dysfonctionnels du système sont analysés afin de calculer les indicateurs et de vérifier la satisfaction des exigences de sûreté. Si la PSSA révèle que l'architecture du système ne respecte pas les exigences, il est nécessaire de modifier l'architecture. Ce processus itératif d'analyse et de renforcement, appelé *durcissement d'architecture*, est appliqué jusqu'à obtenir une architecture respectant les exigences.

La mise au point d'une telle architecture est un problème dit *d'exploration* dans un espace fini d'architectures candidates, dérivées d'une architecture de référence par ajout de mécanismes de sûreté.

Il existe, dans la littérature, des méthodes de résolution automatiques du problème d'exploration, lesquelles seront présentées ultérieurement dans ce manuscrit. Néanmoins, ces solutions présentent des limites, l'objectif de ce manuscrit est donc de définir une approche de résolution automatique et efficace du problème d'exploration basée sur l'utilisation de solveurs du problème de Satisfiabilité Modulo Théorie (SMT). La démarche empruntée pour construire cette approche est présentée dans la section suivante.

1.2 Démarche

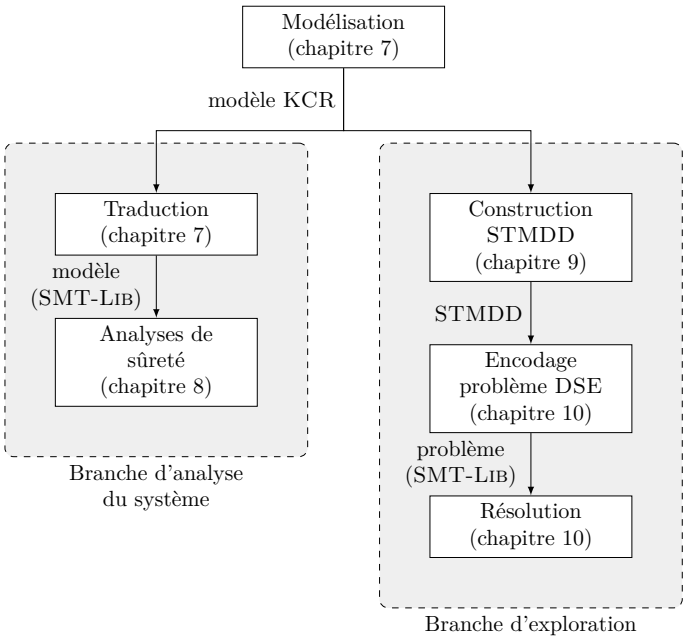


Figure 1.1 – Structure des contributions

Les principales contributions de ce manuscrit, dont l'organisation est illustrée par la figure 1.1, sont les suivantes :

Modélisation Les langages existants de formulation du problème d'exploration ne répondant pas à l'ensemble de nos contraintes, nous introduisons un langage, appelé KCR, permettant de décrire, de manière hiérarchique et modulaire, les comportements dysfonctionnels d'un système ainsi que les exigences de sûreté à respecter.

Branche d'analyse d'un système Nous avons développé un ensemble de méthodes de calcul des indicateurs de sûreté reposant sur la résolution de problèmes SMT. Pour cela, nous définissons la sémantique de KCR par traduction vers le langage SMT-Lib afin de fournir une description du système dans la logique du premier ordre multi-sortée (MSFOL). Grâce à cette description, nous montrons comment calculer des indicateurs de sûreté (tels que les coupes minimales et l'ordre d'un système) à partir de la résolution de problèmes SMT. L'utilisation de solveurs SMT permet alors de réduire le temps de calcul par rapport aux méthodes pré-existantes.

Branche d'exploration Par ailleurs, nous avons développé une méthode complète, correcte et efficace d'exploration reposant sur la résolution de problèmes SMT. Pour cela, la résolution doit reposer sur des méthodes de calcul des indicateurs de sûreté capables d'évaluer exactement et efficacement les candidats lors de l'exploration. Or nous montrons que les méthodes de calcul introduites précédemment ne sont pas les plus adaptées pour analyser les candidats lors de l'exploration. Nous introduisons alors un nouvelle méthode reposant sur la combinaison d'analyses locales des composants du système, dont le résultat est une structure de données

appelée STMDD. Cette structure, automatiquement calculée avant le début de l'exploration, est alors utilisée pour calculer exactement les indicateurs de n'importe quel candidat de l'espace des architectures en un temps linéaire dans la taille du STMDD. Puis, nous développons d'une part une théorie appelée Safety fournissant les prédicats nécessaires pour encoder les contraintes de sûreté et le STMDD en MSFOL ; d'autre part le solveur de la théorie Safety décidant, à l'aide d'une analyse du STMDD, si un candidat donné respecte un ensemble d'exigences de sûreté. Nous introduisons alors une traduction automatique du problème d'exploration, décrit en KCR, vers un problème SMT et déléguons sa résolution à un solveur SMT combiné au solveur de la théorie Safety. Celui-ci résout le problème en élaguant l'espace de recherche par apprentissage de clauses de conflit. La méthode de résolution du problème d'exploration garantit de trouver, en un temps fini, une architecture respectant les exigences si et seulement si celle-ci existe.

1.3 Plan du manuscrit

Afin de positionner les méthodes et outils développés dans ce manuscrit par rapport aux méthodes existantes, nous introduisons, dans le chapitre 2, les méthodes de modélisation et d'analyse de la sûreté de fonctionnement des systèmes classiquement utilisées pour analyser un candidat lors de l'exploration. Puis nous examinons, dans le chapitre 3, les avantages et limitations des différentes approches existantes de résolution du problème d'exploration. Par ailleurs, nous décrivons, dans le chapitre 4, les langages de modélisation des systèmes et de spécification du problème d'exploration. Finalement, nous rappelons, dans le chapitre 5, les concepts élémentaires de la Satisfiabilité Modulo Théorie nécessaires à la lecture de ce manuscrit.

Nous récapitulons ensuite, dans le chapitre 6, les principales limitations des approches existantes de modélisation, d'analyse et de résolution du problème DSE et introduisons une vue d'ensemble du processus de résolution du problème d'exploration développé dans ce manuscrit.

Nous présentons, dans le chapitre 7, le langage de modélisation KCR en décrivant sa syntaxe et sa sémantique. Nous développons, dans le chapitre 8, un ensemble de méthodes de calcul des indicateurs de sûreté basées sur la résolution de problèmes SMT. Puis nous en détaillons les limitations pour évaluer les architectures candidates lors de la résolution d'un problème d'exploration.

Nous introduisons, dans le chapitre 9, une nouvelle analyse permettant de construire le STMDD d'un système, à partir duquel les indicateurs de sûreté de tout candidat appartenant à l'espace des architectures d'un problème d'exploration peuvent être calculés.

Nous développons, dans le chapitre 10, un encodage du problème d'exploration comme un problème SMT et une résolution de ce problème à l'aide d'un solveur SMT combiné au solveur de la théorie Safety.

Nous présentons, dans le chapitre 11, l'outil nommé KCR Analyser implantant l'ensemble des analyses et méthodes de résolution du problème d'exploration présentées dans ce manuscrit.

Finalement, dans le chapitre 12, nous utilisons KCR Analyser pour résoudre différents problèmes d'exploration sur un ensemble de cas d'étude. Les expérimentations illustrent l'efficacité de l'approche de résolution basée SMT qui permet de résoudre plus rapidement les problèmes d'exploration par rapport aux approches génétiques.

Deuxième partie

État de l'art

Chapitre 2

Concepts élémentaires de la sûreté de fonctionnement

Ce chapitre introduit la terminologie (section 2.1) des concepts élémentaires de la sûreté de fonctionnement. Ceux-ci servent à définir le *modèle dysfonctionnel* d'un système, notion qui sera expliquée dans la section 2.2. Les formalismes de modélisation (sections 2.2 et 2.3) des comportements dysfonctionnels des systèmes tirés du livre d'Alain Villemeur [95] sont ensuite détaillés, ainsi que les analyses classiques de sûreté de fonctionnement menées sur ces modèles. Ces concepts serviront de références, tout au long du manuscrit afin de vérifier la conformité d'un système à un ensemble d'exigences de sûreté.

2.1 Terminologie de la sûreté de fonctionnement

Tout d'abord commençons par introduire le cas d'étude ROSACE qui servira d'exemple pour illustrer notre propos.

2.1.1 Système fil rouge : le cas d'étude Rosace

ROSACE [74] est un contrôleur chargé d'asservir l'attitude longitudinale d'un avion de ligne moyen courrier en phase de croisière. Le contrôle longitudinal est une fonction élémentaire de navigation. Sa perte peut entraîner une perte de contrôle de l'appareil et donc un crash. Par conséquent, ce système assure une fonctionnalité *critique*.

La figure 2.1 décrit l'architecture de ROSACE et son interaction avec l'avion, symbolisé par le bloc *Aircraft*. Plus précisément, ce contrôleur régule, d'une part la vitesse verticale V_z vis-à-vis de la consigne V_{zc} à l'aide du contrôleur C_{Vz}. D'autre part il régule la vitesse par rapport au vent V_a vis-à-vis de la consigne V_{ac} à l'aide du contrôleur C_{Va}. Pour cela, le contrôleur envoie des commandes aux élévateurs δ_{thc} et aux moteurs δ_{ec}. Les signaux en provenance des capteurs de l'avion sont filtrés avant d'être envoyés aux contrôleurs C_{Va} et C_{Vz}, afin de limiter la sensibilité du contrôleur aux bruits.

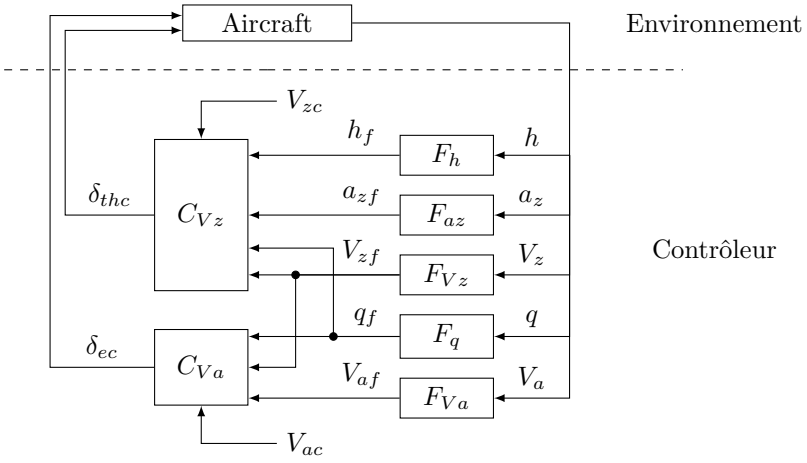


FIGURE 2.1 – Contrôleur longitudinal de vol ROSACE

2.1.2 Concept de défaillance

L'objet d'étude de la sûreté de fonctionnement est l'ensemble des *défaillances* d'un *système*, lequel assure un certain nombre de *fonctions*. Dans ce document, nous définissons un *système* comme un ensemble de *composants* interconnectés. Ces composants assurent des fonctions plus simples dont l'interaction assure les fonctions du système. Autrement dit, les fonctions du système caractérisent ce que le système *doit faire* alors que les fonctions des composants définissent *comment* le système assure ses fonctions. Le concepteur doit alors argumenter que l'interaction permet bien d'assurer les fonctions du système, néanmoins cette question est en dehors du contexte de cette étude.

Exemple 2.1 (Fonctions)**.** *Reprenons le système* ROSACE*, sa fonction est d'assurer le contrôle longitudinal de l'avion en fournissant des commandes aux ailerons et aux moteurs. Pour cela le contrôleur* ROSACE *possèdent :*

- *des filtres dont la fonction est de fournir un signal filtré des paramètres de vol ;*
- *les lois C_{Va} et C_{Vz} qui assurent la régulation de la vitesse par rapport au vent et de la vitesse verticale en fournissant une commande aux ailerons et aux moteurs à partir des paramètres de vol et d'une commande.*

Ainsi dans ROSACE*, les filtres fournissent aux lois les paramètres de vol, utilisés pour réguler la vitesse de l'avion en fournissant une commande aux moteur et aux ailerons, ce qui permet d'assurer la fonction du système c'est-à-dire le contrôle longitudinal de l'avion.*

Pour assurer ses fonctions, un système peut avoir besoin d'un ensemble *d'éléments* issus de son environnement, appelés *entrées* du système. De même, ses *sorties* sont les éléments transmis à son environnement. Par exemple les entrées du contrôleur ROSACE sont les paramètres de vol issus des capteurs lesquels sont indispensables pour réguler le contrôle longitudinal. Par ailleurs, il fournit à son environnement (c'est-à-dire l'avion) des commandes aux ailerons et aux moteurs qui sont ses sorties.

Les relations entre système, composant et fonction sont représentées par le diagramme d'entité-relation de la figure 2.2. Dans ce diagramme, les rectangles correspondent aux entités, une flèche en

diamant de A vers B annotée par un nom s signifie que B est composée d'un ensemble de A appelé s, un flèche en triangle représente une spécialisation, finalement une flèche simple est une relation entre deux entités.

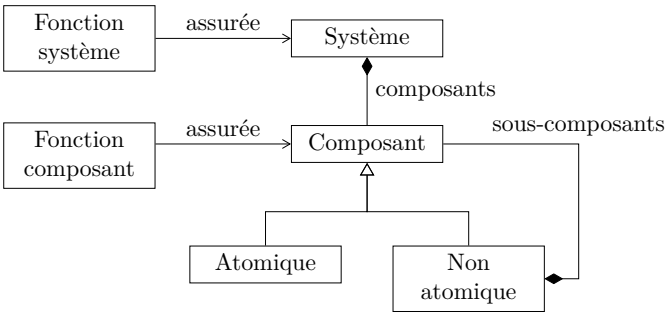
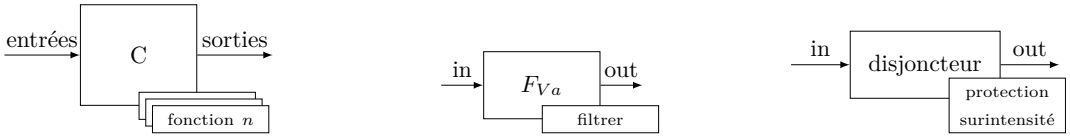


FIGURE 2.2 – Relations entre système et fonctions

La notion de composant varie sensiblement en fonction de la nature du système. Comme les approches présentées dans ce manuscrit ne visent pas un type particulier de systèmes, nous utilisons une définition générique des composants.

Définition 2.1 (Composant)**.** *Un composant C, représenté par la figure 2.3a, est une entité assurant un ensemble de* fonctions*. Les* entrées *de C sont les* éléments *nécessaires à C pour assurer l'ensemble de ses fonctions. De même, ses* sorties *sont les éléments qu'il fournit aux autres composants du système.*



(a) Un composant (b) Le composant F_{Va} de ROSACE (c) Un disjoncteur

FIGURE 2.3 – Notion de composant

Exemple 2.2 (Composant de traitement du signal)**.** *Prenons le composant F_{Va} de* ROSACE *représenté par la figure 2.3b. Puisque sa fonction est de filtrer le signal V_a reçu des capteurs de l'avion, ce signal lui est transmis via l'entrée in. Le signal filtré résultant out doit ensuite être transmis aux lois.*

Exemple 2.3 (Composant électrique)**.** *Illustrons la notion de composant sur un disjoncteur représenté par la figure 2.3c, sa fonction est d'ouvrir le circuit en cas de surintensité, il doit donc avoir accès à l'alimentation du circuit via une entrée in. L'alimentation protégée est alors transmise via la sortie out au reste du système pour pouvoir alimenter ses composants.*

Un composant dit *non-atomique* est décrit comme un ensemble de composants interconnectés appelés *sous-composants*. A l'inverse, un composant dit *atomique*, n'est pas décomposable. Par

exemple, ROSACE est un système qui ne contient que des composants atomiques F_{Va}, F_{Vz}, F_q, F_{az}, F_h, C_{Va}, C_{Vz}.

La figure 2.4 présente les relations entre les différentes notions de *pannes*. Plus précisément, lorsqu'un composant ou un système est dans l'incapacité de fournir une fonction attendue on parle de *défaillance*.

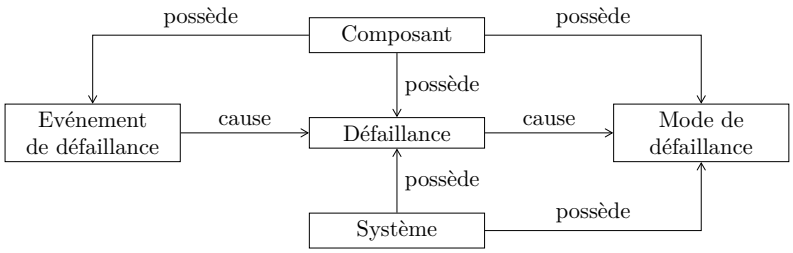


FIGURE 2.4 – Relation entre les concepts de panne

Les causes de la défaillance d'un composant (problème d'implantation, matériel défectueux, environnement agressif, etc) sont modélisées par l'occurrence *d'événements de défaillance*. Par exemple on peut définir l'événement $F_{Va}.l$ dont l'occurrence déclenche l'arrêt de traitement du signal par le filtre F_{Va}. Dans le cas d'un système, les fonctions sont assurées par les fonctions de ses composants. Par conséquent, la défaillance d'un système est causée par un ensemble de défaillances de ces composants. Par soucis de simplicité, nous parlons d'événements de défaillance d'un système pour les événements de défaillance des composants du système.

La défaillance d'un composant ou d'un système induit mécaniquement un changement de comportement. Autrement dit, on observe un *écart* entre les sorties attendues et celles produites, par le composant ou le système, sous l'influence d'une défaillance. Ces *manifestations* de la défaillance sont décrites par les *modes de défaillance* du composant. Par souci de simplicité nous parlons, dans la suite du manuscrit, d'un événement de défaillance e produisant un mode de défaillance *fm*, au lieu d'un événement e causant une défaillance d dont la conséquence est *fm*.

Définition 2.2 (Mode de défaillance)**.** *Soit un composant ou un système A, alors un mode de défaillance de A est une description des conséquences d'une défaillance sur les sorties de A.*

Pour des raisons de lisibilité et de généricité, les modes de défaillance, que nous utilisons dans ce manuscrit, sont des descriptions haut niveau des conséquences d'une défaillance. Par exemple, un ensemble de modes classiques que nous utiliserons par la suite est :

- ***perte*** la sortie x n'est plus disponible (noté x^L) ;
- ***erroné en valeur*** la sortie x est incorrecte (noté x^E).

Exemple 2.4 (Mode de défaillance)**.** *Si la conséquence de la défaillance* traitement incorrect du signal *pour le filtre F_{Va} est l'absence de signal sur la sortie out du filtre alors le mode de défaillance observé est une perte.*

Les défaillances d'un système peuvent amener celui-ci dans un état dangereux qui doit être absolument évité. Ces états sont les *événements redoutés* obtenus par une analyse des risques fonctionnels (FHA) du système et constituent la base des exigences de sûreté.

Définition 2.3 (Événement redouté)**.** *Soit S un système assurant un ensemble de fonctions, alors un événement redouté est un état du système, considéré comme dangereux, où un ensemble de modes de défaillance de ses composants engendrent la perte d'une ou plusieurs fonctions du système.*

Exemple 2.5 (Dysfonctionnements de ROSACE)**.** *Illustrons ces notions sur le cas d'étude* ROSACE*. Nous considérons que les modes de défaillance des composants de* ROSACE *sont l'erroné et la perte. Ainsi à chaque composant C sont associés les événements de défaillance C.e pour* la sortie de C est erronée *et C.l pour* la sortie de C est perdue*. Du fait de la conception des lois de contrôle, celles-ci peuvent fonctionner si un de leurs signaux d'entrée n'est pas correct, hormis pour le signal V_a qui est indispensable. Le fonction de régulation est considérée comme perdue si l'une des deux sorties du contrôleur n'est pas correcte.*

L'événement redouté de ROSACE *est la production d'une commande aux moteurs (δ_{ec}) ou aux ailerons (δ_{thc}) incorrecte provoquée par la défaillance* le contrôleur ne régule plus l'attitude longitudinale de l'avion*. Donc les modes de défaillance décrivant l'événement redouté sont δ^E_{ec} ou δ^L_{ec} ou δ^E_{thc} ou encore δ^L_{thc}, nous noterons ces combinaisons $\{\{\delta^E_{ec}\}, \{\delta^L_{ec}\}, \{\delta^E_{thc}\}, \{\delta^L_{thc}\}\}$*

2.1.3 Modèle dysfonctionnel d'un système

L'un des principaux rôles de l'ingénieur de sûreté est de créer un *modèle dysfonctionnel* (nommé simplement modèle par la suite) du système décrivant l'impact des défaillances des composants sur l'occurrence des événements redoutés. Le choix du formalisme de modélisation est généralement dicté par la nature des comportements dysfonctionnels du système.

L'analyse de ces comportements est fondée sur les occurrences d'événements de défaillance menant à un événement redouté (appelés par la suite *scénarios de défaillance*). Or certains scénarios de défaillance du système peuvent ne pas être modélisables dans le formalisme choisi. Dans ce cas le modèle du système ne pourrait représenter qu'un sous-ensemble des scénarios de défaillance ce qui revient à ignorer ces scénarios durant les analyses. Ceci étant intolérable, les concepteurs doivent s'assurer que le formalisme est capable de représenter l'ensemble des scénarios de défaillance du système. Néanmoins, il est souvent difficile de les représenter exactement, les ingénieurs de sûreté de fonctionnement font alors une modélisation *pessimiste* (aussi appelée *conservative*), c'est-à-dire un modèle dont l'ensemble des scénarios de défaillance contient ceux du système mais également d'autres scénario factices. Pour connaître l'expressivité du formalisme nécessaire pour représenter l'ensemble des scénarios, le concepteur doit déterminer si le système est *dynamique* ou *statique*.

Systèmes dynamiques

Les systèmes dynamiques sont des systèmes où l'ordre d'occurrence des événements de défaillance impacte l'état dysfonctionnel du système. Autrement dit, changer l'ordre d'occurrence d'un ensemble d'événements de défaillance peut déclencher des dysfonctionnements différents du système. La modélisation de cette catégorie de systèmes repose généralement sur les systèmes de transitions.

Exemple 2.6 (Système dynamique)**.** *Considérons un système d'auto-vérification extrait de [6] décrit par la figure 2.5. Considérons qu'une sortie x des composants de ce système est soit correcte (notée x^{ok}), soit incorrecte (notée x^{ko}). Dans ce système, deux composants identiques C_1 et C_2 produisent chacun une sortie* o *potentiellement incorrecte (événements C_i.f symbolisés par un éclair sur C_1 et C_2) envoyée aux entrées in_1 et in_2 d'un sélecteur S. Celui-ci est activé, via l'entrée* s*, par un composant de test T qui vérifie si la sortie de C_1 est correcte. Si oui S renvoie la sortie*

de C_1 sinon celle de C_2. Considérons que le test peut rester bloqué sur la dernière valeur envoyée (événement $T.f$ symbolisé par un éclair sur T). Quels sont alors les scénarios conduisant à envoyer une donnée incorrecte en sortie du système ? Ici le scénario $T.f$ puis $C_1.f$ en fait partie. Or le scénario $C_1.f$ puis $T.f$ n'en fait pas partie puisque le test détecte que C_1 est défaillant avant de tomber en panne. Ce système est donc dynamique.

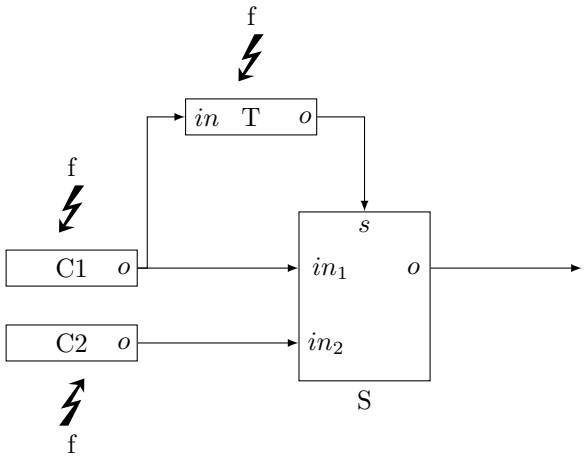


FIGURE 2.5 – Patron d'auto-vérification

Notons que les systèmes dynamiques peuvent également modéliser des *réparations* ou des *reconfigurations* c'est-à-dire des actions rétablissant le fonctionnement du système. En effet, une réparation est vue comme l'occurrence d'un *événement de réparation* créant une transition entre un état de dysfonctionnement et de fonctionnement du système.

Systèmes statiques

Les systèmes statiques sont, à l'inverse, des systèmes où l'ordre d'occurrence des événements de défaillance n'impacte pas l'état dysfonctionnel du système. Autrement dit, quel que soit l'ordre d'occurrence d'un ensemble d'événements de défaillance, le dysfonctionnement déclenché est toujours le même. La modélisation de ces systèmes est alors basée sur la logique booléenne. Plus précisément, [95] montre que les combinaisons d'événements déclenchant un événement redouté sont représentées par une *fonction de structure* donnée dans la définition 2.4. De ce fait, les modèles statiques sont moins expressifs que les modèles dynamiques mais permettent tout de même de représenter un grand nombre de systèmes.

Exemple 2.7 (Système statique)**.** *On peut créer un modèle statique pessimiste de l'exemple précédent en posant* si le test est bloqué et que l'une des deux entrées est incorrecte alors la sortie du sélecteur est incorrecte*. On obtient alors une modélisation pessimiste du système où $C_1.f$ puis $T.f$ et $T.f$ puis $C_1.f$ sont bien deux scénarios menant à l'événement redouté.*

Définition 2.4 (Fonction de structure)**.** *Soit un système S et un événement redouté fc, la fonction de structure φ de S pour fc est une fonction booléenne définie sur les événements de défaillance du système et indiquant si fc est vraie ou non.*

Par souci de lisibilité, nous omettons de préciser, dans le suite de ce document, les entrées de la fonction de structure en l'exprimant directement à partir de propositions correspondant aux événements de défaillance.

Exemple 2.8 (Fonction de structure)**.** *La fonction de structure du système de l'exemple 2.7 pour l'événement redouté* une donnée incorrecte en sortie *est alors :*

$$\varphi = (C_1.f \wedge (T.f \vee C_2.f)) \vee (T.f \wedge C_2.f)$$

Une autre propriété importante est la *monotonie* (aussi appelée *cohérence*) d'un système. Cette propriété assure que seules les occurrences d'événements impactent l'état dysfonctionnel du système. Autrement dit, on ne peut pas *réparer* le système en ajoutant de nouvelles défaillances. Dans le cas d'un système statique, Alain Villemeur [95] montre que la fonction de structure peut s'écrire uniquement à partir des opérateurs $\wedge$ et $\vee$ si et seulement si le système est monotone.

Définition 2.5 (Monotonie)**.** *Soit un système S possédant un ensemble d'événements de défaillance E et fc un événement redouté, alors S est monotone si et seulement si pour tout sous-ensemble $E' \subset E$ dont l'occurrence déclenche fc on a $\forall E'' \subset E \setminus E', E' \cup E''$ déclenche fc.*

Exemple 2.9 (Monotonie)**.** *Reprenons la fonction de structure introduite par l'exemple précédent, soit $E \subset \{C_1.f, C_2.f, T._f\}$ tel que E satisfait φ. Comme φ ne contient pas de négation, on peut prouver que $\forall E' \subset \{C_1.f, C_2.f, T._f\} \setminus E$, $E \cup E'$ satisfait φ, c'est-à-dire que l'occurrence des événements de $E \cup E'$ déclenche l'événement redouté. Ainsi, le système d'auto-vérification est monotone.*

Notons que la notion de réparation peut aussi être modélisée par des systèmes statiques. Comme ceux-ci ne peuvent pas modéliser une précédence, les réparations des systèmes statiques sont modélisées comme des événements probabilistes. Autrement dit, si une combinaison d'événements E déclenche un événement redouté fc, et qu'un événement de réparation r survient alors la combinaison $E \cup \{r\}$ ne déclenche pas fc (système non monotone).

Hypothèse 2.1 (Réparation)**.** *Comme les approches d'analyses développées dans ce document ne prennent pas en compte la réparation, nous considérons par la suite que les systèmes sont non-réparables.*

2.1.4 Indicateurs de sûreté

Afin d'évaluer la tolérance d'un système vis-à-vis des défaillances, les concepteurs de systèmes critiques se basent sur un ensemble d'indicateurs *qualitatifs* et *quantitatifs*.

Indicateur qualitatifs

Les indicateurs qualitatifs décrivent les scénarios d'événements de défaillance menant à un événement redouté. L'un des indicateurs les plus communément utilisés est l'ensemble des coupes minimales. Intuitivement, cet ensemble contient les scénarios de taille minimale qui déclenchent l'événement redouté. Ils capturent les liens entre les occurrences d'événements de défaillance et l'occurrence d'un événement redouté.

Définition 2.6 (Coupes minimales (MCS))**.** *L'ensemble des coupes minimales MCS d'un système S pour un événement redouté fc est l'ensemble des ensembles d'événements appelés* coupes minimales *$MCS = \{mc\}$ tel que $\forall mc \in MCS$,*

1. *si tous les événements $e \in mc$ se déclenchent alors fc se produit ;*
2. *$\forall e \in mc$, si tous les événements $e' \in mc \setminus \{e\}$ se déclenchent alors fc ne se produit pas.*

Exemple 2.10 (Coupes minimales)**.** *Dans le cas de* ROSACE*, les coupes minimales sont les suivantes :*

$$MCS = \left\{ \begin{array}{llllll} \{C_{Va}.e\}, & \{C_{Va}.l\}, & \{F_{Va}.e\}, & \{F_{Va}.l\}, & \{C_{Vz}.e\}, & \{C_{Vz}.l\}, \\ \{F_h.e, F_{az}.e\}, & \{F_h.e, F_{Vz}.e\}, & \{F_h.e, F_q.e\}, & \{F_{az}.e, F_{Vz}.e\}, & \{F_{az}.e, F_q.e\}, & \{F_{Vz}.e, F_q.e\}, \\ \{F_h.l, F_{az}.l\}, & \{F_h.l, F_{Vz}.l\}, & \{F_h.l, F_q.l\}, & \{F_{az}.l, F_{Vz}.l\}, & \{F_{az}.l, F_q.l\}, & \{F_{Vz}.l, F_q.l\} \end{array} \right\}$$

Parmi l'ensemble des coupes appartenant à MCS, les plus petites coupes (en terme de cardinalité) intéressent particulièrement les ingénieurs de sûreté de fonctionnement. En effet, ces coupes capturent les scénarios de défaillance les plus probables. Par exemple dans ROSACE, les plus petites coupes sont $\{C_{Va}.e\}, \{C_{Va}.l\}, \{F_{Va}.e\}, \{F_{Va}.l\}, \{C_{Vz}.e\}, \{C_{Vz}.l\}$.

Définition 2.7 (Ordre du système (mincard))**.** *L'ordre d'un système S pour un événement redouté fc est défini comme la cardinalité de la plus petite coupe minimale de MCS.*

$$mincard = \min_{mc \in MCS}(|mc|)$$

Exemple 2.11 (Ordre du système)**.** *On déduit des coupes calculées précédemment que l'ordre de* ROSACE *est de 1 car il existe plus d'une coupe de cardinalité 1.*

Indicateurs quantitatifs

Les indicateurs quantitatifs sont des mesures probabilistes d'occurrence d'un événement redouté. Les indicateurs utilisés par la suite sont issus des analyses Fiabilité, Maintenabilité, Disponibilité et Sécurité (FMDS ou RAMS en anglais). La fiabilité est définie comme la capacité d'un système à ne pas déclencher un événement redouté pendant un temps donné.

Définition 2.8 (Fiabilité (R))**.** *La fiabilité (reliability en anglais) notée R (respectivement la défiabilité notée $\overline{R}$) d'un système S est la probabilité qu'un événement redouté fc ne se produise pas (respectivement se produise) sur un intervalle de temps $[0, t]$ sachant que S est fonctionnel à $t = 0$. Soit t_{fc} la variable aléatoire modélisant l'instant d'occurrence de fc, la fiabilité est alors*

$$\forall t \in \mathbb{R}^+,\ R(t) \triangleq p(t_{fc} > t) = 1 - p(t_{fc} \leq t) = 1 - \overline{R}(t)$$

Par souci de lisibilité, nous introduisons les notations suivantes : pour tout événement e dont l'instant d'occurrence est modélisé par la variable aléatoire t_e,

$$R_e(t) \triangleq p(t_e > t) = p(e) = 1 - p(\overline{e}) = 1 - p(t_e \leq t) = 1 - \overline{R_e}(t)$$

Lors de l'évaluation des indicateurs, la question de *l'indépendance* des événements de défaillance est souvent abordée. Plus précisément, on parle d'indépendance entre deux événements A et B si l'occurrence de A n'impacte pas la probabilité d'occurrence de B c'est-à-dire si $p(A \cap B) = p(A)p(B)$. Bien entendu, si l'on considère, pendant la conception du système, que des événements sont indépendants alors cette indépendance doit être vérifiée lors de la validation du système afin d'assurer un calcul exact des indicateurs.

Hypothèse 2.2 (Indépendance)**.** *Nous considérons dans la suite de ce document que les événements de défaillance des composants sont indépendants.*

La fiabilité d'un système peut être calculée exactement à partir du Diagramme de Décision Binaire [21] (Binary Decision Diagram en anglais ou BDD).

Définition 2.9 (Décomposition de Shannon)**.** *Soit F une formule booléenne et v une variable booléenne, alors la décomposition de Shannon de F par rapport à v est :*

$$F = (v \wedge F|_{v=\mathbf{T}}) \vee (\neg v \wedge F|_{v=\mathbf{F}})$$

où $F|_{v=\mathbf{T}}$ (respectivement $F|_{v=\mathbf{F}}$) est la formule F où la variable v vaut vrai *(respectivement* faux*).*

Définition 2.10 (BDD)**.** *Un BDD est un graphe orienté et acyclique représentant une formule F. Un* BDD *est constitué d'arcs, de nœuds de décisions et des nœuds terminaux 0 et 1. Chaque nœud de décision est étiqueté par une variable v et possède deux arcs sortant, un menant au fils low et l'autre menant au fils high. Le fils low (respectivement high) est le BDD de la décomposition de Shannon $F|_{v=\mathbf{F}}$ (respectivement $F|_{v=\mathbf{T}}$). Le terminal 1 (respectivement 0) représente alors la formule* **T** *(respectivement* **F***).*

Un BDD est dit ordonné si les variables apparaissent dans le même ordre quel que soit le chemin emprunté. De plus un BDD est dit réduit s'il ne contient pas de sous-graphes structurellement identiques. Dans la suite de ce manuscrit, nous parlerons de BDD pour BDD ordonné réduit. Le BDD d'une formule booléenne est obtenu grâce aux règles de construction définies dans [21]. Le calcul récursif de la fiabilité associée à un BDD, introduit par [86], est :

$$1 - R(t) = p(BDD) = \begin{cases} 0 & \text{si le BDD est le terminal 0} \\ 1 & \text{si le BDD est le terminal 1} \\ p(v)p(F|_{v=\mathbf{T}}) + p(\overline{v})p(F|_{v=\mathbf{F}}) & \text{si } BDD = v \wedge F|_{v=\mathbf{T}} \vee \neg v \wedge F|_{v=\mathbf{F}} \end{cases}$$

Exemple 2.12 (Calcul de la fiabilité par BDD)**.** *Reprenons l'exemple 2.10, considérons pour des raisons de lisibilité que seuls C_{V_a} et F_{Va} peuvent défaillir, alors au vu des coupes données dans l'exemple 2.10, la nouvelle fonction de structure est $\varphi = C_{V_a}.e \vee C_{V_a}.l \vee F_{V_a}.e \vee F_{V_a}.l$. Le BDD de φ est donné par la figure 2.6 où un arc plein (respectivement en pointillés) mène au fils high (respectivement au fils low) du nœud de départ. Considérons que la densité de probabilité de tous les événements de* Rosace *est $p(e) = 1 - e^{-0.001t}$ alors la fiabilité après 10^2 heures de fonctionnement est :*

$$\left.\begin{array}{l} p(A) = p(C_{Va}.e) + p(\overline{C_{Va}.e})p(B) \\ p(B) = p(C_{Va}.l) + p(\overline{C_{Va}.l})p(C) \\ p(C) = p(F_{Va}.e) + p(\overline{F_{Va}.e})p(D) \\ p(D) = p(F_{Va}.l) \end{array}\right\} \quad \begin{array}{ll} 1 - R(10^2) & = p(A) \\ & = p(C_{Va}.e) + p(\overline{C_{Va}.e})\times \\ & [p(C_{Va}.l) + p(\overline{C_{Va}.l})\times \\ & [p(F_{Va}.e) + p(\overline{F_{Va}.e})p(F_{Va}.l)]] \\ & = 1 - (e^{-0.1})^4 \\ R(10^2) & = 0.670 \end{array}$$

La fiabilité est souvent calculée à partir des coupes minimales d'un système. En effet, la probabilité que le système fonctionne sur un intervalle de temps donné revient à ce qu'aucune des coupes minimales ne se produise. Néanmoins le calcul exact est fastidieux (application du principe d'inclusion/exclusion), les ingénieurs de sûreté utilisent plutôt la sous-approximation (donc pessimiste)

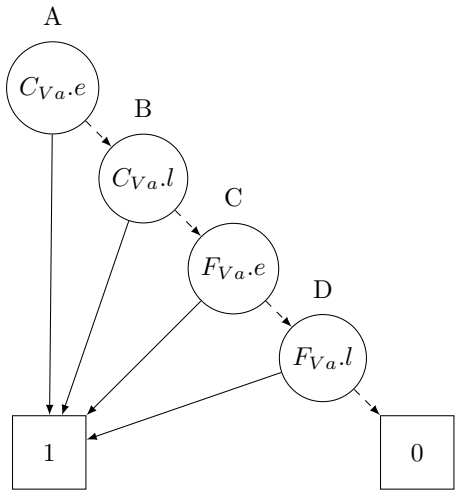


FIGURE 2.6 – BDD de la fonction de structure de l'exemple 2.12

suivante :

$$R(t) = 1 - p\left(\bigcup_{mc \in MCS} \bigcap_{e \in mc} e\right) \geq 1 - \sum_{mc \in MCS} \prod_{e \in mc} p(e)$$

Exemple 2.13 (Calcul de la fiabilité par somme de produits)**.** *Reprenons l'exemple 2.10, l'approximation de la fiabilité par somme de produits est :*

$$\begin{array}{rl} R(t) & \geq 1 - (p(C_{Va}.e) + p(C_{Va}.l) + p(F_{Va}.e) + p(F_{Va}.l)) \\ & \geq 1 - 4(1 - e^{-0.1}) \\ & \geq 0.619 \end{array}$$

Une autre sous-approximation de la fiabilité peut être obtenue par la méthode de calcul d'Esary-Proschan [39]. Soit MCS l'ensemble des coupes minimales d'un système S pour un événement redouté *fc* alors :

$$R(t) \geq \prod_{mc \in MCS} \left(1 - \prod_{e \in mc} p(e)\right)$$

Exemple 2.14 (Calcul de la fiabilité par Esary-Proschan)**.** *En reprenant les coupes calculées précédemment, l'approximation de la fiabilité après* 10^2 *heures de fonctionnement est :*

$$\begin{array}{rcl} R(10^2) & \geq & (1 - p(C_{Va}.e))(1 - p(F_{Va}.e))(1 - p(F_{Va}.e))(1 - p(F_{Va}.l)) \\ & \geq & (e^{-0.1})^4 \\ & \geq & 0.670 \end{array}$$

La fiabilité introduite précédemment évalue la capacité d'un système à être fonctionnel sur un intervalle de temps. Or il est souvent important de connaître la capacité d'un système à être fonctionnel à un instant donné. Cette notion est la *disponibilité* d'un système.

Définition 2.11 (Disponibilité (A))**.** *La disponibilité (availability en anglais) notée A (respectivement l'indisponibilité notée $\overline{A}$) d'un système S est la probabilité que le système soit fonctionnel (respectivement ne le soit pas) à un instant t.*

Comme les systèmes considérés sont non-réparables, [95] montre que la fiabilité et la disponibilité sont égales.

Les indicateurs présentés jusqu'ici donnent une mesure probabiliste de fonctionnement. Néanmoins, il est important d'identifier dans la vie du système les périodes où la probabilité de défaillance est la plus élevée. Cette notion est décrite grâce au *taux de défaillance instantané*. Le taux de défaillance est maximum durant les périodes où l'occurrence d'une défaillance est la plus probable.

Définition 2.12 (Taux de défaillance instantané (Λ))**.** *Le taux de défaillance instantané (*failure rate *en anglais) noté Λ est la probabilité que le système tombe en panne à l'instant $t + \delta t$ sachant qu'il n'était pas tombé en panne sur $[0, t]$. Le taux de défaillance se définit alors comme*

$$\forall t \in \mathbb{R}^+,\ \Lambda(t) = -\frac{\delta R}{\delta t}.R(t)^{-1}$$

Exemple 2.15 (Taux de défaillance instantané)**.** *Reprenons la fiabilité calculée dans l'exemple 2.13. Pour évaluer Λ, nous devons attribuer une distribution de probabilité aux événements de défaillance. Considérons qu'un événement e est modélisé par une loi exponentielle de paramètre λ_e alors :*

$$\Lambda(t) = \frac{\lambda_{C_{Va}}.e^{-\lambda_{C_{Va}}.et} + \lambda_{C_{Va}}.le^{-\lambda_{C_{Va}}.lt} + \lambda_{F_{Va}}.e^{-\lambda_{F_{Va}}.et} + \lambda_{F_{Va}}.le^{-\lambda_{F_{Va}}.lt}}{e^{-\lambda_{C_{Va}}.et} + e^{-\lambda_{C_{Va}}.lt} + e^{-\lambda_{F_{Va}}.et} + e^{-\lambda_{F_{Va}}.lt}}$$

2.1.5 Exigences de sûreté

Les événements redoutés représentent des situations dangereuses pour le système et son environnement. Pour chacun d'eux, les concepteurs doivent alors évaluer leur *sévérité* c'est-à-dire la gravité des conséquences de ces événements. Généralement la sévérité est qualifiée sur une échelle de niveau décrite par la norme visée. Par exemple dans l'ARP4754 [88] la sévérité est classée en cinq niveaux :

Catastrophique (CAT) Multiples décès des occupants, ou blessure mortelle d'un pilote entraînant généralement la perte de l'avion.

Dangereuse (HAZ) Détresse physique ou surcharge de travail empêchant les pilotes d'accomplir leurs tâches de manière précise ou complète, blessure grave ou mortelle d'un passager ou d'un membre de l'équipage de cabine.

Majeure (MAJ) Augmentation significative de la charge de travail de l'équipage, inconfort pour les pilotes, détresse physique ou blessures pour les passagers ou l'équipage de cabine.

Mineure (MIN) Légère réduction des marges de sécurité, légère augmentation de la charge de travail de l'équipage ou inconfort physique pour les passagers ou l'équipage de cabine.

Sans effet (NSE) Aucune incidence sur la capacité opérationnelle de l'avion ou la charge de travail de l'équipage.

Dans le cas de ROSACE, l'événement redouté *le contrôleur produit une commande moteur ou aileron erronée* entraîne un risque de crash. Cet événement est catastrophique d'après l'ARP4754.

Une fois les événements redoutés classifiés, on doit assurer que ces événements sont *acceptables*. Cette notion d'acceptabilité est classiquement définie par une matrice donnant pour chaque sévérité

un seuil sur un ou plusieurs indicateurs de sûreté de l'événement en question. Dans la suite de ce document, nous nous basons sur les seuils fournis par la norme l'ARP4754 sur le taux de défaillance et l'ordre d'un système.

Exemple 2.16 (Acceptabilité)**.** *Dans la matrice de la figure 2.1, on observe qu'un événement de sévérité catastrophique n'est acceptable que si le plus grand taux de défaillance sur l'intervalle de temps* $[0, T]$ *est* $\leq 10^{-9}$ *et que l'ordre est au moins de deux.*

	Exigences :($max_{t\in[0,T]}(\Lambda(t))$, ordre)				
Sévérité	$(> 10^{-3},\ 1)$	$(]10^{-5}, 10^{-3}],\ 1)$	$(]10^{-7}, 10^{-5}],\ 1)$	$(]10^{-9}, 10^{-7}],\ 1)$	$(\leq 10^{-9}, 2)$
Pas d'effet	✓	✓	✓	✓	✓
Mineure	✗	✓	✓	✓	✓
Majeure	✗	✗	✓	✓	✓
Dangereuse	✗	✗	✗	✓	✓
Catastrophique	✗	✗	✗	✗	✓

TABLE 2.1 – Matrice d'acceptabilité

Pour chaque événement redouté, on précise les bornes sur les indicateurs de sûreté devant être respectées. Ces *exigences de sûreté* serviront, en plus des exigences fonctionnelles, de base pour valider les choix de conception.

Exemple 2.17 (Exigence)**.** *Au vu de la matrice de l'exemple 2.16, on peut formuler l'exigence suivante :* la probabilité que le contrôle longitudinal ne soit plus assuré doit être inférieure à 10^{-9} par heure de vol.

2.2 Panorama des formalismes classiques de modélisation

Les risques identifiés par les concepteurs servent de base à la génération d'exigences de sûreté. Or afin de respecter les standards, ces exigences doivent être vérifiées sur la conception choisie du système. Ainsi le concepteur doit sélectionner un formalisme pour modéliser les comportements dysfonctionnels du système. Les formalismes classiques reposent généralement sur une description des combinaisons ou séquences d'événements de défaillance pour modéliser les dysfonctionnements d'un système. Nous allons donc présenter quelques uns des formalismes classiques utilisés dans ce manuscrit.

2.2.1 Formalismes statiques

Les formalismes statiques sont utilisés pour modéliser des systèmes où l'ordre d'occurrence des événements n'a pas d'impact sur l'état dysfonctionnel du système. Les comportements dysfonctionnels peuvent être alors modélisés comme des combinaisons d'occurrence d'événements de défaillance.

L'arbre de défaillance (AdD) est le formalisme le plus connu pour modéliser ces combinaisons. Celui-ci est la représentation graphique de la fonction de structure du système liant les événements de défaillance des composants (appelés événements primaires) aux événements redoutés. Cette représentation est un circuit booléen de portes ET, OU, NON et M-parmi-N (vrai si et seulement si m entrées parmi les n sont vraies). Chaque sortie de porte est annotée et correspond à un événement dit *intermédiaire*. Notons qu'un arbre de défaillance est en réalité un Graphe Acyclique Orienté (DAG) car les événements primaires et intermédiaires peuvent être utilisés par plusieurs portes.

Exemple 2.18 (Arbre de défaillance)**.** *La figure 2.7 représente l'arbre de défaillance du système de l'exemple 2.7. Pour des raisons de lisibilité, les feuilles utilisées par plusieurs portes sont copiées et représentées par des cercles en pointillés. L'événement redouté au sommet de l'arbre est* sortie incorrecte*. Celui-ci est déclenché par deux scénarios distincts,*

1. *soit C_1 est défaillant et C_2 ou T sont défaillants. En effet si C_1 est défaillant alors T doit être fonctionnel pour déclencher le sélecteur S et la valeur fournie par C_2 doit être correcte ;*
2. *soit C_2 et T sont défaillants, en effet si T est défaillant alors par approximation pessimiste, sa sortie est défaillante si l'une des entrées est défaillante ce qui est le cas lorsque C_2 est défaillant.*

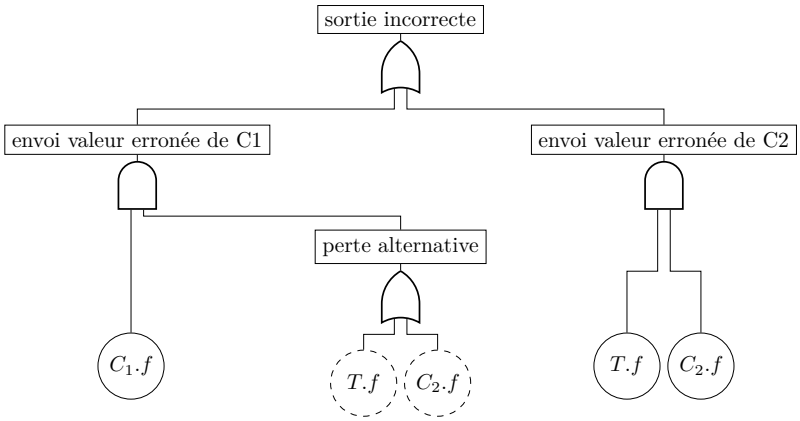


FIGURE 2.7 – Arbre de défaillance du système d'auto-vérification

Les méthodes d'analyses d'un arbre de défaillance, comme celles de [95], sont basées sur le parcours récursif de l'arbre. Néanmoins depuis les années 90, de nombreuses méthodes comme celles de [84] et implantées dans l'outil GRIF [92] mettent à profit les Diagrammes de Décision Binaire (BDD), les solveurs SAT et SMT ou encore la vérification de modèles, pour calculer les indicateurs de sûreté. Décrivons les algorithmes implantés dans les outils d'analyses GRIF [92], HIPHOPS [2] et XFTA [83].

L'algorithme MICSUP (MInimal Cut Sets Upward) [75] fait partie des analyses classiques d'arbres de défaillance de systèmes monotones. Celui-ci construit l'ensemble des coupes minimales en parcourant l'arbre des feuilles vers la racine. Intuitivement, à chaque feuille est associé un booléen, puis pour chaque porte rencontrée lors du parcours de l'arbre, une règle est appliquée pour déduire

les coupes minimales de la porte en fonction des coupes minimales de ses entrées. En général, le calcul des coupes minimales d'une porte est réalisé en deux étapes :

- génération d'une couverture des coupes minimales ;
- simplification pour en déduire les coupes. Cette simplification se base sur l'application des règles d'idempotence ($A \wedge A \Leftrightarrow A$) et d'absorption ($(A \wedge B) \vee A \Leftrightarrow A$).

Soit $|T|$ le nombre de portes d'un arbre de défaillance T et m la complexité du processus de minimisation des coupes à chaque porte, alors la complexité de l'algorithme MICSUP est $m|T|$.

Exemple 2.19 (Calcul MICSUP)**.** *Les coupes calculées sur chaque porte de l'arbre de défaillance de l'exemple 2.18 par l'algorithme MICSUP sont représentés par la figure 2.8. On retrouve bien au sommet de l'arbre les coupes minimales de la fonction de structure.*

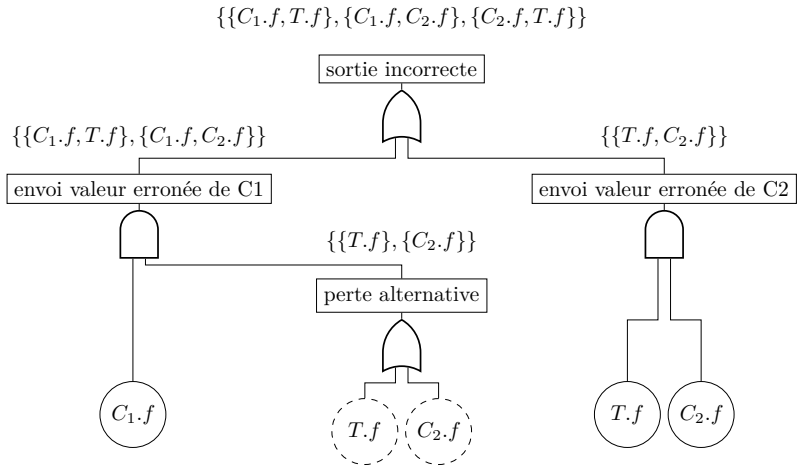


FIGURE 2.8 – Calcul des coupes par l'algorithme MICSUP

La méthode issue de [83] se base sur un schéma *branch and deduce* c'est-à-dire un algorithme récursif explorant la décomposition de Shannon de la fonction de structure représentée par l'arbre de défaillance. Pour une récursion, l'algorithme dispose :

- de la formule courante F ;
- l'ensemble π des variables fixé par les décompositions des récursions précédentes ($v \in \pi$ si v est vraie, $\neg v \in \pi$ si v est fausse, $v \notin \pi$ si la décomposition n'a pas été faite sur v).

Notons $F|_\pi$ la fonction F où les variables v telles que $v \in \pi$ sont interprétées comme vraies et les variables v telles que $\neg v \in \pi$ sont interprétées comme fausses, alors les étapes d'une récursion sont les suivantes :

- si une variable v est strictement nécessaire pour satisfaire la fonction courante en considérant les choix π (c'est-à-dire $F_{\pi \cup \neg v} = \mathbf{F}$) alors ajouter v à π ;
- si pour les choix π, la fonction F est fausse (c'est-à-dire $F_\pi = \mathbf{F}$) alors renvoyer $\emptyset$ (c'est-à-dire pas de coupes) ;
- si pour les choix π, la fonction F est vraie (c'est-à-dire $F_\pi = \mathbf{T}$) alors vérifier qu'il n'existe aucun sous-ensemble de π pour lequel on peut conclure que F est vraie, dans ce cas π est renvoyée car c'est une coupe minimale, autrement $\emptyset$ est retourné (car π n'est pas minimale) ;

– si pour les choix de π on ne peut pas conclure que F est vrai ou fausse alors choisir une variable $v \notin \pi$ et calculer les coupes de F pour $\pi \cup v$ et $\pi \cup \neg v$ et renvoyer l'union des coupes calculées.

Au pire cas, cet algorithme explore la décomposition de Shannon complète de la formule, d'où une complexité pire cas de $\mathcal{O}(2^n)$ où n est le nombre de variables de la formule.

Exemple 2.20 (Calcul branch and deduce)**.** *Illustrons le calcul des coupes minimales par branch and deduce sur l'arbre de défaillance de l'exemple précédent. La fonction de structure représentée par cet arbre est* $\varphi = (C_1.f \wedge (C_2.f \vee T.f)) \vee (C_2.f \wedge T.f)$*. La figure 2.9 représente l'exécution de l'algorithme branch and deduce comme un arbre où une jonction symbolise un appel récursif. L'algorithme est initialisé à* $F = \varphi$ *et* $\pi = \emptyset$*. Comme attendu, l'ensemble* π *initial ne satisfait pas* F *donc l'algorithme choisit de décomposer la formule sur* $C_1.f$ *et d'ajouter* $C_1.f$ *à* π*. De nouveau, une décision est nécessaire. Si l'algorithme choisit d'ajouter* $T.f$ *à* π *alors* π *satisfait* F *puisque pour* $\{C_1.f \mapsto \mathbf{T}, T.f \mapsto \mathbf{T}\}$ F *est vraie, ainsi* $\{\{C_1.f, T.f\}\}$ *est renvoyé. En explorant la décomposition de Shannon on obtient* in fine *les coupes attendues.*

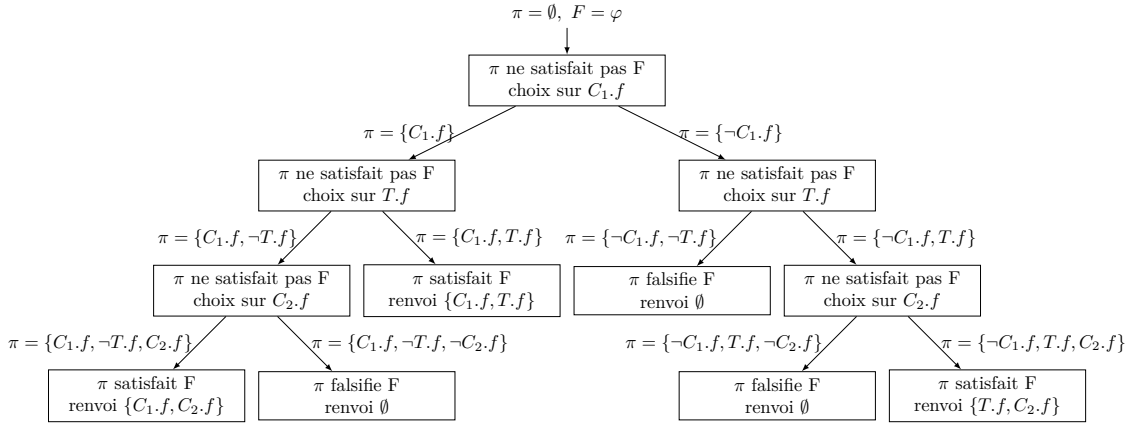


FIGURE 2.9 – Exécution de l'algorithme de calcul des coupes minimales de XFTA

La méthode de calcul des coupes minimales de [84] permet d'extraire les coupes minimales d'une cardinalité inférieure ou égale à une borne k à partir du BDD de la fonction de structure. Pour cela, l'auteur propose d'adapter les règles de construction d'un BDD pour obtenir un BDD dit *tronqué* c'est-à-dire qui représente l'ensemble des valuations contenant au plus k variables vraies satisfaisant une formule F. Pour cela, [84] ajoute une règle de construction du BDD stipulant que si après k décisions positives le BDD résultant n'est pas le terminal 1 alors celui-ci est considéré comme le terminal 0. À partir de ce BDD tronqué, l'auteur de [84] propose l'algorithme parcourant le BDD dont les étapes sont les suivantes :

– si le nœud courant est le terminal 1 alors renvoyer $\{\emptyset\}$;
– si le nœud courant est le terminal 0 alors renvoyer $\emptyset$;
– sinon calculer les coupes minimales du fils low (notées $MCS[F|_{v=\mathbf{F}}]$) et du fils high (notées $MCS[F|_{v=\mathbf{T}}]$) ;
– ajouter la variable v du nœud courant à l'ensemble des coupes de $MCS[F|_{v=\mathbf{F}}]$ (notées $v.MCS[F|_{v=\mathbf{T}}]$) ;

– renvoyer $MCS[F|_{v=\mathbf{F}}] \cup v.MCS[F|_{v=\mathbf{T}}] \setminus MCS[F|_{v=\mathbf{F}}]$

D'après [84], la complexité de la construction du BDD tronqué et de la génération des coupes minimales est polynomiale par rapport aux nombres de variable de la formule. Dans le cas d'un calcul de l'ensemble des coupes minimales d'une fonction F (sans borne k), la seule construction du BDD peut être exponentielle, la complexité pire cas est alors $\mathcal{O}(2^n)$ où n est le nombre de variables de F.

Exemple 2.21 (Calcul des coupes)**.** *La figure 2.10 montre le BDD de la fonction de structure* $\varphi = (C_1.f \wedge (C_2.f \vee T.f)) \vee (C_2.f \wedge T.f)$ *représentée par l'arbre de défaillance de l'exemple 2.18. Les nœuds sont étiquetés par :*

– *les coupes calculées sur le fils* low $MCS[F|_{v=\mathbf{F}}]$
– *les coupes calculées sur le fils* high *auxquelles sont ajoutées la variable* v *du nœud* $v.MCS[F|_{v=\mathbf{T}}]$
– *le résultat du calcul des coupes* MCS

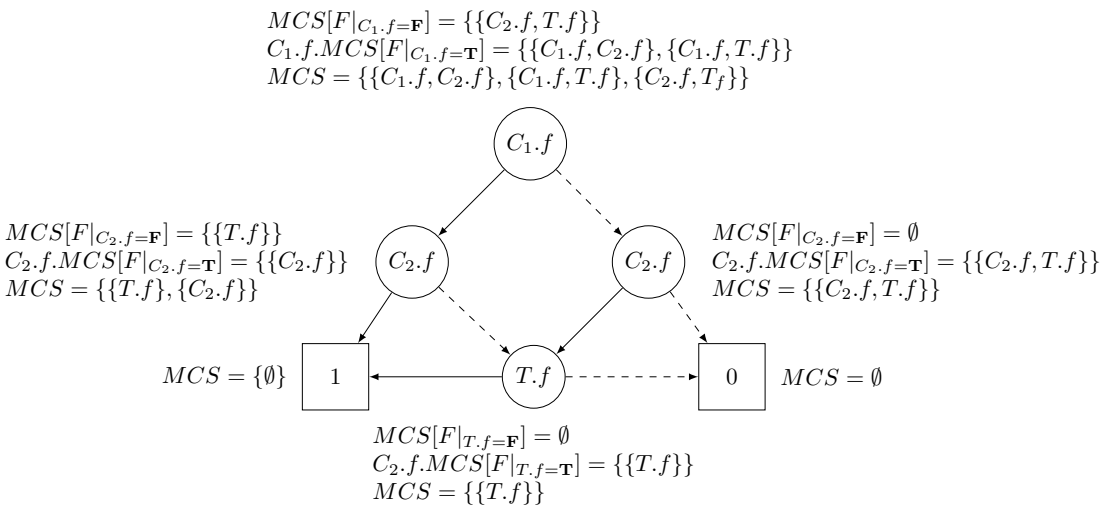


FIGURE 2.10 – BDD de la fonction de structure de l'exemple 4.2

Le diagramme de fiabilité (RBD) représente le système comme une combinaison de circuits séries et parallèles. Un circuit parallèle est fonctionnel si au moins un composant est fonctionnel, tandis qu'un circuit série est fonctionnel si tous ses composants sont fonctionnels. Certains outils comme GRIF [92] étendent ce formalisme avec la connexion M-parmi-N. Ce circuit possède un unique point d'entrée appelé *source* et un ou plusieurs points de sorties appelées *puits*. Ainsi un diagramme de fiabilité représente les ensembles de composants nécessaires au bon fonctionnement du système pour un certain événement redouté. Ces ensembles correspondent aux chemins depuis la source jusqu'au puits schématisant l'événement redouté.

Exemple 2.22 (Diagramme de fiabilité)**.** *La figure 2.11 représente le diagramme de fiabilité du système de l'exemple 2.7. Pour des raisons de lisibilité, les copies des composants sont représentées en pointillés. D'après ce diagramme, le système fonctionne*

1. *si C_1 et C_2 fonctionnent, en effet quel que soit l'état du test, le sélecteur ne pourra transmettre qu'une valeur correcte produite indifféremment par C_1 ou C_2 ;*
2. *ou bien si T et C_1 fonctionnent car dans ce cas quel que soit l'état de C_2, le sélecteur se contente de transmettre la valeur correcte produite par C_1 ;*
3. *ou encore si T et C_2 fonctionnent, en effet si C_1 tombe en panne, alors T détecte la panne et déclenche le sélecteur qui transmet alors la valeur correcte produite par C_2.*

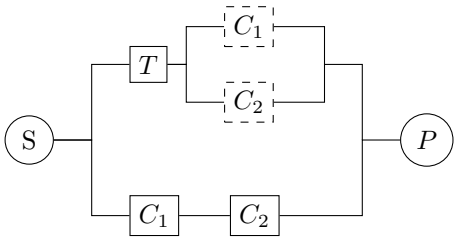


FIGURE 2.11 – Diagramme de fiabilité du système d'auto-vérification

[95] montre que les diagrammes de fiabilité peuvent être traduits en arbres de défaillance, ainsi les méthodes d'analyse d'un diagramme de fiabilité sont des adaptations des méthodes utilisées pour l'analyse des arbres de défaillance.

2.2.2 Formalismes dynamiques

Les formalismes dynamiques sont utilisés pour modéliser des systèmes où l'ordre d'occurrence des événements a un impact sur l'état dysfonctionnel du système. En d'autres termes, les comportements dysfonctionnels peuvent être modélisés comme une séquence d'occurrences d'événements amenant le système dans un état dangereux. Les formalismes dynamiques classiques sont donc généralement fondés sur des systèmes de transitions.

La chaîne de Markov est l'un des formalismes dynamiques les plus connus. Une chaîne de Markov est une machine à état probabiliste dont les états correspondent aux états dysfonctionnels du système (représentés par des cercles) et les transitions encodent les changements d'états occasionnés par l'occurrence d'événements de défaillance ou de réparations (représentés par des arcs). À chaque transition est associée le taux de transition c'est-à-dire la probabilité de franchir la transition depuis l'état courant. Notons que ce formalisme n'est applicable que si l'évolution du système ne dépend que de l'état courant.

Exemple 2.23 (Chaîne de Markov)**.** *La figure 2.12 représente la chaîne de Markov associée au système de l'exemple 2.6 pour l'événement redouté* le système produit une valeur erronée*. Pour pouvoir modéliser le problème à l'aide d'une chaîne de Markov, il faut que les densités de probabilité des événements de défaillance soient* sans mémoire*. Considérons que les densités de probabilité sont exponentielles avec un taux de défaillance constant $\lambda_{C.f}$ où C est un composant et f l'événement de défaillance. Chacun des états est étiqueté soit par l'ensemble des composants défaillants dans le système (noté $\{A, B\}$), soit par une séquence donnant l'ordre dans lequel les composants sont tombés en panne (noté (A, B)). Une transition de la chaîne représente l'occurrence d'un événement*

de défaillance et le taux de transition qui, dans notre cas, est le taux de défaillance. Illustrons cette chaîne sur les états grisés c'est-à-dire ceux où le système déclenche l'événement redouté :

1. *l'état $\{C_1, C_2, T\}$ modélise le cas où tous les composants sont tombés en panne, sans ordre spécifique. Dans ce cas, le sélecteur ne peut pas fournir une sortie correcte ;*
2. *l'état $\{C_1, C_2\}$ représente le cas où C_1 et C_2 sont tombés en panne sans ordre spécifique. En effet ici, le sélecteur ne peut pas fournir une valeur correcte car les deux sources sont défaillantes ;*
3. *l'état (T, C_1) modélise le cas où T puis C_1 tombent en panne. En effet si T tombe avant C_1 alors il ne détecte pas la donnée erronée produite par la suite et ne déclenche pas la redirection vers C_2. Par contre si C_1 tombe avant T alors celui-ci détecte la valeur erronée et déclenche la redirection avant de tomber lui-même en panne, c'est pour cela qu'il est nécessaire de faire la distinction avec l'état (C_1, T) qui lui n'est pas dangereux.*

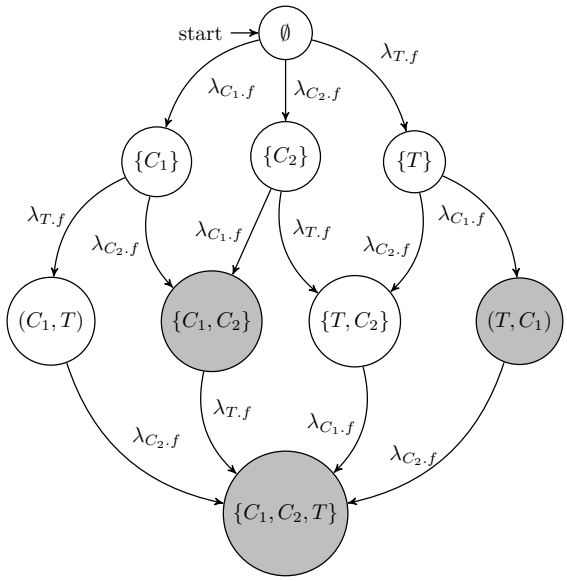


FIGURE 2.12 – Chaîne de Markov du système d'auto-vérification

Il existe une multitude d'autres formalismes dynamiques comme les réseaux de Petri stochastiques généralisé [87], les graphes temporisés de propagation de défaillance [16] ou encore l'extension dynamique des arbres de défaillance. Néanmoins nous ne les présenterons pas car ce document se focalise sur les systèmes statiques, largement utilisés pour l'analyse des systèmes industriels.

Comme rappelé dans [82] les analyses des systèmes dynamiques sont souvent basées sur l'un des principes suivants :

1. une exploration de l'espace d'états consistant à extraire les successions de transitions menant à un état dangereux parmi les états atteignables depuis l'état initial.
2. des simulations stochastiques comme la méthode de Monte-Carlo consistant à tirer aléatoirement un ensemble de séquences d'occurrences d'événements et de simuler le comportement du

système en présence de ces événements. Il est alors possible de faire des statistiques sur les résultats obtenus pour approcher des indicateurs comme la fiabilité.

3. une simulation exhaustive des scénarios de défaillance consistant à simuler toutes les séquences possibles d'événements (d'une taille bornée par un nombre k), puis à extraire les séquences menant à l'événement redouté afin d'estimer les indicateurs de sûreté probabiliste.

2.3 Panorama des formalismes de modélisation fondés sur les modèles

Les formalismes classiques présentés jusqu'ici reposent sur une description très proche des événements de défaillance pour modéliser les dysfonctionnements du système. Or, comme rappelé dans [82], ce type de modélisation bas niveau demande un énorme travail d'abstraction du système. De plus, cette abstraction ne conserve pas la structure architecturale du système initial ce qui rend difficile le lien entre modèle dysfonctionnel et architecture du système.

Un autre point limitant est l'absence de *modularité* de ces modélisations. Autrement dit, il n'est pas possible de décrire des composants réutilisables car la notion de *composant* n'existe pas. Notons que les *composants* des diagrammes de fiabilité ne sont pas des composants au sens où nous l'entendons puisque leur comportement ne peut pas être défini par l'utilisateur (un composant est soit en panne, soit fonctionnel). Cette limitation entraîne non seulement un travail de modélisation considérable qui favorise l'introduction d'erreur dans le modèle, mais aussi rend difficile voire impossible l'ajout de modifications mineures du fonctionnement du système sans avoir à recommencer complètement la modélisation de celui-ci.

Ces limitations ont motivé les ingénieurs de sûreté à mettre en place une approche de modélisation dite fondée sur les modèles (Model Based Safety Assessment en anglais). Plus précisément, l'ingénieur modélise le comportement dysfonctionnel des différents types de composants du système. Puis il les assemble pour former le modèle du système, préservant ainsi la structure du système modélisé. Les études classiques de sûreté sont alors menées directement sur ce modèle.

Ce type d'approches a deux principaux avantages, d'une part le modèle dysfonctionnel est modulaire, hiérarchique et proche de l'architecture du système analysé, ce qui facilite la modélisation et diminue le risque d'erreur. D'autre part, la description locale permet de constituer des librairies de composants mais aussi de modifier simplement le modèle lors d'une mise à jour du fonctionnement du système.

2.3.1 Modélisation

La méthode de modélisation dysfonctionnelle d'un système par les approches fondées sur les modèles est un processus itératif variant assez peu d'une approche à l'autre. Cette méthode de modélisation itérative est principalement composée des étapes suivantes :

1. décomposer le système en un ensemble de composants ;
2. décrire le comportement de ces composants dans le cadre d'occurrences de défaillances ;
3. assembler les composants pour former le modèle du système ;
4. raffiner, si nécessaire, les composants en retournant à l'étape 1 où le système est le composant à raffiner.

Exemple 2.24 (Méthode de modélisation générique)**.** *Appliquons cette méthode de modélisation sur le cas d'étude* ROSACE *:*

1. *le système* ROSACE *est décomposé en cinq filtres* $F_{Va}, F_{Vz}, F_{az}, F_q, F_h$ *et deux contrôleurs* C_{Va} *et* C_{Vz} *;*
2. *le comportement est décrit pour chaque type de composants, par exemple pour un filtre si l'événement de défaillance e se produit alors il fournit des données erronées (mode de défaillance erroné), sinon si l'événement de défaillance l se produit alors il ne fournit plus de données (mode de défaillance perte) sinon il fonctionne normalement ;*
3. *les composants sont connectés de la même manière que le système initial présenté par la figure 2.1 ;*
4. *le raffinage des composants n'est pas considéré comme nécessaire, le modèle obtenu à cette itération est alors le modèle dysfonctionnel de* ROSACE*.*

Cette méthode fournit un modèle dysfonctionnel organisé du système où :
- les composants atomiques sont les composants non-raffinés ;
- les composants non-atomiques sont les composants raffinés en un assemblage de sous-composants.

Comme ce principe de modélisation varie assez peu d'un formalisme à l'autre, nous nous focalisons sur la formalisation des systèmes, composants atomiques et non-atomiques pour chacun des formalismes/langages présentés par la suite.

2.3.2 Formalismes statiques

IF-FMEA (Interface focused FMEA) permet de modéliser des systèmes hiérarchiques et statiques. Dans ce formalisme, un composant est vu comme un ensemble d'équations booléennes. Chacune d'elle donne la règle de génération d'un mode de défaillance sur une sortie donnée à partir des modes de défaillance perçus en entrée et de l'occurrence des événements de défaillance. Les événements de défaillance d'un composant sont donc des variables booléennes. De plus, pour toutes les entrées et sorties x du composant et pour tout mode de défaillance *fm* observable sur x, une variable booléenne est créée pour représenter l'observation ou non du mode de défaillance *fm* sur x. Un système est alors obtenu en interconnectant les composants, c'est-à-dire en ajoutant les règles de propagation des défaillances dans le système. Un lien provenant d'une sortie o et alimentant une entrée *in* impose que les modes de défaillance observés sur *in* soient ceux observés sur o.

Exemple 2.25 (IF-FMEA)**.** *Reprenons le système de l'exemple 2.6. Dans un premier temps il convient de modéliser le comportement des composants du système. Les équations ci-dessous donnent les règles de génération des modes de défaillance pour les composants* C_1*,* C_2*,* S *et* T*. Par exemple le composant* T *produit un ko sur sa sortie o si et seulement si l'événement f se produit ou bien si ko est observé sur in. Puis ceux-ci sont interconnectés, comme montré dans la figure 2.13 où les équations des composants sont représentées comme des circuits booléens, les événements comme des éclairs, les modes de défaillance comme les signaux circulant entre les portes et les liens comme des connexions entre les différents circuits. Par exemple le mode de défaillance ko est observé sur l'entrée* in_1 *du sélecteur* S *si et seulement s'il est observé sur la sortie o de* C_1*. L'ensemble de ces formules donne alors le comportement dysfonctionnel du système.*

$$Composants \begin{cases} C_1 & : & o^{KO} \Leftrightarrow f \\ C_2 & : & o^{KO} \Leftrightarrow f \\ T & : & o^{KO} \Leftrightarrow (f \vee in^{ko}) \\ S & : & o^{KO} \Leftrightarrow ((s^{ko} \wedge in_1^{ko}) \vee (in_1^{ko} \wedge in_2^{ko})) \end{cases} \quad Liens \begin{cases} C_1.o^{KO} \Leftrightarrow S.in_1^{ko} \\ C_2.o^{KO} \Leftrightarrow S.in_2^{ko} \\ T.o^{KO} \Leftrightarrow S.s^{ko} \\ T.in^{KO} \Leftrightarrow C_1.o^{ko} \end{cases}$$

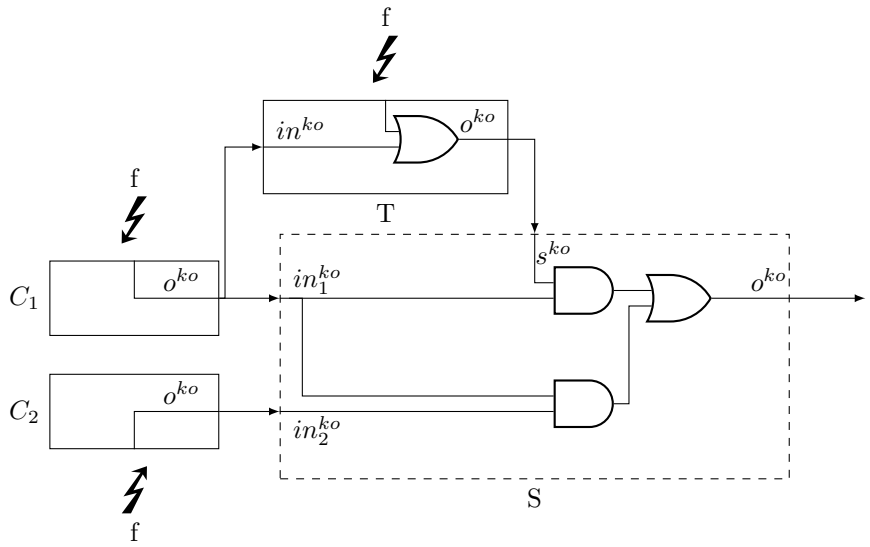


FIGURE 2.13 – Illustration de la IF-FMEA du système d'auto-vérification

2.3.3 Formalismes dynamiques

Les formalismes dynamiques de la MBSA sont souvent fondés sur la théorie des automates. Celle-ci permet de définir le comportent local d'un composant comme un automate, puis de composer et synchroniser ces automates pour former le modèle du système global. Les formalismes des automates finis respectivement des automates de mode [63] adaptés par [85] ont été utilisés par les langages SMV [23] respectivement ALTARICA [7] pour la modélisation des comportements dysfonctionnels.

Un automate fini représente un composant et contient des variables d'entrées et d'états. Les variables d'états représentent les états du composant. Tandis que les variables d'entrées représentent les valeurs potentiellement défaillantes reçues par le composant et ses événements de défaillance. Les transitions possibles sont définies par une relation de transition donnant l'état suivant en fonction de l'état courant et de la valeur des entrées. Finalement les fonctions de sorties donnent les modes de défaillance perçus en sortie en fonction de l'état courant et des entrées.

Pour la conception d'un système à l'aide des automates finis, l'utilisateur définit, dans un premier temps, les automates des différents types de composants atomiques. Puis il définit les composants non-atomiques et les systèmes comme une interconnexion de composants c'est-à-dire comme le produit des automates de ses sous-composants. Plus précisément, ces automates sont *instanciés*, c'est-à-dire que l'on crée des copies des automates des composants initiaux en renommant l'ensemble des variables de l'automate afin d'éviter les conflits d'identifiants (généralement en préfixant les variables par le nom de l'instance). L'interconnexion des sous-composants définit les égalités entre les entrées et sorties des sous-composants. Autrement dit, l'automate d'un composant non-atomique ou d'un système est le produit des automates de ses sous-composants où seules les transitions respectant les égalités sont conservées.

Exemple 2.26 (Automate fini)**.** *Reprenons le système de l'exemple 2.6. Les automates de ses composants sont donnés par la figure 2.14. Prenons celui de T, celui-ci possède une entrée in, un événement de défaillance f et une sortie o donc l'entrée de l'automate est le couple (f, in) et sa sortie est o. Ses transitions sont étiquetées par $\mathbb{B}\times\{ok, ko\}|\{ok, ko\}$ dont la partie gauche correspond à la valeur du couple (f, in) et la partie droite à la valeur de o. Pour des raisons de lisibilité, nous résumons les transitions à l'aide du symbole $*$ représentant n'importe quel symbole pour l'entrée considérée, et du symbole x représentant la valeur lue pour l'entrée concernée. Comme montré par la figure 2.14, l'état initial du composant est l'état ok signifiant que T est fonctionnel. Dans ce cas, la définition de la sortie est simplement de renvoyer l'entrée. Or il est spécifié dans l'exemple 2.6 que si f survient alors la sortie de T reste bloquée sur la dernière valeur de l'entrée. Nous avons alors deux états possibles lorsque f survient : soit l'entrée est ok (état ko_2) ou bien ko (état ko_1).*

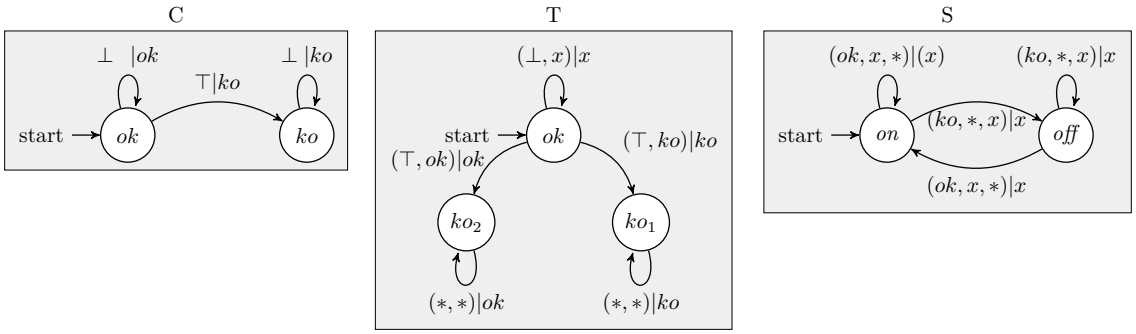


FIGURE 2.14 – Automates des composants du système d'auto-vérification

La figure 2.15 montre comment les automates des composants sont instanciés et connectés pour former l'automate du système. Notons que dans la figure 2.15, l'automate du système est représenté comme plusieurs sous-automates séparés par des pointillés. Cette représentation graphique classique est le produit libre d'automates. Parmi ces sous-automates, on distingue le composant générique C instancié en deux automates C_1 et C_2, tandis que T et S conservent leur nom. Les interconnexions suivantes définissent les transitions légales $T.o = S.s, C_1.o = S.in_1,\ C_2.o = S.in_2,\ C_1.o = T.in$. Pour les représenter, des flèches indiquent comment les entrées et sorties des automates sont contraintes.

Un automate de mode décrit par [85] représente un composant et contient des variables d'entrées et d'états. Contrairement aux automates finis, les événements sont vus comme des objets

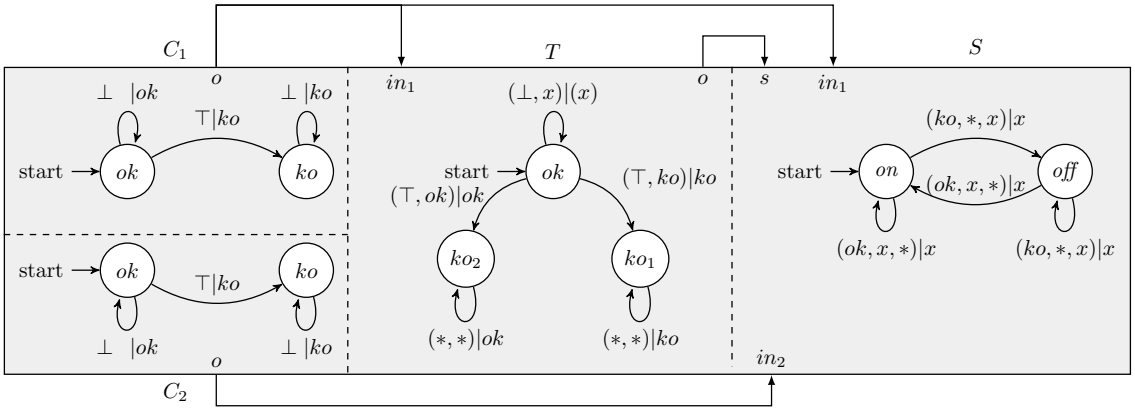


FIGURE 2.15 – Automate du système d'auto-vérification

distincts des entrées. Ces événements sont les déclencheurs des transitions de l'automate de mode. Les transitions contiennent une *garde* c'est-à-dire un prédicat sur les variables d'entrées et d'états du composant et l'événement déclencheur. Une transition de l'automate de mode n'est franchissable que si la garde associée est vraie et que l'événement survient. De plus, l'automate contient des sorties définies par un ensemble de fonctions donnant la valeur des sorties en fonction des entrées et de l'état.

Tout comme les automates finis, l'utilisateur commence par créer les automates de mode des différents types de composants atomiques. De même, un composant non-atomique ou un système est une interconnexion de composants c'est-à-dire le produit libre des automates de mode de ses sous-composants. La différence majeure intervient lors de la description des connexions entre composants. En effet, pour les automates de mode, seuls les événements déclenchent les transitions. Or les événements ne sont pas des entrées de l'automate, donc les contraintes de connexions ne portent pas sur les événements. Par conséquent, toutes les transitions du produit d'automates sont conservées. Par contre les gardes contiennent des variables d'entrée de l'automate. Dans ce cas, connecter une sortie o d'un composant A à une entrée in d'un composant B revient à remplacer dans les gardes et fonctions de sortie de l'automate de B les variables in par $A.o$.

Exemple 2.27 (Automate de mode)**.** *Reprenons le système de l'exemple 2.6. Les automates de mode de ses composants sont donnés par la figure 2.16. Prenons celui de T, celui-ci est très similaire à l'automate fini présenté dans l'exemple 2.26. Par contre les transitions sont étiquetées par un prédicat sur les variables d'entrées et l'événement déclencheur. Par exemple, T passe dans l'état ko_1 depuis l'état ok si l'événement de défaillance f se produit et si l'entrée est à ok.*

La figure 2.17 montre comment les automates des composants sont instanciés et connectés pour former l'automate du système. Notons que dans la figure 2.17, l'automate du système est représenté comme un produit libre d'automates. Parmi ces sous-parties, on distingue le composant générique C instancié en deux automates C_1 et C_2, tandis que T et S conservent leur nom. Puis les remplacements suivants ont été effectués $T.o \rightarrow S.s$, $C_1.o \rightarrow S.in_1$, $C_2.o \rightarrow S.in_2$, $C_1.o \rightarrow T.in$.

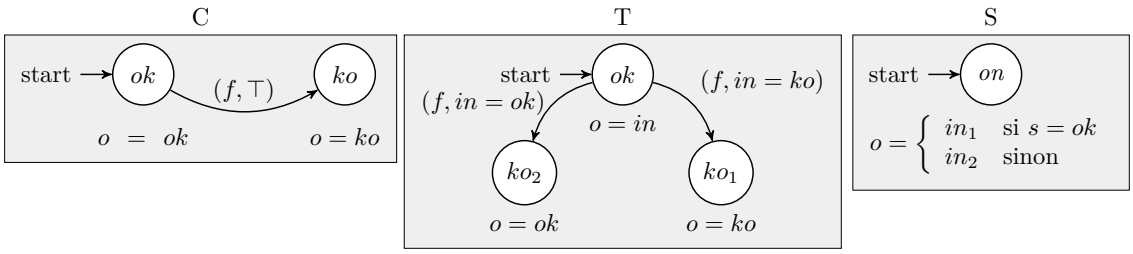


FIGURE 2.16 – Automates de mode des composants du système d'auto-vérification

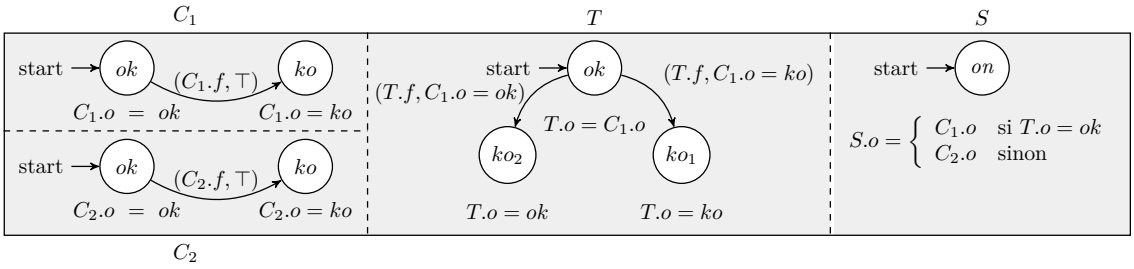


FIGURE 2.17 – Automates de mode du système d'auto-vérification

2.4 Résumé

Nous avons introduit, dans ce chapitre, les concepts élémentaires de sûreté de fonctionnement utilisés lors de la modélisation et l'analyse des défaillances d'un système. Nous avons ensuite défini les indicateurs classiquement utilisés pour évaluer la sûreté d'un système et illustré l'usage de ces indicateurs, notamment lors de la définition d'exigences de sûreté de fonctionnement. Nous avons alors dressé le panorama des formalismes de modélisation et d'analyse utilisés pour calculer ces indicateurs et vérifier qu'un système répond aux exigences de sûreté. Néanmoins nous avons montré que les méthodes classiques ne permettent pas d'assurer une modélisation maintenable du système, ce qui limite leur utilisation pour les systèmes de grande taille. Nous avons alors présenté les approches basées modèles utilisées pour outrepasser ces limitations. Nous focaliserons notre étude bibliographique sur ces langages de modélisation et d'analyse. Dans le prochain chapitre, nous nous intéressons à la phase de *durcissement d'architecture* et notamment aux différentes méthodes existantes pour mener à bien ce processus.

Chapitre 3

Durcissement d'architectures

Les concepteurs de systèmes critiques doivent proposer une architecture de leur système répondant à un ensemble d'exigences de sûreté. Cette phase de conception, appelée *durcissement d'architecture*, revient en fait à résoudre un problème dit *d'exploration de l'espace des architectures* consistant à trouver une architecture répondant aux exigences. Nous présentons dans ce chapitre le problème d'exploration de l'espace des architectures et les méthodes existantes pour résoudre ce problème. Pour cela, nous définissons, dans la section 3.1, le problème d'exploration de l'espace des architectures, notamment les entrées et les différentes étapes de résolution du problème et les différents types de systèmes et d'exigences considérés. Nous présentons ensuite, dans la section 3.2 et 3.3, les différentes méthodes existantes de résolution du problème d'exploration de l'espace des architectures et analysons les avantages et limitations de chacune.

3.1 Introduction du durcissement d'architectures

Les concepteurs de systèmes critiques doivent non seulement construire une architecture respectant les exigences fonctionnelles mais aussi assurer la sûreté de cette architecture en menant une Analyse Préliminaire de la Sûreté du Système (PSSA en anglais) démontrant le respect des exigences de sûreté de fonctionnement. Trouver une telle architecture revient à résoudre un problème *d'exploration de l'espace des architectures*.

3.1.1 Terminologie

Avant de poursuivre, définissons les termes suivants :

Durcissement d'architectures est une phase de développement consistant à construire une architecture respectant un ensemble d'exigences de sûreté de fonctionnement à partir d'une architecture assurant les exigences fonctionnelles (appelée *architecture initiale*).

Problème d'exploration de l'espace des architectures (problème DSE) consiste à trouver une architecture satisfaisant un ensemble de contraintes et/ou optimisant un ensemble de critères parmi un ensemble d'architectures (appelé *espace des architectures*). Pour des raisons de lisibilité nous notons le problème d'exploration de l'espace des architectures : problème DSE (*Design Space Exploration* en anglais).

Méthode de résolution du problème DSE est un processus partiellement ou totalement automatisé de résolution du problème DSE.

Le processus de durcissement d'architectures peut être mené en formulant et en résolvant un problème DSE contenant des contraintes sur les indicateurs de sûreté. Le problème DSE consiste alors à trouver *un candidat* appelé *solution* parmi *l'espace des architectures* respectant un ensemble d'exigences de sûreté. Certaines approches intègrent aussi une notion de coût et recherchent une solution optimale au sens de ces coûts. Définissons les notions de candidats et d'espace des architectures :

les alternatives d'un composant sont l'ensemble des composants atomiques et non-atomiques pouvant remplacer un composant du système initial ;

un candidat est une architecture obtenue par un ensemble de substitutions de composants du système initial par une de leurs alternatives ;

l'espace des architectures est l'ensemble des candidats possibles pour un problème donné. Celui-ci correspond à l'ensemble des systèmes obtenus par substitution des composants du système initial par des alternatives.

Comme illustré par la figure 3.1, les entrées d'une méthode de résolution du problème DSE sont :

- le modèle dysfonctionnel du système initial ;
- l'ensemble des alternatives possibles pour chaque composant du système ;
- un ensemble d'exigence de sûreté de fonctionnement.

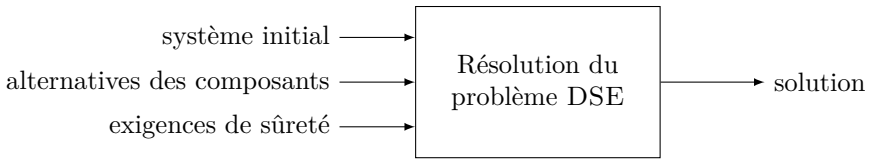


FIGURE 3.1 – Paramètres du problème DSE

Exemple 3.1 (Problème d'exploration sur ROSACE)**.** *Considérons l'événement redouté* la commande δ_{thc} est erronée *du système* ROSACE*, il faut alors trouver une architecture telle que* $R(100) \geq 0.9$*. Imaginons que tous les événements de* ROSACE *sont modélisés par des lois exponentielles* $p(C.e) = 1 - e^{-0.001t}$ *alors on peut montrer que la fiabilité du système* ROSACE *initial est*

$$R(100) \simeq 1 - (p(C_{Va}.e) + p(F_{Va}.e) + p(F_q.e)p(F_{Vz}.e)) \simeq 0.79$$

Il est alors nécessaire de durcir ROSACE *pour obtenir un système respectant cette exigence. Considérons que les composants* C_{Va}*,* F_{Va}*,* F_q *et* F_{Vz} *de* ROSACE *possèdent chacun trois* alternatives

Alternative I *le composant initial où le taux de défaillance pour les événements* e *et* l *est* 10^{-3} *par heure ;*

Alternative A *un composant amélioré où le taux de défaillance pour les événements* e *et* l *est* 5.10^{-4} *par heure ;*

Alternative D *l'application du patron de duplication où le taux de défaillance pour les événements* e *et* l *des deux composants est* 10^{-3} *par heure ;*

Un candidat *est un système obtenu en choisissant une alternative pour les composants C_{Va}, F_{Va}, F_q et F_{Vz}. Ainsi* l'espace des architectures *est l'ensemble des systèmes obtenus en énumérant tous les choix possibles c'est-à-dire $3^4 = 81$ systèmes. Une* solution *du problème* DSE *(si elle existe) est donc une architecture parmi les* 81 *possibles telle que $R(100) \geq 0.9$.*

Les alternatives des composants peuvent intégrer des mécanismes de sûreté spécifique au système étudié [52], néanmoins les alternatives sont généralement issues de constructions classiques appelées *patrons de conception* utilisés pour améliorer la sûreté du système.

3.1.2 Patrons de conception pour les systèmes critiques

Une définition classique de la notion de *patron de conception* est donnée par [42] : « *un patron de conception est une description d'un assemblage de modules et d'objets conçus pour résoudre un problème de conception générique dans un contexte particulier* ». Les patrons de conception pour les systèmes critiques (que nous appelons *patrons de sûreté*) sont des descriptions d'assemblages de composants génériques reconnus pour améliorer la sûreté du système dans un contexte donné.

Exemple 3.2 (Patron de duplication)**.** *Soit un composant C possédant une sortie et pouvant produire des valeurs erronées ou perdues. Pour détecter les erreurs, le patron de duplication de la figure 3.2, consiste à répliquer C en deux composants indépendants C_1 et C_2 et à utiliser un comparateur qui transmet la sortie de C_1 si les sorties de C_1 et C_2 sont égales, sinon le comparateur ne transmet pas la sortie au reste du système. En résumé, la sortie de ce patron est*

- *correcte si C_1 et C_2 fonctionnent correctement ;*
- *erronée si C_1 et C_2 produisent des valeur erronées, car s'ils produisent la même valeur erronée alors le comparateur la transmettra ;*
- *perdue sinon.*

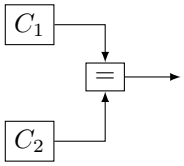


FIGURE 3.2 – Patron de duplication

Il existe différents catalogues comme ceux de [6, 53, 81] qui fournissent des méthodes de classification des patrons. Certains catalogues, comme celui de [6], offrent également une aide à la décision. Néanmoins ces méthodes ne prennent pas en compte l'interaction des composants au sein du système considéré, ce travail revient alors au concepteur qui doit gérer l'aspect combinatoire de la recherche de la solution.

3.1.3 Méthode générale de résolution

Le principe de la résolution est basé sur un processus itératif présenté dans la figure 3.3. Cette méthode est décomposée en deux étapes :

Vérification de la satisfaction des exigences c'est-à-dire mener la PSSA sur l'architecture considérée pour vérifier sa conformité. Si celle-ci est conforme, alors l'architecture est acceptable et est une solution du problème ,autrement l'architecture doit être modifiée ;

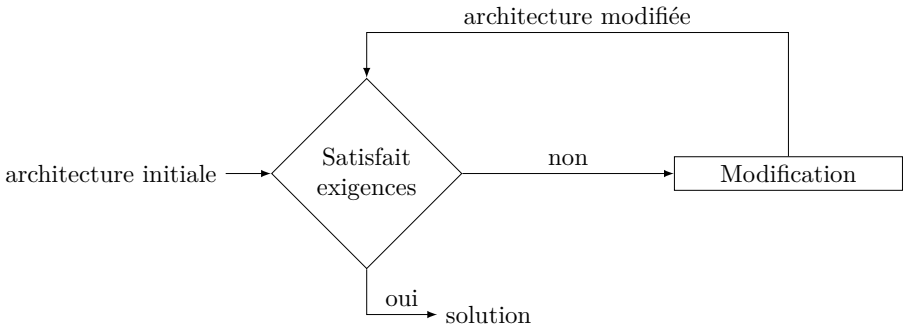


FIGURE 3.3 – Méthode générale de résolution

Modification c'est-à-dire transformer l'architecture pour améliorer sa sûreté. Pour cela, les concepteurs remplacent un sous-ensemble des composants du système par des patrons de sûreté. Le choix d'un patron est en général basé sur l'expérience propre à chaque concepteur mais aussi sur des paramètres comme l'impact sur les indicateurs de sûreté, le coût de mise en œuvre ou encore le cadre d'utilisation.

3.1.4 Automatisation de la résolution du problème DSE

L'approche de résolution itérative présentée précédemment est en général effectuée manuellement. Plus précisément, l'architecture initiale est modélisée et analysée à l'aide des méthodes présentées dans la section 2.2. Cette partie peut être automatisée grâce aux outils de modélisation et d'analyse que nous présenterons dans le chapitre suivant. En revanche, la phase de modification est très largement effectuée manuellement par les concepteurs qui introduisent des patrons de sûreté dans l'architecture initiale. Or cette phase de modification est humainement difficile à traiter car :

- pour un système possédant n composants, où chacun d'eux peut être remplacé par m alternatives, l'espace des architectures résultant contient m^n candidats. La combinatoire du problème est importante même pour des systèmes de taille modeste ;
- il est difficile d'appréhender *a priori* l'effet d'un ensemble de modifications sur le fonctionnement général du système.

Pour ces raisons, le domaine de l'exploration automatique de l'espace des architectures cherche à automatiser la résolution du problème DSE en remplaçant les étapes manuelles de vérification et de modification par des algorithmes de *sélection* et de *génération* des candidats.

Sélection L'espace des architectures étant souvent de taille importante, la méthode de sélection doit être capable de décider efficacement si un candidat ou un ensemble de candidats satisfait les exigences.

Génération Le processus de choix des modifications doit assurer, dans la mesure du possible, que :

- les modifications proposées améliorent la sûreté du système ;
- s'il existe une solution alors celle-ci sera générée, on dit alors que la méthode est *complète*.

Décrivons à présent les contraintes sur les entrées du problème DSE.

3.1.5 Contraintes sur le modèle dysfonctionnel

Le modèle dysfonctionnel doit permette de modéliser et d'analyser des systèmes réalistes, il doit donc être :

Fermé Le comportement ne dépend que des événements de défaillance du système.

Modulaire Les différents types de composants sont d'abord décrits dans une librairie de composants. Puis ceux-ci sont instanciés et connectés pour former le modèle du système. Ce type de modélisation permet de décrire des librairies de composants réutilisables et limite les erreurs de modélisation lors de la description du modèle dysfonctionnel.

Hiérarchique Les composants du système peuvent être non-atomiques, autrement dit contenir des sous-composants. Ceci permet de raffiner itérativement le comportement des composants et d'organiser le modèle.

3.1.6 Contraintes sur les alternatives

Les modifications de l'architecture initiale sont des substitutions c'est-à-dire que la stratégie de modification consiste à remplacer des composants par d'autres composants. Néanmoins remplacer un composant par un autre composant ne possédant pas la même interface est problématique car cette modification nécessite une procédure de reconnexion difficile à automatiser dans le cas général.

Hypothèse 3.1 (Contrainte d'interface)**.** *Les alternatives des composants du système possèdent la même interface que le composant initial*[1]*.*

3.1.7 Exigences de sûreté de fonctionnement

Afin de nous rapprocher des exigences de sûreté de fonctionnement réellement formulées par les concepteurs de systèmes critiques, nous considérons des exigences portant sur la fiabilité et l'ordre du système.

Hypothèse 3.2 (Exigences)**.** *Les exigences de sûreté de fonctionnement considérées sont :*
- *exigences de fiabilité, c'est-à-dire la fiabilité de la solution pour un temps d'opération donné doit être supérieure à la borne donnée par l'exigence ;*
- *exigences d'ordre, c'est-à-dire l'ordre de la solution doit être supérieur à un seuil donné par l'exigence.*

3.2 Méthodes de résolution du problème DSE fondées sur les heuristiques

Présentons dans cette section et la section suivante les différentes méthodes de résolution du problème DSE. Les études bibliographiques de [46] et [4] distinguent deux grandes familles d'approches :
- les approches basées sur l'utilisation *d'heuristiques* de résolution (section 3.2) c'est-à-dire qui ne peuvent pas garantir de trouver une solution au problème DSE même s'il en existe une (méthodes *incomplètes*) ;

1. Cette hypothèse est celle considérée par les outils d'exploration comme HipHOPS

– les approches dites *par contraintes* (section 3.3) formalisent le problème comme un ensemble de contraintes et délèguent la résolution à un solveur garantissant de trouver une solution si et seulement s'il en existe une (méthodes *complètes*).

Pour chaque méthode nous discuterons de son adéquation aux besoins établis dans la section 3.1.5 et en déduirons ses avantages et limitations.

3.2.1 Recherche locale

La recherche locale est une méthode d'optimisation consistant à se déplacer dans le *voisinage* d'un candidat initial pour trouver une solution optimisant un ensemble de critères. Cette méthode repose sur les notions suivantes :

Évaluation les critères à optimiser doivent être évaluables pour chacun des candidats à l'aide d'une *fonction d'évaluation*. Dans notre cas, cette fonction est une analyse de sûreté de fonctionnement donnant la valeur de la fiabilité et de l'ordre d'un système (et potentiellement d'autres critères comme le coût).

Voisinage il faut définir sous quelles conditions deux candidats sont voisins. Dans notre cas, deux candidats sont voisins si et seulement s'il existe un unique composant C au sein de l'architecture telle que l'alternative choisie pour C diffère.

Exemple 3.3 (Voisinage et évaluation)**.** *Considérons un sous-ensemble des candidats du problème* DSE *de* ROSACE *présenté dans l'exemple 3.1. Le graphe de voisinage de la figure 3.4 représente le sous-ensemble des candidats considéré où :*

- *un nœud correspond à un candidat et donne les alternatives choisies pour les composants de l'architecture ainsi que la fiabilité résultante ;*
- *un arc entre deux candidats signifie qu'ils sont voisins.*

Rappelons que l'alternative I est le composant initial de ROSACE *et l'alternative D est une duplication, donc le candidat initial correspondant à l'architecture de* ROSACE *est le candidat 1.*

La figure 3.5 décrit la méthode de recherche locale contenant les étapes suivantes :

1. si la condition d'arrêt (nombre d'itérations, qualité du candidat actuel) est vraie alors la recherche s'arrête, sinon évaluation de l'ensemble des voisins du candidat courant ;
2. exécution de la stratégie de déplacement qui choisit le nouveau candidat parmi les voisins du candidat courant.

Les différentes approches de recherche locale diffèrent vis-à-vis de l'implantation de la stratégie de déplacement. Commençons par une stratégie de déplacement dite *gloutonne*.

Approche basique

L'approche basique est une stratégie gloutonne utilisée par [98] pour résoudre le problème d'allocation de la fiabilité des composants d'une architecture (Reliability Allocation Problem en anglais).

Principe La stratégie consiste à se déplacer vers le voisin optimisant le mieux les critères de recherches, autrement dit à se déplacer en faisant uniquement des choix localement optimaux. La stratégie gloutonne ne revient donc jamais sur ses décisions. Dans notre cas, considérons un candidat s et $N(s)$ l'ensemble de ces voisins, l'algorithme se déplace alors sur un voisin s' ssi $R_{s'} \geq R_s$ et $R_{s'} = max_{s'' \in N(s)}(R_{s''})$ où R_x est la fiabilité associée à un candidat x.

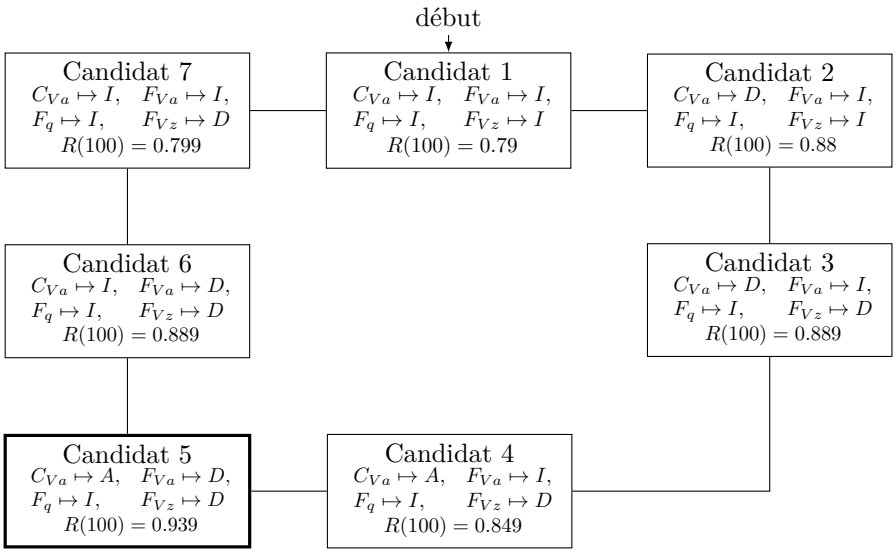


FIGURE 3.4 – Graphe du voisinage des candidats de l'exemple 3.4

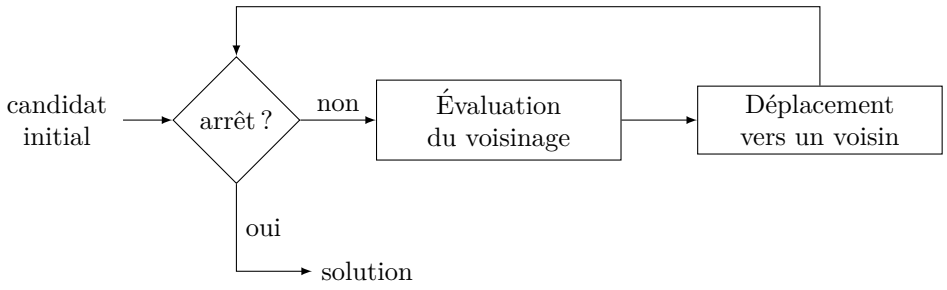


FIGURE 3.5 – Recherche locale générique

Exemple 3.4 (Recherche gloutonne)**.** *Considérons le problème* DSE *de* ROSACE *sur les voisins de l'exemple 3.3. Le chemin d'exploration emprunté par l'algorithme est :*

$$C_1 \to C_2 \to C_3$$

En effet, l'algorithme choisit de se déplacer vers le candidat 2 possédant une meilleure fiabilité, puis vers le candidat 3. Aucun des voisins du candidat 3 ne possède une meilleure fiabilité, l'algorithme s'arrête et renvoie le candidat 3 comme solution optimale du problème. Or le candidat 3 ne respecte pas l'exigence de sûreté $R_{C_3} = 0.889 \leq 0.9$*, donc la résolution du problème échoue.*

Notons que le candidat 5 (en gras dans le graphe) est une solution du problème mais pour l'atteindre, il faut faire des choix locaux non optimaux.

Approche tabou

La recherche tabou a été utilisée pour résoudre le problème d'allocation de la fiabilité par [59] ou encore [72]. Cette méthode peut être vue comme une amélioration des recherches gloutonnes évitant le blocage dans les optimaux locaux.

Principe Comme la recherche gloutonne, une recherche tabou choisit des mouvements localement optimaux mais peut reconsidérer ses choix lors de l'exploration. En effet l'algorithme de recherche tabou maintient une liste de candidats *tabous* c'est-à-dire vers lesquels il est interdit de se déplacer. Cette liste autorise l'algorithme à choisir des mouvements non optimaux et afin d'éviter de rester bloqué dans un optimum local. Il existe différentes politiques de gestion de cette liste, la plus simple consiste à conserver les n derniers candidats rencontrés.

Exemple 3.5 (Recherche tabou)**.** *Considérons que l'algorithme de recherche tabou possède :*
- *une variable best enregistrant le candidat rencontré ayant la plus grande fiabilité ;*
- *une variable c représentant le candidat courant ;*
- *une liste T de candidats tabous contenant les deux derniers candidats visités.*

Nous imposons que l'algorithme continue l'exploration tant qu'il n'a pas trouvé un candidat respectant l'exigence de sûreté $R(100) \geq 0.9$ parmi les candidats du problème DSE *de l'exemple précédent. La trace d'exécution est alors la suivante :*

$$\left\{\begin{array}{l} best = C_1 \\ c = C_1 \\ T = () \end{array}\right. \rightarrow \left\{\begin{array}{l} best = C_2 \\ c = C_2 \\ T = (C_1) \end{array}\right. \rightarrow \left\{\begin{array}{l} best = C_3 \\ c = C_3 \\ T = (C_2, C_1) \end{array}\right. \rightarrow \left\{\begin{array}{l} best = C_3 \\ c = C_4 \\ T = (C_3, C_2) \end{array}\right. \rightarrow \left\{\begin{array}{l} best = C_5 \\ c = C_5 \\ T = (C_4, C_3) \end{array}\right.$$

La recherche tabou emprunte le même chemin que l'approche gloutonne jusqu'au candidat 3, mais ici comme le candidat 2 est tabou, le seul choix possible est de se diriger vers le candidat 4 possédant une plus petite fiabilité. Ainsi, la recherche échappe à l'optimum local et trouve bien le candidat 5 comme une solution du problème.

Approches stochastiques

Les approches stochastiques, notamment le recuit simulé, ont été utilisées par [54] pour résoudre le problème d'allocation de la fiabilité. La stratégie de déplacement de ses approches est aussi basée sur des déplacements améliorant le critère. Néanmoins, celles-ci autorisent des déplacements dégradant le critère avec une certaine probabilité. Ceci évite le blocage dans les optimaux locaux que subissent les approches gloutonnes. Le principal paramètre de ces méthodes est le calcul de la probabilité d'accepter un déplacement dégradant le critère. Une méthode connue est celle du recuit simulé décrite ci-dessous.

Principe Le recuit simulé est basé sur deux notions :
- l'énergie E d'un candidat : valeur de la fonction à minimiser sur le candidat (dans notre cas on souhaite minimiser la défiabilité $\overline{R}$ pour maximiser la fiabilité R) ;
- la température T : paramètre contrôlant le non-déterminisme de la marche, autrement dit plus la température est élevée plus la marche est aléatoire.

La méthode consiste alors à :

1. choisir un voisin du candidat courant avec une probabilité paramétrée par la diminution de l'énergie (notée $\Delta E = E_{courant} - E_{voisin}$), autrement dit plus ΔE est petit plus la probabilité de choisir ce voisin est grande ;
2. se déplacer vers ce voisin si $\Delta E \leq 0$ ou avec une probabilité paramétrée par T et ΔE telle que lorsque T diminue alors la probabilité diminue jusqu'à atteindre 0 lorsque $T = 0$;
3. faire décroître T pour assurer la convergence de l'algorithme, en effet lorsque $T = 0$ l'algorithme ne choisit que des mouvements améliorant le critère (proche de l'approche gloutonne).

Exemple 3.6 (Recuit simulé)**.** *Considérons que la température initiale est 100 et qu'elle décroît de 50 par déplacement. Une trace possible de l'algorithme de recuit simulé sur le problème* DSE *de l'exemple 3.1 est alors :*

$$\left\{ \begin{array}{l} C_1 \\ T = 100 \end{array} \right. \rightarrow \left\{ \begin{array}{l} C_2 \\ T = 50 \end{array} \right. \stackrel{saut}{\rightarrow} \left\{ \begin{array}{l} C_1 \\ T = 0 \end{array} \right. \rightarrow \left\{ \begin{array}{l} C_7 \\ T = 0 \end{array} \right. \rightarrow \left\{ \begin{array}{l} C_6 \\ T = 0 \end{array} \right. \rightarrow \left\{ \begin{array}{l} C_5 \\ T = 0 \end{array} \right.$$

Dans un premier temps , l'algorithme choisit de se diriger vers le deuxième candidat qui améliore bien le critère ($\Delta\overline{R} = -0.09$). Mais à la deuxième itération l'algorithme choisit de retourner au candidat 1 malgré le fait qu'il n'améliore pas le critère, ceci est possible car la température n'est pas nulle. Par la suite l'algorithme choisit toujours un mouvement améliorant le critère car $T = 0$. Finalement, l'algorithme s'arrête sur le candidat 5 car il n'existe pas de mouvement améliorant le critère.

Avantages

L'évaluation des candidats peut être menée par n'importe quel algorithme d'analyse des systèmes. Par conséquent, le problème DSE peut être modélisé comme un problème d'optimisation de la fiabilité et de l'ordre des systèmes. De plus, le paramétrage de la stratégie de déplacement (par exemple la température pour le recuit simulé ou la gestion de la liste tabou) permet d'adapter la recherche au problème considéré. Il existe d'ailleurs des bornes théoriques, notamment sur la décroissance de la température pour le recuit simulé [48], assurant de converger en probabilité vers l'optimum global. Par ailleurs, les auteurs de [8], respectivement [59] proposent une extension du recuit simulé respectivement de la recherche tabou à l'optimisation multi-critères. Finalement la notion de voisinage s'exprime simplement (comme illustré dans l'exemple 3.3) pour le problème DSE.

Limitations

La stratégie de déplacement est basée sur la mesure d'une amélioration du critère de recherche. Or pour des contraintes dures, la fonction d'évaluation ne fait qu'informer l'algorithme de recherche si les contraintes sont respectées ou non (réponse binaire). Cette mesure de l'amélioration ne donne pas d'informations précises pour diriger la recherche. Comme décrit par [58], une tactique classique pour pallier ce problème consiste à ajouter une *fonction de pénalité* évaluant le *degré de falsification* des contraintes par un candidat. Dans notre cas, pour une contrainte sur un indicateur donné, la fonction de pénalité est, par exemple, la différence entre la borne donnée par la contrainte et la valeur de l'indicateur. Cette fonction devient alors un critère de recherche classique qui doit être minimisé.

Par ailleurs, il n'existe pas de garanties fortes de trouver une solution si elle existe (incomplétude). En effet, les bornes théoriques sur les paramètres de recherche sont en pratique difficiles à utiliser car la convergence est trop lente sur des cas d'étude concrets. L'utilisateur doit alors choisir les paramètres empiriquement afin d'explorer suffisamment l'espace des architectures pour éviter les optimaux locaux tout en évitant d'énumérer toutes les architectures.

Finalement, à chaque itération, l'algorithme de recherche doit évaluer l'ensemble du voisinage d'un candidat, ce qui correspond dans notre cas à nm voisins où m est le nombre de composants et n le nombre d'alternatives par composants. Si l'algorithme effectue k itérations, alors le nombre d'appels à la fonction d'évaluation est de nmk ce qui peut être prohibitif pour des systèmes réalistes car le calcul de la fiabilité et des coupes minimales est potentiellement coûteux.

3.2.2 Méthodes d'optimisation peuplées

Les méthodes d'optimisation peuplées sont fondées sur un ensemble *d'agents* explorant l'espace de recherche et dont l'interaction permet de trouver la solution du problème. Présentons en particulier les méthodes d'optimisation par colonies de fourmis et algorithmes génétiques.

Colonies de fourmis

La méthode considère un ensemble d'agents (ici des fourmis) pour trouver la solution du problème. Les colonies de fourmis ont été utilisées par [60] pour résoudre le problème d'allocation de fiabilité.

Principe Une colonie est un ensemble de fourmis explorant les différents candidats du problème par recherche locale. Une itération de l'algorithme est composée de trois phases :

1. Chaque fourmi choisit de manière probabiliste le déplacement vers un voisin du candidat actuel. Cette probabilité est paramétrée par :
   - l'amélioration du critère, dans notre cas l'augmentation de la fiabilité ;
   - la dose de *phéromone* déposée sur le déplacement. La dose de phéromone caractérise l'attractivité de ce déplacement pour les fourmis ayant déjà emprunté ce chemin.
2. Une fois le voisin choisi, la fourmi se déplace et recommence jusqu'à une condition d'arrêt. Lorsque les fourmis ont fini leur chemin, celles-ci déposent une quantité de phéromone proportionnelle à l'attractivité du chemin exploré sur l'ensemble des arrêtes du chemin.
3. Finalement les phéromones sont *atténuées* par une loi d'évaporation décrivant la décroissance de la teneur en phéromone sur les arcs au fil des itérations. Ce processus priorise les informations récentes et contrôle la portée (en terme d'itérations) des choix des fourmis.

Exemple 3.7 (Colonies de fourmis)**.** *Considérons une colonie de 100 fourmis initialement sur le premier candidat, chacune possède trois déplacements pour explorer les candidats possibles du problème* DSE *de l'exemple 3.1. Nous définissons :*

- *la désirabilité du mouvement d'un candidat C_i à un candidat voisin C_j comme $\eta_{ij} = e^{10(R_{C_j} - R_{C_i})}$*
- *la probabilité qu'une fourmi choisisse le déplacement de C_i vers C_j à l'itération k est*

$$p_{ij}^k = \frac{\eta_{ij}\tau_{ij}^k}{\sum_{C_l \in N(C_i)} \eta_{il}\tau_{il}^k}$$

où $N(C_i)$ sont les voisins de C_i et τ_{ij}^k est la dose de phéromone sur l'arc de C_i vers C_j ;

- *le taux d'évaporation des phéromones par itération est* 0.5 *;*
- *le dépôt de phéromone sur les arcs d'un chemin parcouru par une fourmi par* $\Delta\tau = e^{5(R_{best}-R_{req})}$ *où* $R_{req} = 0.9$ *est l'exigence de sûreté et* R_{best} *est la fiabilité maximale des candidats rencontrés sur le chemin.*

La table 3.1 représente les meilleurs candidats trouvés par les fourmis après chacune des quatre itérations imposées. Aux premières itérations, la répartition des fourmis est majoritairement centrée sur les candidats 2 et 3 car sans phéromone, la probabilité de prendre un chemin menant à 2 ou 3 est plus grande que celle menant au candidat 5. En effet, les chemins menant au candidat 5 contiennent des mouvements localement non-optimaux donc peu probables. Or le candidat 5 est plus fiable que le candidat 2 ou 3, donc la dose phéromone déposée par les fourmis ayant rencontré le candidat 5 est plus importante que celle déposée par les autres fourmis. Aux itérations suivantes, la faible attractivité des mouvements menant au candidat 5 est compensée par l'importante dose de phéromone déposée par les fourmis. Par la suite, l'évaporation et les dépôts successifs de phéromone renforce le chemin menant au candidat 5 comme le montre la répartition à la dernière itération.

	Candidats						
k	C_1	C_2	C_3	C_4	C_5	C_6	C_7
1	0	25	55	0	16	4	0
2	0	4	50	0	46	0	0
3	0	0	29	0	71	0	0
4	0	0	16	0	84	0	0

TABLE 3.1 – Répartition des fourmis par meilleur candidat trouvé sur leur chemin

Algorithmes génétiques

Les algorithmes génétiques font aussi partie des approches peuplées et sont fondés sur le processus évolutif d'une population soumise à un processus de sélection. Comme le souligne [4], ce type d'approche est très largement utilisé pour résoudre le problème DSE. Nous pouvons citer l'utilisation de l'algorithme génétique NSGA-II [34] par l'outil d'exploration HIPHOPS [3, 96] présenté en détail dans [78].

Principe Pour les algorithmes génétiques, un candidat (aussi appelé *individu*) est décrit par un *génotype*, c'est-à-dire par un ensemble de *gènes* qui encodent ses *caractéristiques* (dans notre cas ce sont les composants du système). Il existe pour chaque gène un ensemble d'alternatives possibles appelées *allèles* (dans notre cas ce sont les alternatives d'un composant). Le processus évolutif consiste à générer un ensemble d'individus (appelé *population*) de manière aléatoire, puis à effectuer un ensemble d'itérations dont les étapes sont :

1. appliquer les opérateurs de *mutation* et de *croisement* (crossover en anglais) avec une certaine probabilité respectivement p_m et p_r sur chaque individu de la population courante afin d'augmenter la diversité de la population ;
2. appliquer l'opérateur de *sélection* qui extrait de la population courante, un ensemble d'individus formant la nouvelle population.

Le processus se poursuit tant que la condition d'arrêt n'est pas satisfaite, en général, celle-ci est soit un nombre maximum d'itérations ou un critère sur la qualité de la population.

L'opérateur de croisement consiste à créer des candidats *enfants* dont le génotype est une recombinaison des génotypes des parents.

Exemple 3.8 (Croisement)**.** *Soit C_2 et C_7 deux candidats du problème* DSE *de l'exemple 3.1 alors un résultat possible du croisement est :*

$$\left.\begin{array}{cccccc} & C_{Va} & F_{Va} & F_q & F_{Vz} \\ C_2 & D & I & I & I \\ C_7 & I & I & I & D \end{array}\right\} \overset{croisement}{\longrightarrow} \begin{array}{cccccc} & C_{Va} & F_{Va} & F_q & F_{Vz} \\ C_1 & I & I & I & I \\ C_3 & D & I & I & D \end{array}$$

L'opérateur de mutation modifie aléatoirement un ou plusieurs gènes d'un individu afin d'augmenter la diversité d'une population en introduisant des allèles potentiellement absents de la population courante. Cette diversité évite de tomber dans des optimaux locaux.

Exemple 3.9 (Mutation)**.** *Un résultat possible de mutation sur C_2 est :*

$$\left.\begin{array}{cccccc} & C_{Va} & F_{Va} & F_q & F_{Vz} \\ C_2 & D & I & I & I \end{array}\right\} \overset{mutation}{\longrightarrow} \begin{array}{cccccc} & C_{Va} & F_{Va} & F_q & F_{Vz} \\ C_3 & D & I & I & D \end{array}$$

L'opérateur de sélection choisit parmi une population les individus les plus adaptés, en général ceux qui optimisent un ensemble de critère (la fiabilité dans notre cas). Cet opérateur simule la pression de sélection pour diriger l'évolution de la population vers des individus optimisant les critères de la recherche. Un opérateur de sélection simple est la sélection *élitiste* qui ne conserve que les n meilleurs individus s'il n'y a qu'un seul critère, ou les individus non dominés c'est-à-dire tels qu'il n'existe pas d'autres individus meilleurs pour tous les critères.

Exemple 3.10 (Sélection)**.** *Soit la population $P_k = \{C_1, C_2, C_3, C_7\}$ à l'itération k, imaginons que le critère de sélection soit la fiabilité et que la sélection ne conserve que les trois meilleurs individus alors la population de l'itération suivante est $P_{k+1} = \{C_2, C_3, C_7\}$.*

Avantages

Les méthodes d'exploration peuplées n'imposent aucune restriction sur les fonctions d'évaluation (comme la linéarité, continuité, monotonie, etc) des candidats. Ainsi, ces algorithmes sont directement utilisables pour résoudre le problème DSE.

D'ailleurs, les algorithmes génétiques actuels comme NSGA-III [33] et l'extension des colonies de fourmis de [51] intègrent l'optimisation multi-critères et offrent à l'utilisateur les solutions non-dominées c'est-à-dire une estimation du front de Pareto.

Par ailleurs la phase d'exploration de l'espace de recherche par les agents (marche des fourmis, mutation et croisement pour les algorithmes génétiques) peut être parallélisée afin de réduire le temps de calcul.

Finalement, dans le cas des algorithmes génétiques, les contraintes dures peuvent être intégrées à l'opérateur de sélection pour diriger plus efficacement la recherche.

Limitations

Ces approches ne peuvent garantir de trouver une solution si elle existe (incomplétude). Néanmoins, dans le cas des colonies de fourmis, [47] a prouvé la convergence en probabilité de la

colonie vers l'optimum global sous certaines conditions telles que le nombre de fourmis, le dépôt de phéromone et le taux d'évaporation. Mais [47] souligne que ces bornes sont en pratique inutilisables.

La convergence des méthodes d'exploration peuplées vers un optimum global est grandement impactée par l'ensemble des paramètres de l'algorithme. Or comme les bornes théoriques ne sont pas utilisables en pratique, ces paramètres doivent être choisis empiriquement pour chaque problème DSE.

Finalement une recherche effectuée par un ensemble de n agents explorant un candidat par itération et dont la recherche est répétée sur k itérations fait $\mathcal{O}(nk)$ appels au calcul de la fiabilité ce qui peut être coûteux pour des systèmes de taille réaliste.

3.3 Méthodes de résolution du problème DSE fondées sur la programmation par contraintes

La deuxième grande famille des méthodes de résolution du problème DSE identifiée par [4] est la famille des approches dites *de programmation par contraintes*. Celles-ci proposent de formuler le problème DSE dans un formalisme pris en entrée par des solveurs dédiés pour résoudre le problème.

3.3.1 Optimisation non-linéaire

Les techniques d'optimisation de fonctions non-linéaires ou la programmation dynamique ont été utilisées par [71] ou encore [41] pour résoudre un problème d'allocation de fiabilité. Nous ne décrivons pas ces techniques d'optimisation comme la méthode de descente de gradient, le lecteur intéressé pourra se reporter à des ouvrages d'introduction aux méthodes d'optimisation exactes comme [61]. Présentons plus en détail la formulation du problème DSE proposée par [71].

Principe L'idée exposée par les auteurs de [71] est d'exprimer la fiabilité du système comme une fonction dépendant des probabilités des événements de défaillance. Les densités de probabilité associées aux événements sont alors paramétrées par un ensemble de variables. Le problème consiste alors à trouver une valeur de ces variables maximisant la fiabilité.

Exemple 3.11 (Optimisation non-linéaire)**.** *Rappelons que la fiabilité du cas* Rosace *dans le problème* DSE *de l'exemple 3.1 est :*

$$R(t) = 1 - (p(C_{Va}.e) + p(F_{Va}.e) + p(F_q.e)p(F_{Vz}.e))$$

Considérons que la densité de probabilité d'un événement C.e est paramétrée par une valeur $x_{C.e}$ tel que $p(C.e) = 1 - e^{-x_{C.e}\lambda_{C.e}t}$. Considérons les taux de défaillance $\lambda_{C.e} = 0.001$ pour tous les événements. Soit $x_{CVa.e}, x_{FVa.e}, x_{Fq.e}, x_{FVz.e} \in [0, 3]$ les paramètres alors la fiabilité au bout de 100 heures de fonctionnement peut être définie comme :

$$R(100) = 1 - ((1 - e^{-0.1x_{CVa.e}}) + (1 - e^{-0.1x_{FVa.e}}) + (1 - e^{-0.1x_{Fq.e}})(1 - e^{-0.1x_{FVz.e}}))$$

Le problème consiste alors à trouver une valeur des paramètres telle que la fiabilité soit maximale, dans ce cas simple la solution est $\{x_{CVa.e} \mapsto 3, x_{FVa.e} \mapsto 3, x_{Fq.e} \mapsto 3, x_{FVz.e} \mapsto 3\}$.

Avantages Cette méthode repose sur des techniques de résolution exactes pour trouver la solution optimale du problème.

Limitations Cette formalisation ne considère que le cas où les alternatives des composants peuvent être modélisées comme des paramètres venant modifier la probabilité d'occurrence des événements de défaillance des composants du système. Cette hypothèse s'applique si les alternatives ne sont que des versions du composant initial où seuls les taux de défaillance des événements changent (comme pour l'alternative A). Or nous autorisons toutes les alternatives préservant l'interface du composant initial, la fonction de fiabilité peut donc être complètement transformée par la substitution d'un composant. Il n'est donc pas possible de modéliser ces substitutions en ajoutant des paramètres aux distributions de probabilité des événements des composants initiaux. Par conséquent, cette formalisation n'est pas utilisable pour résoudre le problème DSE.

3.3.2 Programmation linéaire à nombres entiers

Les méthodes d'optimisation par Programmation Linéaire à Nombres Entiers (PLNE ou Integer Linear Programming en anglais) ou entiers et réels (Mixed Integer Linear Programming en anglais) ont été utilisées pour résoudre le problème d'allocation de fiabilité par [5] ou encore [27] mais aussi pour résoudre le problème d'allocation de redondance sur des systèmes électriques pour assurer des contraintes d'ordre et de fiabilité par [65].

Par ailleurs, les problèmes pseudo-booléens (qui sont des restrictions du problème PLNE aux variables entières 0/1) ont été utilisés notamment dans notre première version de la méthode d'exploration, présentée dans [35], ne traitant que des exigences sur l'ordre du système.

Principe Pour utiliser la programmation linéaire, il est nécessaire d'exprimer le problème comme un ensemble de contraintes linéaires sur des variables entières ou réelles. Dans notre cas, ces variables encodent les différentes alternatives possibles et les contraintes formalisent les exigences de sûreté. Il est aussi possible de demander à optimiser un ou plusieurs critères linéaires. Soit x le vecteur $n \times 1$ des variables entières, c un vecteur $n \times 1$ de coûts entiers, A une matrice $n \times n$ à coefficients entiers et b un vecteur $n \times 1$ à coefficient entiers alors la forme canonique d'un problème PLNE est :

$$\begin{array}{ll} \text{minimiser} & c^T x \\ \text{tel que} & Ax \leq b \end{array}$$

Un problème PLNE est généralement résolu avec la méthode appelée *branch and cut* [73]. Les étapes principales sont :

1. diviser le problème en sous-problèmes en fixant la valeur de certaines variables ;
2. pour chaque sous-problème tenter de résoudre la relaxation au problème d'optimisation linéaire à variables réelles pour laquelle il existe l'algorithme du simplexe [25] de complexité polynomiale.
   - S'il n'existe pas de solution alors le sous-problème PLNE initial n'a pas de solution et la recherche s'arrête.
   - Sinon l'algorithme cherche à raffiner l'espace des solutions en calculant des *plans de coupes*, c'est-à-dire des contraintes entre les variables préservant les solutions entières mais rejetant des solutions réelles.
3. retourner la meilleure solution (au sens du coût) trouvée dans l'ensemble des sous-problèmes.

La résolution des problèmes pseudo-booléens peut aussi être menée par le *branch and cut* utilisé pour les problèmes PLNE. Néanmoins il existe d'autres méthodes de résolution inspirées du mécanisme de résolution du problème SAT comme le solveur SAT4J [11] que nous avons utilisé dans [35].

Illustrons la formalisation du problème DSE sous des contraintes d'ordres. La méthode consiste à augmenter incrémentalement l'ordre d'une architecture initiale en substituant des composants par des patrons de sûreté. Pour cela, le problème de sélection des substitutions est encodé comme un problème pseudo-booléen. La méthode d'exploration est basée sur un processus itératif décrit par la figure 3.6 dont les principales étapes sont les suivantes :

Évaluation de la sûreté Les coupes minimales de l'architecture sont calculées par l'outil CECILIA-OCAS [29]. Si la cardinalité des coupes calculées précédemment respecte la contrainte d'ordre, alors l'architecture courante est valide. Autrement, il est nécessaire d'effectuer des substitutions dans le but d'améliorer l'ordre.

Choix des substitutions Le choix des substitutions à effectuer est décomposé en deux sous-problèmes :

- une sélection des composants du système sur lesquels une substitution doit être effectuée ;
- une sélection de l'alternative à appliquer sur les composants sélectionnés pour améliorer l'ordre du système.

Le choix des composants et des alternatives est basé sur la résolution d'un problème pseudo-booléen par le solveur SAT4J [11] dont la solution assure que l'ordre du système obtenu après substitution sera strictement supérieur à l'ordre du système courant.

Substitution des composants Une fois les substitutions choisies, l'utilisateur doit effectuer ces substitutions sur le modèle et relancer l'analyse de sûreté.

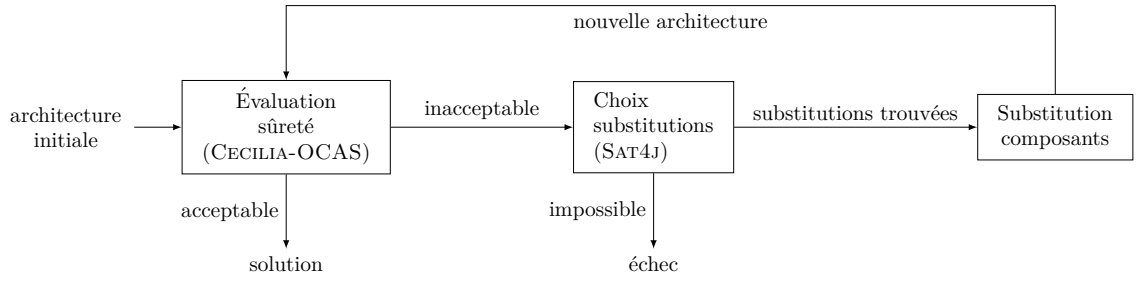


FIGURE 3.6 – Méthode de résolution

Le problème de choix des composants est décrit par un ensemble de contraintes demandant à ce que pour chaque coupe mc ne respectant pas la contrainte d'ordre, au moins un composant C soit choisi tel que $C.e \in mc$ où e est un événement de C.

Par ailleurs, pour assurer que l'alternative choisie améliore l'ordre du système, l'idée est de *classer* les alternatives en fonction de leur tolérance aux fautes. Autrement dit une alternative A est meilleure qu'une alternative B si et seulement si le nombre minimal d'occurrences d'événements pour que A soit défaillante et strictement supérieur à celui de B. L'amélioration est alors encodée par un ensemble de contraintes sur chaque composant assurant que si le composant est choisi alors une alternative meilleure que la précédente doit être choisie. Si le composant n'est pas choisi alors l'alternative courante est conservée.

Exemple 3.12 (Problème pseudo-booléen)**.** *Reprenons le problème* DSE *de l'exemple 3.1 pour la contrainte* l'ordre du système doit être supérieur ou égal à deux*. Les coupes* $\{\{C_{Va}.e\}, \{F_{Va}.e\}\}$ *du système* Rosace *initial ont une cardinalité de un. Il existe trois alternatives par composant* I*,* A *et* D *pour améliorer* Rosace*. Du point de vue de l'ordre, le nombre minimum d'occurrences d'événements nécessaires à l'alternative* I *ou* A *pour être défaillante est de 1 alors que pour la duplication* D*, elle est de 2. Dans ces conditions,* I *et* A *ne sont pas comparables et* D *est meilleure que* I *et* A*. Les variables et constantes du problème de choix sont :*

- x_C *valant 1 si le composant* C *est choisi, 0 sinon ;*
- $x_{C \mapsto C'}$ *valant 1 si le composant* C *est remplacé par l'alternative* C'*, 0 sinon ;*
- $\omega_{C \mapsto C'}$ *le coût de replacement de* C *par* C' *;*
- $b_{X>X'}$ *constante valant 1 si l'alternative* X *est meilleure que X', 0 sinon ;*

Le problème de choix est alors :

$$
\begin{array}{ll}
\textit{minimiser} & \sum_{C \in \{C_{Va}, F_{Va}, F_q, F_{Vz}\}} \sum_{C' \in \{I,A,D\}} \omega_{C \mapsto C'} x_{C \mapsto C'} \\
\textit{tel que} & \\
& \textit{au moins un composant choisi par coupe} \\
& x_{C_{Va}} \geq 1 \\
& x_{F_{Va}} \geq 1 \\
& \\
& \textit{si composant choisi alors alternative meilleure que } I \\
& -x_{C_{Va}} + \sum_{X \in \{I,A,D\}} b_{X>I} x_{C_{Va} \mapsto X} \geq 0 \\
& -x_{F_{Va}} + \sum_{X \in \{I,A,D\}} b_{X>I} x_{F_{Va} \mapsto X} \geq 0 \\
& -x_{F_q} + \sum_{X \in \{I,A,D\}} b_{X>I} x_{F_q \mapsto X} \geq 0 \\
& -x_{F_{Vz}} + \sum_{X \in \{I,A,D\}} b_{X>I} x_{F_{Vz} \mapsto X} \geq 0 \\
& \\
& \textit{une seule alternative choisie par composant} \\
& \sum_{X \in \{I,A,D\}} x_{C_{Va} \mapsto X} = 1 \\
& \sum_{X \in \{I,A,D\}} x_{F_{Va} \mapsto X} = 1 \\
& \sum_{X \in \{I,A,D\}} x_{F_{Vz} \mapsto X} = 1 \\
& \sum_{X \in \{I,A,D\}} x_{F_q \mapsto X} = 1
\end{array}
$$

La solution de ce problème est $\{x_{C_{Va}} \mapsto 1, x_{F_{Va}} \mapsto 1, x_{C_{Va} \mapsto D} \mapsto 1, x_{F_{Va} \mapsto D} \mapsto 1\}$ *et les autres variables à 0. Ceci signifie que dupliquer* C_{Va} *et* F_{Va} *suffit à résoudre le problème.*

Avantages Cette méthode offre une résolution du problème DSE contenant des exigences d'ordre, sans restriction sur le type de système ou d'alternatives. Par ailleurs il est possible d'ajouter des critères de coût afin de trouver la solution optimale respectant les contraintes. De plus, l'espace des architectures peut être simplement encodé à partir des variables entières.

Limitations La principale limitation est la linéarité des contraintes, imposée par le formalisme. En effet, la formule de la fiabilité est en générale non linéaire et donc ne peut pas être modélisée dans un problème PLNE ou MILP. Pour pallier ce problème, la stratégie classique consiste à approximer la fonction objectif de fiabilité par un ensemble de contraintes linéaires (voir l'exemple 3.13). Néanmoins les auteurs appliquent cette technique à un type particulier de système comme les systèmes série-parallèle [13, 27, 5] ou les systèmes grille [65]. Cette méthode n'est, à notre

connaissance, pas extensible à tout type de systèmes.

Exemple 3.13 (Linéarisation de la fonction objectif)**.** *Soit un système série contenant deux composants de fiabilité respective r_1 et r_2. Considérons que chacun peut être redondé x_1 respectivement x_2 fois avec un composant possédant la même fiabilité. Le but est alors de maximiser la fiabilité du système en choisissant une valeur des variables x_1 et x_2 tout en assurant des contraintes de coûts et de poids. Concentrons nous sur la linéarisation de la fonction objectif c'est-à-dire de la fiabilité. L'objectif initial est*

$$max\ R = (1 - (1 - r_1)^{x_1})(1 - (1 - r_2)^{x_2})$$

[27] propose de décomposer l'objective en un ensemble de sous-objectifs :

$$min\{(1 - r_1)^{x_1}, (1 - r_2)^{x_2}\}$$

En appliquant le logarithme on obtient

$$min\{ln(1 - r_1)x_1, ln(1 - r_2)x_2\}$$

En agrégeant les différents objectifs à l'aide de coefficients de poids ω_i, on obtient un objectif exprimé comme une fonction linéaire

$$min\ \gamma_1 x_1 \omega_1 + \gamma_2 x_2 \omega_2\ \ où\ \gamma_i = ln(1 - r_i)$$

3.3.3 Résolution de problèmes SMT

La résolution de problèmes de Satisfiablilité Modulo Théorie (SMT) a été utilisée notamment dans [57] pour analyser les systèmes statiques et dans [80] pour synthétiser des configurations d'un système assurant, entre autres, des exigences de sûreté. La formulation et la résolution des problèmes SMT sont présentées en détail par le chapitre 5. Présentons la formalisation du problème DSE proposée par [80].

Principe L'idée est de représenter l'espace des architectures comme un système où toutes les alternatives des composants et tous les liens possibles entre composants sont décrits. Le but du solveur SMT est alors de sélectionner les alternatives et les liens permettant d'assurer des propriétés dans notre cas sur la fiabilité et l'ordre du système. Notons que les auteurs de [80] ne se restreignent pas à un type de propriété particulier, cette approche est donc utilisable dans divers domaines (temps réel, contrôle-commande, etc).

Plus précisément, chaque composant est décrit par une interface et un ensemble de fonctions, paramètres et contraintes locales qui décrivent les contraintes entre les données d'entrée et de sortie. Les liens entre composants sont alors des contraintes entre les données d'entrée et de sortie des composants. Mais ces contraintes ne s'appliquent que si le lien est *actif*, autrement dit le lien entre un composant C_j et un composant C_i est actif lorsque la variable d'activation $\beta_{C_j}^{C_i}$ est vraie.

Le solveur SMT doit alors trouver une valuation des variables de sélection qui permet d'assurer les propriétés locales et globales du système.

Exemple 3.14 (Exploration avec SMT)**.** *Reprenons le problème* DSE *sur l'ordre du système* ROSACE*. Pour des raisons de lisibilité, considérons que seuls C_{Va} et F_{Va} possèdent les alternatives*

$\{I, A, D\}$. *La représentation graphique de l'espace des architectures (reprise de [80]) est alors donnée par la figure 3.7.*

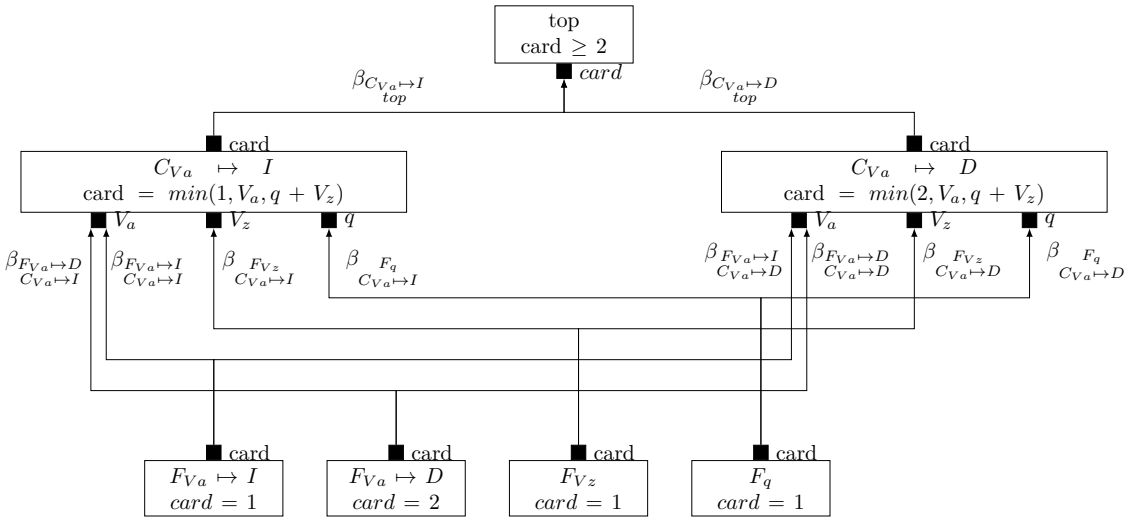


FIGURE 3.7 – Modélisation du problème d'exploration

Le système contient :

- *l'ensemble des alternatives possibles pour les composants C_{Va} et F_{Va} (si l'alternative pour F_{Va} est I alors le composant est noté $F_{Va} \mapsto I$) ;*
- *les liens possibles entres les différentes alternatives avec les variables de sélection correspondantes ;*
- *les propriétés et paramètres des composants ainsi que la propriété générale donnée par le composant top.*

Avantages La méthode de modélisation du problème DSE proposée par [80] est applicable à tout système basé sur la notion de composants assurant des propriétés, dans notre cas de sûreté de fonctionnement. Cette modélisation capture une grande variété de systèmes et en particulier les modèles dysfonctionnels des systèmes statiques.

De plus les propriétés peuvent être de natures très différentes, ceci permet d'explorer l'espace des architectures sous des contraintes de sûreté, de coûts, mais aussi d'autres types, tant que celles-ci sont exprimables dans la MSFOL.

Finalement, l'utilisation de solveurs SMT pour résoudre le problème DSE permet de profiter des différents mécanismes et heuristiques notamment :

- la génération de clauses de conflits lors de l'exploration qui élague l'espace de recherche sans écarter de solutions ;
- les heuristiques de décisions dynamiques guidées par les conflits.

Si le problème est exprimable dans une logique décidable alors le solveur assure de trouver, en un temps fini, une solution si et seulement si elle existe.

Limitations Néanmoins, l'approche présentée ne donne pas de méthodes de traitement des contraintes métiers comme celles de la sûreté de fonctionnement. Celles-ci sont directement exprimées comme des propriétés arithmétiques (arithmétique réelle non linéaire pour la fiabilité et arithmétique entière linéaire pour la cardinalité) dont les procédures de décisions (si elles existent) sont coûteuses. La résolution du problème DSE sur des systèmes réalistes est alors peu efficace. Les auteurs de [80] ont d'ailleurs identifié cette limitation et proposent d'y remédier en définissant des théories spécialisées aux contraintes métiers. Ce constat est une des motivations principales de la définition d'une théorie dédiée à la sûreté de fonctionnement qui fait partie de nos contributions.

3.3.4 Récapitulatif

La table 3.2 résume les avantages et les limitations des méthodes présentées dans ce chapitre. Nous nous focalisons notamment sur la :

Formalisation du problème c'est-à-dire capacité de la méthode à traiter des problèmes portant sur les systèmes, exigences et alternatives adressés dans la section 3.1.5 ;

Résolution du problème c'est-à-dire sur la capacité à offrir des garanties de résolution à savoir la complétude, la possibilité d'optimiser des coûts et la garantie de converger vers un résultat quelque soit le réglage des paramètres de la méthode (si elle en possède).

		Formalisation du problème			Résolution du problème		
	Méthode	systèmes modélisables	exigences modélisables	alternatives modélisables	complétude	optimisation coûts	convergence assurée
Heuristique	Glouton	✓	✓	✓	✗	✓	✓
	Tabou	✓	✓	✓	✗	✓	✗
	Recuit	✓	✓	✓	en probabilité	✓	✓
	Fourmis	✓	✓	✓	en probabilité	✓	✗
	Génétique	✓	✓	✓	✗	✓	✗
Contrainte	Optimisation non-linéaire	restrictions	restrictions	restrictions	✓	✓	✓
	PLNE	restrictions	restrictions	restrictions	✓	✓	✓
	SMT	✓	✓	✓	✓	✓	✓

TABLE 3.2 – Tableau comparatif des méthodes d'exploration

Méthodes basées heuristiques La table 3.2 met en évidence que les méthodes d'exploration basées sur les heuristiques offrent une formalisation du problème couvrant les systèmes, exigences et alternatives que nous considérons. En effet, ces méthodes ne dépendent pas de la manière dont les candidats sont évalués, autrement dit tout outil d'analyse de la sûreté de fonctionnement peut être utilisé pour donner à ces méthodes les valeurs des indicateurs pour un candidat donné. De plus les substitutions opérées sur l'architecture initiale sont encodées par des choix sur des variables de décisions (par exemple choix d'un chemin pour les fourmis, gènes pour les algorithmes génétiques), ce qui facilite la représentation de l'espace des architectures.

Néanmoins les méthodes basées heuristiques offrent des garanties limitées sur la résolution du problème. En effet ces méthodes peuvent au mieux assurer la convergence en probabilité vers la solution sous des contraintes sur les paramètres de ces méthodes (recuit simulé et fourmis). Or ces contraintes ne peuvent être respectées en pratique car elles réduisent drastiquement la vitesse

de convergence des méthodes (décroissance de température pour le recuit et nombre de fourmis pour les colonies de fourmis). De plus ces méthodes sont ajustables à l'aide de paramètres (par exemple pour les algorithmes génétiques : population initiale, taux de mutation) influençant la vitesse de convergence et pour certaines la capacité à converger vers une solution. Or les auteurs de ces méthodes ont déjà identifiés la difficulté de choisir des paramètres assurant *a priori* une résolution rapide du problème.

Notons qu'il existe des approches hybrides mêlant les techniques d'optimisation fondées sur les heuristiques et sur la satisfaction de contraintes. Nous pouvons notamment citer la recherche locale dirigée par les conflits introduite par [50]. Ces méthodes utilisent les informations données par la génération et l'apprentissage de clauses de conflit (traditionnellement utilisés par les solveurs SMT) pour guider la recherche. Néanmoins, il n'existe pas, à notre connaissance, d'utilisation de ces techniques pour la résolution du problème DSE.

Méthodes basées contraintes A l'inverse, la majorité des méthodes basées contraintes imposent des restrictions sur les systèmes, les exigences et les alternatives considérées. En effet, pour ces méthodes l'évaluation des indicateurs de sûreté doit être exprimée dans le formalisme utilisé par la méthode (un ensemble de contraintes linéaires pour PLNE par exemple) ce qui complique la formalisation du problème car :

- il est difficile de donner une formalisation des indicateurs de sûreté permettant de les calculer pour tous les candidats de l'espace des architectures. Ces approches se focalisent sur des sous-groupes de systèmes (comme les systèmes série-parallèle) dont la structure permet de formaliser le calcul des indicateurs.
- les exigences portent sur la fiabilité et l'ordre dont les méthodes de calcul ne reposent pas sur les mêmes formalismes (arithmétique réelle non linéaire pour la fiabilité et arithmétique entière linéaire pour l'ordre) ce qui complexifie la formalisation. Par exemple pour les méthodes MILP, l'exigence d'ordre se traduit simplement en ensemble de contraintes linéaires, par contre la fiabilité doit être linéarisée pour pouvoir être analysée.

A l'inverse, les méthodes basées contraintes sont complètes et quel que soit le réglage des paramètres de ces méthodes (par exemple pour SMT heuristique de décision, période de redémarrage, etc) celles-ci convergent vers une solution.

3.4 Résumé

Dans ce chapitre, nous avons introduit le problème DSE et le besoin d'automatisation pour résoudre ce problème sur des systèmes réalistes. Nous avons ensuite défini les types de systèmes, d'exigences de sûreté de fonctionnement et d'alternatives considérés dans ce document. Nous avons alors présenté les méthodes existantes de résolution du problème DSE et comparé les avantages et limitations de chacune. Nous en avons déduit que la résolution de problèmes SMT offre un cadre de formalisation et de résolution du problème DSE répondant à nos besoins. Ainsi, nous nous inspirons de cette méthode pour définir une approche de résolution basée SMT et spécialisée dans l'exploration sous contraintes de sûreté de fonctionnement qui constitue une des contributions principales de ce manuscrit. Mais avant cela, nous identifions dans le chapitre suivant, un langage offrant les constructions nécessaires à la formulation d'un problème DSE.

Chapitre 4

Langages de modélisation et outils d'analyse de la sûreté de fonctionnement

Le formalisme de modélisation a un impact sur la représentativité du modèle, l'effort de modélisation à fournir pour le réaliser et la qualité des analyses pouvant être menées. Ainsi, l'objectif du chapitre est d'identifier le langage et le formalisme les mieux adaptés à la formulation du problème d'exploration. Pour cela, nous présentons dans la section 4.1 les besoins de modélisation nécessaires à la formulation d'un problème DSE. Par la suite nous confrontons, dans les sections 4.2 et 4.3, les langages existants aux exigences établies et montrons alors que les limitations identifiées justifient l'introduction d'un nouveau langage spécialement dédié à la résolution du problème DSE.

4.1 Identification des besoins de modélisation

Notre objectif principal est de fournir un ensemble de méthodes de résolution du problème DSE. Pour cela, le langage de modélisation doit permettre de décrire le modèle dysfonctionnel du système, les exigences de sûreté et l'espace des architectures. Ce langage doit avoir une syntaxe et une sémantique clairement établies afin de fonder nos analyses sur un modèle formellement défini.

Paramètres du problème DSE Le langage de modélisation doit pouvoir décrire :

- le modèle de l'architecture initiale ;
- l'ensemble des candidats envisagés lors de l'exploration ;
- les exigences de sûreté à respecter.

Séparation des exigences et du modèle Les paramètres de l'exploration doivent être séparés syntaxiquement du modèle dysfonctionnel afin de réutiliser un même modèle pour différents paramètres d'exploration.

FVa
Internal malfunctions : e : 0.001 (failures per hour) l : 0.001 (failures per hour)
Output failure modes : ERR-out = e LOST-out = l

Listing 4.1 – Annotation HIPHOPS du filtre F_{Va}

4.2 Langage de modélisation et d'analyse des systèmes statiques

Confrontons le langage de modélisation et d'analyse HIPHOPS présenté dans [77] aux besoins établis dans la section précédente.

Modélisation HIPHOPS est un langage de modélisation des comportements dysfonctionnels par annotation de l'architecture du système modélisée par des outils comme Simulink, EAST-ADL ou encore SimulationX. Les annotations ne donnent que des informations sur la génération et la propagation des modes de défaillance au sein des composants du système. En général, seuls les composants atomiques du systèmes sont annotés, puisque les informations sur l'interconnexion des composants au sein d'un système sont contenues dans le modèle de l'architecture. Les annotations d'un composant atomique contiennent :

1. les événements de défaillance (champ *internal malfunctions*) et leur distribution de probabilité ;
2. la logique de génération des modes de défaillance (champ *output failure modes*) décrite par une formule propositionnelle.

Exemple 4.1 (Modélisation d'un composant en HIPHOPS)**.** *Reprenons le capteur F_{Va} de* ROSACE*. Pour que* ROSACE *soit clos, nous considérons par la suite que les filtres reçoivent des signaux corrects, nous ne modélisons donc pas ces entrées. Le filtre est soumis aux événements e et l engendrant la production d'un signal erroné ou perdu (voir exemple 2.5). Le code* HIPHOPS *de ce composant est donné par le listing 4.1.*

Puisque le formalisme sous-jacent à HIPHOPS est la IF-FMEA, la conjonction des règles et des liens entre composants est la fonction de structure du système. L'arbre de défaillance est donc obtenu par simple représentation graphique de cette fonction.

Exemple 4.2 (Calcul de l'arbre de défaillance)**.** *Pour calculer l'arbre de défaillance pour l'événement* la sortie δ_{ec} de ROSACE est erronée*, nous donnons un sous-ensemble des équations modélisant* ROSACE*. Notons que nous utilisons la convention de notation de* HIPHOPS *où fm-C.x signifie que*

le mode de défaillance fm est observé sur l'entrée/sortie x du composant C.

$$\textit{Liens}\begin{cases} ERR-C_{Va}.q \Leftrightarrow ERR-F_q.o \\ ERR-C_{Va}.Vz \Leftrightarrow ERR-F_{Vz}.o \\ ERR-C_{Va}.Va \Leftrightarrow ERR-F_{Va}.o \\ ERR-\delta_{ec} \Leftrightarrow ERR-C_{Va}.o \end{cases} \quad \textit{Composants}\begin{cases} ERR-C_{Va}.o \Leftrightarrow & C_{Va}.e \vee ERR-C_{Va}.Va \\ & \vee(ERR-C_{Va}.q \wedge ERR-C_{Va}.Vz) \\ ERR-F_q \Leftrightarrow & F_q.e \\ ERR-F_{Vz} \Leftrightarrow & F_{Vz}.e \\ ERR-F_{Va} \Leftrightarrow & F_{Va}.e \end{cases}$$

En assemblant les équations, on obtient la fonction de structure φ pour $ERR-\delta_{ec}$.

$$\varphi = C_{Va}.e \vee F_{Va}.e \vee (F_q.e \wedge F_{Vz}.e)$$

Par ailleurs, HipHOPS permet de modéliser le problème DSE. Pour cela, l'utilisateur peut fournir, pour chaque composant du système, un ensemble d'annotations (appelées *implantations*) correspondant aux différents modèles dysfonctionnels des alternatives du composant. De plus, l'utilisateur peut décrire un ensemble de coûts pour chaque implantation.

Outils L'outil d'analyse (aussi intitulé HipHOPS) est présenté comme un plug-in pour l'une des plateformes Simulink, EAST-ADL ou SimulationX. Celui-ci permet d'annoter les composants du système et de calculer les indicateurs de sûreté notamment :

- les coupes minimales à l'aide de l'algorithme MICSUP (cf section 2.1.4) ;
- l'approximation de la disponibilité par la méthode d'Esary-Proschan (cf section 2.1.4).

Par ailleurs, HipHOPS offre la possibilité de résoudre un problème DSE où l'utilisateur spécifie les exigences de sûreté sur l'indisponibilité que doit respecter la solution. L'outil se charge ensuite de fournir un ensemble de solutions Pareto-optimales pour les coûts définis par l'utilisateur et respectant les exigences en utilisant un algorithme génétique.

Confrontations aux besoins HipHOPS répond à la majorité de nos exigences puisque qu'il permet de décrire des systèmes statiques de manière hiérarchique et de formuler le problème DSE (cf section 3.1.1). Néanmoins HipHOPS est un langage d'annotation et doit donc être utilisé avec un outil de modélisation d'architecture. En choisissant ce langage nous serions dépendant des interfaces graphiques d'outils propriétaires, ce que nous souhaitons éviter. De plus, puisqu'il n'existe pas, à notre connaissance, de notion de *classe* de composants, HipHOPS n'est pas modulaire. Cette limitation a été étudiée dans [97] mais n'est, à notre connaissance, pas encore intégrée dans l'outil. De plus, les différentes implantations d'un composant sont directement définies dans le composant. Il est alors nécessaire de modifier les composants du système pour modifier les paramètres du problème DSE.

4.3 Langages de modélisation et d'analyse des systèmes dynamiques

Confrontons les langages de modélisation et d'analyse des systèmes dynamiques les plus connus aux besoins établis précédemment. Nous concluons cette section avec un tableau récapitulant les besoins remplis par les langages présentés.

4.3.1 SMV

Le langage SMV, développé par l'institut FBK [23], permet de définir des automates finis et l'extension FEI du langage permet d'injecter de fautes afin de modéliser le comportement dysfonctionnel. Ces modèles sont analysables avec l'outil XSAP [15] lui-même basé sur nuXMV [22] (basé sur SMT) ou l'outil nuSMV (basé sur les BDDs).

Modélisation Dans l'approche [15], la modélisation repose sur :

1. la modélisation d'un comportement nominal du composant en SMV
2. la modélisation des effets des défaillances par un langage d'injection de fautes appelé FEI. Il existe plusieurs classes de comportements prédéfinis (panne permanente, temporaire, etc) sur lesquelles l'utilisateur peut s'appuyer.

La modélisation des systèmes basée sur SMV est proche de celle décrite dans la section 2.3.1 :

- un composant atomique est un automate où :
  - les entrées sont les entrées du composant, les événements fonctionnels et les événements de défaillance (champ *IVAR*) ;
  - les sorties sont les sorties du composant (champ *DEFINE*) ;
  - l'état du composant est défini par un ensemble de variables d'états (champ *VAR*) ;
  - les transitions sont définies soit en extension en donnant la relation de transition (champ *ASSIGN*) ou bien en intention en donnant un invariant représentant l'ensemble des transitions légales de l'automate (champ *INVAR*).
- un système ou un composant non-atomique est un automate où :
  - les sous-composants sont décrits comme des variables d'états (champ *VAR*) ;
  - les connexions entre composants sont faites par passage d'argument c'est-à-dire que le sous-composant est défini comme un automate où les entrées sont contraintes par les sorties d'autres automates.

Exemple 4.3 (Modélisation d'un composant en SMV)**.** *Reprenons les filtres de* Rosace*. En* SMV*, leurs entrées sont les événements de défaillance e et l. Rappelons que le modèle donné dans le listing 4.2 ne décrit que le comportement nominal du composant.*

```
1  MODULE Filter (in)
2
3   // declaration des evenements de
      defaillance
4   IVAR
5     e : boolean;
6     l : boolean;
7
8   // declaration de la variable d'etat
9   VAR
10    s : {ERR, LOST, OK};
11
12  // definition de la relation de
      transition nominale
13  ASSIGN
14    init(s) := OK;
15    next(s) := s
16
17  // definition de la sortie
18  DEFINE
19    o := (s = OK) ? in : s;
```

Listing 4.2 – Code SMV des filtres de Rosace

Exemple 4.4 (Modélisation des défaillances en FEI)**.** *L'effet des événements de défaillance e et l sur les variables d'états d'un capteur est décrit dans le modèle du listing 4.3. Dans notre cas si*

e (respectivement l) survient alors l'état s reste bloqué dans le mode ERR (respectivement LOST) et produit donc des données erronées (respectivement perdues).

```
FAULT EXTENSION System

  // defaillances de Filter
  EXTENSION OF MODULE Filter

    // defaillances de Filter impactant s
    SLICE Filter_faults AFFECTS s WITH

      // premier mode: si e survient alors s reste bloque a ERR
      MODE e_hasOccurred : Permanent StuckAtByValue_D(
        data term << ERR,
        data input << s,
        data varout >> s,
        event failure >> e);

      // deuxieme mode: si l survient alors s reste bloque a LOST
      MODE l_hasOccurred : Permanent StuckAtByValue_D(
        data term << LOST,
        data input << s,
        data varout >> s,
        event failure >> l);
```

Listing 4.3 – Effet des événements e et l sur un filtre

Exemple 4.5 (Modélisation de ROSACE en SMV)**.** *Le modèle de* ROSACE*, présenté dans le listing 4.4, est obtenu en instanciant les filtres et les contrôleurs comme des variables d'états de l'automate en connectant les sorties des filtres aux entrées des contrôleurs. Précisons que pour étudier* ROSACE*, il est nécessaire de faire des hypothèses sur ses entrées, lesquelles sont supposées correctes.*

```
MODULE Rosace

  //declaration des composants
  VAR
    FVa: Filter(OK);
    FVz: Filter(OK);
    Fq : Filter(OK);
    Faz : Filter(OK);
    Fh : Filter(OK);
    CVa : CVA(FVz.o,Fq.o,FVa.o);
    CVz : CVZ(Fh.o,Faz.o,Fq.o,FVz.o);

  //definition des sorties
  DEFINE
    deltaEC:= CVa.o;
    deltaTHC:= CVz.o;
```

Listing 4.4 – Code SMV de ROSACE

Outils L'outil XSAP [15] permet de calculer les séquences minimales du système pour un certain événement redouté. Cette analyse est fondée sur une exploration symbolique de l'espace d'état introduite par [19] et basée sur le *model checker* NUXMV.

Confrontations aux besoins Le langage SMV de base n'offre pas la possibilité de représenter des composants alternatifs. Néanmoins les auteurs de [43] proposent d'utiliser des langages de description d'architectures comme OCRA [26] pour représenter l'espace des architectures. La structure du système est alors définie sans donner les implantations des composants. Néanmoins, ce langage ne permet pas d'assurer simplement que les modèles sont statiques.

4.3.2 Altarica

Altarica est un langage de modélisation basé sur les automates de mode présentés dans la section 2.3. Ce langage introduit dans les années 90 par [7] est très largement utilisé [20] pour la modélisation et l'analyse de la sûreté de fonctionnement.

Modélisation La modélisation avec Altarica est proche de celle décrite en section 2.3.1 :

- un composant atomique est appelé *nœud*, celui-ci contient :
  1. des entrées et des sorties (champ *flows*) dont les valeurs peuvent représenter les modes de défaillance observés, des entiers ou encore des booléens de contrôle ;
  2. des événements (champ *event*) modélisant à la fois les événements de défaillance mais aussi des événements fonctionnels ;
  3. des variables d'états (champ *states*) dont la valeur initiale est donnée par le champ *init* ;
  4. les transitions de l'automate (champ *trans*) décrites comme un ensemble de triplets (garde, événement, action) où la garde est un prédicat sur les entrées et variables d'états du composant et l'action donne l'état d'arrivée (la transition est franchie si la garde est vraie et que l'événement survient) ;
  5. les fonctions de sorties encodées par des *assertions* (champ *assert*), c'est-à-dire par un ensemble de contraintes sur les valeurs des sorties en fonctions de l'état courant et des entrées.
- un composant non atomique est aussi un nœud (parfois appelé *équipement*) et contient en plus des entrées/sorties :
  1. un ensemble de sous-composants (champ *sub*) ;
  2. un ensemble d'assertions contraignant les entrées et sorties des sous-composants pour encoder la notion de connexion.

Exemple 4.6 (Modélisation d'un composant en Altarica)**.** *Modélisons les filtres de* Rosace *soumis aux événements e et l engendrant la production d'un signal erroné ou perdu. Le code* Altarica *correspondant est donné par le listing 4.5.*

Les connexions sont ensuite réalisées en *branchant* les automates entre eux. Ceci est généralement fait à partir d'une interface graphique où les connexions sont de simples liens entre les entrées et sorties des composants.

Exemple 4.7 (Modélisation d'un système en Altarica)**.** *Le système* Rosace *est obtenu par l'instanciation et la connexion de ses composants comme montré par le listing 4.6.*

Pour spécifier l'événement redouté, il est nécessaire de décrire un observateur signalant l'occurrence de l'événement redouté en fonction des modes de défaillance observés sur les composants.

Outils L'outil Cecilia-OCAS [29] développé par Dassault permet de calculer les séquences d'occurrences d'événements déclenchant l'événement redouté. Ces séquences contiennent à la fois des événements de défaillance mais aussi des événements fonctionnels. Pour cela, l'outil procède à une simulation exhaustive des séquences de taille bornée k menant à un événement redouté. D'autres techniques, comme celles présentées dans [45], enrichissent le modèle initial avec des observateurs

```
node Filter

  //declaration des entrees et sorties
  flows
    o: {ERR,LOST,OK}: out;
    i:{ERR,LOST,OK}: in;

  //declaration de la variable d'etat
  state
    s: {ERR,LOST,OK};

  //declaration des evenements de
    defaillance
  event
    e,l;

  //etat initial de l'automate
  init
    s := OK;

  //transitions de l'automate
  trans
    s = OK |- e -> s := ERR;
    s = OK |- l -> s := LOST;

  //fonction de sortie
  assert

    o = case{
          s = ERR : ERR,
          s = LOST : LOST,
          else i
        };
end
```

Listing 4.5 – Code Altarica des filtres de Rosace

```
node Rosace

  //declaration des sorties
  flows
    deltaEC: {ERR,LOST,OK}: out;
    deltaTHC: {ERR,LOST,OK}: out;

  //declaration des composants
  sub
    FVa: Filter;
    FVz: Filter;
    Fq : Filter;
    Faz : Filter;
    Fh : Filter;
    CVa : CVA;
    CVz : CVZ;


  //connexion des composants
  assert
    FVa.i = OK;
    FVz.i = OK;
    Fq.i = OK;
    Faz.i = OK;
    Fh.i = OK;
    CVa.Vaf = FVa.o;
    CVa.Vzf = FVz.o;
    CVa.qf = Fq.o;
    deltaEC= CVa.o;
    CVz.Vaf = FVa.o;
    CVz.Vzf = FVz.o;
    CVz.hf = Fh.o;
    CVz.azf = Faz.o;
    deltaTHC= CVa.o;

end
```

Listing 4.6 – Code Altarica de Rosace

afin d'encoder l'occurrence d'événements de défaillance et procèdent ensuite à une analyse d'atteignabilité des états dangereux de l'automate. L'outil calcule alors les séquences à partir de la valeur des observateurs.

Par ailleurs, des papiers comme [85] proposent une méthode alternative consistant à compiler les automates de mode en arbres de défaillance, puis d'utiliser les techniques d'analyse des arbres (cf section 2.1). Cette manipulation permet d'obtenir rapidement les coupes minimales de l'approximation statique du système au lieu d'effectuer la simulation exhaustive. Bien entendu, cette compilation détruit l'information sur l'ordre d'occurrence des événements mais donne tout de même une sur-approximation [85] des séquences d'événements menant à la défaillance du système. De plus, la méthode de [85] offre la possibilité de modéliser des systèmes statiques en Altarica puis d'utiliser les méthodes d'analyse classiques des système statiques.

Confrontations aux besoins ALTARICA est modulaire et hiérarchique mais n'assure pas modéliser des systèmes statiques. Par conséquent, du point de vue de l'outil, il est nécessaire de vérifier que le système modélisé est en réalité statique. De plus, comme l'événement redouté est un observateur du modèle, il n'y a pas de séparation claire entre paramètres d'analyse et modèle. Finalement, il n'y a pas en ALTARICA de notion de composants alternatifs, il est alors impossible de représenter l'ensemble des candidats considérés par l'exploration.

4.3.3 Récapitulatif

Les propriétés des différents langages analysés dans cette section sont résumées dans la table 4.1. Au final, aucun des langages étudiés ne répond complètement à nos attentes. Néanmoins HIPHOPS offre un bon cadre de modélisation du problème DSE et ALTARICA offre une description modulaire et hiérarchique. Par conséquent, nous proposons de définir un nouveau langage nommé KCR, à mi-chemin entre ALTARICA et HIPHOPS.

Notons que nous n'avons pas décrit en détails les langages de modélisation comme GRIF ou les formats d'échange d'arbres de défaillance comme OPENPSA [90] car ceux-ci sont basés sur les formalismes classiques peu adaptés à la modélisation de systèmes complexes (cf section 2.3).

Langage	Besoins				
	Statique	Hiérarchique	Modulaire	Candidats représentables	Séparation paramètres/modèle
HIPHOPS	✓	✓	✗	✓	✗
ALTARICA	✗	✓	✓	✗	✗
SMV	✗	✓	✓	✗	✓

TABLE 4.1 – Récapitulatif de la confrontation des langages

4.4 Résumé

Nous avons identifié, dans ce chapitre, les besoins de modélisation pour pouvoir formuler le problème DSE. Nous avons alors présenté les langages et outils de modélisation et d'analyse de la sûreté de fonctionnement les plus utilisés. En confrontant ces langages aux besoins, nous avons constaté qu'ils ne répondent pas complètement à nos attentes. Ce constat motive l'introduction d'un nouveau langage de modélisation appelé KCR, inspiré des langages HIPHOPS et ALTARICA, permettant d'analyser les systèmes et de résoudre le problème DSE à l'aide de solveurs SMT. Pour cela nous introduisons dans le prochain chapitre, les concepts élémentaires de la formulation et la résolution de problème SMT.

Chapitre 5

Introduction à la Satisfiabilité Modulo Théorie

Ce chapitre introduit les notions élémentaires de la formulation et la résolution de problèmes de Satisfiabilité Modulo Théorie (SMT), cadre dans lequel nous allons formaliser les analyses de la sûreté de fonctionnement des systèmes et le problème d'exploration. Pour cela, nous donnons, dans la section 5.1, un aperçu informel des différents concepts de la Satisfiabilité Modulo Théorie. Puis nous définissons formellement ces concepts, dans la section 5.2, en donnant la syntaxe et la sémantique de la logique du premier ordre multi-sortée utilisée pour formuler les problèmes SMT. Nous présentons ensuite, dans la section 5.3, les problèmes SAT, SMT et le concept de théorie. Puis nous exposons, dans la section 5.4, les procédures de résolution classiques de ces problèmes. Finalement nous présentons dans la section 5.5, le langage standard SMT-LIB utilisé par la suite pour la spécification de problèmes SMT.

5.1 Concepts élémentaires de la Satisfiabilité Modulo Théorie

L'idée principale de la formulation d'un problème de Satisfiabilité Modulo Théorie (SMT) est d'exprimer un problème comme un ensemble de contraintes décrites dans la logique du premier ordre. Présentons le problème classique de Satisfiabilité de la logique propositionnelle (problème SAT). Puis montrons comment celui-ci est étendu au problème SMT en fournissant un aperçu des différentes notions liées à la formulation, l'interprétation et la résolution de tels problèmes.

5.1.1 Le problème SAT

Un problème est décrit par la logique propositionnelle, c'est-à-dire comme un ensemble de contraintes sur des variables booléennes, nous utilisons donc les notions de :

- conjonction $\wedge$, disjonction $\vee$ et négation $\neg$ classiques sur $\mathbb{B}$;
- propositions (généralement notées par des lettres $a, b, \cdots$) vues comme des éléments de $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$
- constantes vrai $\mathbf{T}$ et faux $\mathbf{F}$.

Par exemple une énoncé de la logique propositionnelle est $(a \wedge b) \vee (\neg a \wedge c)$.

Le problème SAT est alors de savoir s'il existe une interprétation des propositions telle que l'énoncé soit vrai. Si oui alors l'énoncé est dit *satisfiable*. Par exemple, $(a \vee b) \wedge (\neg a \vee c)$ est satisfiable car pour $\{a \mapsto \mathbf{T}, b \mapsto \mathbf{F}, c \mapsto \mathbf{T}\}$ l'énoncé est vrai. Le but est alors de trouver une *procédure de décision*, c'est-à-dire un algorithme capable de déterminer, en un temps fini, si un énoncé est satisfiable ou non. Ce problème a fait l'objet de nombreuses recherches qui ont abouti aux procédures de décision DPLL et CDCL (présentées en section 5.4). Ces procédures sont à la base des outils appelés *solveurs* SAT, permettant de résoudre le problème SAT. Néanmoins, ce type de problème possède une expressivité limitée car ne résout que des problèmes sur des énoncés de la logique propositionnelle.

5.1.2 Formuler un problème SMT

Dans le cadre SMT les contraintes sont décrites dans la logique du premier ordre multi-sortée, c'est-à-dire que les éléments de la logique appartiennent à des types appelés *sortes*. Pour pouvoir formaliser un problème, il est nécessaire de définir les symboles utilisables pour l'exprimer. Ces symboles sont donnés par une *signature multi-sortée* (cf section 5.2). Intuitivement, celle-ci regroupe les sortes, les opérateurs et les constantes (qui sont des fonctions sans argument). De plus, la signature contient une relation associant, à chaque symbole de fonction f, son domaine et son codomaine (appelé *rang* de f).

Exemple 5.1 (Signature)**.** *Par exemple la signature des problèmes d'arithmétique réelle linéaire contient :*

- *les sortes des booléens Bool et des réels Real ;*
- *les constantes décimales comme* 1.345*, vrai* $\top$ *et faux* $\bot$*, notons que les constantes sont vues comme des fonctions sans argument dites interprétées c'est-à-dire* 1.345 *est la fonction renvoyant toujours l'élément* 1.345 *de sorte Real*
- *l'opérateur* $+$*, et les relations* $>, \geq, =, \leq, <$
- *les rangs classiquement associés à ces opérateurs, par exemple* $+$ *prend deux éléments de sorte Real (domaine) et renvoie un élément de Real (codomaine).*

Il faut alors définir les énoncés bien formés, ce qui revient à donner les règles syntaxiques du langage de la MSFOL. Par exemple, $(1 \times (+2))$ n'est pas bien formé car un seul réel est donné à l'opérateur $+$ au lieu des deux réels prévus par la signature, par conséquent cet énoncé n'est pas syntaxiquement correct. Un autre exemple comme $3 \geq \top$ n'est pas bien formé car l'opérateur $\geq$ reçoit un réel et un booléen au lieu des deux réels attendus, cet énoncé n'est pas correctement typé. Par contre $1 + 3$ est syntaxiquement correct et correctement typé, $1 + 3$ est alors appelé *terme*. Un terme de sorte *Bool* est appelé une *formule* et une fonction dont le codomaine est *Bool* est appelé un *prédicat*. Néanmoins les énoncés bien formés comme $1 + 3$ n'ont pas encore de sens car la sémantique des symboles comme $1, 3, +$ n'est pas établie. Présentons les notions associées à la sémantique des termes.

5.1.3 Interpréter un problème SMT

Afin de donner un sens aux symboles de la signature, il est nécessaire d'associer à chacun d'eux une définition mathématique. Par exemple le symbole $+$ est décrit par l'addition classique sur les réels. Cette étape peut sembler triviale mais il n'est pas rare qu'un symbole ait des sens très différents en fonction de l'environnement dans lequel il est défini. Par exemple $+$ peut être l'addition sur les réels ou alors sur les entiers. Ainsi, il existe des symboles :

interprétés c'est-à-dire dont la sémantique est définie, ce sont souvent les sortes, fonctions et constantes classiques comme $+$, 1 et *Real* dont le sens est bien défini,

non-interprétées c'est-à-dire dont la sémantique n'est pas imposée, ce sont les inconnues du problèmes comme x, y ou encore une fonction inconnue $g : Real^2 \to Real$.

La sémantique d'une signature est définie par une *structure* (cf section 5.2). Celle-ci associe un ensemble à chaque sorte et associe une fonction au sens mathématique à chaque symbole de fonction.

Exemple 5.2 (Structure)**.** *Reprenons les symboles définis pour l'arithmétique réelle linéaire, la sorte Real est toujours interprétée comme* $\mathbb{R}$*, les booléens comme* $\mathbb{B}$*, et les symboles* $+, >, <, \cdots$ *sont l'addition et les relations classiques définies sur* $\mathbb{R}$*. Par contre, pour une formule* $x+1 = 3$ *contenant une constante non-interprétée* x*, il existe autant de structures que d'interprétations possibles pour* x*. Par exemple, dans la structure où* $x \mapsto 1$ *la formule est fausse, alors que dans la structure où* $x \mapsto 2$ *la formule est vraie.*

Le problème de satisfiabilité consiste alors à décider s'il existe une structure (appelée *modèle*) telle que la formule soit vraie (cf section 5.3).

En général, il n'est pas intéressant d'obtenir un modèle donnant une interprétation totalement arbitraire aux symboles de la formule. En effet, il est plus intéressant de fournir un modèle où l'interprétation des symboles est cohérente. Par exemple, on souhaite que le modèle de la formule $x + 1 = 3$ interprète $+$ et $=$ comme l'addition et l'égalité classique sur les réels.

Autrement dit, pour une signature donnée, il existe des classes (au sens de la théorie des ensembles) de structures où l'interprétation des symboles interprétés est toujours la même. Pour une classe donnée, il existe des formules appelées *théorèmes* qui sont vraies quelle que soit la structure choisie, l'ensemble de ces formules forment alors une *théorie*. Dans ce document, nous nous intéressons aux théories tel qu'il existe un algorithme capable de déterminer, avec certitude et en temps fini, la satisifiabilité d'une conjonction de termes de la théorie. On dit alors qu'une théorie est décidable et cet algorithme est le solveur associé à la théorie.

Exemple 5.3 (Théorie *Core*)**.** *La théorie Core est la théorie de la logique propositionnelle, qui sert de base à l'ensemble des énoncés de la* MSFOL*. La signature associée contient les opérateurs* $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$*, les constantes* $\top$, $\bot$ *et la sorte Bool. Les structures de Core interprètent Bool comme* $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$*,* $\top$ *comme* $\mathbf{T}$*,* $\bot$ *comme* $\mathbf{F}$*,* $\wedge$ *comme la conjonction,* $\vee$ *la disjonction,* $\neg$ *la négation, l'implication* $\Rightarrow$ *et* $\Leftrightarrow$ *l'équivalence classique définie sur* $\mathbb{B}$*. La théorie Core est alors l'ensemble des tautologies, comme* $a \vee \neg a$ *qui est vraie quelle que soit la structure choisie.*

Exemple 5.4 (Théorie *EQ*)**.** *La théorie de l'égalité est aussi une base des énoncés de la* MSFOL*. Elle ne fait que définir l'opérateur* $=$ *prenant deux éléments d'une même sorte* S *et retournant un élément de Bool. Les structures associées à cette théorie sont celles où l'opérateur* $=$ *est transitif, réflexif et symétrique. Cette théorie est décidable par la construction de classes d'équivalence* union-find*.*

Exemple 5.5 (Théorie *UF*)**.** *La théorie des fonctions non interprétées (UF en anglais) ne définit pas de nouveaux opérateurs mais impose que l'interprétation de l'opérateur* $=$ *respecte la règle de congruence :* $\forall x, y, x = y \Rightarrow f(x) = f(y)$*. Cette théorie est décidable par l'algorithme de* congruence closure *[67].*

Exemple 5.6 (Théorie *LRA*)**.** *Dans la théorie de l'arithmétique réelle linéaire (LRA en anglais)*

– *la sorte Real est interprétée comme* $\mathbb{R}$
– *les constantes décimales sont interprétées comme des éléments de* $\mathbb{R}$
– *les opérations* $+, <, \leq, >, \geq$ *sont interprétées comme les opérateurs classiques sur les réels.*

La théorie est alors l'ensemble des théorèmes de l'arithmétique réelle linaire, par exemple nous avons vu que $x + 3 = 1$ *peut être faux pour certaines structures (c'est-à-dire certaines valeurs de* x*). Par contre la formule* $(x > 1 \wedge x + y < 1) \Rightarrow y < 0$ *est vraie quelle que soit l'interprétation de* x *et* y *donc cette formule fait partie de la théorie LRA. Cette théorie est décidable avec l'algorithme du simplexe [25].*

Le but est alors de combiner ces solveurs de théories pour former une procédure de décision pour une combinaison de théories appelée *logique*. Afin d'assurer qu'une logique est décidable il est souvent nécessaire de contraindre syntaxiquement les formules. Par exemple, dans la logique des fonctions non-interprétées et de l'arithmétique réelle linéaire, les formules ne possèdent pas de quantificateurs $\forall, \exists$.

On peut alors étendre le problème de satisifiabilié au problème de Satisifiabilité Modulo Théorie (SMT), c'est-à-dire trouver un modèle d'une formule appartenant à une certaine logique.

5.1.4 Résoudre un problème SMT

Le processus de résolution d'un problème SMT peut être vu comme la communication entre un solveur SAT et l'ensemble des solveurs des théories de la logique.

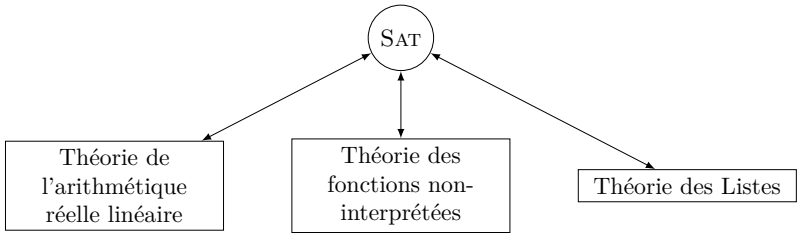


FIGURE 5.1 – Illustration de l'architecture d'un solveur SMT

Illustrons le principe de résolution (cf section 5.4) sur la formule :

$$(x \geq 0) \wedge (y = x + 1) \wedge ((y > 2) \vee (y < 1))$$

Dans un premier temps, les termes dont l'interprétation est associée à une théorie sont identifiés et remplacés par des booléens.

$$\underbrace{(x \geq 0)}_{p_1} \wedge \underbrace{(y = x + 1)}_{p_2} \wedge (\underbrace{(y > 2)}_{p_3} \vee \underbrace{(y < 1)}_{p_4}) \overset{\text{structure booléenne}}{\Longrightarrow} p_1 \wedge p_2 \wedge (p_3 \vee p_4)$$

Le problème est alors résolu par un solveur SAT qui fournit un modèle pour les booléens. Imaginons que le modèle est $\{p_1 \mapsto \mathbf{T}, p_2 \mapsto \mathbf{T}, p_4 \mapsto \mathbf{T}\}$. Il faut alors vérifier qu'il existe un couple (x,y) tel que ce modèle est vrai du point de vue de la théorie, ce qui n'est pas le cas ici, en effet

$$(x \geq 0) \wedge (y = x + 1) \wedge (y < 1) \overset{Simplexe}{\Rightarrow} \text{unsat} \overset{\text{clause de conflit}}{\Longrightarrow} \neg p_1 \vee \neg p_2 \vee \neg p_4$$

L'explication de l'incohérence, ici $\neg p_1 \vee \neg p_2 \vee \neg p_4$, est appelée clause de conflit. Cette nouvelle information est prise en compte par le solveur SAT qui est alors relancé sur la conjonction de la formule et de la clause (cette nouvelle formule est équivalente à la formule initiale).

$$p_1 \wedge p_2 \wedge (p_3 \vee p_4) \wedge (\neg p_1 \vee \neg p_2 \vee \neg p_4)$$

Le nouveau modèle est maintenant $\{p_1 \mapsto \mathbf{T}, p_2 \mapsto \mathbf{T}, p_3 \mapsto \mathbf{T}, p_4 \mapsto \mathbf{F}\}$. On vérifie de nouveau la cohérence et on obtient

$$x \geq 0 \wedge y = x + 1 \wedge y > 2 \overset{Simplexe}{\Rightarrow} \text{sat} \overset{\text{modèle}}{\Longrightarrow} \{x \mapsto 2, y \mapsto 3\}$$

Puisque le modèle booléen est cohérent du point de vue de la théorie, le solveur de théorie fournit le modèle du problème. Après cette présentation intuitive, présentons formellement l'interprétation et la résolution des problèmes SMT.

5.2 Logique du premier ordre multi-sortée (MSFOL)

Les problèmes SMT sont formulés dans la logique du premier ordre multi-sortée. Présentons dans un premier temps la syntaxe et la sémantique de la MSFOL. Les notations et définitions de ce chapitre sont inspirées des travaux de [24] et [10].

5.2.1 Syntaxe de la MSFOL

Afin de définir la syntaxe de la MSFOL, il convient d'introduire les différents symboles admis par la logique. Ces symboles sont définis par la *signature multi-sortée* de la logique, formellement définie par la définition 5.1 issue de [24].

Définition 5.1 (Signature)**.** *Une signature mutli-sortée Σ est un n-uplet $(Sort, Var, Fun, Rank)$ où*
- *$Sort$ est l'ensemble des sortes de la logique contenant au moins $Bool$;*
- *Var est un ensemble dénombrable de symboles de variables ;*
- *Fun est un ensemble de symboles de fonctions tel que $Fun \cap Var = \emptyset$ et contenant au moins $\vee, \wedge, \neg, \top, \bot$ et $=$;*
- *$Rank \subseteq Fun \times Sort^+$ est une relation totale à gauche, la liste de sortes $Sort^+$ associée à f par $Rank$ où $f \in Fun$ est appelée rang de f.*

Par soucis de lisibilité, nous introduisons les notations et définitions suivantes :
- $f(\sigma_1, \cdots, \sigma_n) : \sigma$ signifie que $(f, \sigma_1, \cdots, \sigma_n, \sigma) \in Rank$;
- si $(f, \sigma) \in Rank$, alors f est une *constante* notée $f : \sigma$. Les constantes d'une signature sont rassemblées dans l'ensemble $Const$;
- si $f(\sigma_1, \cdots, \sigma_n) : Bool$ on dit alors que f est un prédicat.

Notons que $Rank$ est une relation, par conséquent, un symbole f peut être en relation avec plusieurs rangs (ce qui est typiquement le cas de $=$). Si un symbole possède plusieurs rangs où seule la sorte de retour σ diffère, alors nous désambiguïserons avec la notation f^σ.

Exemple 5.7 (Signature)**.** *Illustrons ce concept sur la signature de la logique propositionnelle enrichie de l'égalité appelée $Core = (\{Bool\}, \emptyset, Fun_{Core} = \{\wedge, \vee, \neg, =, \Rightarrow, \Leftrightarrow, \top, \bot\}, Rank_{Core})$ où*

$Rank_{Core}$ est définie comme l'ensemble

$$\left\{\begin{array}{lcl} \vee(Bool, Bool) & : & Bool, \\ \wedge(Bool, Bool) & : & Bool, \\ \neg(Bool) & : & Bool, \\ \Leftrightarrow(Bool, Bool) & : & Bool, \end{array}\quad\begin{array}{lcl} =(S,S) & : & Bool \text{ pour toute sorte } S \in Sort, \\ \top & : & Bool, \\ \bot & : & Bool \end{array}\right\}$$

Définissons maintenant la signature de l'arithmétique réelle linéaire $LRA = (\{Bool, Real\}, \emptyset, Fun_{Core} \cup \{\geq, \leq, >, <, + \text{ et les décimaux}\}, Rank_{LRA})$ *où* $Rank_{LRA}$ *contient* $Rank_{Core}$*, les décimaux comme* 1.345 *sont de rang Real et*

$$\left\{\begin{array}{lcl} +(Real, Real) & : & Real, \\ >(Real, Real) & : & Bool, \\ <(Real, Real) & : & Bool, \end{array}\quad\begin{array}{lcl} \geq(Real, Real) & : & Bool, \\ \leq(Real, Real) & : & Bool \end{array}\right\}$$

Toute signature Σ doit contenir au moins la signature *Core*. On dit alors qu'une signature $\Sigma = (Sort, Var, Fun, Rank)$ est *légale* si et seulement si $Bool \in Sort$, $Fun_{Core} \subset Fun$, $Rank_{Core} \subset Rank$ et $\forall \sigma \in Sort,\ =(\sigma, \sigma) : Bool$. Par la suite nous omettrons le terme *légal* et nous dirons *signature* pour *signature légale*.

Les signatures de base ne contiennent que les symboles couramment utilisés pour décrire des problèmes de la MSFOL. Ainsi, il est possible d'étendre une signature Σ en introduisant un ensemble S de nouveaux symboles de fonctions, de variables et de sortes n'interférant pas avec les symboles initiaux de Σ, on parle alors *d'extension de signature* notée $\Sigma(S)$.

Définition 5.2 (Syntaxe)**.** *Comme montré dans [24], on peut définir la syntaxe des termes T associés à une signature $\Sigma = (Sort, Var, Fun, Rank)$ par la forme Backus-Naur étendue (EBNF [64]) suivante :*

$$T ::= v \mid f\ T^* \mid f^\sigma\ T^* \mid \exists (x : \sigma)^+\ T \mid \forall (x : \sigma)^+\ T \mid let\ (x = T)^+\ in\ T \mid (T)$$

où $v \in Var$, $f \in Fun$, $\sigma \in Sort$, $f\ T^*$ et $f^\sigma\ T^*$ sont des applications *de f, let $(x = T)^+$ in T est appelé* let binding *et $\forall, \exists$ sont appelés* quantificateurs*.*

Notons que la construction *let* $(x = T)^+$ *in* T est une commodité syntaxique. En effet, toute construction *let* $(x_1\ t_1) \cdots (x_n\ t_n)$ *in* T peut être supprimée en remplaçant les occurrences des x_i dans t par le terme t_i correspondant. Nous considérons donc par la suite que les termes ne contiennent pas de *let* $(x = T)^+$ *in* T. De plus, la syntaxe classique de l'application de fonction en MSFOL est dite *préfixe* (c'est-à-dire que l'opérateur est donné au début de l'application), ce qui facilite la description des règles syntaxiques. Or comme cette notation est difficilement lisible, nous écrirons les termes des exemples dans la notation infixe habituelle.

Exemple 5.8 (Syntaxe)**.** *Considérons que la signature LRA définie précédemment est étendue avec les symboles $f(Bool, Bool) : Real$, $b : Bool, x : Real$ et la variable $v : Bool$. Alors le terme $\forall v : Bool, f(v\ b) \geq x$ est syntaxiquement correct. Par contre, le terme $x(b)$ n'est pas correct car x est une constante, non une fonction prenant un booléen en entrée.*

Comme les logiques multi-sortées possèdent une notion de type, on peut définir inductivement les termes correctement sortés d'une signature $\Sigma = (Sort, Var, Fun, Rank)$. Or dans les termes de la forme $Q(x : \sigma)\ t$ où $Q \in \{\forall, \exists\}$, le quantificateur Q introduit une nouvelle variable x de sorte σ qui,

du point de vue de t, doit être *libre* c'est-à-dire non liée à un autre quantificateur. Ainsi, pour établir la sorte d'un terme, il est nécessaire d'introduire un environnement de typage donnant la sorte de chaque variable libre du terme considéré, cet environnement est noté $\Gamma = \{v_1 : \sigma_1, \cdots, v_n : \sigma_n\}$. Le typage d'un terme est alors déduit récursivement sur la structure du terme :

- une variable v est un terme de sorte σ si et seulement si $v : \sigma \in \Gamma$
- une constante $c \in \mathit{Const}$ est un terme de sorte σ si et seulement si $c : \sigma$
- une application $f(t1, \cdots, t_n)$ est un terme de sorte σ si $f \in \mathit{Fun}$, $t_1 : \sigma_1, \cdots, t_n : \sigma_n$ et $f(\sigma_1, \cdots, \sigma_n) : \sigma$
- un terme $Q(x_1 : \sigma_1, \cdots, x_n : \sigma_n)\ t$ est un terme de sorte *Bool* si $Q \in \{\forall, \exists\}$ et t est un terme de sorte *Bool* dans l'environnement $\Gamma \cup \{x_1 : \sigma_1, \cdots, x_n : \sigma_n\}$.

Par la suite, nous parlons d'un terme de sorte S pour un terme correctement typé de sorte S. Reprenons les définitions introduites dans [10] utilisées dans ce document :

une formule est un terme de sorte *Bool* ;

un atome est une formule ne contenant ni $\wedge$ (conjonction), ni $\vee$ (disjonction), ni $\neg$ (négation) ;

un littéral est un atome a ou $\neg a$;

un cube est une conjonction de littéraux ;

une clause est une disjonction de littéraux ;

une Forme Normale Disjonctive (DNF) est une disjonction de cubes ;

une Forme Normale Conjonctive (CNF) est une conjonction de clauses ;

une Forme Normale Négative (NNF) est une formule où l'opérateur $\neg$ n'est appliqué que sur des atomes.

5.2.2 Sémantique de la MSFOL

Pour donner un sens aux termes syntaxiquement corrects, il est nécessaire de donner la *sémantique* des sortes et fonctions constituant le terme. Pour cela, une structure M est associée à la signature Σ considérée. Cette structure associe à chaque sorte de Σ un ensemble et associe à chaque symbole de fonction de Σ une fonction au sens mathématique du terme. Cette notion de structure est introduite dans la définition 5.3.

Définition 5.3 (Structure)**.** *Soit une signature Σ, une structure M est un couple $(\mathcal{C}, \mathcal{I})$ où*

- *$\mathcal{C}$ est une fonction associant à chaque $s \in \mathit{Sort}$ un ensemble porteur*
- *$\mathcal{I}$ est la fonction d'interprétation associant à chaque $f \in \mathit{Fun}$ de rang $f(\sigma_1, \cdots, \sigma_n) : \sigma$ une fonction totale $\mathcal{C}(\sigma_1) \times \cdots \times \mathcal{C}(\sigma_n) \to \mathcal{C}(\sigma)$. Si $\mathcal{I}$ est une fonction partielle sur Fun alors la structure est dite* partielle.

Exemple 5.9 (Structure)**.** *Donnons la structure $M_{Core} = (\mathcal{C}_{Core}, \mathcal{I}_{Core})$ associée classiquement à la signature Core. L'ensemble porteur associé à la sorte Bool est l'ensemble des booléens $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$*

donc $\mathcal{C}_{Bool}(Bool) = \mathbb{B}$. *La fonction d'interprétation* $\mathcal{I}_{Core}$ *est alors définie pour chaque* $f \in Fun_{Core}$ *:*

$$
\begin{array}{lcl}
\mathcal{I}_{Core}(\top) & \triangleq & \mathbf{T} \\
\mathcal{I}_{Core}(\bot) & \triangleq & \mathbf{F} \\
\mathcal{I}_{Core}(\neg)(p) & \triangleq & \textit{si } p \textit{ est } \mathbf{T} \textit{ alors } \mathbf{F} \textit{ sinon } \mathbf{T} \\
\mathcal{I}_{Core}(\wedge)(p,q) & \triangleq & \textit{si } p \textit{ et } q \textit{ sont } \mathbf{T} \textit{ alors } \mathbf{T} \textit{ sinon } \mathbf{F} \\
\mathcal{I}_{Core}(\vee)(p,q) & \triangleq & \textit{si } p \textit{ ou } q \textit{ sont } \mathbf{T} \textit{ alors } \mathbf{T} \textit{ sinon } \mathbf{F} \\
\mathcal{I}_{Core}(\Leftrightarrow)(p,q) & \triangleq & \textit{si } p \textit{ et } q \textit{ sont } \mathbf{T} \textit{ ou si } p \textit{ et } q \textit{ sont } \mathbf{F} \textit{ alors } \mathbf{T} \textit{ sinon } \mathbf{F} \\
\mathcal{I}_{Core}(=)(p,q) & \triangleq & \textit{si } p \textit{ et } q \textit{ correspondent au même élément d'un ensemble alors } \mathbf{T} \textit{ sinon } \mathbf{F}
\end{array}
$$

Si un terme est correctement typé dans un environnement Γ non vide, alors il est nécessaire de donner la sémantique des variables de Γ à l'aide d'une *valuation*.

Définition 5.4 (Valuation)**.** *Soit une structure* $(\mathcal{C}, \mathcal{I})$ *de* Σ *et un environnement* $\Gamma = \{v_1 : \sigma_1, \cdots, v_n : \sigma_n\}$*, alors une valuation* A *de* Γ *est une fonction associant à chaque* v_i *une valeur* $val_i \in \mathcal{C}(\sigma_i)$*. De plus on notera* $A' = A[v_0 \mapsto val_0]$ *la valuation où* $A'(v) = val_0$ *si* $v = v_0$ *et* $A(v)$ *sinon.*

La structure et la valuation associées à une signature multi-sortée Σ permettent d'interpréter les termes de Σ.

Définition 5.5 (Interprétation)**.** *Soit une signature* Σ*, une structure* $M = (\mathcal{C}, \mathcal{I})$*, un terme correctement typé* t *dans un environnement* Γ *de* Σ *et* A *une valuation de* Γ*, alors l'interprétation de* t *par* M *et* A *notée* $[\![t]\!]^{\mathcal{I}}_A$ *est définie comme suit :*

$$
\begin{array}{lcl}
[\![v]\!]^{\mathcal{I}}_A & \triangleq & A(v) \textit{ si } v \in Var \\
[\![c]\!]^{\mathcal{I}}_A & \triangleq & \mathcal{I}(c) \textit{ si } c \in Const \\
[\![f\ t_1\ \cdots\ t_n]\!]^{\mathcal{I}}_A & \triangleq & \mathcal{I}(f)([\![t_1]\!]^{\mathcal{I}}_A, \cdots, [\![t_n]\!]^{\mathcal{I}}_A) \textit{ si } f \in Fun \\
[\![\exists(v_1 : \sigma_1, \cdots, v_n : \sigma_n)\ \phi]\!]^{\mathcal{I}}_A & \triangleq & \mathbf{T} \textit{ si et seulement si pour une certaine } val_1 \in \mathcal{C}(\sigma_1), \cdots, val_n \in \mathcal{C}(\sigma_n), \\
 & & [\![\phi]\!]^{\mathcal{I}}_{A[v_1 \mapsto val_1] \cdots [v_n \mapsto val_n]} = \mathbf{T} \\
[\![\forall(v_1 : \sigma_1, \cdots, v_n : \sigma_n)\ \phi]\!]^{\mathcal{I}}_A & \triangleq & \mathbf{T} \textit{ si et seulement si pour tout } val_1 \in \mathcal{C}(\sigma_1), \cdots, val_n \in \mathcal{C}(\sigma_n), \\
 & & [\![\phi]\!]^{\mathcal{I}}_{A[v_1 \mapsto val_1] \cdots [v_n \mapsto val_n]} = \mathbf{T}
\end{array}
$$

Comme l'interprétation des symboles de fonctions $\mathcal{I}_{core}$ et des sortes $\mathcal{C}_{core}$ est toujours la même, nous omettrons celle-ci lors de la définition d'une structure. Par ailleurs, la plupart des termes considérés dans la suite du manuscrit sont *clos* c'est-à-dire ne possèdent pas de variables libres. Puisque l'interprétation d'un terme t clos ne dépend que de la structure, nous noterons $\mathcal{I}(t)$ pour $[\![t]\!]^{\mathcal{I}}_{\emptyset}$.

Exemple 5.10 (Interprétation)**.** *Soit le terme* $(a \wedge b) \vee c$ *de la signature étendue* $Core(\{a : Bool,\ b : Bool,\ c : Bool\})$ *et la structure* $(\mathcal{C}, \mathcal{I}) = (\emptyset, \{a \mapsto \mathbf{T},\ b \mapsto \mathbf{F},\ c \mapsto \mathbf{F}\})$ *alors l'interprétation du*

terme est :

$$\begin{aligned}\mathcal{I}((a \wedge b) \vee c) &= \mathcal{I}(\vee)(\mathcal{I}(a \wedge b), \mathcal{I}(c)) \\ &= \mathcal{I}(\vee)(\mathcal{I}(\wedge)(\mathcal{I}(a), \mathcal{I}(b)), \mathbf{F}) \\ &= \mathcal{I}(\vee)(\mathcal{I}(\wedge)(\mathbf{T}, \mathbf{F}), \mathbf{F}) \\ &= \mathbf{F}\end{aligned}$$

Prenons un autre exemple avec quantificateur, soit $(\exists d : Bool, a \wedge d) \wedge (\exists d : Bool, a \wedge \neg d)$ *l'interprétation du terme est alors :*

$$\mathcal{I}((\exists d : Bool, a \wedge d) \wedge (\exists d : Bool, a \wedge \neg d)) = \mathcal{I}(\wedge)(\mathcal{I}(\underbrace{\exists d : Bool, a \wedge d}_{A}), \mathcal{I}(\underbrace{\exists d : Bool, a \wedge \neg d}_{B}))$$

Ici, il faut choisir une valuation de la variable d *des termes* A *et* B *tel que leurs interprétations soient vraies, ainsi :*

$$\begin{aligned}\mathcal{I}(A) &= [\![a \wedge d]\!]^{\mathcal{I}}_{\{d \mapsto \mathbf{T}\}} \\ &= \mathcal{I}(\wedge)([\![a]\!]^{\mathcal{I}}_{\{d \mapsto \mathbf{T}\}}, [\![d]\!]^{\mathcal{I}}_{\{d \mapsto \mathbf{T}\}}) \\ &= \mathcal{I}(\wedge)(\mathbf{T}, \mathbf{T}) \\ &= \mathbf{T}\end{aligned} \qquad \begin{aligned}\mathcal{I}(B) &= [\![a \wedge \neg d]\!]^{\mathcal{I}}_{\{d \mapsto \mathbf{F}\}} \\ &= \mathcal{I}(\wedge)([\![a]\!]^{\mathcal{I}}_{\{d \mapsto \mathbf{F}\}}, [\![\neg d]\!]^{\mathcal{I}}_{\{d \mapsto \mathbf{F}\}}) \\ &= \mathcal{I}(\wedge)(\mathbf{T}, \mathbf{T}) \\ &= \mathbf{T}\end{aligned}$$

Il existe bien une interprétation de la variable d *pour les termes* A *et* B *telle que* $\mathcal{I}(A) = \mathbf{T}$ *et* $\mathcal{I}(B) = \mathbf{T}$*, d'où*

$$\mathcal{I}((\exists d : Bool, a \wedge d) \wedge (\exists d : Bool, a \wedge \neg d)) = \mathcal{I}(\wedge)(\mathbf{T}, \mathbf{T}) = \mathbf{T}$$

Notons ici que la valuation choisie pour la variable d *n'est pas la même pour les termes* A *et* B*. Ceci n'est pas un problème puisque la variable* d *est liée à deux quantificateurs distincts, par conséquent la variable* d *du terme* A *n'est pas la même que celle du terme* B *(malgré le fait qu'elle porte le même nom). Cet exemple illustre la portée des variables au sein des termes.*

5.3 Les problèmes SAT et SMT

5.3.1 Problème de satisfiabilité

Un problème classique basé sur l'interprétation des formules d'une signature Σ consiste à trouver une structure appelée *modèle* où une formule de Σ est vraie. On l'appelle *problème de satisfiabilité.*

Définition 5.6 (Modèle)**.** *Soit une signature* Σ*, une structure* $M = (\mathcal{C}, \mathcal{I})$ *de* Σ*, une formule* ϕ *de* Σ *correctement typée dans un environnement vide, alors* M *est un modèle de* ϕ *noté* $M \models \phi$ *si et seulement si* $\mathcal{I}(\phi)$ *est* $\mathbf{T}$*.*

Définition 5.7 (Satisfiabilité)**.** *Soit une signature* Σ *et une formule* ϕ *de* Σ *correctement typée dans un environnement vide, alors* ϕ *est* satisfiable *(ou plus simplement* SAT*) si et seulement s'il existe une structure* M *de* Σ *tel que* $M \models \phi$*. Dans le cas contraire on dit que* ϕ *est* insatisfiable *(*UNSAT*).*

De plus considérons deux formules ϕ et φ d'une signature Σ, s'il existe un modèle M, tel que $M \models \phi$ si et seulement s'il existe un modèle M' tel que $M' \models \varphi$, alors on dit que ϕ et φ sont équisatisfiables. Par abus de notation, nous notons $\phi \models \varphi$ si tout modèle de ϕ est aussi un modèle de φ. Par ailleurs, si $\phi \models \varphi$ et $\varphi \models \phi$ alors φ et ϕ sont sémantiquement équivalentes noté $\phi \equiv \varphi$.

Exemple 5.11 (Satisfiabilité)**.** *Reprenons la formule $\phi = (a \wedge b) \vee c$, nous savons que la structure $(\mathcal{C}, \mathcal{I}) = (\emptyset, \{a \mapsto \mathbf{T},\ b \mapsto \mathbf{F},\ c \mapsto \mathbf{F}\})$ n'est pas un modèle de ϕ puisque $\mathcal{I}(\phi)$ est $\mathbf{F}$. Par contre la structure $(\mathcal{C}, \mathcal{I}') = (\emptyset, \{a \mapsto \mathbf{T},\ b \mapsto \mathbf{T},\ c \mapsto \mathbf{F}\})$ est un modèle de ϕ car $\mathcal{I}'(\phi)$ est $\mathbf{T}$*

5.3.2 Le problème SAT

À partir des définitions précédentes, on peut introduire le problème SAT qui consiste à décider s'il existe un modèle d'une formule propositionnelle (c'est-à-dire de la signature appelée *Core*). Ce problème a de nombreuses applications en satisfaction de contraintes, planification ou encore model-checking. Par exemple, un problème SAT consiste à déterminer s'il existe une structure donnant une interprétation des propositions a et b telle que la formule $\phi = (a \wedge b) \vee (\neg a \wedge \neg b)$ soit vraie.

Le but consiste alors à déterminer un algorithme appelé *procédure de décision* décidant, en un temps fini, si une formule propositionnelle ϕ est satisfiable. Dans le cas du problème SAT, il existe plusieurs algorithmes de résolution du problème de satisfiabilité (le problème SAT est dit *décidable*). Par la suite, nous parlons de *solveur* SAT pour désigner une procédure de décision du problème SAT. L'un des enjeux fondamentaux de la résolution de ces problèmes est de trouver une procédure de décision la plus *efficace* possible c'est-à-dire résolvant le problème dans un temps raisonnable. Ceci n'est pas le cas pour notre approche *naïve* présentée dans l'exemple 5.12 qui construit une table contenant 2^n cellules où n est le nombre de propositions dans la formule. Ce problème est un champs de recherche très actif et [28] a prouvé que le problème SAT est un problème NP-complet.

Exemple 5.12 (Procédure de décision)**.** *Proposons une procédure de décision naïve pour la théorie $\mathcal{T}_{Core} = (\Sigma_{Core}, \mathcal{M}_{Core})$ définie précédemment. Soit $S_\phi \subset S$ l'ensemble des atomes contenus dans une formule ϕ de Σ_{Core}. Alors on peut construire la table de vérité d'une formule ϕ par rapport aux atomes de S_ϕ. Dans ce cas, on peut en déduire l'interprétation de ϕ à partir de la structure où l'interprétation des atomes de S_ϕ est donnée par la table, tandis que les autres symboles sont libres d'interprétation. Par conséquent, la formule est satisfiable si et seulement s'il existe une case de la table où l'interprétation est $\mathbf{T}$, son modèle est alors celui associé à cette case. S'il existe plusieurs cases où l'interprétation est $\mathbf{T}$, alors la formule possède plusieurs modèles et l'un d'entre eux (sans préférence particulière) est renvoyé.*

Prenons la formule $\phi = (a \vee b) \wedge (\neg a \vee \neg b)$, alors $S_\phi = \{a, b\}$, la table de vérité est :

a	b	$a \vee b$	$\neg a \vee \neg b$	$(a \vee b) \wedge (\neg a \vee \neg b)$
F	**F**	**F**	**T**	**F**
F	**T**	**T**	**T**	**T**
T	**F**	**T**	**T**	**T**
T	**T**	**T**	**F**	**F**

Donc ϕ est satisfiable par le modèle $(\emptyset, \{a \mapsto \mathbf{T},\ b \mapsto \mathbf{F}\})$.

5.3.3 Théorie et problème SMT

Généralement, il existe des contraintes supplémentaires sur les modèles obtenus par résolution d'un problème de satisfiabilité. En effet, ces modèles doivent interpréter certains symboles d'une manière bien définie, tandis que d'autres restent libres d'interprétation. On distingue deux types de symboles d'une signature Σ :

interprétés : ceux qui ont une interprétation imposée par une structure M de Σ

non-interprétés : ceux qui n'ont pas d'interprétation définie.

Ainsi on peut constituer, pour une signature donnée, l'ensemble des structures représentant les interprétations possibles des symboles non-interprétés mais partageant la même interprétation pour les symboles interprétés. Les formules vraies sur l'ensemble de ces structures forment alors une *théorie*.

Définition 5.8 (Théorie)**.** *Soit Σ une signature, $\mathcal{M}$ une classe (au sens de la théorie des ensembles) de structures de Σ et $\mathcal{F}$ l'ensemble des formules de Σ, alors une théorie $\mathcal{T}$ engendrée par Σ et $\mathcal{M}$ est $\mathcal{T} = \{\phi \in \mathcal{F} \mid \forall M \in \mathcal{M},\ M \models \phi\}$, les formules $\phi \in \mathcal{T}$ sont appelées les théorèmes de $\mathcal{T}$.*

Exemple 5.13 (Théorie)**.** *Considérons un ensemble de symboles S, on peut alors définir la théorie $\mathcal{T}_{Core}$ comme l'ensemble des formules de $\Sigma_{prop} = Core(\{b : Bool | b \in S\})$ tel que pour toute structure M dont l'interprétation des opérateurs booléens est conforme à celle donnée dans M_{Core}, nous avons $M \models \phi$. Autrement dit, $\mathcal{T}_{Core}$ est l'ensemble des tautologies.*

Soit $\mathcal{T}$ une théorie engendrée par une signature Σ et une classe de structures $\mathcal{M}$, nous dirons par la suite

- que les symboles de Σ n'appartenant pas à Σ_{Core} sont appelés *symboles de théorie* (nous parlerons notamment de prédicat de théorie par la suite) ;
- qu'un modèle M appartient à $\mathcal{T}$ si et seulement si $M \in \mathcal{M}$.

Le problème de satisfiabilité est alors étendu à une théorie $\mathcal{T}$ de la manière suivante :

Définition 5.9 (Satisfiabilité modulo théorie)**.** *Soit une théorie $\mathcal{T}$ et ϕ une formule de Σ alors ϕ est* satisfiable modulo $\mathcal{T}$ *si et seulement s'il existe une structure M de $\mathcal{T}$ telle que l'interprétation de ϕ par M soit* **T** *(notée $M \models_{\mathcal{T}} \phi$).*

Une fois encore, un problème fondamental est de déterminer s'il existe une procédure de décision pour résoudre le problème SMT. Malheureusement ceci n'est pas le cas pour toutes les théories, par exemple il n'existe pas de procédure de décision pour l'arithmétique entière [93]. Néanmoins, dans ce document nous nous intéressons aux théories où les sortes sont des ensembles finis. Dans ce cas il existe toujours une procédure de décision naïve consistant à énumérer toutes les structures possibles et à vérifier si l'une d'elle est un modèle de la formule. Les théories considérées sont donc décidables et la procédure de décision associée à une théorie est appelé *solveur* de théorie.

L'étape suivante consiste à combiner ces procédures de décision afin de décider si des formules utilisant des symboles de plusieurs théories sont satisfiables. Il existe plusieurs méthodes de combinaison des solveurs de théories comme celles présentées dans [66, 18]. Néanmoins ces méthodes imposent des restrictions sur les signatures des théories à combiner. Par exemple pour [66], les théories ne doivent partager aucun symbole. Une combinaison de théories telle qu'il existe une procédure de décision du problème de satisfiabilité pour un sous-ensemble des formules $\mathcal{F}$ d'une signature est appelée une *logique*.

Définition 5.10 (Logique). *Soit $\mathcal{T}_1, \cdots \mathcal{T}_n$ un ensemble de théories sur $\Sigma_1 = (Sort_1, Var_1, Fun_1, Rank_1)$, $\cdots$, $\Sigma_n = (Sort_n, Var_n, Fun_n, Rank_n)$ et $\mathcal{F}$ un ensemble de formules de $\Sigma = (Sort_1 \cup \cdots \cup Sort_n, Var_1 \cup \cdots \cup Var_n, Fun_1 \cup \cdots \cup Fun_n, Rank_1 \cup \cdots \cup Rank_n)$, alors une logique $\mathcal{L}$ est une combinaison de théories notée $\mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n$ telle qu'il existe une procédure de décision déterminant si une formule $\phi \in \mathcal{F}$ est satisfiable.*

Exemple 5.14 (Logique). *Soit $\mathcal{T}_{LRA}$ la théorie de l'arithmétique linéaire des réels et $\mathcal{T}_{UF}$ la théorie des fonctions non-interprétées alors la logique QF_UFLRA est la combinaison $\mathcal{T}_{UF} \cup \mathcal{T}_{LRA}$ pour laquelle il existe une procédure de décision déterminant s'il existe une structure M telle que $M \models \phi$ où $\phi \in \Sigma$ avec :*

- $\Sigma = (Sort_{UF} \cup Sort_{LRA}, \emptyset, Fun_{UF} \cup Fun_{LRA}, Rank_{UF} \cup Rank_{LRA})$
- $M = (M_{UF}, M_{LRA})$
- *$\mathcal{F}$ est l'ensemble des formules de Σ ne contenant pas de quantificateurs.*

5.4 Résolution des problèmes de satisfiabilité

Le problème de Satisfiabilité modulo Théorie (SMT) consiste à décider si une formule d'une logique $\mathcal{L}$ est satisfiable. Comme dans ce manuscrit les ensembles porteurs des sortes sont finis, il existe toujours une procédure de décision naïve consistant à énumérer toutes les structures possibles. Alors la satisfiabilité modulo $\mathcal{T}$ d'une conjonction d'un ensemble de littéraux par un modèle M, notée $M \models_{\mathcal{T}} l_1 \wedge \cdots \wedge l_n$, est décidable.

Pour résoudre le problème de satisfiabilité, la plupart des solveurs, comme Z3 [32], SharpCDCL [62] ou SAT4J [11], se basent sur la procédure de décision DPLL pour Davis-Putman-Logemann-Loveland [30]. Cette procédure décide si une formule CNF est satisfiable en construisant un modèle partiel de la formule. Notons qu'une formule propositionnelle ϕ est traduisible en une forme CNF ϕ' équisatisfiable en un temps linéaire [94].

Pour des raisons de lisibilité, nous introduisons les notations et définitions suivantes.

- $C = \{l_1, \neg l_2, \cdots, l_n\}$ représente la clause $C = l_1 \vee \neg l_2 \cdots \vee l_n$ comme un ensemble de littéraux ;
- $C = C' \vee l$ où $l \in C$ et $C' = C \setminus \{l\}$;
- une *valuation partielle* est un ensemble de littéraux M tel que si $l \in M$ alors l est évalué à **T**, si $\neg l \in M$ alors l est évalué à **F**, autrement l n'a pas d'interprétation définie. On dit alors que $M \models \phi$ si la structure $(\emptyset, M) \models \phi$.

Les travaux de [69] introduisent une formalisation de la procédure de décision DPLL [30] comme un système de transition d'états appelé DPLL Abstrait.

Définition 5.11 (DPLL Abstrait). *Soit une formule CNF ϕ, alors la procédure DPLL est une machine à état où les états sont étiquetés par le couple $M \| \phi'$ où M est une valuation partielle des littéraux de ϕ et ϕ' est une formule CNF. L'état initial de la machine est $\emptyset \| \phi$. Les états finaux sont soit S_{fail} signifiant que la formule n'est pas satisfiable et $M \| \phi$ signifiant que $M \models \phi$. Une transition d'un état s vers un états s' de la machine est notée $s \longrightarrow s'$.*

Intuitivement, cette procédure consiste à fabriquer incrémentalement un modèle M d'une formule ϕ :

- soit par déduction de la valeur d'un littéral l à partir de M et ϕ, c'est-à-dire que l'on sait que si $l \notin M$ alors M n'est pas un modèle de ϕ ;
- soit par décision de la valeur d'un littéral l n'appartenant pas encore à M, c'est-à-dire un choix arbitraire de la valeur d'un littéral l.

Si, lors de la construction d'un modèle M, tous les littéraux d'une clause de ϕ sont faux, alors M n'est pas un modèle de ϕ, et certaines décisions sont inversées pour résoudre le problème.

5.4.1 Résolution de SAT

Expliquons maintenant les règles de construction d'un modèle M d'une formule CNF ϕ de la logique propositionnelle. Si une clause C de ϕ ne contient plus qu'un seul littéral l permettant de la satisfaire pour le modèle courant M alors l doit être ajouté à M, c'est la règle de propagation.

$$\text{Propagate} \quad M \| \phi, C \vee l \quad \longrightarrow \quad Ml \| \phi, C \vee l \quad si \begin{cases} M \models \neg C \\ l, \neg l \notin M \end{cases}$$

Lorsque cette règle n'est pas applicable, il est nécessaire d'ajouter *arbitrairement* un *littéral de décision* (noté l^d) à M pour continuer la résolution.

$$\text{Decide} \quad M \| \phi \quad \longrightarrow \quad Ml^d \| \phi \quad si \begin{cases} l \text{ ou } \neg l \text{ appartient à une clause de } \phi \\ l, \neg l \notin M \end{cases}$$

Comme ce choix est basé sur une heuristique, il peut générer un *conflit* c'est-à-dire qu'il existe une clause C, appelée *clause de conflit*, de ϕ telle que $M \models \neg C$. Un des littéraux de décision de M peut être la cause de ce problème. Dans ce cas on applique la règle de *backtrack* qui inverse le dernier littéral de décision $l^d \in M$. Bien entendu les littéraux de propagation s'appuyant sur cette décision sont supprimés de M.

$$\text{Backtrack} \quad Ml^d N \| \phi, C \quad \longrightarrow \quad M \neg l \| \phi, C \quad si \begin{cases} Ml^d N \models \neg C \\ l^d \text{ est le dernier littéral de décision} \end{cases}$$

Si cette règle échoue alors il n'existe aucun littéral de décision expliquant le conflit, ϕ est alors *insatisfiable*.

$$\text{Fail} \quad M \| \phi, C \quad \longrightarrow \quad S_{fail} \quad si \begin{cases} M \models \neg C \\ M \text{ ne contient pas de littéraux de décision} \end{cases}$$

Définition 5.12 (DPLL)**.** *Pour résoudre le problème de satisfiablité les transitions possibles de la machine sont données par les règles Propagate, Decide, Backtrack, Fail.*

Exemple 5.15 (DPLL)**.** *Appliquons la procédure DPLL sur la formule* $(\wedge\ (\vee\ a\ b)\ (\vee\ \neg b\ d)\ (\vee\ \neg b\ \neg d)\ (\vee\ \neg a\ c))$*. Voici la trace d'exécution de la procédure DPLL et les règles utilisées pour*

résoudre le problème de satisfiabilité :

$$
\begin{array}{lll}
\emptyset & \|(\wedge\ (\vee\ a\ b)\ (\vee\ \neg b\ d)\ (\vee\ \neg b\ \neg d)\ (\vee\ \neg a\ c)) & \xrightarrow{Decide} \\
\{a^d\} & \|(\wedge\ (\vee\ a\ b)\ (\vee\ \neg b\ d)\ (\vee\ \neg b\ \neg d)\ \mathbf{(\vee\ \neg a\ c)}) & \xrightarrow{Popagate} \\
\{a^d,\ c\} & \|(\wedge\ (\vee\ a\ b)\ (\vee\ \neg b\ d)\ (\vee\ \neg b\ \neg d)\ (\vee\ \neg a\ c)) & \xrightarrow{Decide} \\
\{a^d,\ c,\ b^d\} & \|(\wedge\ (\vee\ a\ b)\ \mathbf{(\vee\ \neg b\ d)}\ (\vee\ \neg b\ \neg d)\ (\vee\ a\ c)) & \xrightarrow{Propagate} \\
\{a^d,\ c,\ b^d,\ d\} & \|(\wedge\ (\vee\ a\ b)\ (\vee\ \neg b\ d)\ \mathbf{(\vee\ \neg b\ \neg d)}\ (\vee\ \neg a\ c)) & \xrightarrow{Backtrack} \\
\{a^d,\ c,\ \neg b\} & \|(\wedge\ (\vee\ a\ b)\ (\vee\ \neg b\ d)\ (\vee\ \neg b\ \neg d)\ (\vee\ \neg a\ c)) &
\end{array}
$$

Dans un premier temps, la procédure décide d'ajouter a^d au modèle. Or pour que la clause $\{\neg a, c\}$ soit vraie dans M, il faut ajouter c à M, ce qui est fait par la règle de propagation. Une fois de plus, un choix est nécessaire, disons que b^d est ajouté à M. Alors par la clause $\{\neg b, d\}$, on sait que d doit appartenir à M. Or il existe une autre clause $\{\neg b, \neg d\}$ qui implique que $\neg d$ doit aussi être ajouté à M. Ceci n'étant pas possible, la règle de retour est alors utilisée pour résoudre le conflit. On change alors la décision sur b et on obtient un modèle du problème $M = \{a^d,\ c,\ \neg b\}$.

DPLL est une procédure de décision du problème SAT pour les formules CNF [69]. Autrement dit, il n'existe pas de séquences infinies d'états à partir de l'état initial. De plus l'état final S_{fail} est atteignable si et seulement si la formule est insatisfiable, inversement l'état final $M\|\phi$ est atteignable si et seulement si $M \models \phi$. Par conséquent toute implantation respectant les règles introduites par la définition 5.12 est une procédure de décision du problème SAT.

Cette procédure, datant des années 60, a été améliorée en ajoutant des règles exploitant plus efficacement les conflits rencontrés lors de la résolution. Plus précisément, le but est d'*apprendre* des clauses de conflit afin d'éviter de commettre d'autres erreurs de décision pendant la résolution. Ce type de résolution est appelé *apprentissage de clauses dirigé par le conflit* (CDCL en anglais) [12]. Présentons les nouvelles règles ajoutées par CDCL.

La première règle appelée *Backjump* est une version améliorée de la règle initiale de *Backtrack*.

$$
\text{Backjump} \quad Ml^dM'\|\phi, C \quad \longrightarrow \quad Ml'\|\phi, C \quad si \begin{cases} Ml^dM' \models \neg C \\ \text{il existe } C' \vee l' \text{ tel que } \phi \models C' \vee l' \\ M \models \neg C' \\ l' \text{ ou } \neg l' \text{ appartient à une clause de } \phi \\ l', \neg l' \notin M \end{cases}
$$

Cette règle génère une nouvelle clause de conflit $C' \vee l'$ et ajoute le littéral l' au lieu d'ajouter la négation du dernier littéral de décision. En effet, la cause d'un conflit peut venir d'un ensemble de mauvaises décisions ajoutées en amont, il est donc judicieux d'analyser les raisons de ce conflit c'est-à-dire mettre en place des techniques de génération des clauses de conflit. Ces techniques ont fait l'objet de nombreuses recherches et sont au cœur de l'amélioration des performances des solveurs SAT. Néanmoins la présentation de ces techniques est en dehors du cadre de ce document, nous pouvons tout de même citer la génération par *analyse du graphe d'implication* présentée dans [99].

La règle suivante consiste à ajouter à la formule originale ϕ les clauses de conflits générées

pendant la résolution (si C est une clause de conflit alors $\phi \wedge C$ est équivalente à ϕ).

$$\text{Learn} \quad M \| \phi \quad \longrightarrow \quad M \| \phi, C \quad si \begin{cases} \phi \models C \\ \text{tous les atomes de } C \text{ appartiennent à } \phi \end{cases}$$

Afin de contenir l'explosion de l'espace mémoire nécessaire au stockage de ces nouvelles clauses, la règle *Forget* efface les clauses apprises durant la résolution.

$$\text{Forget} \quad M \| \phi, C \quad \longrightarrow \quad M \| \phi \quad \text{si } \phi \models C$$

Lors de la résolution, il est parfois plus rapide de recommencer complètement l'analyse à partir du début. En effet, l'heuristique de choix peut avoir construit un modèle partiel à partir duquel il est difficile de construire un modèle complet ou de prouver que les décisions mènent à une impasse.

$$\text{Restart} \quad M \| \phi, C \quad \longrightarrow \quad \emptyset \| \phi \quad \text{déclenchée par une heuristique}$$

Définition 5.13 (CDCL)**.** *La procédure CDCL est constituée des règles Propagate, Decide, Backjump, Fail, Learn, Forget, Restart.*

La stratégie d'application de ces nouvelles règles est la suivante :
- appliquer les règles classiques entre chaque apprentissage
- appliquer le redémarrage de moins en moins souvent au cours de la résolution
- appliquer l'apprentissage juste après un retour

Pour conserver la terminaison de la procédure classique il faut assurer que les règles d'oubli et d'apprentissage ne sont appliquées qu'un nombre fini de fois, sinon il est possible d'apprendre et d'oublier la même clause indéfiniment. De même, le redémarrage est appliqué avec une périodicité croissante au cours de la résolution, ce qui évite de recommencer sans cesse la résolution du problème.

5.4.2 Résolution de SMT

Nous allons voir maintenant comment les auteurs de [69] étendent la procédure CDCL pour résoudre le problème SMT pour une logique $\mathcal{L}$. Pour cela, considérons qu'il existe une procédure de décision basée sur l'interaction des solveurs de théorie de la logique. Ainsi dans une formule $\phi \in \mathcal{F}$, les atomes sont soit des propositions, soit des prédicats d'une des théories de $\mathcal{L}$. On peut alors extraire le squelette booléen ϕ_{Core} de ϕ en remplaçant les prédicats par de nouvelles propositions. Par conséquent, la procédure CDCL vérifie si ϕ_{Core} est satisfiable dans $\mathcal{L}_{Core}$, c'est-à-dire si $M \models \phi_{Core}$. Si elle ne l'est pas alors *a fortiori* elle ne l'est pas non plus dans $\mathcal{L}$. Par contre si ϕ_{Core} est SAT dans $\mathcal{L}_{Core}$, il est nécessaire de demander aux $\mathcal{T}$-solveurs si la conjonction des prédicats de théorie de M est satisfiable modulo $\mathcal{T}$. Si oui alors M est dit $\mathcal{T}$-consistant, autrement la procédure est relancée en rejetant le modèle, c'est-à-dire en analysant la formule $\phi \wedge \varphi$ où $\varphi \models \neg M$.

Plus précisément, il faut adapter les règles de la CDCL pour prendre en compte ces nouvelles informations. Par conséquent, lorsqu'un modèle est rejeté, la clause résultante est ajoutée par la règle $\mathcal{T}$-Learn.

$$\mathcal{T} - \text{Learn} \quad M \| \phi \quad \longrightarrow \quad M \| \phi, C \quad si \begin{cases} \phi \models_{\mathcal{T}} C \\ \text{tous les atomes de } C \text{ appartiennent à } \phi \end{cases}$$

Il faut alors adapter la règle Forget pour pouvoir oublier des clauses générées par le solveur de théorie.

$$\mathcal{T} - \text{Forget} \quad M \| \phi, C \quad \longrightarrow \quad M \| \phi \quad \text{si } \phi \models_{\mathcal{T}} C$$

Finalement, lors de la détection d'un conflit, on peut utiliser le solveur de théorie pour trouver les décisions ayant menées à un modèle inconsistant de son point de vue.

$$\mathcal{T} - \text{Backjump} \quad M l^d M' \| \phi, C \quad \longrightarrow \quad M l' \| \phi, C \quad si \begin{cases} M l^d M' \models \neg C \\ \text{il existe } C' \vee l' \text{ tel que } \phi \models_{\mathcal{T}} C' \vee l' \\ M \models \neg C' \\ l' \text{ ou } \neg l' \text{ appartient à une clause de } \phi \\ l', \neg l' \notin M \end{cases}$$

Pour améliorer la résolution d'un problème SMT, il est possible d'intégrer les solveurs de théories au niveau de la règle de propagation de CDCL. En effet, au lieu d'attendre que le solveur SAT propose un modèle complet pour pouvoir le vérifier, il est plus efficace d'intégrer le solveur dans la construction du modèle. C'est pour cela que la procédure CDCL($\mathcal{T}$) introduit la règle de *propagation de théorie*. Intuitivement, cette règle stipule que si un littéral l est une conséquence logique de M du point de vue de $\mathcal{T}$ alors celui-ci est ajouté à M. En général, cette règle possède la priorité d'application la plus élevée afin de prendre en compte le plus rapidement possible les conséquences d'un modèle du point de vue de $\mathcal{T}$.

$$\mathcal{T}\text{−}Propagate \quad M \| \phi \quad \longrightarrow \quad M l \| \phi \quad si \begin{cases} M \models_{\mathcal{T}} l \\ l, \neg l \notin M \\ l \text{ ou } \neg l \text{ appartient à une clause de } \phi \end{cases}$$

Définition 5.14 (CDCL($\mathcal{T}$))**.** *Soit $\mathcal{T}$ une théorie, alors les nouvelles transitions de l'extension de CDCL($\mathcal{T}$) sont définies par les règles $\mathcal{T}$-Backjump, $\mathcal{T}$-Learn, $\mathcal{T}$-Forget, $\mathcal{T}$-Propagate.*

Exemple 5.16 (CDCL($\mathcal{T}_{LRA}$))**.** *Soit la logique $\mathcal{L}_{LRA} = (\Sigma_{LRA}(\{x : Int, y : Int\}), \mathcal{T}_{LRA}, \mathcal{F}_{LRA})$, étudions le problème de satisfiabilité de la formule suivante :*

$$(\underbrace{(x \geq 0)}_{p_1} \vee \underbrace{(y = x + 3)}_{p_2}) \wedge (\underbrace{(y > x)}_{p_3} \vee \underbrace{(y \leq 2)}_{p_4})$$

Imaginons que l'heuristique de décision choisisse d'ajouter p_2^d à M. Dans ce cas, une première propagation de $\mathcal{T}_{LRA}$ se produit car $\{(y = x + 3)^d\} \models_{\mathcal{T}_{LRA}} (y > x)$ donc p_3 est ajouté à M. En résumé, une décision et propagation suffisent à produire le modèle p_2^d, p_3. Le simplexe établit que $\{x \mapsto 1, y \mapsto 4\} \models_{LRA} (y = x + 3) \wedge (y > x)$ c'est-à-dire que $\{x \mapsto 1, y \mapsto 4\}$ est un modèle de la formule.

Il existe d'autres améliorations plus récentes de la procédure CDCL, notamment celle présentée par [31] intitulée *calcul de satisfiabilité par construction de modèles* (mcSAT). Le modèle M, utilisé

par la résolution classique CDCL, contient non seulement des littéraux mais aussi des valuations des variables et constantes MSFOL du problème. À l'inverse des anciennes méthodes de résolution, mcSAT offre la possibilité aux analyses de conflit de générer des clauses contenant de nouveaux littéraux (appartenant à une base finie pour assurer la terminaison) créées à partir des valuations contenues dans le modèle. Les auteurs de [31] adaptent alors les règles de CDCL pour gérer la création et l'utilisation de ces nouveaux littéraux. Nous ne présenterons pas en détails ces adaptions car nous n'utilisons pas cette approche par la suite, néanmoins le lecteur intéressé peut se reporter à [31] pour plus de détails.

5.5 Présentation du standard SMT-Lib

Pour harmoniser le format d'entrée des différents solveurs SMT existants, un langage standard de description des problèmes SMT appelé SMT-Lib a été introduit par [9]. Ce standard donne une syntaxe des termes d'une signature et propose un ensemble de logiques prédéfinies. Par exemple la logique des Fonctions non Interprétées et des Vecteurs de Bits (UFBV) avec structure de données algébrique proposée par SMT-Lib est utilisée dans la suite de ce manuscrit. Par conséquent, nous utilisons ce standard pour décrire les problèmes SMT sur lesquels nous nous basons pour analyser un système.

5.5.1 Syntaxe des termes

Pour écrire un problème SMT, il est souvent nécessaire d'étendre la signature de la logique utilisée avec des fonctions et des sortes. Dans SMT-Lib ces sortes et fonctions sont soit

interprétées c'est-à-dire que l'on décrit le symbole et l'interprétation de celui-ci

non-interprétées c'est-à-dire que l'on donne seulement le symbole sans y associer d'interprétation, ce sont les inconnues du problème SMT.

Dans la logique UFBV le langage SMT-Lib fournit les sortes prédéfinies comme les booléens (`Bool`) et les vecteurs de bits (`BitVec`) de taille fixe mais arbitrairement grande. Il est aussi possible de définir de nouvelles sortes à partir des sortes prédéfinies avec la commande `define-sort`. Par exemple la sorte représentant les vecteurs de bits de taille 8 est définie comme suit :

```
(define-sort BV8 () (_ BitVec 8))
```

Il est aussi possible de déclarer une sorte *structure de données* avec la commande `declare-datatype`. Par exemple, une sorte `Tuple2`, représentant les couples, est déclarée comme suit :

```
(declare-datatypes (T1 T2) ((Tuple2 (mkTuple2 (field1 T1) (field2 T2)))))
```

Cette commande déclare la sorte `Tuple2`, son constructeur `mkTuple2`, paramétré par les sortes T_1 et T_2, et les accesseurs `field1` (respectivement `field2`) retournant le premier (respectivement le second) élément d'un `Tuple2`. Par exemple, (`mkTuple2 a b`) renvoie un `Tuple2` contenant a, b.

Une fonction interprétée $f(v_1 : \sigma, \cdots, v_n : \sigma_n) : \sigma$ dont l'interprétation est définie comme un terme `body` de sorte σ, est définie par la commande `define-fun` comme suit :

```
(define-fun f ((v1 sigma_1)...(vn sigma_n)) sigma body)
```

Par exemple, une fonction `BV8_inter` retournant le ET bit-à-bit de deux `BV8` est définie comme :

```
(define-fun BV8_inter ((a BV8) (b BV8)) BV8 (bvand a b))
```

SMT-Lib fournit un ensemble de fonctions interprétées de base pour la logique UFBV :
- (`and` (*Bool Bool*) *Bool*) pour la conjonction
- (`or` (*Bool Bool*) *Bool*) pour la disjonction
- (`=>` (*Bool Bool*) *Bool*) pour l'implication
- (`not` (*Bool*) *Bool*) pour la négation
- (`<=>` (*Bool Bool*) *Bool*) pour l'équivalence
- (`=` (`S S`) *Bool*) pour l'égalité (où S est une sorte)
- (`bvor` ((`_BitVec n`) (`_BitVec n`)) (`_BitVec n`)) pour la disjonction bit-à-bit entre deux vecteurs de bits
- (`bvand` ((`_BitVec n`) (`_BitVec n`)) (`_BitVec n`)) pour la conjonction bit-à-bit entre deux vecteurs de bits
- (`let`((`x`$_1$ `t`$_1$)$\cdots$(`x`$_n$ `t`$_n$)) `t`) pour *let* $((x_1\ t_1), \cdots, (x_n\ t_n))$ *in* t
- (`ite b t e`) pour le branchement conditionnel *si* b *alors* t *sinon* e

Une fonction non-interprétée $g(\sigma, \cdots, \sigma_n)$: σ est déclarée par la commande `declare-fun` comme suit :

```
(declare-fun g (sigma_1 ... sigma_n) sigma)
```

Par exemple une fonction non-interprétée g prenant deux `BV8` et renvoyant un booléen est déclarée par :

```
(declare-fun g (BV8 BV8) Bool)
```

Si la fonction est une constante c de sorte σ alors les raccourcis suivants sont utilisés :

```
(declare-const c sigma)
(define-const c sigma val)
```

Nous rappelons que les formules d'un problème sont soit des applications de prédicats, d'opérateurs booléens, d'égalité de termes et de quantificateurs. En SMT-Lib une application d'une fonction f s'écrit (`f t1...tn`) où les sortes de $t_1, \cdots, t_n$ respectent le rang de f. Les quantificateurs s'utilisent comme suit :

```
(forall ((x1 T1)...(xn Tn)) b) (exists ((x1 T1)...(xn Tn)) b)
```

Un `forall` (respectivement `exists`) est vrai si et seulement si b est vraie pour toute interprétation (respectivement pour une interprétation) des symboles $x_1, \cdots, x_n$.

5.5.2 Commandes de résolution des problèmes SMT

Comme introduit dans la section 5.2.2, le but du solveur SMT est de trouver un modèle des fonctions non-interprétées satisfaisant un ensemble de formules, appelées *assertions* en SMT-Lib. Formellement la formule à satisfaire est la conjonction des assertions. Une assertion a est ajoutée au problème via la commande (`assert a`). Puis, le problème SMT est résolu via la commande (`check-sat`). Deux cas sont alors possibles
- le problème est Sat alors un modèle peut être fourni par la commande (`get-model`).
- le problème est Unsat alors il n'existe pas de modèle satisfaisant les assertions et une explication *minimale* peut être générée avec la commande (`get-unsat-core`).

Illustrons la syntaxe des termes et les commandes de résolution de SMT-Lib. Le script ci-dessous représente un problème SMT basé sur l'arithmétique entière linéaire. Celui-ci contient deux entiers

non-interprétés `x` et `y`. Pour exprimer le problème, nous avons introduit une fonction `min` qui renvoie le plus petit des deux entiers passés en paramètre. La formule à satisfaire est la conjonction de deux assertions. La dernière commande demande de résoudre le problème.

```
; declaration des inconnues
(declare-const x Int)
(declare-const y Int)

; definition de la fonction min
(define-fun min
  ((a Int) (b Int))
  Int
  (ite (<= a b) a b))

; definition des assertions
(assert
  (<= (min x y) 2))

(assert
  (let ((z (+ x y)))
    (>= z 2)))

; resolution du probleme
(check-sat)
```

Listing 5.1 – Exemple de script SMT-LIB

Dans ce cas, le solveur répond SAT, et le modèle fourni par $\texttt{get} - \texttt{model}$ est le suivant $M = \{x \mapsto 0, y \mapsto 2\}$.

5.6 Résumé

Nous avons présenté, dans ce chapitre, la logique du premier ordre multi-sortée en exposant la construction et le typage des termes mais aussi la sémantique des formules de la MSFOL. Nous avons alors formellement introduit le problème de Satsifiabilité Modulo Théorie et les mécanismes de résolution de ce problème. Finalement, nous avons présenté le langage standard SMT-LIB, utilisé dans ce manuscrit pour formuler et résoudre les problèmes SMT. Nous allons maintenant proposer un ensemble de contributions ayant pour objectif de lever les limitations des approches existantes de résolution du problème DSE.

Troisième partie

Contributions

Chapitre 6

Problématique

Nous rappelons, dans ce chapitre, les principales limitations des approches existantes de modélisation, d'analyse et de résolution du problème DSE. Afin de pallier ces limitations, nous proposons un nouveau langage ainsi qu'un ensemble d'analyses et de méthodes de résolution du problème DSE basées sur la résolution de problèmes SMT. Pour cela nous rappelons, dans la section 6.1, les différentes limitations dégagées dans l'état de l'art et introduisons, en section 6.2, une vue d'ensemble de notre processus de modélisation, d'analyse et de résolution du problème DSE.

6.1 Limitations des approches existantes de résolution du problème DSE

Approches de résolution du problème DSE Les limitations des approches de résolution existantes, identifiées dans la section 3.3.4, sont les suivantes :

formalisation Les approches basées contraintes imposent des restrictions sur les types de systèmes, d'exigences et d'alternatives considérés, réduisant ainsi le champs d'application de ces approches. À l'inverse, les approches basées heuristiques offrent une modélisation du problème sans aucune restriction sur les systèmes, alternatives ou exigences considérées.

résolution Les approches basées heuristiques ne fournissent pas la garantie de résoudre, en un temps fini, un problème DSE. À l'inverse, les méthodes basées contraintes garantissent de trouver la solution si et seulement si elle existe (sous condition que le problème respecte les restrictions de la méthode utilisée).

La méthode de résolution basée SMT de [80] semble fournir un bon compromis entre restrictions de formalisation et garanties de résolution. En effet, cette technique propose :

- de modéliser le problème en optant pour une traduction automatique du système et des propriétés de sûreté en MSFOL ;
- de bénéficier des mécanismes de résolution des solveurs SMT ;
- de définir des théories spécifiques pour traiter efficacement les exigences de sûreté de fonctionnement.

Pour toutes ces raisons, nous nous sommes inspirés de cette méthode pour définir une exploration basée SMT et spécialisée dans la résolution de contraintes de sûreté de fonctionnement.

Approches de modélisation des systèmes Afin d'évaluer la sûreté de fonctionnement d'un candidat lors du processus de sélection, nous nous sommes focalisés, dans la section 4.3.3, sur les langages de modélisation et d'analyse des comportements dysfonctionnels des systèmes. Nous avons établi les limitations suivantes :

modélisation Les langages permettant de décrire des modèles hiérarchiques et modulaires comme Altarica et SMV sont destinés à la modélisation de systèmes dynamiques. Ainsi, pour utiliser ces langages nous devrions établir une procédure vérifiant si les systèmes modélisés sont bien statiques (en dépit de leur modélisation à base d'automates). Puis traduire ces systèmes dans un formalisme statique.

analyses Par ailleurs, les analyses de sûreté de fonctionnement pour les systèmes dynamiques sont basées sur l'exploration de l'espace d'états des automates, qui peut être coûteuse pour calculer exactement les indicateurs probabilistes. À l'inverse, les méthodes de calcul des indicateurs pour les systèmes statiques (Grif, OpenPSA, HipHOPS) ont été utilisées avec succès pour évaluer ces indicateurs pour des systèmes de grande taille.

Pour ces raisons, nous souhaitons définir un nouveau langage de modélisation hiérarchique et modulaire de systèmes statiques. Nous nous inspirons de la modélisation proposée par Altarica et HipHOPS pour créer un nouveau langage appelé KCR.

6.2 Positionnement du manuscrit

Afin de répondre à ces différentes problématiques nous proposons un nouveau processus de modélisation, d'analyse des systèmes et de résolution du problème DSE illustré par la figure 6.1 dont les principales étapes sont les suivantes :

Modélisation KCR est un langage de modélisation modulaire des systèmes hiérarchiques et statiques. Les étapes de modélisation d'un système sont basées sur le processus décrit dans la section 2.3.1, c'est-à-dire :

Modèles des composants L'utilisateur donne, pour chaque composant, les modes de défaillance produits en sortie du composant en fonction des modes de défaillance en entrée et des événements de défaillance du composant.

Instanciation & connexion Puis, les modèles des composants sont instanciés et connectés pour former le modèle du système.

Configuration Finalement, l'utilisateur fournit les paramètres d'analyse et d'exploration au sein d'une structure appelée *configuration*. Plus précisément, cette structure contient les différentes alternatives possibles pour les composants du système, l'événement redouté et les exigences de sûreté à respecter.

Analyses Nous proposons d'intégrer les solveurs SMT dans le processus d'analyse du système de la manière suivante :

Traduction Nous proposons de définir la sémantique formelle de KCR par traduction vers la MSFOL ce qui permet :

- une formulation simplifiée des problèmes SMT puisque le modèle du système est déjà exprimé dans la MSFOL ;
- d'utiliser la logique des fonctions non-interprétées et des vecteurs de bits (UFBV) pour modéliser la propagation des modes de défaillance dans le système (BV) et la synthèse de la fonction de structure à partir du modèle (UF).

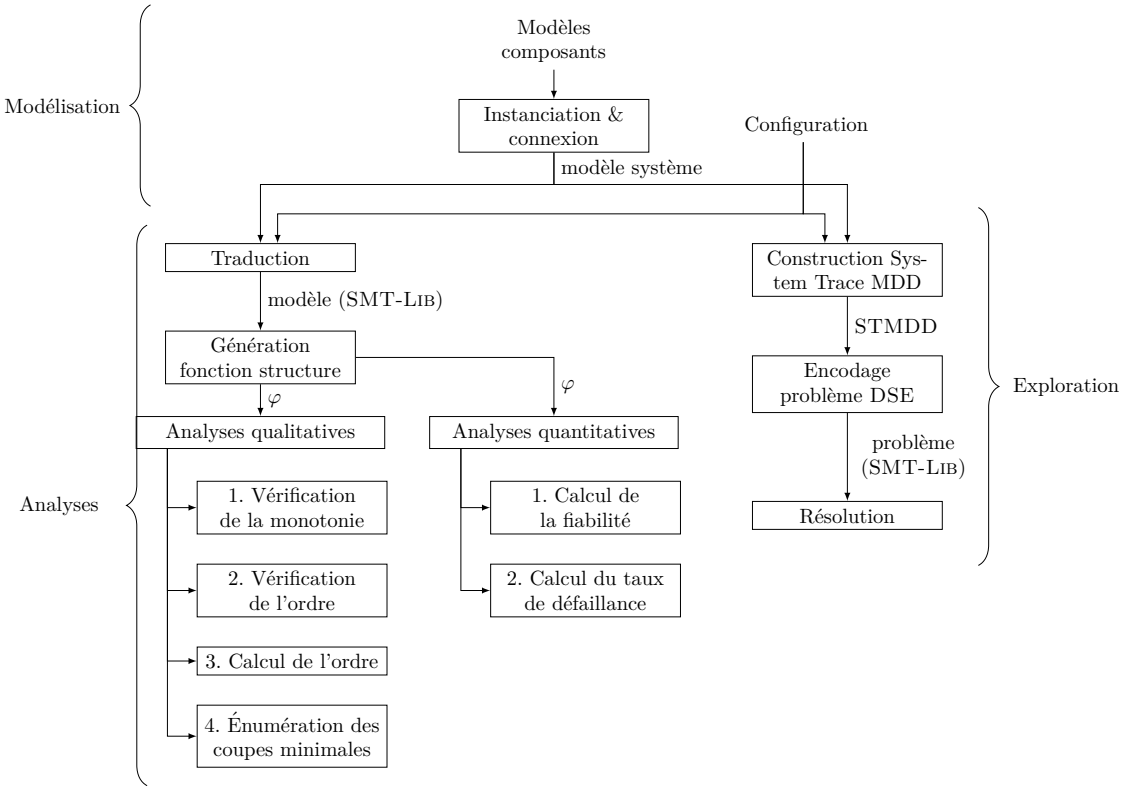


Figure 6.1 – Processus de modélisation, d'analyse et de résolution du problème DSE

Génération fonction structure Puisque la fonction de structure permet de calculer l'ensemble des indicateurs de sûreté (section 2.1.4), cette fonction est construite à partir de la traduction MSFOL du modèle du système.

Analyses qualitatives Nous formalisons alors un ensemble de problèmes SMT à partir de cette fonction de structure pour

- calculer ou vérifier la conformité d'un système par rapport à des exigences sur l'ordre du système ;
- déterminer la monotonie du système ;
- énumérer les coupes minimales du système.

Analyses quantitatives La fiabilité et le taux de défaillance du système peuvent aussi être évalués à partir des méthodes classiques présentées précédemment.

Exploration Finalement, nous proposons une méthode de résolution du problème DSE reposant sur la résolution de problèmes SMT. Pour cela :

Construction System Trace MDD Nous introduisons une nouveau type d'analyse dont le résultat est une structure de données appelée *System Trace MDD* (STMDD). Nous montrons alors que le STMDD du système initial est utilisable pour calculer les indicateurs de sûreté de n'importe quel candidat de l'espace des architectures.

Encodage du problème DSE Nous introduisons une théorie (au sens SMT), appelée

Safety, fournissant un ensemble de prédicats et de sortes utilisés pour encoder le problème DSE à partir du STMDD du système initial et des exigences de sûreté données dans la configuration.

Résolution Finalement, nous utilisons le solveur SMT Z3 combiné au solveur de la théorie Safety pour résoudre le problème DSE.

Présentons, dans le chapitre suivant, le langage KCR utilisé pour formuler les problèmes DSE.

Chapitre 7

Modélisation des systèmes avec le langage KCR

Ce chapitre présente le langage de modélisation KCR [36], spécialement conçu pour décrire les comportements dysfonctionnels des systèmes statiques. Pour cela nous présentons la syntaxe de KCR dans la section 7.1 et expliquons comment les structures du langage sont employées pour décrire la logique de propagation et de génération des modes de défaillance d'un système. Puis la section 7.2 présente la sémantique de KCR par traduction vers le langage SMT-Lib.

7.1 Syntaxe

La syntaxe de KCR est exprimée dans la Forme Backus-Naur Étendue (EBNF en anglais) définissant les règles grammaticales. Les identificateurs du langage appartiennent aux ensembles suivants :

- *ValId* identificateurs de valeurs,
- *TypeId* identificateurs de types,
- *CompId* identificateurs de composants,
- *EvtId* identificateurs d'événements de défaillance,
- *InstId* identificateurs d'instances de composant,
- *FlowId* identificateurs de flots,
- *ConfId* identificateurs de configurations,
- *PkgId* identificateurs de paquets.

Par souci de lisibilité, introduisons une macro-expression désignant les listes d'éléments délimités par un séparateur *sep* :

$NEList(x, sep)$	::=	$x\ (sep\ x)^*$	liste non vide
$List(x, sep)$	::=	$[NEList(x, sep)]$	liste potentiellement vide

Dans le langage KCR, le séparateur des éléments d'une liste est $\ll , \gg$, nous écrivons donc $NEList(x)$ (resp. $List(x)$) pour $NEList(x, \ll , \gg)$ (resp. $List(x, \ll , \gg)$).

7.1.1 Programme KCR (*Program*)

Un *programme* KCR est un ensemble de déclarations contenues dans un fichier `.kcr`.

$$\langle Program\rangle ::= (\langle Decl\rangle)^+$$

Une *déclaration* est soit une définition de *composant, de type, de configuration, de paquet* ou une *ouverture de paquet*.

$$\langle Decl\rangle ::= \langle CDecl\rangle \mid \langle TDecl\rangle \mid \langle Conf\rangle \mid \langle PkgDecl\rangle \mid \langle OpenDecl\rangle$$

Un *paquet* est un contenant permettant d'encapsuler un ensemble de définitions et donc de structurer les modèles. Soit $pid \in PkgId$ alors :

$$\langle PkgDecl\rangle ::= \texttt{package}\ pid\ (\langle Decl\rangle)^+\ \texttt{end}$$

L'*ouverture de paquet* importe tous les identificateurs de composant, types et configurations définis dans le paquet. Le système d'import n'est pas récursif, seules les définitions du paquet sont introduites. De plus si le paquet importé contient une définition dont l'identificateur est déjà utilisé par une définition de l'environnement courant alors la définition courante est masquée par celle de l'import. Soit $pid \in PkgId$ alors :

$$\langle OpenDecl\rangle ::= \texttt{open}\ pid\ ;$$

7.1.2 Déclaration de type (*TDecl*)

Une *déclaration de type* définit un ensemble de modes de défaillance pouvant survenir sur les composants modélisés. Cette notion de type permet de rassembler les modes de défaillance pertinents pour une famille de composants.

$$\langle TDecl\rangle ::= \texttt{type}\ t := \{NEList(v)\}\ ;$$

Exemple 7.1 (*TDecl*)**.** *Les composants du système* Rosace *peuvent produire les modes de défaillance* erroné *(ERR) et* perte *(LOST). Par ailleurs un composant électrique peut produire les modes de défaillance* pas de tension *(NT) et surtension (ST). Ainsi, le type* `t` *(respectivement* `t'`*) définit l'ensemble de modes de défaillance de* Rosace *(respectivement d'un composant électrique) comme suit :*

```
1 type t := {ERR, LOST};
2 type t' := {ST, NT};
```

7.1.3 Déclaration de composant (*CDecl*)

Une *déclaration de composant* définit les modes de défaillance produits sur les sorties du composant en fonction des modes de défaillance perçus sur les entrées du composant et des événements de défaillance du composant.

Le mot-clé `primary` est ajouté si le composant est atomique c'est-à-dire s'il ne contient pas de sous-composants. Soit $t \in \mathit{TypeId}$, $v \in \mathit{ValId}$, $\mathit{name} \in \mathit{CompId}$, $e \in \mathit{EvtId}$ alors :

$$\begin{aligned}\langle \mathit{CDecl} \rangle \quad ::= \quad & [\texttt{primary}]\ \texttt{comp}\ \mathit{name}\ (\mathit{List}(\langle \mathit{FDecl} \rangle))\ \texttt{returns}\ (\mathit{NEList}(\langle \mathit{FDecl} \rangle))\ \{ \\ & [\ \texttt{evts} : \mathit{NEList}(e)\ ;\ ] \\ & [\ \texttt{locs} : \mathit{NEList}(\langle \mathit{FDecl} \rangle)\ ;\ ] \\ & \texttt{defs} : (\langle \mathit{FDef} \rangle\ ;)^{+}\}\ ;\end{aligned}$$

Une déclaration de composant contient :
- les entrées du composant,
- les sorties du composant,
- les événements de défaillance du composant dans le champ `evts`,
- des variables locales (champ `locs`),
- la définition des sorties et des variables locales (champ `defs`).

Exemple 7.2 (*CDecl*)**.** *Les filtres de* ROSACE *sont des composants atomiques soumis aux événements e et l dont l'occurrence produit un comportement erroné ou de perte. Ainsi en* KCR*, un filtre possède :*
- *une entrée* `in` *de type* `t`*,*
- *une sortie* `out` *de type* `t`*,*
- *deux événements de défaillance* `e` *et* `l`*,*
- *la définition des modes de défaillance produits par la sortie en fonction de l'occurrence de* `e` *et* `l`*.*

```
1 primary comp Filter(in:t) returns (out: t) {
2   evts: e,l;
3   defs:
4     out := if e then ERR
5            else if l then LOST
6            else in;
7 };
```

7.1.4 Déclaration de flot (*FDecl*)

Les entrées, sorties et variables d'un composant sont appelées *variables de flots* ou plus simplement *flots*. Un *flot* f est décrit par une *déclaration de flot* donnant les modes de défaillance pouvant survenir sur f. Une déclaration de flot associe donc un type à f :

$$\langle \mathit{FDecl} \rangle ::= f : t$$

7.1.5 Définition de flot (*FDef*)

Un flot f est ensuite défini par une *expression de flot* F ou bien par une *instanciation de composant* (obligatoirement nommée lors de l'instanciation). Soit $f \in \mathit{FlowId}$, $t \in \mathit{TypeId}$, $c \in \mathit{CompId}$, $id \in \mathit{InstId}$ alors :

$$\langle \mathit{FDef} \rangle ::= f := \langle F \rangle \mid \mathit{NEList}(f) := c\ @\ id\ (\mathit{List}(f))$$

Une variable de flot est une variable contenant un *ensemble de modes de défaillance* (potentiellement vide), c'est-à-dire un élément de l'ensemble des parties de son type. La valeur $empty \in ValId$ représente le cas où un flot ne transporte aucun mode de défaillance c'est-à-dire que le flot n'est pas défaillant.

Une définition de flot par instanciation d'un composant (identifiée par l'identificateur *id*) consiste à assigner les sorties de l'instance à un n-uplet de variables de flots (un flot par sortie). Une définition de flot par expression consiste simplement à assigner la valeur de l'expression au flot.

Exemple 7.3 (*FDef*)**.** *Le système* ROSACE *est défini en* KCR *comme une interconnexion de ses composants. Pour cela, la définition des flots est donnée par* instanciation *de composant comme montré dans le code* KCR *suivant. Prenons la définition de* `Vaf`*, celle-ci indique que* `Vaf` *est défini par le mode de défaillance retourné par l'instance* `FVa` *d'un filtre* `Filtre`*. Notons que l'on considère que les filtres reçoivent des données correctes en entrées.*

```
comp Rosace() returns (delta_ec, delta_thc: t) {
  locs: Vaf,Vzf,qf, hf, azf: t;
  defs:
    Vaf := Filter@FVa(empty);
    hf := Filter@Fh(empty);
    azf := Filter@Faz(empty);
    Vzf := Filter@FVz(empty);
    qf := Filter@Fq(empty);

    delta_ec := LawVa@CVa(Vaf,Vzf,qf);
    delta_thc := LawVz@CVz(hf,azf,Vzf,qf);
};
```

7.1.6 Expression de flot (F)

Les opérateurs utilisables au sein d'une expression de flot sont :

1. `inter`, `union`, $?, =$ respectivement l'intersection, l'union, l'inclusion et l'égalité sur les ensembles ;
2. le branchement conditionnel (if-then-else) sur des conditions booléennes où les expressions booléennes sont des événements de défaillance du composant, le test d'appartenance à un ensemble, l'égalité/inclusion d'ensembles ou les connecteurs booléens classiques $\#, \&, !$ respectivement le OU, ET et NON logique ;

Pour des raisons de lisibilité, une valeur $v \in ValId$ représente
– un mode de défaillance appartenant à l'ensemble donné par le type de v ;
– mais aussi le singleton $\{v\}$ lorsqu'il est utilisé dans une expression de flot.
Soit $e \in EvtId$, $v \in ValId$, $f \in FlowId$, alors une expression de flot est définie comme suit :

$$\begin{array}{lcl} \langle F \rangle & ::= & v \mid f \mid (\langle F \rangle) \mid \langle F \rangle \texttt{ union } \langle F \rangle \mid \langle F \rangle \texttt{ inter } \langle F \rangle \mid \texttt{if } \langle B \rangle \texttt{ then } \langle F \rangle \texttt{ else } \langle F \rangle \\ \langle B \rangle & ::= & e \mid \langle F \rangle = \langle F \rangle \mid \langle B \rangle \ \& \ \langle B \rangle \mid \langle B \rangle \ \# \ \langle B \rangle \mid !\langle B \rangle \mid (\langle B \rangle) \mid \langle F \rangle \ ? \ \langle F \rangle \end{array}$$

Exemple 7.4 (F)**.** *Soit* $e \in EvtId$, $ERR \in ValId$ *et* $f_1, f_2 \in FlowId$*, alors une expression de flot valable pour* KCR *est :*

```
if e # in=ERR then f1 union f2
```

Fournir un opérateur d'union sur les flots (monotone par rapport à l'inclusion d'ensemble) permet de modéliser les systèmes cohérents en 1) donnant les règles de production des différents modes de défaillance séparément, 2) en les fusionnant dans une sortie à l'aide de l'union (modélisation proche de celle de HIPHOPS). Par ailleurs, l'opérateur if-then-else permet de décrire des comportements où les modes de défaillance sont mutuellement exclusifs (modélisation proche d'ALTARICA).

7.1.7 Définition de la configuration (*Conf*)

Les exigences de sûreté de fonctionnement, les paramètres des analyses et les alternatives des composants d'un problème DSE sont décrits par une *configuration* identifiée par un nom `cid`. Plus précisément, une configuration définit :

- `root` : le système à analyser (appelé *composant racine*) c'est-à-dire l'architecture initiale utilisée par le problème DSE ;
- `failure condition` : l'événement redouté comme une expression booléenne sur les sorties du système à analyser ;
- `duration` : le temps d'exposition du système ;
- `minorder` : l'ordre minimal de la solution ;
- `minreliability` : la fiabilité minimale de la solution pour le temps d'exposition donné ;
- `design space` : l'ensemble des alternatives des instances de composant du système définissant l'espace des architectures du problème DSE.

Les instances de composant sont localisées dans l'*arbre d'instances* par rapport au composant racine `root` par les identificateurs *Path*. Un nœud de l'arbre correspond à une instance de composant et un arc correspond à une instanciation d'un composant (nœud d'arrivée) au sein d'une instance (nœud de départ). Formellement, un arbre d'appel est défini comme l'ensemble d'arcs :

$$CT = \{(V_1, V_2) \in InstId^2 \mid V_2 \text{ appelle } V_1\}$$

Un identificateur *Path* est alors la représentation syntaxique d'un chemin dans l'arbre d'instances c'est-à-dire une liste d'identificateurs d'instances de composant séparés par un point. Notons que le composant racine n'est pas une instance, donc les chemins sont donnés relativement à celui-ci (c'est-à-dire que le nom du composant racine n'apparaît pas dans les chemins). Le mot clé `init` identifie alors la version initiale des composants du système où le composant donné dans la configuration doit être le même que celui donné lors de l'instanciation.

Le composant racine ne doit pas avoir d'entrées (c'est-à-dire que les systèmes analysés sont clos *cf* section 3.1.5). L'événement redouté n'est exprimé que sur les sorties du composant `root`.

Soit $rootId \in CompId$, $e \in EvtId$, $name \in InstId, cid \in ConfId$ alors :

$$
\begin{array}{lcl}
\langle Conf \rangle & ::= & \texttt{config}\ cid\ \{ \\
& & \quad \texttt{root} := rootId; \\
& & \quad \texttt{failure condition} := B; \\
& & \quad \texttt{duration} := Int\ ; \\
& & \quad [\texttt{minorder} := Int\ ;] \\
& & \quad [\texttt{minreliability} := Real\ ;] \\
& & \quad \texttt{design space}\ \{ \\
& & \qquad List(\langle Path \rangle \rightarrow \{\texttt{init}\ cid(NEList(e = Real)), \\
& & \qquad\qquad NEList(cid(NEList(e = Real))\} \\
& & \quad \}; \\
& & \}; \\
\\
\langle Path \rangle & ::= & name\ (.\ name)^*
\end{array}
$$

Exemple 7.5 (*Conf*)**.** *Considérons le problème* DSE *suivant :*

- *l'architecture initiale est donnée par le composant* `Rosace` *;*
- *l'événement redouté est* « l'une des sorties de `Rosace` n'est pas correcte » *;*
- *le temps d'exposition est de une heure ;*
- *l'ordre de la solution doit être supérieur ou égal à 2 ;*
- *la fiabilité de la solution doit être supérieure ou égale à 0.999999999 ;*
- *les composants de* ROSACE *possèdent deux versions avec différents taux de défaillance des événements et peuvent être dupliqués ou tripliqués (on considère toutes les combinaisons possibles des deux versions des composants dans les patrons de réplication).*

La configuration pour ce problème DSE *est donnée par le listing 7.1. Par souci de lisibilité nous n'affichons pas l'ensemble des alternatives pour chaque composant. Néanmoins on voit ici que l'instance* `FVa` *peut être remplacée par une version où* $\lambda_e = \lambda_l = 0.001$*, une version où* $\lambda_e = \lambda_l = 0.0005$ *ou encore par une duplication.*

```
config Conf {
  root := C;
  failure condition:= (delta_ec != empty) # (delta_thc != empty);
  duration := 1;
  minorder:=2;
  minreliability:=0.999999999;
  design space{
    FVa -> {
      init Filter(e=0.001, l=0.001),
      Filter(e=0.0005, l=0.0005),
      DupFilter(e1=0.001,e2=0.01,l1=0.001,l2=0.001),
      ...
    }...
  }
};
```

Listing 7.1 – Configuration de ROSACE

7.2 Sémantique

Avant d'exposer la sémantique de KCR, notons que les séparateurs des éléments des listes SMT-Lib est « ␣ » , donc par la suite nous écrivons $NEList(x)$ (respectivement $List(x)$) pour $NEList(x,\ll\,␣\,\gg)$ (respectivement $List(x,\ll\,␣\,\gg)$). La sémantique de KCR est définie par traduction vers la logique UFBVexprimée en SMT-Lib. La fonction de traduction est nommée Tr.

Rappelons que les déclarations de paquets et les ouvertures de paquets définissent des contenants et importent des définitions dans l'environnement lexical d'un programme KCR. Ainsi, la sémantique d'un programme KCR est la sémantique de ses définitions de types, de composants et de configurations. Soit $Program = TDecl_1 \cdots TDecl_n\ CDecl_1 \cdots CDecl_m\ Conf_1 \cdots Conf_k$ alors :

$$Tr(Program) \triangleq Tr(TDecl_1) \cdots Tr(TDecl_n)\ Tr(CDecl_1) \cdots Tr(CDecl_m)\ Tr(Conf_1) \cdots Tr(Conf_k)$$

7.2.1 Traduction d'un type ($TDecl$)

Un type `t` est traduit comme un type de vecteurs de bits de taille $card(t)$. Chaque constante v_i de type `t` est traduite comme un vecteur de bits constant où tous les bits hormis le ième sont mis à zéro. La valeur `empty` est traduite comme un vecteur où tous les bits sont à zéro. Soit $TDecl$ = `type` $t := \{v_1, \cdots, v_n\}$; où $v_i \in ValId$, $t \in TypeId$ et x_i la représentation chaîne de caractères de taille n représentant 0 si $i = 0$ et la valeur binaire 2^{i-1} sinon , alors :

$$\begin{array}{rcl} Tr(TDecl) & \triangleq & (\texttt{declare-sort}\ t\ (\_\ \texttt{BitVec}\ n)) \\ & & (\texttt{define-const}\ empty\ Tr(t)\ \#\text{b}x_0) \\ & & (\texttt{define-const}\ v_1\ Tr(t)\ \#\text{b}x_1) \\ & & \cdots(\texttt{define-const}\ v_n\ Tr(t)\ \#\text{b}x_n) \end{array}$$

Exemple 7.6 (Traduction d'un type)**.** *Reprenons le type* `t` *de* Rosace *contenant les modes ERR et LOST, celui-ci est alors traduit comme une sorte de vecteurs de bits de taille 2. De plus chaque mode est traduit comme un vecteur de bits constant de sorte* `t` *valant 01 (*`#b01` *en* SMT-Lib*) pour ERR, 10 (*`#b10` *en* SMT-Lib*) pour LOST. Finalement la valeur empty est un vecteur de bits constant de sorte* `t` *valant 00 (*`#b00` *en* SMT-Lib*).*

```
1 (define-sort t () (_ BitVec 2))
2 (define-const ERR t #b01)
3 (define-const LOST t #b10)
4 (define-const empty t #b00)
```

7.2.2 Traduction d'un composant ($CDecl$)

Une déclaration de composant est traduite comme une fonction interprétée prenant en entrée :
- un n-uplet de vecteurs de bits correspondant aux flots d'entrée du composant,
- un n-uplet de vecteurs de booléens correspondant aux événements de défaillance du composant.

Cette fonction produit un n-uplet de vecteurs de bits correspondant aux flots de sortie du composant.

Si la déclaration d'un composant contient des instanciations de composant (c'est-à-dire possède des sous-composants) alors le rang de la fonction est étendu récursivement avec les événements de défaillance de ses sous-composants. Afin d'éviter les conflits d'identificateurs, les événements sont

renommés en utilisant une fonction *rnm* qui préfixe l'identificateur d'événement par l'identificateur de l'instance dont il est issu c'est-à-dire $rnm(\texttt{compId@id}, \texttt{e}) = \texttt{id.e}$. Soit le composant

$$
\begin{array}{l}
\texttt{comp}\ name\ (FDecl_1, \cdots, FDecl_n)\ \texttt{returns}\ (FDecl_1, \cdots, FDecl_m)\ \{ \\
\quad \texttt{evts} : e_1, \cdots, e_k\ ; \\
\quad \texttt{locs} : FDecl_1, \cdots, \cdots, FDecl_l\ ; \\
\quad \texttt{defs} : FDef_1; \cdots FDef_{m+l}; \\
\}\ ;
\end{array}
$$

Soit $List((Tr(e')\ \texttt{Bool}))$ la liste des identificateurs d'événements extraits des sous-composants de *name*, `TupleM` la sorte des n-uplets de taille m et $t_1, \cdots, t_m$ les sortes des sorties $o_1, \cdots, o_m$ définies dans les déclarations $\langle FDecl \rangle_1, \cdots, \langle FDecl \rangle_m$ alors :

$$
\begin{array}{lll}
Tr(CDecl) & \triangleq & (\texttt{define-fun}\ name \\
& & \left.\begin{array}{l} (Tr(FDecl_1)\ \cdots\ Tr(FDecl_n) \\ (Tr(e_1)\ \texttt{Bool})\ \cdots\ (Tr(e_k)\ \texttt{Bool}) \\ List((Tr(e')\ \texttt{Bool}))) \end{array}\right\} \text{entrées} \\
& & (\texttt{TupleM}\ Tr(t_1) \cdots Tr(t_m)) \quad \}\text{sorties} \\
& & (Tr(FDef_1; \cdots FDef_{m+l};)) \quad \}\text{corps}
\end{array}
$$

Dans le langage KCR, notons que les déclarations $FDecl_1, \cdots, FDecl_l$ des flots locaux sont seulement utilisées pour assurer que les définitions correspondantes sont correctement *typées* c'est-à-dire que la définition d'un flot local produit bien des modes de défaillance du type donné dans sa déclaration. Si ce n'est pas le cas, une erreur de typage est remontée à l'utilisateur, autrement celles-ci sont ignorées dans la traduction.

Exemple 7.7 (Traduction d'un composant)**.** *Le composant atomique* `Filtre` *de* ROSACE *possède une entrée, une sortie de type* `t` *et deux événements de défaillance. Comme ce composant n'a qu'une seule sortie, il n'est pas nécessaire d'utiliser la sorte n-uplet en sortie du composant. La fonction interprétée de* `Filtre` *est alors :*

```
(define-fun Filter
  ((in t) (e Bool) (l Bool))
  t
  (let ((out (ite e ERR (ite l LOST in))))
    out))
```

7.2.3 Traduction d'une déclaration de flot $(FDecl)$

La sémantique d'une déclaration de flot est une association d'un type et d'un identificateur de flot. Soit $FDecl = f : t$ où $f \in FlowId,\ t \in TypeId$ alors :

$$Tr(FDecl) \triangleq (Tr(f)\ Tr(t))$$

7.2.4 Traduction d'une définition de flot $(FDef)$

Soit $FDef_1; \cdots FDef_n;$ où $FDef_i = f_i{:=}F_i;$, $f_i \in FlowId$, F_i est une expression de flot et `mkTupleM` la fonction de construction des n-uplets de taille m. Considérons que les m dernières

définitions sont les définitions des sorties du composant alors :

$$\begin{array}{rcl} Tr(FDef_1;\cdots FDef_n;) & \triangleq & (\texttt{let } Tr(FDef_1) \\ & & \cdots(\texttt{let } Tr(FDef_n) \\ & & (\texttt{mkTupleM } Tr(f_{n-m})\cdots Tr(f_n))\cdots) \end{array}$$

Si la définition de flot est donnée par une expression de flot F. Soit $f := F$ alors :

$$Tr(FDef) \triangleq (Tr(f)\ Tr(F))$$

Sinon la définition est donnée par l'instanciation d'un composant où un identifiant de flot est donné pour chaque sortie du composant. Soit

- $FDef = List(f) := c\ @\ name\ (List(f'))$ où $f, f' \in FlowId,\ c \in CompId$;
- $I = List(Tr(f'))\ List(Tr(e'))$ où $List(e')$ sont les événements extraits des sous-composants de $name$;
- i est la position de f dans $List(f)$ alors :

$$Tr(FDef) \triangleq List((Tr(f)\ (\texttt{field}_\texttt{i}\ (Tr(c)\ I))))$$

7.2.5 Traduction d'une expression de flot (F)

Une expression de flot peut contenir des opérations sur les modes de défaillance et sur les occurrences des événements de défaillance exprimées comme des expressions booléennes. Rappelons qu'un mode de défaillance (à ne pas confondre avec les événements de défaillance vus comme des constantes booléennes non interprétées) est encodé comme un vecteur de bits avec un unique bit à 1. Par conséquent, un ensemble de modes de défaillance est encodé par un vecteur de bits où le ième bit est vrai si et seulement si le ième mode de défaillance du type associé appartient à l'ensemble. L'union (respectivement l'intersection) de deux ensembles est encodée par un OU bit-à-bit (`bvor`) (respectivement le ET bit-à-bit `bvand`). Soit $e \in EvtId,\ v \in ValId,\ f \in FlowId,\ t \in TypeId,\ c \in CompId,\ f \in FlowId$ alors :

$$\begin{array}{lcl} Tr(v) & \triangleq & \texttt{v de sorte } Tr(t) \\ Tr(c) & \triangleq & \texttt{c un identificateur de fonction} \\ Tr(e) & \triangleq & \texttt{e de sorte Bool} \\ Tr(t) & \triangleq & \texttt{t identificateur de sorte vecteur de bits} \\ Tr(f) & \triangleq & \texttt{f de sorte } Tr(t) \\ Tr(!\ B) & \triangleq & (\texttt{not } Tr(B)) \\ Tr(B_1\ \&\ B_2) & \triangleq & (\texttt{and } Tr(B_1)\ Tr(B_2)) \\ Tr(B_1\ \#\ B_2) & \triangleq & (\texttt{or } Tr(B_1)\ Tr(B_2)) \\ Tr(F_1\ \texttt{union}\ F_2) & \triangleq & (\texttt{bvor } Tr(F_1)\ Tr(F_2)) \\ Tr(F_1 = F_2) & \triangleq & (= Tr(F_1)\ Tr(F_2)) \\ Tr(F_1\ \texttt{inter}\ F_2) & \triangleq & (\texttt{bvand } Tr(F_1)\ Tr(F_2)) \\ Tr(F_1\ ?\ F_2) & \triangleq & Tr((F_1 \cap F_2) = F_1) \\ Tr(\texttt{if } B\ \texttt{then } F_1\ \texttt{else } F_2) & \triangleq & (\texttt{ite } Tr(B)\ Tr(F_1)\ Tr(F_2)) \end{array}$$

Exemple 7.8. *Reprenons le système* ROSACE *pour illustrer la définition de flot par instanciation. Ce système produit deux sorties et possède des sous-composants, donc le rang de la fonction interprétée doit être étendu avec les événements de ses sous-composants. La traduction du système*

Rosace *est :*

```
(define-fun Rosace
  ;; entrees
  ((FVa.e Bool) (FVa.l Bool) (Fh.e Bool) (Fh.l Bool) (Faz.e Bool) (Faz.l Bool)
  (FVz.e Bool) (FVz.l Bool) (Fq.e Bool) (Fq.l Bool) (CVa.e Bool) (CVa.l Bool)
  (CVz.e Bool) (CVz.l Bool))

  ;;sorties
  (Tuple2 t t)

  ;;corps
  (let
    ((Vaf (Filter empty FVa.e FVa.l)) (hf (Filter empty Fh.e Fh.l))
     (azf (Filter empty Faz.e Faz.l)) (Vzf (Filter empty FVz.e FVz.l))
     (qf (Filter empty Fq.e Fq.l)))
     (let
      ((delta_ec (LawVa Vaf Vzf qf CVa.e CVa.l))
      (delta_thc (LawVz hf azf Vzf qf CVz.e CVz.l )))
      (mkTuple2 delta_ec delta_thc))))
```

7.2.6 Traduction d'une configuration (*Conf*)

Soit une configuration nommée *cid* et $o_1, \cdots, o_n$ les sorties du composant `root` de la configuration alors le champ `failure condition` est traduit comme un prédicat sur les sorties du système comme suit :

$$Tr(\texttt{failure condition} := B) \triangleq \begin{array}{l}(\texttt{define-fun cid.fail} \\ ((Tr(o_1)\ Tr(t_1)) \cdots (Tr(o_n)\ Tr(t_n))) \\ \texttt{Bool}\ Tr(B))\end{array}$$

Exemple 7.9. *Le prédicat encodant l'événement redouté de la configuration de l'exemple 7.5 est :*

```
(define-fun Conf.fail
  ((delta_ec t)(delta_thc t))
  Bool
  (or (not (= delta_ec empty)) (not (= delta_thc empty))))
```

La traduction des champ `duration`, `minreliability`, `minorder` et `design space` (correspondants aux entrées du problème DSE) génère l'encodage du problème DSE comme un problème SMT. Néanmoins nous ne donnons pas ici la traduction de ces champs car celle-ci repose sur l'utilisation de prédicats et de sortes d'une théorie spécifique appelée *Safety* que nous présentons dans le chapitre 10.

7.3 Résumé

Nous avons défini la syntaxe du langage KCR et sa sémantique par traduction vers la logique UFBV exprimée dans le langage SMT-Lib. Nous allons maintenant utiliser cette traduction pour développer un ensemble de méthodes d'analyse de la sûreté de fonctionnement des systèmes KCR basées sur la résolution de problèmes SMT.

Chapitre 8

Analyses de sûreté de fonctionnement sur les modèles KCR

Ce chapitre développe un ensemble de méthodes de calcul des indicateurs de sûreté. Comme illustré par la figure 8.1, ces méthodes, publiées dans [36], sont basées sur la résolution de problèmes SMT construits à partir du modèle KCR traduit en SMT-Lib. Pour cela, les sections 8.1, 8.2, 8.3, 8.4 et 8.5 introduisent les problèmes SMT et algorithmes calculant les indicateurs qualitatifs et quantitatifs classiques de la sûreté de fonctionnement (*cf* 2.1.4). De plus nous établissons dans la section 8.6 les avantages et les limitations de ces méthodes de calcul pour évaluer les architectures candidates lors de la résolution d'un problème DSE.

8.1 Fonction de structure

Les modèles de la fonction de structure φ d'un système représentent les occurrences d'événements de défaillance du système pour lesquelles l'événement redouté se produit. Or par traduction d'un programme KCR, nous disposons de :

- la fonction `rootId` donnant les modes de défaillance observés sur les sorties du système à analyser en fonction de ses événements de défaillance ;
- la fonction `cid.fail` encodant l'événement redouté de la configuration *cid* à analyser, où l'événement redouté est exprimé sur les modes de défaillance perçus sur les sorties du système.

Soit $\texttt{field}_\texttt{i}$ les fonctions permettant d'accéder à la ième composante d'un tuple, m le nombre de sorties du composant racine `rootId` et n le nombre d'événements de défaillance de `rootId`, alors la fonction SMT-Lib de φ est :

```
(define-fun phi ((e1 Bool) ... (en Bool))
                Bool
                (cid.fail (field1 (rootId e1 ... en))
                          ...(fieldm (rootId e1 ... en))))
```

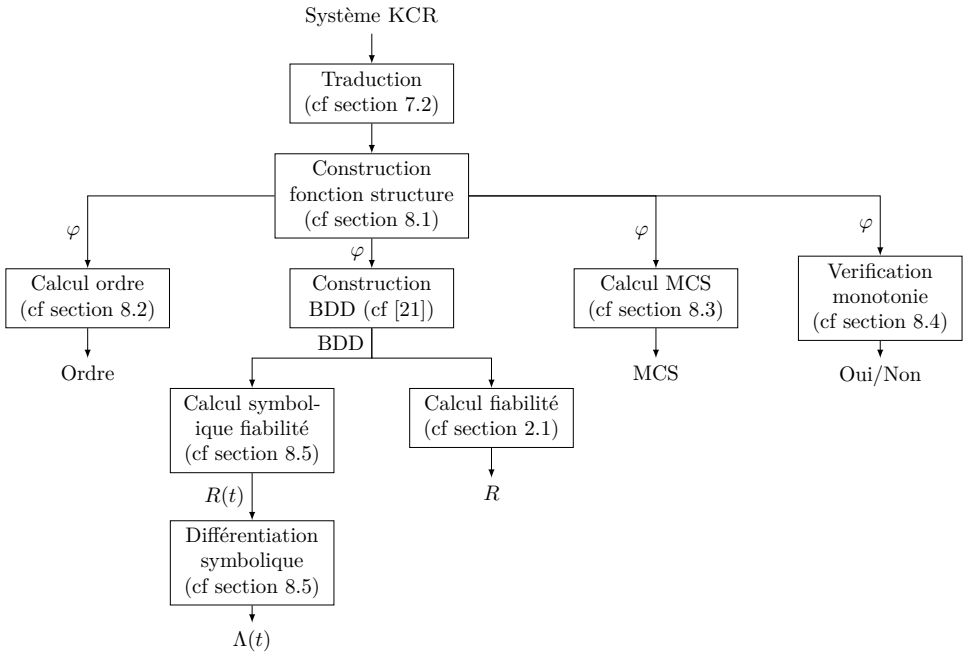


FIGURE 8.1 – Processus de calcul des indicateurs

Si l'on souhaite synthétiser φ comme une formule sur les événements de `rootId` alors il suffit de demander à un solveur SMT (comme Z3) de trouver le modèle d'une fonction non-interprétée satisfaisant la définition 8.1.

Définition 8.1 (Fonction de structure)**.** *Soit* `rootId` $= Tr($`rootId`$)$ *et* `cid.fail` $= Tr(failure := B)$ *alors la fonction de structure* `phi` *est donnée par la résolution du problème suivant :*

```
(declare-fun phi (Bool ... Bool) Bool)
(assert (forall ((e1 Bool) ... (en Bool))
    (= (phi e1 ... en)
       (cid.fail (field1 (rootId e1 ... en))...(fieldm (rootId e1 ... en))))))
```

Exemple 8.1 (Fonction de structure)**.** *Pour l'exemple* ROSACE *où seuls* C_{Va} *et* F_{Va} *peuvent défaillir, la fonction de structure synthétisée par le problème précédent est donnée dans le listing suivant :*

```
(define-fun phi
          ((CVa.e Bool)(CVa.l Bool)(FVa.e Bool)(FVa.l Bool))
          Bool
          (or CVa.e CVa.l FVa.e FVa.l ))
```

8.2 Ordre

Vérification de l'ordre Les exigences de sûreté imposent en général une borne minimale sur l'ordre du système, comme par exemple l'absence de points de défaillance uniques pour une fonction critique. Ainsi, nous proposons une méthode permettant de vérifier que l'ordre du système est bien supérieur ou égal à une borne donnée. Par ailleurs, cette vérification n'entraîne pas un calcul explicite de l'ordre, ni des coupes minimales. Pour cela encodons le problème de vérification comme un problème SMT.

En effet, le problème de vérification revient à prouver qu'il n'existe aucune combinaison d'événements de défaillance contenant au plus $k-1$ événements et déclenchant l'événement redouté. L'encodage de ce problème est basé sur un opérateur comptant le nombre de booléens à vrai parmi un ensemble de booléens. De nombreux encodages de cet opérateur sont proposés dans la littérature notamment dans [40].

Définition 8.2 (Vérification de l'ordre)**.** *Soit* `phi` *la fonction de structure sur les événements* $e_1, \cdots, e_n$, $\texttt{atMost}_{\texttt{k-1}}$ *un encodage de l'opérateur de cardinalité et k l'exigence sur l'ordre. Alors le système est conforme si et seulement si le problème suivant est* UNSAT.

```
(declare-const e1 Bool)...(declare-const en Bool)
(assert (phi e1 ... en))
(assert (atMost_{k-1} e1 ... en))
```

Exemple 8.2 (Vérification de l'ordre)**.** *Considérons une contrainte d'ordre de 2. Le problème de vérification est* SAT *et le modèle renvoyé est* $\{C_{Va}.e \mapsto \mathbf{T}\}$. *En effet, si* $C_{Va}.e$ *se produit alors les commandes* δ_{thc} *envoyées par* ROSACE *sont erronées. Il existe donc un point de défaillance unique ce qui viole la contrainte d'ordre comme nous l'avions identifié dans l'exemple 2.11.*

Calcul de l'ordre Rappelons que l'ordre est la taille minimale des coupes minimales d'un système. La méthode consiste simplement à résoudre le problème de vérification précédent en itérant sur la borne k à partir de $k = 0$. Chaque itération comporte les étapes suivantes :

- si le problème de vérification est UNSAT pour le k courant, alors il n'existe pas de combinaisons contenant au plus k événements déclenchant l'événement redouté, l'ordre du système est donc supérieur à k. La méthode recommence alors la résolution pour $k + 1$;
- sinon il existe une combinaison contenant k événements déclenchant l'événement redouté, or la résolution du problème à l'itération précédente prouve qu'il n'existe pas de combinaisons d'événements contenant au plus $k - 1$ événements, par conséquent la méthode s'arrête et l'ordre du système est k.

Exemple 8.3 (Calcul de l'ordre)**.** *Dans le cas de* ROSACE *:*

- *pour* $k = 0$, *le problème de vérification est* UNSAT, *la méthode continue*
- *pour* $k = 1$, *le problème est* SAT, *l'ordre de* ROSACE *est donc 1.*

Notons que les problèmes précédents peuvent être encodés dans la logique propositionnelle ou le formalisme pseudo-booléen en traduisant la fonction de structure φ comme une formule CNF à l'aide de la transformation de Tseitin [94]. Il est alors possible d'utiliser des solveurs SAT ou pseudo-booléens comme SAT4J [11] pour résoudre ces problèmes. Les avantages des solveurs SAT par rapport aux solveurs SMT sont notamment :

- énumération de modèles gérée par le solveur (`sharpCDCL` [56]) ;
- possibilité de parallélisation (`clingo` [44]).

8.3 Coupes minimales

La plupart des analyses de sûreté de fonctionnement des systèmes statiques sont basées sur l'analyse des coupes minimales. Ainsi, nous proposons une méthode d'énumération des coupes minimales basée sur la résolution de problèmes SAT. Nous supposons que l'utilisateur spécifie une borne k sur le nombre maximum d'événements contenus dans une coupe minimale.

La méthode de calcul ENUMERATECUTSETS est présentée dans l'algorithme 1 et réutilise le problème de vérification de l'ordre présenté précédemment. La méthode consiste à itérer sur la taille $i \in [0, k]$ des coupes recherchées, en calculant pour chaque i toutes les coupes possédant exactement i événements. Pour cela chaque itération contient les étapes suivantes :

- la borne i contenue dans la contrainte $\Sigma_j e_j \leq i$ sur les événements e_j du système est mise à jour dans le problème de vérification courant pour calculer les coupes de taille i (ligne 5) ;
- puis chaque modèle trouvé par le solveur est restreint à ses littéraux positifs (c'est-à-dire que les valuations $e \mapsto \mathbf{F}$ sont ignorées) et sauvegardé comme une coupe (ligne 7).
- la négation du modèle restreint est ensuite ajoutée au problème courant (ligne 9) empêchant le solveur de trouver deux fois le même modèle (on parle de *bloquer* le modèle).
- les modèles sont énumérés jusqu'à ce que le problème soit UNSAT (ligne 6) ; la borne i est ensuite incrémentée (ligne 4) et l'itération est de nouveau exécutée jusqu'à ce que i atteigne la borne k fixée par l'utilisateur.

Les modèles réduits M produits par l'algorithme sont des coupes minimales car pour un modèle donné :

- M déclenche l'événement redouté
- tout sous-ensemble d'événements de M ne peut pas déclencher l'événement redouté. En effet si un sous-ensemble d'événements déclenchait l'événement redouté, alors celui-ci aurait été trouvé et bloqué aux itérations précédentes.

Algorithme 1 Algorithme d'énumération des coupes minimales

```
1:  procedure enumerateCutSets(cardPb, k)
2:      result ← ∅
3:      problem ← cardPb
4:      for i ∈ [0, k] do
5:          problem ←replaceCard(problem, i)
6:          while solver.checkSat(problem) = SAT do
7:              cut ←restrictPositive(solver.getModel)
8:              result ← result ∪ cut
9:              problem ←addClause(problem, Not(cut))
10:         end while
11:     end for
12:     return result
13: end procedure
```

8.4 Vérification de la monotonie

La monotonie (ou cohérence) du système revient à vérifier la monotonie de la fonction de structure φ [95]. Nous proposons de vérifier la monotonie à l'aide d'une résolution de problème SMT

comme suit.

Définition 8.3 (Vérification de la monotonie)**.** *Soit* `phi` *la fonction de structure du système alors le système est monotone si et seulement si le problème suivant est* UNSAT*.*

```
(assert (exists ((e1 Bool) ... (en Bool) (e'1 Bool) ... (e'n Bool))
  (and (and (=> e1 e'1) ... (=> en e'n))
       (not (=> (phi e1 ... en) (phi e'1 ... e'n))))))
```

Exemple 8.4 (Vérification de la monotonie)**.** *Pour la fonction de structure de l'exemple 8.1, le problème est* UNSAT*,* ROSACE *simplifié est donc monotone.*

8.5 Analyses quantitatives

Les indicateurs quantitatifs de sûreté sont évalués à l'aide des méthodes classiques présentées dans le chapitre 2.1. Plus précisément le calcul de la fiabilité est basée sur la construction et l'analyse du BDD de la fonction de structure du système KCR comme décrit par le processus de la figure 8.1.

Par ailleurs, le taux de défaillance est obtenu par différentiation symbolique de la formule de la fiabilité. Pour cela la méthode SYMBR présentée par l'algorithme 2, génère la formule symbolique de la fiabilité par parcours du BDD de φ. Soit N le nœud racine du BDD de φ et L une association telle que $L(e)$ renvoi la valeur du taux de défaillance de l'événement e donné par l'utilisateur alors $\text{SYMBR}(N, L)$ génère la formule de la fiabilité $R(t)$. Puis des librairies de différentiation symbolique comme SYMPY [91] peuvent être utilisées pour générer la formule du taux de défaillance et calculer sa valeur pour un temps d'opération donné.

Algorithme 2 Méthode de génération de la formule symbolique de la fiabilité

function $\text{SYMBR}(N, L)$
  N **match**
    case `BDD`$[\mathbf{F}] \Rightarrow$ **return** 0
    case `BDD`$[\mathbf{T}] \Rightarrow$ **return** 1
    case $v \wedge F|_{v=\mathbf{T}} \vee \neg v \wedge F|_{v=\mathbf{F}} \Rightarrow$
      $f_1 \leftarrow \text{SYMBR}(F|_{v=\mathbf{T}}, L)$
      $f_0 \leftarrow \text{SYMBR}(F|_{v=\mathbf{F}}, L)$
      $t_1 \leftarrow$ **if** $f_1 = 0$ **then** 0 **else** $(1 - e^{L(v).t}).f_1$ **end if**
      $t_0 \leftarrow$ **if** $f_0 = 0$ **then** 0 **else** $e^{L(v).t}.f_1$ **end if**
      return $t_1 + t_0$
end function

8.6 Limitations des analyses classiques pour l'exploration

Les analyses de sûreté de fonctionnement que nous avons présentées dans ce chapitre ainsi que les analyses existantes présentées dans la section 2.1 sont basées sur l'analyse des combinaisons d'événements déclenchant l'événement redouté. Or nous montrons dans cette section que ce type

d'analyses ne permet pas d'évaluer efficacement la conformité des candidats lors de l'exploration de l'espace des architectures.

8.6.1 Analyses basées sur les événements de défaillance

Les indicateurs de sûreté sont généralement définies par rapport aux événements de défaillance (par exemple la probabilité d'occurrence de l'événement redouté, nombres d'occurrences d'événements avant défaillance). Les analyses classiques consistent alors à construire une structure de données représentant les combinaisons d'événements menant à l'événement redouté (arbre de défaillance, BDD, diagrammes de fiabilité, etc) pour pouvoir en déduire la valeur des indicateurs.

Exemple 8.5 (Combinaison d'événements)**.** *Reprenons* ROSACE*, pour des raisons de lisibilité, considérons que seuls C_{Va} et F_{Va} peuvent produire des données erronées ou perdues comme représenté par la figure 8.2. Alors les combinaisons d'événements déclenchant la perte des ordres δ_{thc} sont $\{\neg C_{Va}.e, C_{Va}.l\}$ et $\{\neg F_{Va}.e, F_{Va}.l\}$.*

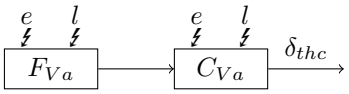


FIGURE 8.2 – ROSACE simplifié

8.6.2 Analyses et substitution

Pour l'analyse d'un unique système, la construction de ces structures de données est en effet judicieuse car elles permettent d'évaluer les indicateurs de sûreté. Néanmoins lors de l'exploration, la méthode d'évaluation doit analyser de nombreux candidats obtenus par substitution des composants de l'architecture initiale par d'autres composants possédant le même interface. Il serait donc pertinent de pouvoir réutiliser les analyses effectuées sur l'architecture initiale pour analyser les candidats. Or les événements de défaillance, les règles de propagation et de génération des modes de défaillance des alternatives ne sont pas les mêmes que ceux du composant initial. Par conséquent, les combinaisons d'événements menant à l'événement redouté sont différentes de celles de l'architecture initiale. Or comme les structures de données classiques représentent ces combinaisons d'événements, celles-ci doivent être recalculées après chaque substitution. Cette reconstruction systématique des structures de données limite donc l'efficacité de la méthode d'exploration quelle que soit la méthode d'exploration employée.

Exemple 8.6 (Impact substitution)**.** *Considérons que le contrôleur C_{Va} est remplacé par un patron de duplication présenté dans l'exemple 3.2 comme montré dans la figure 8.3. Dans ce cas les combinaisons d'événements déclenchant la perte des ordres δ_{thc} sont $\{C_{Va1}.e\}, \{C_{Va1}.l\}, \{C_{Va2}.e\}, \{C_{Va2}.l\}$ et $\{\neg F_{Va}.e, F_{Va}.l\}$.*

Exemple 8.7 (Impact substitution)**.** *La figure 8.4 illustre l'impact de l'application du patron de duplication sur C_{Va} sur le BDD de* ROSACE*.*

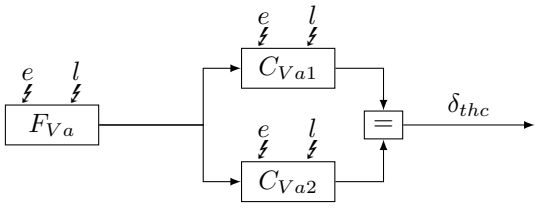


FIGURE 8.3 – ROSACE après application de la duplication

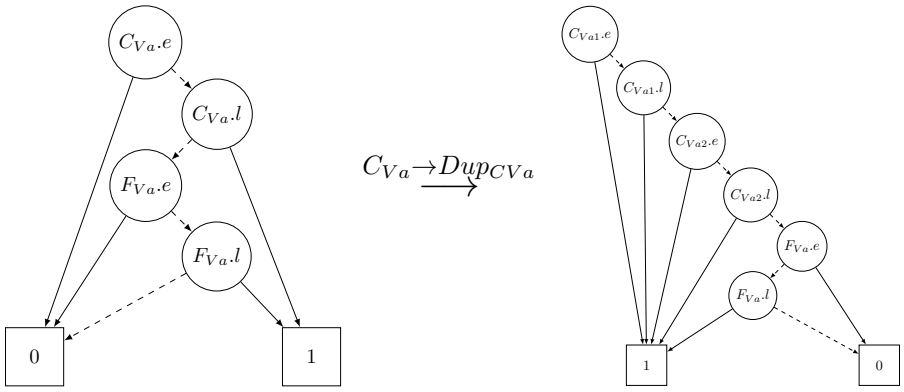


FIGURE 8.4 – Impact de la substitution sur le BDD de ROSACE

8.7 Résumé

Nous avons développé, dans ce chapitre, un ensemble de méthodes de calcul des indicateurs de sûreté basées sur la résolution de problèmes SAT et SMT. De plus, nous avons présenté une méthode permettant de vérifier la monotonie d'un système. Néanmoins nous avons montré que ces méthodes, tout comme les méthodes existantes, ne sont pas adaptées à l'évaluation des candidats lors de l'exploration de l'espace des architectures. En effet, ces méthodes nécessitent une analyse complète pour chaque nouveau candidat à évaluer. Ainsi, nous introduisons, dans le chapitre suivant, une nouvelle structure de données et de nouvelles méthodes d'analyse permettant d'évaluer les indicateurs de tous les candidats en réutilisant les analyses de l'architecture initiale.

Chapitre 9

Analyses de sûreté pour l'exploration

L'approche de résolution du problème DSE pourrait réutiliser les analyses introduites dans le chapitre 8. Malheureusement, nous avons établi précédemment qu'une exploration basée sur ces analyses souffrirait d'une grande inefficacité. C'est pourquoi nous introduisons, dans ce chapitre, une nouvelle forme d'analyse des systèmes, présentée dans [37], permettant de calculer efficacement les indicateurs de sûreté lors d'une exploration de l'espace des architectures. Pour cela, la section 9.1 introduit informellement les concepts sur lesquels reposent ces analyses et leurs avantages par rapport aux approches classiques. Par la suite, nous introduisons dans la section 9.2, un interpréteur du langage KCR que nous utilisons, dans les sections 9.3 et 9.4 pour décrire formellement les analyses. Nous expliquons notamment en quoi ces dernières permettent de calculer les indicateurs de sûreté et montrons l'équivalence entre les analyses introduites dans ce chapitre et les analyses classiques. Finalement dans la section 9.5, nous introduisons une nouvelle structure de données appelée STMDD que nous utilisons dans la section 9.6 pour calculer les indicateurs de sûreté de tout système appartenant à l'espace des architectures d'un problème DSE.

9.1 Idée générale

Plutôt que de réévaluer les indicateurs de sûreté de fonctionnement sur chaque candidat, nous proposons une nouvelle méthode, illustrée par la figure 9.2, fondée sur l'analyse des modes de défaillance observés sur les interfaces des composants du système (que nous appelons *traces système*). À la différence des méthodes classiques, l'objet permettant de calculer les indicateurs (appelé MDD des traces système ou plus simplement STMDD) peut être généré sans connaître les substitutions opérées. Ces dernières ne sont utilisées qu'au moment du calcul des indicateurs. Par conséquent le STMDD généré n'a pas besoin d'être recalculé pour évaluer les indicateurs de sûreté.

Exemple 9.1 (Trace versus événements)**.** *Considérons l'architecture de* Rosace *simplifiée où seuls C_{Va} et F_{Va} peuvent défaillir et que F_{Va} est modélisé comme un filtre sans entrées (car son entrée est toujours correcte). La figure 9.3 représente alors un comportement dysfonctionnel où la sortie δ_{ec} est erronée. La représentation classique de ce scénario est $\{F_{Va}.e\}$, qui devient $(F_{Va}, \{out \mapsto ERR\}), (C_{Va}, \{in \mapsto ERR,\ out \mapsto ERR\})$ dans nos analyses.*

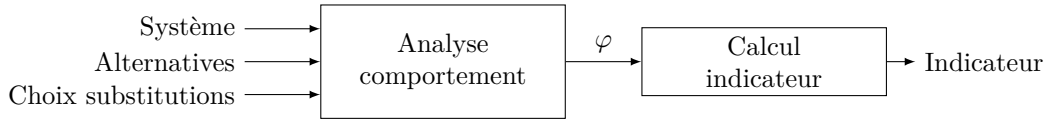


FIGURE 9.1 – Analyse classique (basée événement)

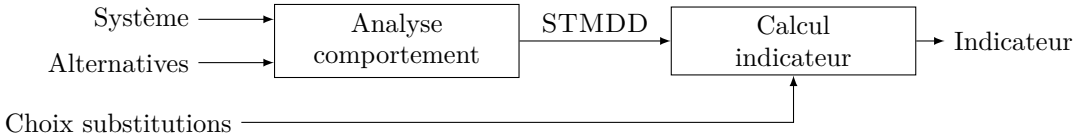


FIGURE 9.2 – Analyse basée trace

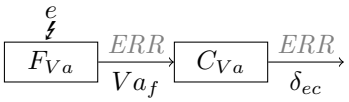


FIGURE 9.3 – Trace versus événements

Dans les sections suivantes, nous présentons les différentes étapes et les outils, représentés par la figure 9.4, permettant de construire le STMDD d'un système :

Interprétation La construction du STMDD est basée sur deux analyses parallèles, une concernant le système et l'autre concernant les alternatives considérées dans le problème DSE. Pour ces deux analyses, la première étape consiste à calculer les modes de défaillance pouvant être produits sur les flots internes (entrées, sorties et flots locaux) du système (respectivement des alternatives). Ce calcul est effectué par un interpréteur du langage KCR présenté dans la section 9.2.

Calcul des traces composant Nous introduisons alors, dans la section 9.3, la notion de *trace de composant* c'est-à-dire les modes de défaillance observés sur l'interface d'un composant. Ainsi, pour chaque alternative, ses traces de composant (notées *traceOf*) sont construites à partir des interprétations de ses flots calculées par l'interpréteur. Par ailleurs les combinaisons des événements de l'alternative déclenchant une trace donnée sont déduites des interprétations et encodées sous la forme de BDDs (notés *trig*). Notons qu'une alternative est analysée en isolation, c'est-à-dire que les traces et les BDDs sont calculés lorsque les entrées de l'alternative sont libres.

Caractérisation des traces composant Dans un second temps nous caractérisons, dans la section 9.3.2, les traces de composant en définissant et calculant la *probabilité d'occurrence* (notée $prob(tr)$) et *l'ordre* (noté $order(tr)$) d'une trace tr à partir des BDDs calculés précédemment. Ces indicateurs *locaux* nous permettrons de calculer la fiabilité et l'ordre d'un système.

Construction du STMDD Nous formalisons, dans la section 9.4, le concept de traces système et montrons que si pour tout composant du système, et pour chacune de ses alternatives, les traces de l'alternative sont contenues dans celles du composant initial (propriété d'inclusion) alors les traces système permettent de calculer la fonction de structure de tout système obtenu par substitution des composants du système initial. Par conséquent, l'analyse des

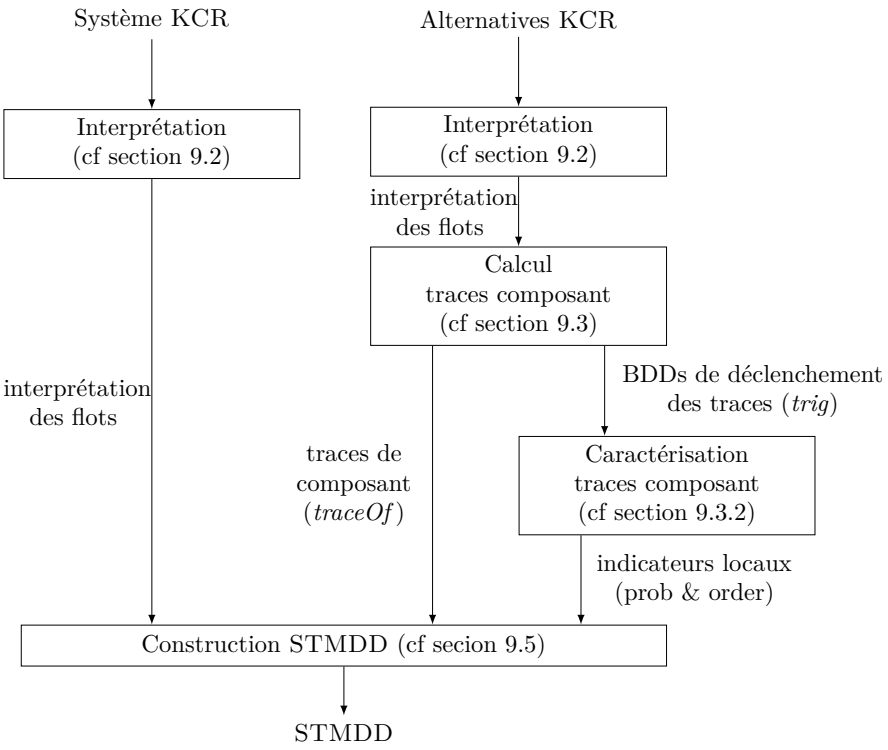


FIGURE 9.4 – Analyse du comportement basée trace

traces permet de calculer exactement les indicateurs de sûreté de tout candidat de l'espace des architectures d'un problème DSE. Pour mener ces analyses nous développons, dans la section 9.5, un encodage des traces système comme un Diagramme de Décision Multi-valué (STMDD) à partir des traces composant des alternatives, de leurs indicateurs locaux et de l'interprétation des flots du système.

Une fois le STMDD construit, nous montrons, dans le section 9.6, comment calculer exactement la fiabilité et l'ordre de n'importe quel candidat de l'espace des architecture à partir du STMDD du système initial et du choix des substitutions opérées sur les composants du système initial (cf figure 9.5). Ces calculs sont basés sur une traversée du STMDD, leur complexité est donc linéaire dans la taille du STMDD. Ceci nous permet d'assurer que le temps de calcul des indicateurs ne dépend pas du choix de substitution, ce qui fournit une méthode d'évaluation efficace des candidats lors de la résolution d'un problème DSE.

La propriété d'inclusion, nécessaire à l'évaluation des indicateurs par STMDD, pourrait limiter la portée de l'approche. Néanmoins, puisque les alternatives d'un composant sont des instanciations de patrons de sûreté, celles-ci doivent mitiger la production de certains modes de défaillance ou diminuer leur probabilité d'occurrence mais ne produisent pas de nouveaux modes de défaillance. Autrement dit, nous considérons par la suite que l'ensemble des modes de défaillance observables sur une instance de patron est un sous-ensemble de ceux produits pas le composant initial. Nous montrons, dans la section 9.3.2, comment assurer formellement que les alternatives vérifient cette

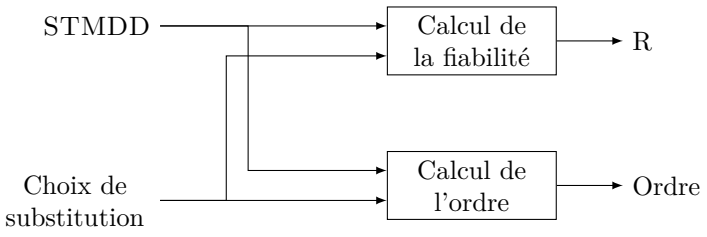


FIGURE 9.5 – Calcul de la fiabilité et de l'ordre par analyse du STMDD

hypothèse.

Hypothèse 9.1 (Inclusion)**.** *L'ensemble des modes de défaillance observables sur une instance de patron est un sous-ensemble de ceux produits par le composant initial.*

Par ailleurs, rappelons qu'un système est une interconnexion d'instances de composant atomique et non atomique. Afin de simplifier les analyses, nous considérons que la hiérarchie du système a été *aplatie* en *inlinant* les définitions des instances de composant non-atomique au sein du système.

Hypothèse 9.2 (Aplatissement)**.** *Un système est une interconnexion d'instances de composant atomique.*

9.2 Interpréteur du langage KCR

Puisque nous disposons de la traduction SMT du système, nous pourrions énumérer les traces système à l'aide d'un solveur, c'est-à-dire générer les valeurs possibles des flots du système lorsqu'un ensemble arbitraire d'événements se produit. Néanmoins, nous avons constaté en pratique que cette énumération est coûteuse en temps de calcul et ne peut être appliquée pour analyser des systèmes comme ROSACE.

Pour pallier ce problème, nous proposons de définir un interpréteur de KCR. Sa particularité est d'interpréter un flot comme un ensemble de couples (v, ϕ) où v est une valeur possible du flot et ϕ est l'encodage BDD des combinaisons d'événements pour lesquelles cette valeur est produite. Puisque les systèmes de KCR sont clos, l'information contenue dans ϕ est suffisante pour calculer ces ensembles de valeurs pour les différents opérateurs des expressions de flots de KCR, à l'aide des règles de calcul présentées dans la section 9.2.3. Ainsi, au lieu d'énumérer les valeurs possibles d'un flot f, celles-ci sont construites à partir de l'évaluation, par l'interpréteur, de l'expression définissant f.

Notons que définir cet interpréteur revient à décrire une sémantique opérationnelle pour KCR distincte de la sémantique obtenue par traduction du modèle en SMT-LIB. Par faute de temps, nous ne donnons pas la preuve d'équivalence entre ces deux sémantiques. Néanmoins nous montrons, dans la section 9.2.4, que pour un système donné, prouver pour toute valuation de ses événements que les fonctions MSFOL obtenues par traduction des définitions des flots du système produisent les mêmes valeurs que celles calculées par l'interpréteur revient à résoudre un problème SMT.

9.2.1 Valeurs d'interprétation des expressions de KCR

Soit $\mathcal{BDD}$ l'ensemble des BDDs pouvant être construits sur les événements de défaillance du système, $\mathcal{P}(A)$ l'ensemble des parties d'un ensemble A, alors les interprétations possibles d'une expression KCR prenant une valeur dans un ensemble X sont

$$\mathcal{S}_X = \mathcal{P}(\{(v, \phi) | v \in X, \phi \in \mathcal{BDD}\})$$

Le langage KCR contient deux types d'expressions :

Expressions de flots rappelons qu'en KCR, un flot $f : t$, où t est un type défini comme `type` $t = \{c_1, \cdots, c_n\}$, peut prendre pour valeur tout ensemble A tel que $A \subseteq \{c_1, \cdots, c_n\}$. Autrement dit A appartient à l'ensemble $\mathcal{P}(\{c_1, \cdots, c_n\})$ que nous notons simplement t. Son interprétation est donc une valeur $S \in \mathcal{S}_t$.

Expressions booléennes dans ce cas l'interprétation est une valeur $S \in \mathcal{S}_{\mathbb{B}}$ où $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$.

Pour des raisons de lisibilité, nous ne ferons pas de distinctions entre une formule booléenne et son BDD. Nous ne différencions donc pas les opérateurs booléens (conjonction, disjonction et négation) de leurs opérateurs correspondants sur les BDDs.

9.2.2 Fonctions de simplification, de combinaison et projection des interprétations

Afin d'alléger l'écriture des règles d'interprétation, nous introduisons une fonction de simplification, de combinaison et de projection.

Fonction de simplification notée $\texttt{simp} : \mathcal{S}_X \to \mathcal{S}_X$ construit une interprétation S' à partir d'une interprétation S. Pour tout $(v, \phi) \in S$, soit $A = \{(v, \phi') \in S\}$ alors :
- si $\bigvee_{(v,\phi)\in A} \phi \not\equiv \mathbf{F}$ alors $(v, \bigvee_{(v,\phi)\in A} \phi) \in S'$;
- sinon $(v, \phi) \notin S'$.

Fonction de combinaison notée $\texttt{apply} : \mathcal{S}_X^2 \times \mathcal{F}_{X^2 \to X'} \to \mathcal{S}_{X'}$ où $\mathcal{F}_{X^2 \to X'}$ représente l'ensemble des fonctions de $X^2 \to X'$. Soit $(S_1, S_2) \in \mathcal{S}_X^2$ et $f \in \mathcal{F}_{X^2 \to X'}$, alors la fonction `apply` construit la nouvelle interprétation S' de l'expression comme suit :

$$S' = \{(f(v_1, v_2), \phi_1 \wedge \phi_2) | (v_1, \phi_1) \in S_1, (v_2, \phi_2) \in S_2\}$$

Fonction de projection notée $\texttt{snd} : X \times Y \to Y$ renvoie simplement son deuxième paramètre, c'est-à-dire $\texttt{snd}(x, y) = y$.

9.2.3 Règles d'interprétation des expressions KCR

Les règles suivantes définissent l'interprétation des différents opérateurs contenus dans les expressions F et B (introduits dans la section 7.2.5) pour un environnement d'interprétation γ. Cet environnement contient un ensemble d'énoncés notés $x \mapsto S$ pour « l'interprétation de x est S ». Par ailleurs, puisque les identifiants des flots et événements sont locaux à un composant, la variable c correspond au composant au sein duquel les interprétations sont opérées. Ainsi, $(c, \gamma) \vdash x \mapsto S$ indique que l'interprétation de x dans le composant c pour l'environnement γ est S.

L'interprétation d'une constante de flot v est toujours l'ensemble $\{v\}$ quels que soit les événements de défaillance observés.

$$\text{Règle } v : \frac{}{(c,\gamma) \vdash v \mapsto \{(\{v\}, \mathbf{T})\}}$$

L'interprétation d'un événement e correspond aux deux valeurs possibles, **T** si e se produit, **F** si e ne se produit pas.

$$\text{Règle } e : \frac{}{(c,\gamma) \vdash e \mapsto \{(\mathbf{T}, e), (\mathbf{F}, \neg e)\}}$$

Pour les identificateurs de flot f, s'il existe une définition *FDef* de f au sein du composant c, c'est-à-dire $FDef \in Def_c$ alors l'interprétation est obtenue en interprétant la partie droite de la définition. Pour une définition de la forme $f{:=}F$ l'interprétation de f est celle de F.

$$\text{Règle } f : \frac{f{:=}F \in Def_c \qquad (c,\gamma) \vdash F \mapsto S}{(c,\gamma) \vdash f \mapsto S}$$

Pour une définition par instanciation $f_1, \cdots, f_{i-1}, f, f_{i+1}, \cdots, f_m{:=}c'@inst(f'_1, \cdots, f'_n)$ où le flot à interpréter est f alors l'interprétation est celle de la sortie o_i du composant c, interprétée sachant que l'interprétation des entrées $in_1, \cdots, in_n$ est celle des flots $f'_1, \cdots, f'_n$.

$$\text{Règle } inst : \frac{\begin{array}{c}(c,\gamma) \vdash f'_1 \mapsto S_1 \qquad \cdots \qquad (c,\gamma) \vdash f'_n \mapsto S_n \\ f_1, \cdots, f_{i-1}, f, f_{i+1}, \cdots, f_m{:=}c'@inst(f'_1, \cdots, f'_n) \in Def_c \quad (c', \{in_1 \mapsto S_1, \cdots, in_n \mapsto S_n\}) \vdash o_i \mapsto S\end{array}}{(c,\gamma) \vdash f \mapsto S}$$

Finalement l'identificateur peut être une entrée libre c'est-à-dire $f \in inputs_c$. Dans ce cas l'interpréteur crée toutes les valeurs possibles v que peut prendre le flot f. Puisque f est une entrée libre, sa valeur ne dépend d'aucun événement de défaillance, donc nous créons un ensemble d'événements factices $e_{f \mapsto v}$ modélisant le fait que « l'entrée f vaut v ». Puisqu'une entrée f d'un type t ne peut avoir qu'une seule valeur $v \in t$, le BDD associé à v est $e_{in \mapsto v} \wedge \bigwedge_{v' \in t, v \neq v'} \neg e_{in \mapsto v'}$. Lors de l'évaluation d'un système, cette règle ne s'applique jamais puisque les systèmes analysables par KCR sont clos. Autrement dit, toute entrée de composant est connectée à la sortie d'un autre, donc chaque évaluation d'un identificateur $f \in inputs_c$ est précédée par l'application d'une règle *inst* qui aura assigné une interprétation à toutes les entrées de c. Cependant, cette règle permet d'analyser un composant en isolation c'est-à-dire où ses entrées sont libres.

$$\text{Règle } in : \frac{f \in inputs_c}{(c,\gamma) \vdash f \mapsto \{(v, \phi) | v \in t, \phi = e_{in \mapsto v} \wedge \bigwedge_{v' \in t, v \neq v'} \neg e_{in \mapsto v'}\}}$$

Exemple 9.2 (Interprétation d'une entrée)**.** *Prenons l'entrée in d'un composant LawVa de* ROSACE *simplifié (cf exemple 9.1), où in est libre. Alors son interprétation est donnée par la dérivation suivante :*

$$\frac{in \in inputs_{LawVa}}{(LawVa,\gamma) \vdash in \mapsto \left\{ \begin{array}{llllllll} (\{\emptyset\}, & e_{in\mapsto\emptyset} & \wedge & \neg e_{in\mapsto ERR} & \wedge & \neg e_{in\mapsto LOST} & \wedge & \neg e_{in\mapsto\{ERR,LOST\}}), \\ (\{ERR\}, & \neg e_{in\mapsto\emptyset} & \wedge & e_{in\mapsto ERR} & \wedge & \neg e_{in\mapsto LOST} & \wedge & \neg e_{in\mapsto\{ERR,LOST\}}), \\ (\{LOST\}, & \neg e_{in\mapsto\emptyset} & \wedge & \neg e_{in\mapsto ERR} & \wedge & e_{in\mapsto LOST} & \wedge & \neg e_{in\mapsto\{ERR,LOST\}}), \\ (\{ERR, LOST\}, & \neg e_{in\mapsto\emptyset} & \wedge & \neg e_{in\mapsto ERR} & \wedge & \neg e_{in\mapsto LOST} & \wedge & e_{in\mapsto\{ERR,LOST\}}) \end{array} \right\}}$$

Soit $\neg, \vee, \wedge$ les opérateurs classiques de négation, de disjonction et de conjonction définis sur les booléens et $\cup, \cap, \subset, =$ les opérateurs classiques d'union, d'intersection, d'inclusion et d'égalité sur les ensembles. Alors l'interprétation des différents opérateurs de KCR consiste à appliquer la fonction de combinaison avec l'opérateur approprié.

$$\text{Règle } \# : \frac{(c,\gamma) \vdash B_1 \mapsto S_1 \qquad (c,\gamma) \vdash B_2 \mapsto S_2}{(c,\gamma) \vdash B_1 \ \# \ B_2 \mapsto \texttt{simp}(\texttt{apply}(S_1, S_2, \vee))}$$

$$\text{Règle } \& : \frac{(c,\gamma) \vdash B_1 \mapsto S_1 \qquad (c,\gamma) \vdash B_2 \mapsto S_2}{(c,\gamma) \vdash B_1 \ \& \ B_2 \mapsto \texttt{simp}(\texttt{apply}(S_1, S_2, \wedge))}$$

$$\text{Règle } ! : \frac{(c,\gamma) \vdash B \mapsto S}{(c,\gamma) \vdash !B \mapsto \{(v, \neg\phi) | (v, \phi) \in S\}}$$

$$\text{Règle } ? : \frac{(c,\gamma) \vdash F_1 \mapsto S_1 \qquad (c,\gamma) \vdash F_2 \mapsto S_2}{(c,\gamma) \vdash F_1 \ ? \ F_2 \mapsto \texttt{simp}(\texttt{apply}(S_1, S_2, \subset))}$$

$$\text{Règle } \texttt{union} : \frac{(c,\gamma) \vdash F_1 \mapsto S_1 \qquad (c,\gamma) \vdash F_2 \mapsto S_2}{(c,\gamma) \vdash F_1 \ \texttt{union} \ F_2 \mapsto \texttt{simp}(\texttt{apply}(S_1, S_2, \cup))}$$

$$\text{Règle } \texttt{inter} : \frac{(c,\gamma) \vdash F_1 \mapsto S_1 \qquad (c,\gamma) \vdash F_2 \mapsto S_2}{(c,\gamma) \vdash F_1 \ \texttt{inter} \ F_2 \mapsto \texttt{simp}(\texttt{apply}(S_1, S_2, \cap))}$$

$$\text{Règle } = : \frac{(c,\gamma) \vdash F_1 \mapsto S_1 \qquad (c,\gamma) \vdash F_2 \mapsto S_2}{(c,\gamma) \vdash F_1 \ = \ F_2 \mapsto \texttt{simp}(\texttt{apply}(S_1, S_2, =))}$$

Pour donner l'interprétation d'un `if` B `then` F_1 `else` F_2, nous devons donner les règles d'interprétation pour les différentes interprétations possibles de la condition booléenne B. Or pour toute interprétation $S \in \mathcal{S}_X$, la règle de simplification assure que le nombre de couples contenus dans S est inférieur au nombre de valeurs contenues dans X. Pour toute expression booléenne B, il n'existe donc que trois ensembles de couples :

1. Pour B toujours vraie : $S = \{(\mathbf{T}, \mathbf{T})\}$,
2. Pour B toujours fausse : $S = \{(\mathbf{F}, \mathbf{T})\}$,
3. Pour B parfois vraie, parfois fausse : $S = \{(\mathbf{T}, \phi_{\mathbf{T}}), (\mathbf{F}, \phi_{\mathbf{F}})\}$.

Si la condition B est toujours vraie (respectivement toujours fausse), alors l'interprétation est celle de F_1 (respectivement de F_2). Autrement l'interprétation du *if-then-else* est l'union de celle de F_1 lorsque B est vraie et de celle de F_2 lorsque B est fausse.

$$\text{Règle } \texttt{if then else} : \frac{(c,\gamma) \vdash B \mapsto \{(\mathbf{T}, \phi_{\mathbf{T}}), (\mathbf{F}, \phi_{\mathbf{F}})\} \qquad (c,\gamma) \vdash F_1 \mapsto S_1 \qquad (c,\gamma) \vdash F_2 \mapsto S_2}{\begin{array}{l}(c,\gamma) \vdash \texttt{if} \ B \ \texttt{then} \ F_1 \ \texttt{else} \ F_2 \mapsto \texttt{simp}(\quad \texttt{apply}(\{(\mathbf{T}, \phi_{\mathbf{T}})\}, S_1, \texttt{snd}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cup \ \texttt{apply}(\{(\mathbf{F}, \phi_{\mathbf{F}})\}, S_2, \texttt{snd}))\end{array}}$$

$$\frac{(c,\gamma) \vdash B \mapsto \{(\mathbf{T}, \mathbf{T})\} \qquad (c,\gamma) \vdash F_1 \mapsto S_1 \qquad (c,\gamma) \vdash F_2 \mapsto S_2}{(c,\gamma) \vdash \texttt{if} \ B \ \texttt{then} \ F_1 \ \texttt{else} \ F_2 \mapsto F_1}$$

$$\frac{(c,\gamma) \vdash B \mapsto \{(\mathbf{F},\mathbf{T})\} \qquad (c,\gamma) \vdash F_1 \mapsto S_1 \qquad (c,\gamma) \vdash F_2 \mapsto S_2}{(c,\gamma) \vdash \texttt{if } B \texttt{ then } F_1 \texttt{ else } F_2 \mapsto F_2}$$

Exemple 9.3 (Interprétation). *Prenons l'expression* `if` e `then` ERR `else` $LOST$*, l'interprétation dans l'environnement* $\gamma = \emptyset$ *pour un composant* c *est* $\{(\{ERR\}, e), (\{LOST\}, \neg e)\}$ *comme montré par la dérivation suivante :*

$$\frac{\overline{(c,\emptyset) \vdash e \mapsto \{(\mathbf{T},e),(\mathbf{F},\neg e)\}} \qquad \overline{(c,\emptyset) \vdash ERR \mapsto \{(\{ERR\},\mathbf{T})\}} \qquad \overline{(c,\emptyset) \vdash LOST \mapsto \{(\{LOST\},\mathbf{T})\}}}{(c,\emptyset) \vdash \texttt{if } e \texttt{ then } ERR \texttt{ else } LOST \mapsto \{(\{ERR\},e),(\{LOST\},\neg e)\}}$$

9.2.4 Équivalence entre interpréteur et traduction SMT

Pour un système s donné, nous souhaitons vérifier que les règles de traduction, données dans la section 7.2, et les règles d'interprétation définies précédemment, produisent les mêmes modes de défaillance sur les flots locaux et de sortie de s pour toutes valuations des événements de s. Pour cela, nous définissons le prédicat MSFOL `is_a_tr` prenant en entrée une valuation E des événements et une valuation F des variables locales et sorties de s. `is_a_tr` est vrai si et seulement si la valuation F est bien égale à celle produite par les fonctions MSFOL obtenues par traduction des définitions de flot de s.

Définition 9.1. *(* `is_a_tr`*) Soit un système* s*,* $e_1, \cdots, e_k$ *la liste des événements extraits des sous-composants de* s*,* $t_1, \cdots, t_n$ *les sortes (respectivement* $MSFOLDef_1 = Tr(FDef_1)$*,* $\cdots$*,* $MSFOLDef_n = Tr(FDef_n)$ *la traduction des définitions) des flots locaux et de sortie* $f_1, \cdots, f_n$ *de* s *alors* `is_a_tr` *est :*

```
(define-fun is_a_tr ((f_1' t1) ... (f_n' tn)
                     (e_1 Bool) ... (e_k Bool))
                     Bool
                     (= (mkTupleN f_1' ... f_n')
                        (let (MSFOLDef1 ... MSFOLDefn)
                             (mkTupleN f_1 ... f_n))))
```

De plus nous disposons d'un prédicat `is_an_interp` prenant en entrée une valuation E des événements et une valuation F des variables locales et sorties de s. `is_an_interp` est vrai si et seulement si la valuation F a été calculée par l'interpréteur pour E.

Définition 9.2. *(* `is_an_interp`*) Soit un système* s*,* $e_1, \cdots, e_k$ *la liste des événements extraits des sous-composants de* s*,* $t_1, \cdots, t_n$ *les sortes (respectivement* $S_1 = \{(v_{1,1}, \phi_{1,1}), \cdots, (v_{1,n}, \phi_{1,m})\}$*,* $\cdots$*,* $S_n = \{(v_{n,1}, \phi_{n,1}), \cdots, (v_{n,m'}, \phi_{n,m'})\}$ *les interprétations) des flots locaux et de sortie* $f_1, \cdots, f_n$ *de* s*,* `toFormula` *la fonction transformant un* BDD *en formule* SMT-LIB *et* `toBV` *la fonction transformant un ensemble de modes de défaillance en vecteur de bits alors* `is_an_interp` *est :*

```
(define-fun is_an_interp ((f_1' t1) ... (f_n' tn)
                          (e_1 Bool) ... (e_k Bool))
                          Bool
                          (and (= f_1 (ite (toFormula phi_11) (toBV v_11)
                                           (ite (toFormula phi_12) (toBV v_12)
                                           ...
                                           (ite (toFormula phi_1{m-1}) (toBV v_1{m-1})
```

```
                                        v1m) ...)
                                  ...
                              (= f_n (ite (toFormula phi_n1) (toBV v_n1)
                                          (ite (toFormula phi_n2) (toBV v_n2)
                                          ...
                                          (ite (toFormula phi_n{m'-1}) (toBV v_n{m'-1})
                                          v_nm') ...)
```

Soit $e_1, \cdots, e_k$ les événements , $f_1, \cdots, f_n$ les flots locaux et de sortie du système et t_{f_i} la sorte du flot f_i alors la traduction SMT et le simulateur sont équivalents si et seulement si le problème du listing 9.1 est UNSAT.

```
(declare-const e1 Bool)...(declare-const ek Bool)
(declare-const f1 t_f1)...(declare-const fn t_fn)
(assert (not (= (is_a_tr f1...fn e1...ek ) (is_an_interp f1...fn e1...ek))))
```

Listing 9.1 – Problème d'équivalence

Exemple 9.4 (Vérification de l'équivalence)**.** *Le problème d'équivalence pour la version simplifiée de* ROSACE *est donné par le listing 9.2 qui est* UNSAT.

```
;; evenements de rosace simplifie
(declare-const FVa.e Bool) (declare-const FVa.l Bool) (declare-const CVa.e Bool)
(declare-const CVa.l Bool)


;; variables de flots de rosace
(declare-const Vaf t) (declare-const delta_ec t)

;; valeurs produites par traduction smt
(define-fun is_a_tr ((Vaf' t) (delta_ec' t) (FVa.e Bool) (FVa.l Bool)
                     (CVa.e Bool) (CVa.l Bool))
                    Bool
                    (= (mkTuple2 Vaf' delta_ec')
                       (let ((Vaf (Filter empty FVa.e FVa.l))
                             (delta_ec (LawVa Vaf CVa.e CVa.l)))
                             (mkTuple2 Vaf delta_ec))))

;; valeurs produites par l'interpreteur
(define-fun is_an_interp ((Vaf t) (delta_ec t) (FVa.e Bool) (FVa.l Bool)
                          (CVa.e Bool) (CVa.l Bool))
                         Bool
                         (and
                           (= Vaf (ite FVa.e ERR
                                       (ite (and FVa.l (not FVa.e)) LOST empty)))
                           (= delta_ec (ite (or CVa.e (and FVa.e (not CVa.l))) ERR
                                            (ite (and (not CVa.e)
                                                      (or CVa.l (and FVa.l (not FVa.e))))
                                                 LOST
                                                 empty)))))

;; contrainte d'equivalence
(assert (not (= (is_an_interp Vaf delta_ec FVa.e FVa.l CVa.e CVa.l)
                (is_a_tr Vaf delta_ec FVa.e FVa.l CVa.e CVa.l))))
```

Listing 9.2 – Problème d'équivalence pour ROSACE

9.3 Traces composant

9.3.1 Définition et génération

Une *trace composant* est une valuation des modes de défaillance observés sur les entrées et sorties du composant, étudié en isolation, et provoquée par une ou plusieurs combinaisons d'événements. Autrement dit, les valeurs de la valuation doivent être contenues dans les interprétations des flots d'entrée/sortie calculées par l'interpréteur et se produire simultanément.

Définition 9.3 (Traces composant)**.** *Soit $in_1, \cdots, in_n$ les entrées et $o_1, \cdots, o_m$ les sorties d'un composant c. On note pour toute entrée/sortie f de c, S_f l'interprétation de f par l'interpréteur c'est-à-dire $(c, \emptyset) \vdash f \mapsto S_f$. Alors les traces de c sont :*

$$traceOf_c = \{\{in_1 \mapsto v_1, \cdots, o_m \mapsto v_{m+n}\} | (v_1, \phi_1), \cdots, (v_{n+m}, \phi_{n+m}) \in S_{in_1} \times \cdots \times S_{o_m}, \phi_1 \wedge \cdots \wedge \phi_{n+m} \not\equiv \mathbf{F}\}$$

Exemple 9.5 (Traces composant)**.** *Pour le composant Filtre de* Rosace *simplifié, nous avons $(Filtre, \emptyset) \vdash out \mapsto S_{out}$ où*

$$S_{out} = \{\ (\{\emptyset\}, \neg e \wedge \neg l),\quad (\{ERR\}, e),\quad (\{LOST\}, \neg e \wedge l)\ \}$$

Donc les traces de Filtre sont

$$traceOf_{Filtre} = \{\{out \mapsto ERR\}, \{out \mapsto LOST\}, \{out \mapsto \emptyset\}\}$$

Pour le composant LawVa de Rosace *simplifié nous avons $(LawVa, \emptyset) \vdash in \mapsto S_{in}$ et $(LawVa, \emptyset) \vdash out \mapsto S_{out}$ où*

$$S_{in} = \left\{ \begin{array}{lllllllll} (\{\emptyset\}, & e_{in\mapsto\emptyset} & \wedge & \neg e_{in\mapsto ERR} & \wedge & \neg e_{in\mapsto LOST} & \wedge & \neg e_{in\mapsto\{ERR,LOST\}}), \\ (\{ERR\}, & \neg e_{in\mapsto\emptyset} & \wedge & e_{in\mapsto ERR} & \wedge & \neg e_{in\mapsto LOST} & \wedge & \neg e_{in\mapsto\{ERR,LOST\}}), \\ (\{LOST\}, & \neg e_{in\mapsto\emptyset} & \wedge & \neg e_{in\mapsto ERR} & \wedge & e_{in\mapsto LOST} & \wedge & \neg e_{in\mapsto\{ERR,LOST\}}), \\ (\{ERR, LOST\}, & \neg e_{in\mapsto\emptyset} & \wedge & \neg e_{in\mapsto ERR} & \wedge & \neg e_{in\mapsto LOST} & \wedge & e_{in\mapsto\{ERR,LOST\}}) \end{array} \right\}$$

$$S_{out} = \left\{ \begin{array}{llllllllllllll} (\{\emptyset\}, & \neg l & \wedge & \neg e & \wedge & e_{in\mapsto\emptyset} & \wedge & \neg e_{in\mapsto ERR} & \wedge & \neg e_{in\mapsto LOST} & \wedge & \neg e_{in\mapsto\{ERR,LOST\}}), \\ (\{ERR\}, & e & \vee & (\neg l & \wedge & \neg e_{in\mapsto\emptyset} & \wedge & e_{in\mapsto ERR} & \wedge & \neg e_{in\mapsto LOST} & \wedge & \neg e_{in\mapsto\{ERR,LOST\}})), \\ (\{LOST\}, & \neg e & \wedge & (l & \vee & \neg e_{in\mapsto\emptyset} & \wedge & \neg e_{in\mapsto ERR} & \wedge & e_{in\mapsto LOST} & \wedge & \neg e_{in\mapsto\{ERR,LOST\}}))), \\ (\{ERR, LOST\}, & \neg e & \wedge & \neg l & \wedge & \neg e_{in\mapsto\emptyset} & \wedge & \neg e_{in\mapsto ERR} & \wedge & \neg e_{in\mapsto LOST} & \wedge & e_{in\mapsto\{ERR,LOST\}}) \end{array} \right\}$$

Donc les traces de LawVa sont

$$traceOf_{LawVa} = \left\{ \begin{array}{ll} \{in \mapsto \emptyset, out \mapsto \emptyset\}, & \{in \mapsto \emptyset, out \mapsto ERR\}, \\ \{in \mapsto \emptyset, out \mapsto LOST\}, & \{in \mapsto ERR, out \mapsto ERR\}, \\ \{in \mapsto ERR, out \mapsto LOST\}, & \{in \mapsto LOST, out \mapsto LOST\}, \\ \{in \mapsto LOST, out \mapsto ERR\}, & \{in \mapsto \{ERR, LOST\}, out \mapsto ERR\}, \\ \{in \mapsto \{ERR, LOST\}, out \mapsto LOST\}, & \{in \mapsto \{ERR, LOST\}, out \mapsto \{ERR, LOST\}\} \end{array} \right\}$$

Notons que $\phi_1 \wedge \cdots \wedge \phi_{n+m}$ encode les combinaisons d'événements pour lesquelles la trace se produit. Or comme un composant, étudié en isolation, n'est pas forcément clos, des événements n'appartenant pas aux événements de défaillance du composant ont été ajoutés lors de la construction de l'ensemble des valeurs possibles des entrées (cf règle *in* de 9.2.3). Nous définissons donc la fonction $trig_c(tr)$ donnant le BDD encodant les combinaisons d'événements d'un composant c produisant la trace tr sachant que les valeurs des entrées, fournies par la trace, se produisent.

Définition 9.4 (Occurrence de trace)**.** *Soit* $tr = \{in_1 \mapsto v_1, \cdots, o_m \mapsto v_{m+n}\}$*,* ϕ_i *le* BDD *des événements produisant* v_i*,* $e_{in \mapsto v}$ *l'événement introduit par l'interpréteur pour « la variable d'entrée in vaut* v *» et* $\texttt{restrict}(\phi, \{x_1 \mapsto y_1, \cdots, x_n \mapsto y_n\})$ *le BDD de* ϕ *où les variables* $x_1, \cdots, x_n$ *sont remplacées par les valeurs* $y_1, \cdots, y_n$ *(opération de restriction définie dans [21]) alors :*

$$trig_c(tr) = \begin{cases} \texttt{restrict}(& \phi_1 \wedge \cdots \wedge \phi_{n+m}, \\ & \{e_{in \mapsto v} = \mathbf{T} | in \mapsto v \in tr\} \cup \{e_{in \mapsto v} = \mathbf{F} | in \mapsto v \notin tr\}) \quad si\ tr \in traceOf_c \\ \mathbf{F} & sinon \end{cases}$$

Exemple 9.6 (Occurrence de trace)**.** *Pour le composant Filtre, prenons la trace* $tr = \{out \mapsto ERR\}$*, celle-ci est issue de l'interprétation* $(\{ERR\}, e)$ *et comme Filtre n'a pas d'entrée, les BDDs ne contiennent pas d'événements* $e_{in \mapsto v}$*, d'où :*

$$trig_{Filtre}(tr) = \texttt{restrict}(e, \emptyset) = e$$

Pour le composant LawVa de Rosace *simplifié, considérons la trace* $tr = \{in \mapsto \emptyset, out \mapsto \emptyset\}$*, cette trace est issue des interprétations*

- $(\{\emptyset\}, e_{in \mapsto \emptyset} \wedge \neg e_{in \mapsto ERR} \wedge \neg e_{in \mapsto LOST} \wedge \neg e_{in \mapsto \{ERR,LOST\}})$ *pour in*
- $(\{\emptyset\}, \neg l \wedge \neg e \wedge e_{in \mapsto \emptyset} \wedge \neg e_{in \mapsto ERR} \wedge \neg e_{in \mapsto LOST} \wedge \neg e_{in \mapsto \{ERR,LOST\}})$ *pour out*

Comme LawVa possède une entrée in, nous devons éliminer les événements $e_{in \mapsto v}$ *pour lesquels* $v \neq \emptyset$ *d'où la restriction suivante :*

$$\begin{array}{rcl} trig_c(tr) & = & \texttt{restrict}(\neg l \wedge \neg e \wedge e_{in \mapsto \emptyset} \wedge \neg e_{in \mapsto ERR} \wedge \neg e_{in \mapsto LOST} \wedge \neg e_{in \mapsto \{ERR,LOST\}}, \\ & & \{e_{in \mapsto LOST} = \mathbf{F}, e_{in \mapsto ERR} = \mathbf{F}, e_{in \mapsto \emptyset} = \mathbf{T}, e_{in \mapsto \{ERR,LOST\}} = \mathbf{F}\}) \\ & = & \neg e \wedge \neg l \end{array}$$

Donc sachant que $in = \emptyset$*, la trace* $\{in \mapsto \emptyset, out \mapsto \emptyset\}$ *se produit lorsque les événements e et l ne se produisent pas.*

9.3.2 Caractérisation

Les indicateurs de sûreté sont définis en fonction des occurrences d'événements de défaillance déclenchant un événement redouté. Par conséquent, nous devons redéfinir les méthodes de calcul des indicateurs à partir des traces composant.

Probabilité et ordre d'une trace Nous introduisons la notion de *probabilité d'une trace* $prob_c(tr)$ (respectivement *d'ordre* $order_c(tr)$) comme la probabilité qu'une des combinaisons d'événements encodées par $trig_c(tr)$ se produisent (respectivement la cardinalité minimale des coupes minimales de $trig_c(tr)$). Pour cela la probabilité de $trig_c(tr)$ est calculée par l'approche classique présentée dans la section 2.1. L'ordre est déduit en calculant le nombre minimal d'occurrences d'événements nécessaires pour atteindre le terminal 1 dans le BDD. Pour cela nous utilisons le calcul récursif suivant, soit F un BDD alors

$$order(F) = \begin{cases} 0 & \text{si F est le terminal 1} \\ +\infty & \text{si F est le terminal 0} \\ min(order(F|_{v=\mathbf{F}}), 1 + order(F|_{v=\mathbf{T}})) & \text{si } F = v \wedge F|_{v=\mathbf{T}} \vee \neg v \wedge F|_{v=\mathbf{F}} \end{cases}$$

Exemple 9.7. *Pour $tr = \{(out, ERR)\}$ d'un composant Filtre, la probabilité (respectivement l'ordre) est $prob_{F_{Va}}(tr) = p(e)$ (respectivement $order_{F_{Va}}(tr) = order(e) = 1$).*

Analyse des alternatives Nous avons analysé jusqu'ici des composants, or un système KCR est une interconnexion d'instances de composant. Nous devons donc adapter les définitions précédentes aux instances. Ainsi, les traces produites par une instance *inst* d'un composant c sont celles produites par c c'est-à-dire $traceOf_c^{inst} = traceOf_c$. Par contre, nous avons vu dans la définition de la sémantique du langage (voir 7.2) que lors d'une instanciation les événements de défaillance du composant sont renommés et ajoutés aux événements du système. Donc les combinaisons d'événements $trig_c^{inst}(tr)$ permettant de déclencher une trace tr pour une instance *inst* d'un composant c sont celles de c où les événements sont renommés.

Exemple 9.8 (Renommage)**.** *Prenons l'instance F_{Va} d'un filtre dans le système* Rosace*, et $tr = \{out \mapsto ERR\}$ une trace de F_{Va} alors :*

$$trig_{Filter}^{F_{Va}}(tr) = F_{Va}.e$$

Par ailleurs une instance *inst* d'un composant c peut être remplacée par un instance du composant c'. Par conséquent, nous calculons les valeurs $prob_{c'}^{inst}(tr)$ et $order_{c'}^{inst}(tr)$ pour chaque alternative c' de *inst* et trace $tr \in traceOf_c^{inst}$. Nous stockons ces valeurs dans des tableaux attachés à chaque trace du composant initial comme montré dans l'exemple 9.9.

Définition 9.5 (Caractérisation)**.** *Soit l'instance inst d'un composant c et $tr \in traceOf_c^{inst}$ alors le tableau prob (respectivement order) donne pour chaque alternative c' de inst la valeur de $prob_{c'}^{inst}(tr)$ (respectivement $order_{c'}^{inst}(tr)$).*

Exemple 9.9 (Caractérisation)**.** *Si DupF est une alternative de F_{Va} alors l'analyse des traces de DupF est la suivante :*

- $trig_{DupF}(\{out \mapsto LOST\}) = (F_1.e \vee F_1.l \vee F_1.e \vee F_2.l) \wedge \neg(F_{Va1}.e \wedge F_2.e)$ *donc*
  $\Rightarrow order_{DupF}(\{out \mapsto LOST\}) = 1$
  $\Rightarrow prob_{DupF}(\{out \mapsto LOST\}) = 0.04$
- $trig_{DupF}(\{out \mapsto ERR\}) = F_1.e \wedge F_2.e$ *donc*
  $\Rightarrow order_{DupF}(\{out \mapsto LOST\}) = 2$
  $\Rightarrow prob_{DupF}(\{out \mapsto ERR\}) = 10^{-4}$
- $trig_{DupF}(\{out \mapsto \emptyset\}) = \neg(F_1.e \vee F_1.l \vee F_2.e \vee F_2.l)$ *donc*
  $\Rightarrow order_{DupF}(\{out \mapsto LOST\}) = 0$
  $\Rightarrow prob_{DupF}(\{out \mapsto ERR\}) = 0.9599$

Les tableaux construits pour chacune des traces de F_{Va} sont illustrés par la figure 9.6.

$\{out \mapsto ERR\}$

	Filter	DupF
order	1	2
prob	10^{-2}	10^{-4}

$\{out \mapsto \emptyset\}$

	Filter	DupF
order	0	0
prob	0.98	0.9599

$\{out \mapsto LOST\}$

	Filter	DupF
order	1	1
prob	10^{-2}	4.10^{-2}

Figure 9.6 – Caractérisation des alternatives de F_{Va}

Par ailleurs l'hypothèse 9.1 suppose que les alternatives d'une instance produisent un sous-ensemble des modes de défaillance du composant initial de l'instance. Soit *inst* une instance du composant c pouvant être remplacée par l'instance d'un composant c' alors, en se basant sur la notion de trace, nous pouvons vérifier l'hypothèse 9.1 pour cette substitution en assurant que $traceOf_{c'} \subset traceOf_c$ (on parle de substitution acceptable). L'hypothèse 9.1 revient alors à considérer que toutes les alternatives sont acceptables (cf hypothèse 9.3).

Définition 9.6 (Alternative acceptable)**.** *Soit c' une alternative d'une instance inst d'un composant c, alors celle-ci est acceptable si et seulement si*

$$traceOf_{c'} \subset traceOf_c$$

Exemple 9.10 (Alternative acceptable)**.** *Les traces de la duplication DupF d'un filtre calculées par l'interpréteur sont :*

$$traceOf_{DupF} = \{\{out \mapsto ERR\}, \{out \mapsto LOST\}, \{out \mapsto \emptyset\}\}$$

Par conséquent $traceOf_{DupF} = traceOf_{Filter}$*, or comme* F_{Va} *est une instance de Filtre, DupF est une alternative acceptable pour* F_{Va}*.*

Hypothèse 9.3. *(Acceptabilité) Toutes les alternatives des instances de composant sont acceptables.*

9.4 Traces système

Nous nous intéressons maintenant à l'analyse d'un système qui par l'hypothèse 9.2 est une interconnexion d'instances de composant atomique. Pour des raisons de lisibilité nous omettrons l'indice c dans $traceOf_c^{inst}$, $trig_c^{inst}(tr)$, $order_c^{inst}(tr)$ et $prob_c^{inst}(tr)$ lorsque le composant c correspond au composant dont est issue l'instance *inst* dans le système initial.

9.4.1 Définition

Une *trace système* associe à chaque instance de composant du système une trace composant telles que ces traces peuvent être produites par une ou plusieurs combinaisons des événements de défaillance du système. Pour définir ces traces système, nous devons connaître le lien entre les identificateurs des entrées/sorties d'une instance de composant et les flots locaux du système utilisés lors de la définition de cette instance.

Définition 9.7 (Connexion)**.** *Soit s un système, inst une instance de composant contenue dans s, f un flot local ou de sortie de s et g un identificateur d'entrée/sortie de inst. Alors le prédicat* `isConnected`$(f, g, inst, s)$ *est vrai si et seulement si :*

- *f est utilisé pour récupérer la sortie g de inst lors de son instanciation c'est-à-dire* $f_1, \cdots, f_{i-1}, f, f_{i+1}, \cdots, f_m := c@inst(f'_1, \cdots, f'_n)$ *est une définition contenue dans s et l'identificateur de la ième sortie de inst est g ;*
- *ou bien si f est utilisé pour assigner l'entrée g de inst lors de son instanciation c'est-à-dire* $f_1, \cdots, f_m := c@inst(f'_1, \cdots, f'_{i-1}, f, f'_{i+1}, \cdots, f'_n)$ *est une définition contenue dans s et l'identificateur de la ième entrée de inst est g ;*

Exemple 9.11 (Connexion). *Dans* ROSACE*, considérons la sortie out du filtre* F_{Va} *et le flot* Va_f *du système. Dans la modélisation* KCR *de* ROSACE *de l'exemple 7.3, la définition de* Va_f *est donnée par instanciation du filtre* F_{Va} *d'où*

$$\texttt{isConnected}(Va_f, out, F_{Va}, Rosace) = \mathbf{T}$$

Considérons le flot az_f *du système, la définition de* az_f *n'est pas donnée par instanciation de* F_{Va}*, d'où*

$$\texttt{isConnected}(az_f, out, F_{Va}, Rosace) = \mathbf{F}$$

Définition 9.8 (Traces système). *Soit* $inst_1, \cdots, inst_n$ *les instance de composant d'un système s,* $S_{f_1}, \cdots, S_{f_m}$ *l'interprétation des flots locaux et de sortie* $f_1, \cdots, f_m$ *de s. Alors les traces de s sont (a) les ensembles de traces composant (b) tels qu'il existe une valuation des flots du système (c) pouvant être produite par une valuation des événement de s (d) où la valuation des flots correspond aux valeurs attribuées aux entrées et sorties des instances de composant du système :*

$$\begin{aligned} traceOf_s = \quad & \{\{inst_1 \mapsto tr_1, \cdots, inst_n \mapsto tr_n\} | \\ & tr_1 \in traceOf^{inst_1}, \cdots, tr_n \in traceOf^{inst_n}, && (a) \\ & \exists (v_1, \phi_1) \in S_{f_1}, \cdots, (v_m, \phi_m) \in S_{f_m}, && (b) \\ & \phi_1 \wedge \cdots \wedge \phi_m \not\equiv \mathbf{F} && (c) \\ & \wedge\ \forall i \in [1,n], j \in [1,m], (f \mapsto v) \in tr_i, \texttt{isConnected}(f_j, f, inst_i, s) \Rightarrow v_j = v\} && (d) \end{aligned}$$

Exemple 9.12 (Traces système). *Considérons l'ensemble de traces suivant pour les instances de composant de* ROSACE *simplifié :* $tr = \{F_{Va} \mapsto \{out \mapsto \emptyset\}, C_{Va} \mapsto \{in \mapsto \emptyset, out \mapsto \emptyset\}\}$*. Parmi les interprétations* S_{Va_f} *et* $S_{\delta_{ec}}$ *des flots de* ROSACE*, considérons les valuations suivantes :*

- (v_1, ϕ_1) *pour* Va_f *où* $v_1 = \emptyset$ *et* $\phi_1 = \neg F_{Va}.e \wedge \neg F_{Va}.l$
- (v_2, ϕ_2) *pour* δ_{ec} *où* $v_2 = \emptyset$ *et* $\phi_2 = \neg F_{Va}.e \wedge \neg F_{Va}.l \wedge \neg C_{Va}.e \wedge \neg C_{Va}.l$

Le prédicat de connexion n'est vrai que pour les cas suivants : $\texttt{isConnected}(Va_f, out, F_{Va}, Rosace)$, $\texttt{isConnected}(Va_f, in, C_{Va}, Rosace)$, $\texttt{isConnected}(\delta_{ec}, out, F_{Va}, Rosace)$*. Or l'ensemble de ces flots sont évalués à* $\emptyset$ *par les traces composant et par les valuations* v_1 *et* v_2 *donc* $(b), (d)$ *sont satisfaits. De plus,*

$$\phi_1 \wedge \phi_2 = \neg F_{Va}.e \wedge \neg F_{Va}.l \wedge \neg C_{Va}.e \wedge \neg C_{Va}.l \not\equiv \mathbf{F}$$

Donc (c) *est satisfait. Par ailleurs,* $\{out \mapsto \emptyset\} \in traceOf_{F_{Va}}$ *et* $\{in \mapsto \emptyset, out \mapsto \emptyset\} \in traceOf_{C_{Va}}$ *donc* (a) *est satisfait. Comme* $(a), (b), (c)$ *et* (d) *sont satisfaits nous avons :*

$$tr \in traceOf_{Rosace}$$

Lorsque les modes de défaillance produits par le système pour une trace donnée déclenchent l'événement redouté alors cette la trace est dite *dangereuse*. Par la suite, nous ne considérons que les traces dangereuses, donc nous parlons de *traces système* pour *traces système dangereuses*. Ainsi, $traceOf_s$ représente l'ensemble des traces dangereuses d'un système s pour un événement redouté *fc*.

9.4.2 Trace de système et fonction de structure

Nous développons, dans les sections suivantes, les méthodes de calcul des indicateurs de sûreté basées sur l'analyse des traces système. Pour cela introduisons la *fonction caractéristique des*

traces c'est-à-dire la formule représentant les combinaisons d'événements déclenchant les traces d'un système.

Définition 9.9 (Fonction caractéristique des traces)**.** *Soit s un système, fc un événement redouté et* $traceOf_s$ *les traces de s déclenchant fc alors la fonction caractéristique est :*

$$\phi = \bigvee_{tr \in traceOf_s} \bigwedge_{(inst \mapsto tr') \in tr} trig^{inst}(tr')$$

Propriété 9.1 (Équivalence)**.** *Soit s un système, fc un événement redouté,* φ *la fonction de structure de s pour fc et* ϕ *la fonction caractéristique alors* φ *et* ϕ *sont sémantiquement équivalentes.*

Démonstration. Soit un système s, $inst_1, \cdots, inst_n$ les instances de composant contenues dans s et $f_1, \cdots, f_m$ les flots (locaux et de sortie) de s.

1. Montrons que $\varphi \models \phi$. Soit E une valuation des événements du système telle que $E \models \varphi$, soit $V = \{(v_1, \phi_1) \in S_{f_1}, \cdots, (v_m, \phi_m) \in S_{f_m}\}$ la valuation des flots lorsque E se produit c'est-à-dire telle que $E \models \phi_1 \wedge \cdots \wedge \phi_m$. Considérons l'ensemble de traces composant $tr_s = \{inst_1 \mapsto tr_1, \cdots, inst_n \mapsto tr_n\}$ où les valeurs $v_1, \cdots, v_m$ des flots de s sont associées aux entrées/sorties des composants correspondants c'est-à-dire telles que

   $$\forall i \in [1,n], j \in [1,m], (f \mapsto v) \in tr_i, \texttt{isConnected}(f_j, f, inst_i, s) \Rightarrow v_j = v \qquad (1)$$

   Par ailleurs, $E \models \phi_1 \wedge \cdots \wedge \phi_m$ donc

   $$\phi_1 \wedge \cdots \wedge \phi_m \not\equiv \mathbf{F} \qquad (2)$$

   Or pour V nous avons (1) et (2) donc

   $$tr_s \in traceOf_s$$

   De plus, E déclenche $v_1, \cdots, v_m$. Or ces valeurs correspondent à celles données aux entrées/sorties dans les traces de composant de tr_s. Par conséquent, toutes les traces de composant de tr_s sont déclenchées pour E d'où

   $$E \models \bigwedge_{(inst \mapsto tr') \in tr_s} trig^{inst}(tr')$$

   Or

   $$\phi = \bigvee_{tr \in traceOf_s} \bigwedge_{(inst \mapsto tr') \in tr} trig^{inst}(tr')$$

   Donc

   $$E \models \phi$$

2. Montrons que $\phi \models \varphi$. Soit E une valuation des événements du système telle que $E \models \phi$, ce qui revient à dire qu'au moins un des termes $\bigwedge_{(inst \mapsto tr') \in tr} trig^{inst}(tr')$ est vrai. Donc

   $$\exists tr_s = \{inst_1 \mapsto tr_1, \cdots, inst_n \mapsto tr_n\} \in traceOf_s, E \models \bigwedge_{(inst \mapsto tr') \in tr} trig^{inst}(tr')$$

Par définition des traces système, il existe une valuation $V = \{(v_1, \phi_1) \in S_{f_1}, \cdots, (v_m, \phi_m) \in S_{f_m}\}$ de flots de s telle que

$$\phi_1 \wedge \cdots \wedge \phi_m \not\equiv \mathbf{F}$$
$$\wedge\ \forall i \in [1,n], j \in [1,m], (f \mapsto v) \in tr_i, \texttt{isConnected}(f_j, f, inst_i, s) \Rightarrow v_j = v$$

Or comme $E \models \bigwedge_{(inst \mapsto tr') \in tr} trig^{inst}(tr')$, toutes les traces de composant contenues dans tr_s se produisent. Puisque s est clos, tous les flots de s sont définis par une sortie de composant donc les valeurs $v_1, \cdots, v_m$ déclenchant l'événement redouté sont observées sur les flots de s pour E d'où

$$E \models \varphi$$

□

La propriété 9.1 nous assure que les combinaisons d'événements encodées par φ sont les mêmes que celles encodées par ϕ. Donc les indicateurs de sûreté peuvent être calculés à partir d'une analyse de ϕ.

9.4.3 Calcul des indicateurs

Introduisons les méthodes de calcul des indicateurs de sûreté comme la fiabilité et l'ordre d'un système à partir des traces système et des caractérisations des traces composant. Afin de simplifier le calcul des indicateurs étudions les propriétés des combinaisons d'événements déclenchant les traces du système.

Propriété 9.2 (Incompatibilité des traces système)**.** *Soit un système s et $tr_s, tr'_s \in traceOf_s$ telles que $tr_s \neq tr'_s$, alors*

$$\bigwedge_{(inst \mapsto tr) \in tr_s} trig^{inst}(tr) \wedge \bigwedge_{(inst \mapsto tr') \in tr'_s} trig^{inst}(tr') \equiv \mathbf{F}$$

Démonstration. Considérons $tr_s, tr'_s \in traceOf_s$ telles que $tr_s \neq tr'_s$, soit $D = \{inst | (inst \mapsto tr) \in tr_s, (inst \mapsto tr') \in tr'_s, tr \neq tr'\}$ l'ensemble des instances de composant pour lesquelles tr_s et tr'_s n'associent pas la même trace de composant. Ordonnons D suivant un ordre topologique du graphe d'appel et prenons la première instance notée $inst_0$.

Opérons par disjonction de cas sur l'interface de $inst_0$, si $inst_0$ ne possède pas d'entrées alors les traces de composant tr (respectivement tr') associées à $inst_0$ par tr_s (respectivement tr'_s) n'associent pas les mêmes modes de défaillance aux sorties de $inst_0$. Puisque $inst_0$ n'a pas d'entrées et que les modes de défaillance produits en sortie d'un composant ne dépendent que de ceux perçus en entrée et des événements de défaillance, alors produire des sorties différentes revient à choisir deux valuations distinctes des événements de $inst_0$. Par conséquent il n'existe pas de valuation des événements de $inst_0$ permettant de produire à la fois tr et tr' d'où

$$trig^{inst_0}(tr) \wedge trig^{inst_0}(tr') \equiv \mathbf{F}$$

Or par définition de D, $(inst_0 \mapsto tr) \in tr_s$ et $(inst_0 \mapsto tr') \in tr'_s$ donc

$$\bigwedge_{(inst \mapsto tr) \in tr_s} trig^{inst}(tr) \wedge \bigwedge_{(inst \mapsto tr') \in tr'_s} trig^{inst}(tr') \equiv \mathbf{F}$$

Si $inst_0$ possède des entrées, comme le système est clos nous savons que ces entrées sont définies par des instance de composant précédant $inst_0$ dans l'ordre topologique. Or par définition de D, $inst_0$ est la première instance de composant pour lesquelles tr_s et tr'_s n'associent pas la même trace de composant. Donc les modes de défaillance associés aux entrées de $inst_0$ sont les mêmes dans tr_s et tr'_s. Par ailleurs comme $inst_0 \in D$, les traces de composant tr (respectivement tr') associées à $inst_0$ par tr_s (respectivement tr'_s) sont différentes. Donc, tr et tr' n'associent pas les mêmes modes de défaillance aux sorties de $inst_0$ tout en associant les mêmes modes de défaillance aux entrées de $inst_0$. Or les modes de défaillance produits en sortie d'un composant ne dépendent que de ceux perçus en entrée et des événements de défaillance, donc le seul moyen de produire des sorties différentes est de choisir deux valuations distinctes des événements de $inst_0$. Par conséquent, il n'existe pas de valuation des événements de $inst_0$ permettant de produire à la fois tr et tr' d'où

$$trig^{inst_0}(tr) \wedge trig^{inst_0}(tr') \equiv \mathbf{F}$$

Or par définition de D, $(inst_0 \mapsto tr) \in tr_s$ et $(inst_0 \mapsto tr') \in tr'_s$ donc

$$\bigwedge_{(inst \mapsto tr) \in tr_s} trig^{inst}(tr) \wedge \bigwedge_{(inst \mapsto tr') \in tr'_s} trig^{inst}(tr') \equiv \mathbf{F}$$

□

Propriété 9.3 (Indépendance des traces composant)**.** *Soit $inst, inst' \in InstId$ deux instances distinctes, $tr \in traceOf^{inst}$, $tr' \in traceOf^{inst'}$ et* `evt` *la fonction renvoyant les événements contenus dans une formule alors :*

$$\texttt{evt}(trig^{inst}(tr)) \cap \texttt{evt}(trig^{inst'}(tr')) = \emptyset$$

Démonstration. Par définition $trig^{inst}(tr)$ encode les combinaisons d'événements de $inst$ déclenchant la trace tr. Or les événements sont propres à chaque instance d'où

$$\texttt{evt}(trig^{inst}(tr)) \cap \texttt{evt}(trig^{inst'}(tr')) = \emptyset$$

□

Calcul de la défiabilité Par la propriété 9.1 nous assure que $\varphi \equiv \phi$ donc

$$\begin{aligned}
\overline{R}(t) = p(\varphi) = p(\phi) &= p\left(\bigvee_{tr \in traceOf_s} \bigwedge_{(inst \mapsto tr') \in tr} trig^{inst}(tr')\right) \\
&\overset{\text{par propriété 9.2}}{=} \sum_{tr \in traceOf_s} p\left(\bigwedge_{(inst \mapsto tr') \in tr} trig^{inst}(tr')\right) \\
&\overset{\text{par propriété 9.3}}{=} \sum_{tr \in traceOf_s} \prod_{(inst \mapsto tr') \in tr} p(trig^{inst}(tr')) \\
&= \sum_{tr \in traceOf_s} \prod_{(inst \mapsto tr') \in tr} prob^{inst}(tr')
\end{aligned}$$

La défiabilité est donc calculable à partir des indicateurs locaux $prob^{inst}(tr')$ introduits dans la

section 9.3.

Calcul de l'ordre L'ordre d'un système correspond au nombre minimal d'événements nécessaires pour satisfaire φ. Or comme $\varphi \equiv \phi$ ceci revient à calculer le nombre minimal d'événements nécessaires pour satisfaire ϕ. Puisque ϕ est une disjonction de cubes mutuellement exclusifs (cf propriété 9.2), une valuation d'événements ne peut satisfaire qu'une seule de ses clauses c'est-à-dire ne produire qu'une trace. Les nombre minimal d'événements nécessaire pour satisfaire ϕ est donc le nombre minimal pour déclencher une des traces du système. De plus, pour déclencher une trace système tr, il faut déclencher toutes les traces composant contenues dans tr. Or par la propriété 9.3, nous savons qu'aucun événement ne permet de déclencher deux traces issues d'instances différentes. Donc le nombre d'événements pour déclencher une trace système est la somme des nombres minimaux d'événements nécessaires pour déclencher les traces composant c'est-à-dire $order_{inst}(tr_{inst})$. La formule de l'ordre d'un système est donc :

$$mincard = \min_{tr \in traceOf_s} \left(\sum_{(inst \mapsto tr') \in tr} order^{inst}(tr') \right)$$

9.4.4 Impact des substitutions

Évaluons maintenant l'impact d'une substitution de composant sur les traces du système. Nous supposons par l'hypothèse 9.3 que les alternatives des composants du système sont acceptables. Par conséquent, toutes les traces de l'alternative sont aussi des traces du composant initial. Nous montrons alors que les traces produites par le système obtenu par substitution sont aussi des traces du système initial (voir propriété 9.4).

Propriété 9.4 (Inclusion traces système)**.** *Soit s un système et s' le système obtenu en substituant une instance $inst_{sub}$ d'un composant c par une instance d'un composant c' alors*

$$traceOf_{s'} \subset traceOf_s$$

Démonstration. Soit $tr_{s'} = \{inst_1 \mapsto tr_1, \cdots, inst_n \mapsto tr_n\}$ telle que $tr_{s'} \in traceOf_{s'}$. Comme $tr_{s'} \in traceOf_{s'}$ alors par définition des traces système il existe une valuation $V = \{(v_1, \phi_1) \in S_{f_1}, \cdots, (v_m, \phi_m) \in S_{f_m}\}$ des flots $f_1, \cdots, f_m$ de s' telle que :

$$\forall i \in [1, n], j \in [1, m], (f \mapsto v) \in tr_i, \texttt{isConnected}(f_j, f, inst_i, s) \Rightarrow v_j = v$$

De même, par définition des traces système, il existe une valuation E des événements de s' telle que $E \models \phi_1 \wedge \cdots \wedge \phi_m$ c'est-à-dire produisant V. Soit tr_{sub} la trace de $inst_{sub}$ dans $tr_{s'}$, par l'hypothèse 9.3, $traceOf_{c'}^{inst} \subset traceOf_c^{inst_{sub}}$ donc

$$tr_{sub} \in traceOf_c^{inst_{sub}}$$

Puisque $tr'_{sub} \in traceOf_c^{inst_{sub}}$, il existe une valuation E_{inst} des événements de défaillance de $inst_{sub}$ produisant tr'_i lorsque $inst_{sub}$ est une instance de c. En remplaçant la valuation des événements de $inst_{sub}$ par $E_{inst_{sub}}$ dans E on obtient une valuation E' des événements de s produisant les valeurs $v_1, \cdots, v_m$ sur les flots $f_1, \cdots, f_m$ de s donc il existe une valuation $(v_1, \phi'_1) \in S_{f_1}, \cdots, (v_m, \phi'_m) \in$

S_{f_m} des flots de s telle que :

$$\forall i \in [1,n], j \in [1,m], (f \mapsto v) \in tr_i, \texttt{isConnected}(f_j, f, inst_i, s) \Rightarrow v_j = v$$

Et comme E' produit les valeurs $v_1, \cdots, v_m$ sur les flots $f_1, \cdots, f_m$ on a

$$\begin{array}{l} E' \models \phi'_1 \wedge \cdots \wedge \phi'_m \\ \Rightarrow \phi'_1 \wedge \cdots \wedge \phi'_m \not\equiv \mathbf{F} \end{array}$$

Donc

$$tr_{s'} \in traceOf_s$$

□

Grâce à la propriété 9.4, nous pouvons utiliser l'ensemble des traces produites par le système initial pour analyser un système obtenu par substitution.

Propriété 9.5 (Substitution et fonction caractéristique)**.** *Soit s un système et s' le système obtenu en substituant une instance $inst_{sub}$ d'un composant c par une instance d'un composant c', ϕ_s (respectivement $\phi_{s'}$) la fonction caractéristique de s (respectivement s') alors :*

$$\begin{array}{rcl} \phi_{s'} & = & \displaystyle\bigvee_{tr \in traceOf_s} \bigwedge_{(inst \mapsto tr') \in tr} \left\{ \begin{array}{ll} trig_{c'}^{inst_{sub}}(tr') & si\ inst = inst_{sub} \\ trig^{inst}(tr') & sinon \end{array} \right. \\ & = & \phi_s[trig_{c'}^{inst_{sub}} \mapsto trig_c^{inst_{sub}}] \end{array}$$

Démonstration. Comme $inst_{sub}$ est la seule instance à être substituée nous avons :

$$\phi_{s'} = \bigvee_{tr \in traceOf_{s'}} \bigwedge_{(inst \mapsto tr') \in tr} \left\{ \begin{array}{ll} trig_{c'}^{inst_{sub}}(tr') & \text{si } inst = inst_{sub} \\ trig^{inst}(tr') & \text{sinon} \end{array} \right.$$

Or par la propriété 9.4 $traceOf_{s'} \subset traceOf_s$ donc il existe un ensemble de traces $D = \{tr \in traceOf_s | tr \notin traceOf_{s'}\}$. On a alors

$$\phi_{s'} = \bigvee_{tr \in traceOf_s \setminus D} \bigwedge_{(inst \mapsto tr') \in tr} \left\{ \begin{array}{ll} trig_{c'}^{inst_{sub}}(tr') & \text{si } inst = inst_{sub} \\ trig^{inst}(tr') & \text{sinon} \end{array} \right.$$

Étudions

$$\phi_D = \bigvee_{tr \in D} \bigwedge_{(inst \mapsto tr') \in tr} \left\{ \begin{array}{ll} trig_{c'}^{inst_{sub}}(tr') & \text{si } inst = inst_{sub} \\ trig^{inst}(tr') & \text{sinon} \end{array} \right.$$

Puisque toutes les instances de composant de s' sont les mêmes que celles de s hormis pour $inst_{sub}$, $tr \in D$ implique que $\exists (inst_{sub} \mapsto tr') \in tr, tr' \notin traceOf_{c'}^{inst_{sub}}$. Or pour cette trace $trig_{c'}^{inst_{sub}}(tr') \equiv \mathbf{F}$ donc

$$\phi_D = \bigvee_{tr \in D} \bigwedge_{(inst \mapsto tr') \in tr} \left\{ \begin{array}{ll} \mathbf{F} & \text{si } inst = inst_{sub} \\ trig^{inst}(tr') & \text{sinon} \end{array} \right. \equiv \mathbf{F}$$

On peut alors écrire

$$\begin{aligned}\phi_{s'} &= \phi_D \vee \bigvee_{tr \in traceOf_s \setminus D} \bigwedge_{(inst \mapsto tr') \in tr} \begin{cases} trig_{c'}^{inst_{sub}}(tr') & \text{si } inst = inst_{sub} \\ trig^{inst}(tr') & \text{sinon} \end{cases} \\ &= \bigvee_{tr \in traceOf_s} \bigwedge_{(inst \mapsto tr') \in tr} \begin{cases} trig_{c'}^{inst_{sub}}(tr') & \text{si } inst = inst_{sub} \\ trig^{inst}(tr') & \text{sinon} \end{cases}\end{aligned}$$

□

La propriété 9.5 nous assure que remplacer les fonctions *trig* lors d'une substitution permet d'obtenir la fonction caractéristique du nouveau système. Or le calcul des indicateurs est basé sur la fonction caractéristique, donc pour recalculer la valeurs des indicateurs pour le nouveau système il suffit de sélectionner dans les tableaux order et prob (cf définition 9.5) les indicateurs locaux pour le choix de substitution considéré.

9.5 Construction du Diagramme de Décision Multi-Valué des traces système

Afin de représenter et d'analyser efficacement les traces d'un système nous proposons d'utiliser les Diagrammes de Décision Multi-Valués (MDD). Pour cela nous introduisons, dans la section 9.5.1, les concepts élémentaires de construction et de manipulation des MDDs. Puis nous montrons, dans la section 9.5.2, comment construire le Diagramme de Décision Multi-Valué des traces système à l'aide de l'interpréteur de KCR. Finalement, afin de réduire la taille du MDD obtenu, nous introduisons, dans la section 9.5.3, une adaptation du MDD appelée STMDD.

9.5.1 Introduction aux MDDs

Les diagrammes de décisions multi-valués (MDD) introduits dans [89] sont utilisés pour encoder des fonctions à n entrées et une seule sortie dont la signature est de la forme :

$$f : P_1 \times \cdots \times P_n \to Q$$

où P_i est l'ensemble fini des valeurs possibles de la ième entrée de f et Q l'ensemble fini des valeurs de sortie de f.

Définition 9.10 (MDD)**.** *Un MDD est un graphe orienté acyclique défini par le n-uplet suivant :*

$$MDD = \langle \mathcal{V}, \mathcal{N}, \mathcal{T}, \mathcal{E}, \sigma, \tau, dom, \Gamma, < \rangle$$

où

$\mathcal{V}$ *est l'ensemble des variables d'entrée ;*

$\mathcal{N}$ *est l'ensemble des nœuds ;*

$\mathcal{T}$ *est l'ensemble des terminaux ;*

$\mathcal{E}$ *est l'ensemble d'arcs orientés tel que* $E \subset \mathcal{N}^2$ *où un arc* e *d'un nœud* N *vers* N' *est* $e = (N, N')$ *;*

σ *est la fonction d'étiquetage* $\mathcal{N} \to \mathcal{V}$ *donnant la variable étiquetée sur un nœud ;*

τ *est la fonction d'étiquetage* $\mathcal{T} \mapsto Q$ *donnant la valeur étiquetée sur un terminal ;*

dom est la fonction donnant le domaine d'une variable $v \in V$

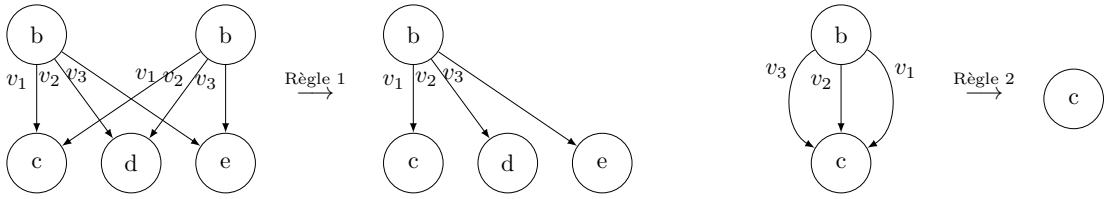


FIGURE 9.7 – Règles de simplification du MDD

Γ *est la famille de fonctions d'étiquetage* $\{\gamma_N : \mathcal{E}_N \to P | N \in \mathcal{N}\}$ *où* $P = dom(\sigma(N))$ *est le domaine de la variable étiquetée sur* N *et* $\mathcal{E}_N = \{(N, N') \in \mathcal{E}\}$ *l'ensemble de arcs issus de* N*. Soit un arc* $e = (N, N')$ *alors* $\gamma_N(e)$ *est la valeur de la variable* $\sigma(N)$ *étiquetée sur* e *;*

$<$ *est l'ordre total sur* V *;*

Pour simplifier les notations, nous introduisons les associations $sons_N = \{c \mapsto N' | (N, N') \in \mathcal{E}, c = \gamma_N((N, N'))\}$ donnant le fils du nœud N pour chacune des valeurs c de la variable étiquetée sur N.

Dans un MDD, un chemin depuis la racine jusqu'à un terminal encode un ensemble de décisions sur les entrées de la fonction et la valeur étiquetée sur le terminal contient la valeur de la fonction pour ces décisions.

Un MDD est dit *ordonné* si et seulement si pour tout arc $(N, N') \in \mathcal{E}$, alors $\sigma(N) < \sigma(N')$. De même, un MDD est dit *réduit* si et seulement si, lors de sa construction, les règles suivantes (illustrées par la figure 9.7) sont appliquées :

1. si le MDD contient un ensemble de sous-graphes isomorphes alors seul l'un d'entre eux est conservé et les arcs pointant sur les anciens sous-graphes sont redirigés vers celui-ci ;
2. si tous les arcs issus d'un nœud N pointent vers un même fils N' alors ces arcs sont redirigés vers N' et N est supprimé.

Pour construire un MDD, les auteurs de [89] définissent un opérateur appelée *case*. Soit F le MDD d'une fonction dont le codomaine est $Q = \{c_1, \cdots, c_m\}$ et $\{G_1, \cdots, G_m\}$ un ensemble de MDDs, alors $H = case(F, \{v_1 \mapsto G_1, \cdots, v_m \mapsto G_m\})$ construit le MDD H correspondant à la fonction h suivante :

$$h(x_1, \cdots, x_n) = \begin{cases} g_1(x_1, \cdots, x_n) & \text{si } f(x_1, \cdots, x_n) = c_1 \\ \vdots & \\ g_m(x_1, \cdots, x_n) & \text{si } f(x_1, \cdots, x_n) = c_m \end{cases}$$

Exemple 9.13. *Prenons la fonction* $f : \{1, 2, 3\} \times \{0, 1\}^2 \to \{0, 1, 2\}$ *définie comme suit :*

$$f(a, b, c) = \begin{cases} b + c & si\ a \leq 2 \\ c + 1 & sinon \end{cases}$$

En décomposant la fonction sur les variables a, b, c *à l'aide de l'opérateur case avec* $a < b < c$ *on obtient le MDD de la figure 9.8.*

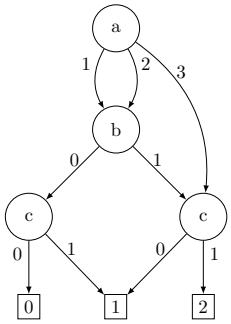


FIGURE 9.8 – MDD de l'exemple 9.13

9.5.2 Construction du MDD

Encodons l'ensemble des traces d'un système $traceOf_s$ comme un MDD appelé MDD des traces système. Soit $inst_1, \cdots, inst_n$ les instance de composant du système triées par un ordre topologique du graphe d'appel (cf section 7.1), alors la fonction multi-valuée f représentée par le MDD est :

$$f : traceOf^{inst_1} \times \cdots \times traceOf^{inst_n} \to \{0, 1, \bot\}$$

Les variables $\mathcal{V} = \{inst_1, \cdots, inst_n\}$ sont les instances de composant et les arcs E indiquent quelle est la trace composant produite par l'instance. Le domaine d'une variable du MDD est donc l'ensemble des traces composant que peut produire l'instance c'est-à-dire $dom(inst_i) = traceOf^{inst_i}$.

Définition des terminaux La fonction f possède donc trois valeurs de sortie dont la signification est la suivante

0 la trace est une trace système mais n'est pas dangereuse ;

1 la trace appartient aux traces système dangereuses ;

$\bot$ la trace ne respecte pas la définition d'une trace système c'est-à-dire il n'existe pas d'interprétations des flots du système pouvant être produites simultanément et correspondants aux valeurs assignées dans les traces (cf définition 9.8).

L'ensemble des terminaux est alors $\mathcal{T} = \{Terminal(0), Terminal(1), Terminal(\bot)\}$

Interprétation du système Pour construire le MDD nous utilisons l'interpréteur comme suit :

1. Nous interprétons le flots locaux et de sortie $f_1, \cdots, f_n$ du système et préservons les valuations pour lesquelles l'événement redouté se produit. Soit $S_{f_1}, \cdots, S_{f_n}$ les interprétations obtenues, nous créons alors l'ensemble suivant :

$$A = \{f_1 \mapsto S_{f_1}, \cdots, f_n \mapsto S_{f_n}\}$$

2. Puis nous reconstruisons les interprétations des entrées et sorties des instances de composant à partir de A. Soit $f_1, \cdots, f_m{:=}c$ @ $inst(f'_1, \cdots, f'_n)$ l'instanciation $inst$ où in_1, $\cdots$, in_n, o_1, $\cdots$, o_m sont les flots d'entrée/sortie de $inst$, alors les interprétations de ces flots sont

représentées par l'association :

$$inst \mapsto \{in_1 \mapsto S_{f'_1}, \cdots, in_n \mapsto S_{f'_n}, o_1 \mapsto S_{f_1}, \cdots, o_m \mapsto S_{f_m}\}$$

Soit $(inst_1, \cdots, inst_n)$ la liste des instances de composant du système classées dans un ordre topologique du graphe d'appel (cf section 7.1), alors nous appliquons la transformation précédente pour chacune des instances de composant du système et obtenons la liste suivante :

$$l = (inst_1 \mapsto \{in_1 \mapsto S_{in_1}, \cdots, o_m \mapsto S_{o_m}\}, \cdots, inst_n \mapsto \{in'_1 \mapsto S_{in'_1}, \cdots, o'_{m'} \mapsto S_{o'_{m'}}\})$$

Construction du MDD à partir de l'interprétation du système Pour calculer et encoder les traces système nous introduisons la function BUILDMDD décrite par l'algorithme 3. Celui-ci construit le MDD récursivement à partir de la liste l des interprétations. Cet algorithme est basé sur les fonctions suivantes :

toTrace construit les valuations d'un ensemble de flots pouvant se produire simultanément à partir de leur interprétation. Soit $interp = \{f_1 \mapsto S_{in_1}, \cdots, f_n \mapsto S_{f_n}\}$ une interprétation d'un ensemble de flots alors :

$$\begin{aligned} \text{TOTRACE}(interp) = \quad & \{(\{f_1 \mapsto v_1, \cdots, f_n \mapsto v_n\}, \phi_1 \wedge \cdots \wedge \phi_n) | (v_1, \phi_1) \in S_{f_1}, \cdots, \\ & (v_n, \phi_n) \in S_{f_n}, \phi_1 \wedge \cdots \wedge \phi_n \not\equiv \mathbf{F}\} \end{aligned}$$

isPossible est vrai si et seulement si la trace d'un composant possède bien la même valuation des entrées que celles fournies par un ensemble de traces. Soit $c \in CompId$, $tr = \{in_1 \mapsto v_1, \cdots, o_m \mapsto v_{n+m}\}$ une trace de c, $in_1, \cdots, in_n$ les entrées de c, T un ensemble de traces de c et $tr(x)$ la valeur associée au flot x dans la trace tr, alors

$$\text{ISPOSSIBLE}(c, tr, T) = \forall (tr' \in T), tr(in_1) = tr'(in_1) \wedge \cdots \wedge tr(in_n) = tr'(in_n)$$

applyCase applique l'opérateur *case* pour construire un MDD. Soit $inst \in InstId$, $sons = \{tr_1 \mapsto G_1, \cdots, tr_n \mapsto G_n\}$ une association de MDD pour chaque trace $tr \in traceOf^{inst}$ et I le MDD possédant un nœud étiqueté par $inst$ et relié à n terminaux correspondant aux traces contenues dans $traceOf^{inst}$ alors

$$\text{APPLYCASE}(inst, sons) = case(I, \{tr_1 \mapsto G_1, \cdots, tr_n \mapsto G_n\})$$

Notons que les règles de simplification, mentionnées dans la section 9.5.1, sont implicitement appliquées lors de la construction d'un MDD.

La fonction BUILDMDD prend en entrée
- la liste l des interprétations des entrées/sorties des instances de composant du système ;
- le BDD ϕ sur les événements de défaillance du système indiquant les combinaisons d'événements ayant produit le nœud parent.

BUILDMDD construit récursivement le MDD encodant l'ensemble des traces système à partir de l. Si la liste est vide l'algorithme renvoie le terminal 1, sinon il effectue les étapes suivantes :

1. construire les traces de l'instance en tête de la liste l à partir des interprétations des flots données par l'interpréteur et ne conserver que celles pouvant se produire lorsque le nœud parent peut se produire (ligne 5).

2. pour chacune des traces $tr \in traceOf^{inst}$, si tr est une trace produite par l'interpréteur telle que celle-ci et le nœud parent peuvent être produits par une combinaison d'événements alors construire le fils pour cette trace (ligne 9) ;
3. sinon si tr possède la même valuation des entrées de l'instance que les traces produites par l'interpréteur, alors celle-ci peut se produire. En effet, comme l'ensemble des prédécesseurs de l'instance courante ont été évalués (construction suivant l'ordre topologique) alors exactement une trace a été choisie pour chacun d'eux. Or les sorties alimentant les entrées de l'instance courante sont forcément des sorties des prédécesseurs, donc les combinaisons d'événements encodées par ϕ imposent une unique valeur aux entrées. Néanmoins tr n'appartient pas aux traces générées par l'interpréteur donc elle ne mène pas à une trace système dangereuse, le fils est donc le terminal 0 (ligne 11) ;
4. sinon tr ne peut pas être produite donc le fils est le terminal $\bot$;
5. construire le nœud à partir des fils calculés (ligne 16).

Algorithme 3 Algorithme de construction du MDD

```
1:  function buildMDD(l, φ)
2:      l match
3:          case Nil ⇒ return getTerminal(1)
4:          case (inst ↦ interp) :: t ⇒
5:              traces ← {(tr, φ') ∈ toTraces(interp) | φ' ∧ φ ≢ F}
6:              sons ← ∅
7:              for tr ∈ traceOf_inst do
8:                  if ∃(tr', φ') ∈ traces, tr = tr' then
9:                      sons ← sons ∪ {tr ↦ buildMDD(t, φ ∧ φ')}
10:                 else if isPossible(c, tr, traces) then
11:                     sons ← sons ∪ {tr ↦ getTerminal(0)}
12:                 else
13:                     sons ← sons ∪ {tr ↦ getTerminal(⊥)}
14:                 end if
15:             end for
16:             return applyCase(inst, sons)
17: end function
```

Un chemin depuis la racine du MDD vers le terminal 1 représente bien une trace système dangereuse car celui-ci est construit si et seulement s'il existe une valuation des flots du système telle que :

- les valuations des flots déclenchent l'événement redouté,
- celle-ci puisse se produire (ligne 5),
- que les valeurs contenues dans les traces composant sont les mêmes que celles données par la valuation (ligne 8).

9.5.3 Adaptation du MDD à l'encodage des traces système (STMDD)

Un MDD construit avec la méthode buildMDD peut contenir un nœud N étiqueté par l'instance $inst$ où les arcs sortant pointent soit vers un même fils N' soit vers le terminal $\bot$. En effet, cette

configuration ne permet pas d'utiliser les règles de simplification. Néanmoins, N indique seulement que quelle que soit la trace composant que l'on peut produire dans le contexte donné (c'est-à-dire celles ne pointant pas vers le terminal $\perp$) alors le prochain fils est toujours N'. Par conséquent, le choix de la trace produite par *inst* n'influe pas sur le fait que la trace système soit dangereuse ou non. Nous introduisons alors le STMDD correspondant à un MDD classique où la deuxième règle de construction des MDDs est adaptée de la manière suivante :

Définition 9.11. *(Règle de simplification) Soit N un nœud du MDD étiqueté par une instance inst où $S = \{tr \mapsto son \in sons_N | son \neq Terminal(\perp)\}$ est l'ensemble de ses fils hormis le terminal $\perp$, alors si $\exists N' \in \mathcal{N}, \forall tr \mapsto son \in S, son = N'$ alors le nœud N est supprimé et remplacé par N'.*

Cette règle de simplification est spécialement conçue pour éviter de représenter des sous-graphes menant uniquement aux terminaux 1 et $\perp$, cas fréquent lors de la construction du MDD. En effet, il existe souvent un composant (nommons le c) dont la défaillance entraîne la défaillance du système quelque soit le comportement des autres composants (nommons les A). Or sans la simplification précédente, le MDD représente tous ces comportements sous la forme d'un graphe n'accédant qu'au terminaux 1 et $\perp$. Représenter ce graphe a) n'a aucun intérêt puisqu'il ne fait qu'informer que tous les comportements possibles des composants de A (c'est-à-dire n'accédant pas au terminal $\perp$) sont des traces système, b) est coûteux en mémoire puisque ce graphe peut avoir une taille exponentielle en fonction du nombre de composants contenus dans A. Or, sans la règle de la définition 9.11, ces graphes ne peuvent pas être simplifiés puisque tous les arcs ne sont pas dirigés vers le même terminal. Conserver ces graphes aurait donc un fort impact sur la taille du MDD limitant ainsi son utilisation pour des systèmes réalistes.

Exemple 9.14 (STMDD de Rosace)**.** *Reprenons l'architecture de* Rosace *simplifiée de l'exemple 9.1, considérons qu'un filtre (respectivement un contrôleur) peut être remplacé par une duplication DupF (respectivement une duplication DupLawVa) alors le* STMDD *de* Rosace *est représenté par la figure 9.9. Par des raisons de lisibilité, nous ne représentons pas les traces pointant vers le terminal $\perp$.*

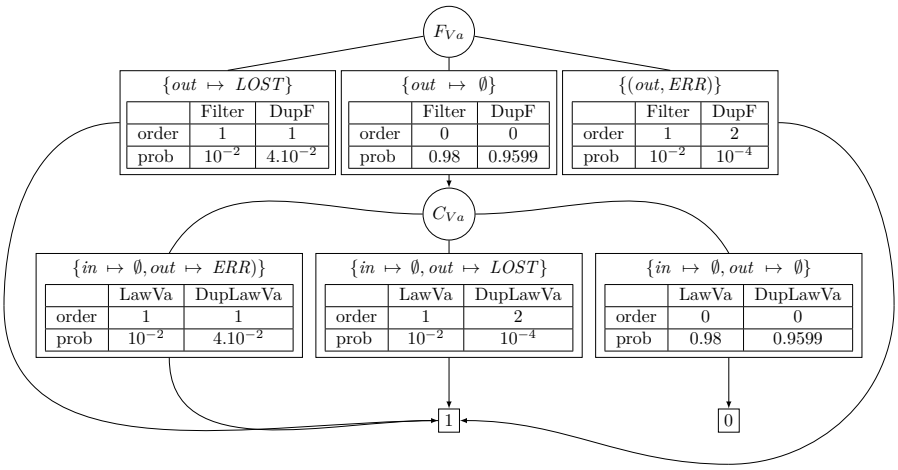


Figure 9.9 – STMDD de Rosace

9.6 Calcul des indicateurs

Adaptons les algorithmes de calcul des indicateurs de sûreté pour mener l'analyse directement sur le STMDD.

9.6.1 Fiabilité

Soit N un nœud du STMDD, alors par la propriété 9.2 on sait que les fils de N sont incompatibles deux à deux car ils représentent des traces système différentes. De plus, comme les fils de N sont étiquetés par une autre instance de composant que étiquetée sur N, nous savons par la propriété 9.3 que les traces composant de ses fils sont indépendantes de celles de N. Donc le calcul de la probabilité d'un nœud N est :

$$p(N) = \begin{cases} 0 & \text{si } N \text{ est le terminal } \perp \text{ ou } 0 \\ 1 & \text{si } N \text{ est le terminal } 1 \\ \sum_{(tr \mapsto son) \in sons_N} prob^{inst}(tr) p(son) & \text{si } N \text{ est un nœud étiquetté par } inst \end{cases} \tag{9.1}$$

Étudions maintenant l'impact de la règle de simplification introduite par la définition 9.11 sur le calcul de la probabilité. Soit N un nœud étiqueté par *inst* tel que pour toute association $(tr \mapsto son) \in sons_N$, *son* est soit un même fils N' soit le terminal $\perp$. Ceci indique donc que toutes les traces de *inst*, conformes à la valuation des entrées imposée par ses prédécesseurs, mènent au même fils N'. Or pour une valuation fixée des entrées, le seul moyen de produire toutes les traces possibles de *inst* est de considérer les sorties produites pour toutes les combinaisons possibles des événements de *inst* d'où :

$$\sum_{\substack{(tr \mapsto son) \in sons_N \\ son \neq Terminal(\perp)}} prob^{inst}(tr) = 1 \tag{9.2}$$

En reprenant l'équation 9.1, nous avons

$$p(N) = \sum_{\substack{(tr \mapsto son) \in sons_N \\ son \neq Terminal(\perp)}} prob^{inst}(tr) p(son) + \sum_{\substack{(tr \mapsto son) \in \in sons_N \\ son = Terminal(\perp)}} prob^{inst}(tr) p(\perp)$$

Or $p(\perp) = 0$ d'où

$$p(N) = p(N') \sum_{\substack{(tr \mapsto son) \in sons_N \\ son \neq Terminal(\perp)}} prob^{inst}(tr)$$

Comme par l'équation 9.2 on en déduit que $p(N) = p(N')$. La règle de simplification ne modifie pas le calcul de la probabilité d'occurrence des traces représentées par un MDD. Soit *Root* le nœud racine du STMDD d'un système alors :

$$\overline{R}(t) = p(\phi) = p(Root) \tag{9.3}$$

9.6.2 Ordre

Par les propriétés 9.2 et 9.3 nous pouvons calculer le nombre minimal d'événements nécessaires pour déclencher l'une des traces système encodées par un nœud N comme suit :

$$order(N) = \begin{cases} 0 & \text{si } N \text{ est le terminal 1} \\ +\infty & \text{si } N \text{ est le terminal 0 ou } \perp \\ \min\limits_{(tr \mapsto son) \in sons_N} (order(son) + order^{inst}(tr)) & \text{si } N \text{ est un nœud étiquetté par } inst \end{cases} \tag{9.4}$$

Étudions l'impact de la règle de simplification sur le calcul de l'ordre. Soit N un nœud étiqueté par *inst* tel que pour toute association $(tr \mapsto son) \in sons_N$, *son* est soit un même fils N' soit le terminal $\perp$. Donc toutes les traces de *inst*, conformes à la valuation des entrées imposée par ses prédécesseurs, mènent au même fils N'. De plus, nous savons que le seul moyen de produire toutes les traces possibles de *inst* est de considérer les sorties produites pour toutes les combinaisons possibles des événements de *inst*. Considérons, en particulier, la trace tr' produite lorsque aucun événement de défaillance ne se produit, par conséquent $order^{inst}(tr') = 0$. Reprenons l'équation 9.4 sur N

$$order(N) = min\left(\min_{(tr \mapsto son) \in sons_N, tr \neq tr'} (order(son) + order^{inst}(tr)), (order^{inst}(tr') + order(N'))\right)$$

Or $order^{inst}(tr') = 0$, $order(Terminal(\perp)) = +\infty$ et tous les arcs ne pointant pas vers $\perp$ pointent vers N' d'où :

$$\begin{aligned} order(N) &= min\left(order(N') + \min_{(tr \mapsto son) \in sons_N, tr \neq tr'} (order^{inst}(tr)), order(N')\right) \\ &= order(N') + min\left(\min_{(tr \mapsto son) \in sons_N, tr \neq tr'} (order^{inst}(tr)), 0\right) \\ &= order(N') \end{aligned}$$

La règle de simplification ne modifie donc pas le calcul de l'ordre, soit N le nœud racine du STMDD d'un système alors

$$\text{mincard} = order(N) \tag{9.5}$$

Rappelons que les instance de composant du système peuvent être remplacées par des instances d'autres composants. Or par la propriété 9.5 nous savons qu'il suffit de récupérer la valeur des indicateurs locaux dans les tableaux prob et order pour adapter le calcul des indicateurs.

Exemple 9.15. *Considérons que F_{Va} n'est pas substitué et que C_{Va} est remplacé par une duplication DupLawVa , alors les cellules grisées des tableaux order et prob dans le* STMDD *de* Rosace *(figure 9.10) sont utilisées dans les équations 9.1 et 9.4 pour calculer l'ordre et la défiabilité. Nous avons*

$$mincard = min(1, 0 + min(1, 1, +\infty), 1) = 1$$

$$\overline{R} = 10^{-2} + 0.98(4.10^{-2} + 10^{-4}) + 10^{-2} \simeq 6.10^{-2}$$

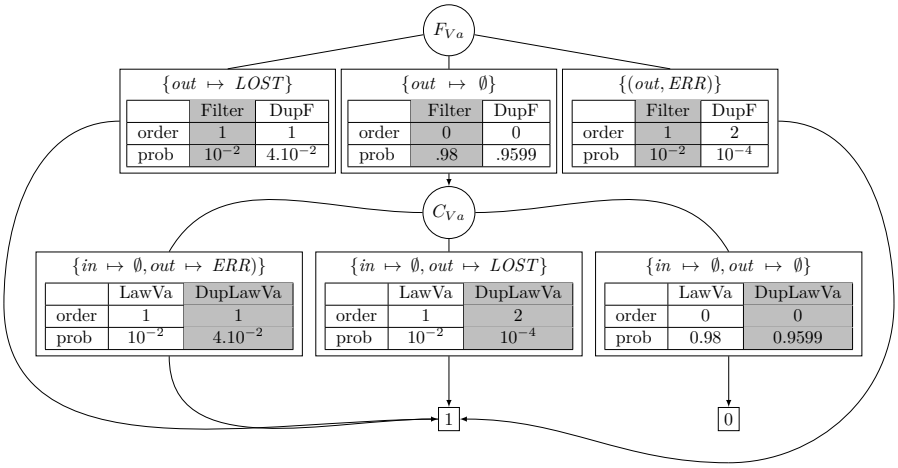


FIGURE 9.10 – STMDD de ROSACE

9.7 Résumé

Nous avons développé une nouvelle forme d'analyse des systèmes statiques fondée sur l'étude des traces dangereuses du système. Nous avons alors montré que les indicateurs de tous les candidats de l'espace des architecture d'un problème DSE peuvent être calculés à partir de l'analyse du système initial et des alternatives. Ainsi nous avons utilisé un encodage des traces du système initial par un MDD (appelé STMDD) pour calculer la fiabilité et l'ordre des candidats. Nous allons maintenant utiliser cette structure de données pour encoder et résoudre le problème DSE à l'aide des solveurs SMT.

Chapitre 10

Durcissement automatique des modèles KCR

Nous présentons, dans ce chapitre, une formalisation et une résolution du problème DSE à l'aide des solveurs SMT et de l'analyse du STMDD. Pour cela nous développons, dans la section 10.2, une théorie appelée *Safety*, présentée dans [37], permettant d'utiliser le STMDD pour traiter les contraintes de sûreté décrites comme des prédicats de la MSFOL. Puis nous utilisons dans la section 10.3, les prédicats et sortes fournis par cette théorie pour encoder le problème DSE comme un problème SMT. Finalement la section 10.4 décrit l'implantation du solveur de la théorie Safety.

10.1 Processus de résolution

La figure 10.1 expose le processus de résolution d'un problème DSE repose sur :

- une théorie (au sens SMT du terme, voir 5.3) appelée *Safety* présentée dans la section 10.2, fournissant :
  - des sortes MSFOL pour représenter l'espace des architectures et le STMDD du système ainsi que des prédicats utilisés pour modéliser les exigences de sûreté ;
  - une procédure pour décider si un candidat respecte les exigences de sûreté.
- Le solveur SMT Z3 pour résoudre le problème DSE à l'aide de la théorie Safety.

Les principales étapes du processus de résolution d'un problème DSE sont :

Construction du STMDD Cette phase préliminaire génère le STMDD du système initial à l'aide des analyses introduites dans le chapitre 9 et vérifie que les alternatives données par l'utilisateur dans la configuration sont acceptables (voir définition 9.6).

Encodage de l'espace des architectures Les informations contenues dans la configuration KCR et le STMDD sont encodées comme un problème SMT (voir section 10.3) possédant un modèle si et seulement s'il existe un choix des alternatives (c'est-à-dire un candidat de l'espace des architectures) permettant de satisfaire les exigences de sûreté spécifiées par la configuration. Pour cela, l'espace des architectures formé par les alternatives est encodé par un ensemble de variables de décision appartenant à la théorie Safety indiquant les alternatives choisies pour les composants du système. Un modèle du problème SMT final, c'est-à-dire une valuation de ces variables, représente alors un candidat de l'espace des architectures.

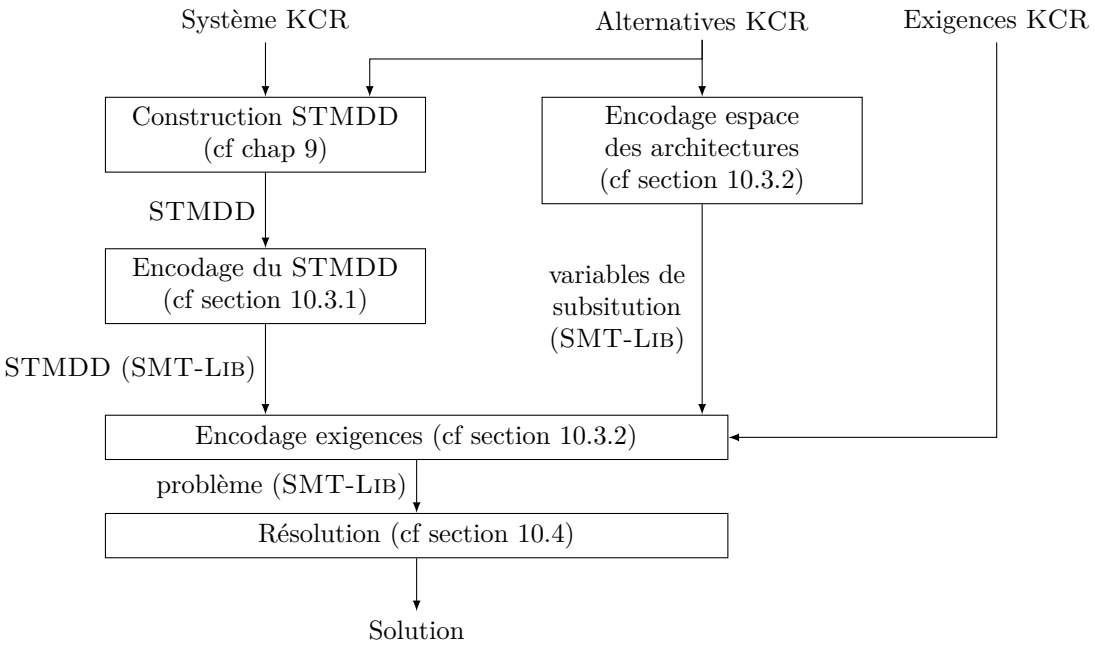


Figure 10.1 – Aperçu du process de résolution d'un problème DSE

Encodage STMDD Le STMDD est encodé comme une constante interprétée utilisée par la théorie Safety pour mener les analyses de sûreté.

Encodage exigences Finalement les exigences de sûreté contenues dans la configuration sont traduites vers des prédicats de la théorie Safety et constituent les assertions du problème SMT.

Résolution Le problème SMT obtenu est alors résolu par le solveur SMT Z3 étendu avec un solveur de la théorie Safety (voir section 10.4) utilisé de manière *paresseuse* comme montré par la figure 10.2. Plus précisément, le solveur transforme le problème SMT en problème Sat en remplaçant les prédicats de Safety par des propositions. Le solveur résout dans un premier temps le problème Sat puis appelle le solveur de Safety à la fin du processus de résolution pour vérifier si la conjonction des prédicats de Safety construite à partir du modèle obtenu est satisfiable. Si oui, un modèle des constantes non-interprétées est renvoyé par le solveur de Safety. Sinon une clause de conflit est renvoyée à Z3 qui l'ajoute aux assertions. Si, par ajout de clauses, le problème Sat devient Unsat alors le problème n'a pas de solution.

10.2 Théorie Safety

Puisque les indicateurs de sûreté sont calculables sur le STMDD du système initial, la théorie Safety permet de formuler des contraintes sur ce STMDD afin d'encoder les exigences de sûreté contenues dans un problème DSE. Nous donnons la signature et l'axiomatique de cette théorie mettant à disposition les sortes, fonctions et prédicats utilisés pour encoder le problème DSE.

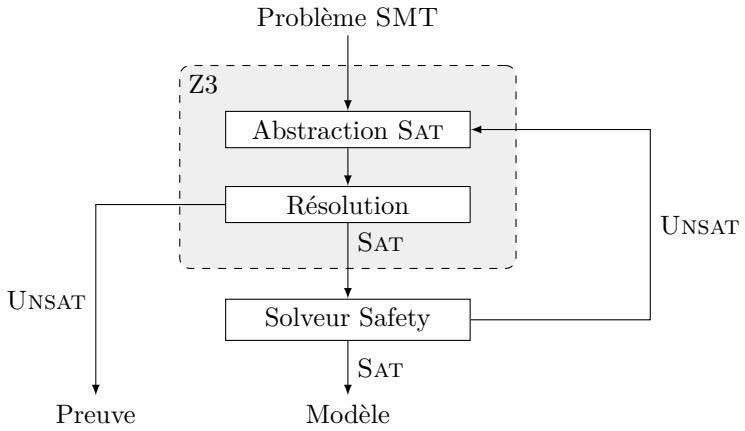


FIGURE 10.2 – Résolution SMT paresseuse

10.2.1 Définition

Les sortes et prédicats sur lesquels sont exprimées les formules de la théorie Safety sont donnés par la signature suivante :

$$\Sigma_{Safety} = \left\{ \begin{array}{ll} Sort_{Safety} & = \{STMDD, SonList, Label, Sub, InfInt, Real, Int, Bool\}, \\ Const_{Safety} & = \{nil, one, zero, bottom\}, \\ Fun_{Safety} = & \{cons, choose, isSafeOrder, isSafeR, +, \times, \leq, minList, minInf, \\ & geqInf, sumList, addInf, orderOf, probOf\}, \\ Rank_{Safety} & \end{array} \right\}$$

$$Rank_{Safety} = \left\{ \begin{array}{llllll} choose(Sub, Int) & : & Bool, & isSafeR(STMDD, Real) & : & Bool, \\ isSafeOrder(Sub, Int) & : & Bool, & probOf(STMDD) & : & Real, \\ orderOf(STMDD) & : & InfInt, & \times(Real, Real) & : & Real, \\ +(Real, Real) & : & Real, & addInf(InfInt, InfInt) & : & InfInt, \\ \leq(Real, Real) & : & Bool, & geqInf(InfInt, InfInt) & : & Bool, \\ minInf(InfInt, InfInt) & : & InfInt, & minList(SonList, Sub) & : & InfInt, \\ sumList(SonList, Sub) & : & Real, & cons(Label, SonList) & : & SonList \end{array} \right\}$$

Les sortes de Safety sont définies à l'aide de sortes des théorie NIRA, des types algébriques et des tableaux, tandis que les fonctions et prédicats sont définis par un ensemble d'axiomes.

Sub représente les variables de décision sur le choix substitution pour les instances de composant du système, elle est définie comme un alias des entiers de la manière suivante :

```
(define-sort Sub () Int)
```

InfInt est la sorte des entiers enrichie d'une valeur *infinity* représentant $+\infty$, celle-ci est décrite comme un type algébrique.

```
(declare-datatypes () ((InfInt infinity (mkInfInt (val Int)))))
```

STMDD représente les nœuds d'un STMDD et est défini comme un type algébrique pouvant être soit un terminal *one*, *zero* ou *bottom*, soit un nœud *node* contenant une variable de substitution *var* et une liste de fils *sons* ;

SonList est une liste des fils d'un nœud N du STMDD encodés par des triplets (order, prob, son) représentant les arcs issus de N. Le triplet encode les informations données par l'arc du STMDD à savoir le nœud d'arrivée *son* et les tableaux order et prob (voir définition 9.5) associés à la trace étiquetée sur l'arc. Notons que cette définition utilise la théorie des tableaux, cependant ceux-ci sont seulement des constantes interprétées initialisées lors de la définition d'une liste de fils.

```
(declare-datatypes () (
    (STMDD one zero bottom  (node (var Sub) (sons SonList)))
    (SonList nil (cons (head Label) (tail SonList)))
    (Label (mkLabel (order (Array Sub InfInt)) (prob (Array Sub Real)) (son
   STMDD)))))
```

choose est vrai si et seulement si le choix de substitution est le ième, c'est-à-dire soit *var* une variable de sélection et *val* un entier alors (`choose` *var val*) est vrai si et seulement si $var = val$;

```
(forall ((var Sub) (val Int)) (= (choose var val) (= var val)))
```

minInf sur *InfInt* est défini par les axiomes suivants :

```
(forall ((x InfInt)) (= (minInf x infinity) x))
(forall ((x InfInt)) (= (minInf infinity x) x))
(forall ((x Int) (y Int))
        (= (minInf (mkInfInt x) (mkInfInt y)) (mkInfInt (min x y))))
```

addInf sur *InfInt* est défini par les axiomes suivants :

```
(forall ((x InfInt)) (= (addInf x infinity) infinity))
(forall ((x InfInt)) (= (addInf infinity x) infinity))
(forall ((x Int) (y Int))
        (= (addInf (mkInfInt x) (mkInfInt y)) (mkInfInt (+  x y))))
```

geqInf sur *InfInt* est défini par les axiomes suivants :

```
(forall ((x InfInt)) (geqInf infinity x))
(forall ((x Int) (y Int))
        (= (geqInf (mkInfInt x) (mkInfInt y)) (>=  x y)))
```

minList fournit l'ordre minimal parmi ceux calculés sur un liste de nœuds *SonList* pour un choix *Sub* :

```
(forall ((l SonList) (var Sub))
        (= (minList l var)
        (ite (= l nil) infinity
                        (minInf (addInf (select (order (head l)) var)
                                        (orderOf (son (head l))))
                                (minList (tail l) var)))))
```

sumList additionne l'ensemble des probabilités calculées sur une liste de nœuds *SonList* pour un choix *Sub* :

```
(forall ((l SonList) (var Sub))
        (= (sumList l var)
           (ite (= l nil) 0
                          (+ (* (select (prob (head l)) var)
                                (probOf (son (head l))))
                          (sumList (tail l) var)))))
```

orderOf fournit l'ordre d'un STMDD :

```
(= (orderOf one) (mkInfInt 0))
(= (orderOf zero) infinity)
(= (orderOf bottom) infinity)
(forall ((N STMDD)) (= (orderOf N) (minList (sons N) (var N))))
```

probOf fournit la défiabilité d'un STMDD :

```
(= (probOf one) 1)
(= (probOf zero) 0)
(= (probOf bottom) 0)
(forall ((N STMDD)) (= (probOf N) (sumList (sons N) (var N))))
```

isSafeOrder est vrai si et seulement si le STMDD N satisfait l'exigence d'ordre *req*. Ceci revient (voir section 9.5) à vérifier que l'ordre calculé sur le N est supérieur ou égale à la borne *req* :

```
(forall ((N STMDD) (req Int))
  (= (isSafeOrder N req) (geqInf (orderOf N) (mkInfInt req))))
```

isSafeR vrai si et seulement si le STMDD N satisfait l'exigence de fiabilité *req*. Ceci revient à vérifier si la défiabilité calculée sur un N est inférieure ou égale à $1 - req$:

```
(forall ((N STMDD) (req Real))
  (= (isSafeR N req) (<= (probOf N) (- 1 req))))
```

De plus nous imposons que les formules ne contiennent pas de quantificateurs sur les sortes *STMDD* et *Sub* et que les constantes de sorte *STMDD* soient interprétées.

10.2.2 Décidabilité

La formalisation de l'axiomatique de la théorie Safety est exprimée dans des fragments logiques plus riches que ce qui est nécessaire pour encoder les problèmes DSE. En effet, l'axiomatique repose :

1. sur l'arithmétique linéaire entière avec quantificateurs, qui n'est pas décidable,
2. sur des définitions récursives sur la structure du STMDD. Puisque la preuve par induction n'est pas traitée par Z3, la théorie Safety ne peut pas être décrite par son axiomatique dans Z3.

Dans *Safety*, un STMDD est une constante interprétée et les prédicats *isSafeR* et *isSafeOrder* sont des inégalités sur les indicateurs calculés sur un STMDD. Or il existe un nombre fini d'alternatives par instance de composant du système initial, donc les variables de substitution appartiennent à des ensembles finis. Par conséquent, il est possible de calculer les indicateurs pour tous les choix possibles de substitution et de les comparer aux bornes fixées par l'utilisateur. De ce fait, la satisfiabilité d'une conjonction de prédicats de la théorie Safety est décidable.

10.3 Encodage du problème DSE

10.3.1 Traduction du STMDD

La première étape de l'encodage consiste à traduire le STMDD du système initial comme une constante interprétée de sorte *STMDD*. Pour cela nous introduisons la méthode de traduction translateSTMDD, donnée par l'algorithme 4, d'un STMDD vers une constant interprétée de sorte *STMDD*. Cette traduction repose sur les éléments suivants :

- la liste $l = (N_1, \cdots, N_n)$ des nœuds du STMDD triés par un ordre topologique inverse du STMDD ;
- ConstantDefinition type représentant une définition de constante interprétée dans SMT-Lib ;
- getSubVarOf(N) renvoyant la variable de substitution associée à l'instance de composant étiquetée sur le nœud N ;
- getNodeName(N) renvoyant le nom de la constante MSFOL représentant le nœud N ;
- getSonsOf(N) retourne les fils du nœud N
- les fonctions de construction des termes MSFOL correspondant à l'application d'une commande SMT-Lib ;
- getOrder(N, tr) (respectivement getProb) renvoyant le tableau order (respectivement prob) attachés à la trace tr.

La racine du STMDD est alors le dernier nœud de l (puisqu'il est le premier dans un ordre topologique), donc la constante interprétée représentant le STMDD est le dernier élément de translateSTMDD(l).

Notons que les tableaux MSFOL représentant order et prob sont indexés par des entiers (*Sub* est un alias des entiers) et non des identificateurs de composant. Par conséquent, l'index d'une alternative c pour une instance *inst* est donné par la position de c dans la liste des alternatives donnée par la configuration du problème. Par la suite, nous utilisons les entiers $nbAlt_{inst}$ représentant le nombre d'alternative pour l'instance *inst* c'est-à-dire la taille des tableaux order et prob.

10.3.2 Encodage de l'espace des architectures et des exigences de sûreté

Le problème SMT encodant le problème DSE contient alors :

- des variables de substitution de sorte *Sub* pour chaque instance de composant contenue dans le système étudié ;
- le STMDD du système encodé comme une constante interprétée de sorte *STMDD* ;
- les contraintes d'ordre et de fiabilité exprimées à l'aide des prédicats *isSafeOrder* et *isSafeR* ;
- des contraintes sur les variables de substitution afin de s'assurer qu'au plus une alternative est choisie pour chaque instance de composant.

Soit $inst_1, \cdots, inst_n$ les instances de composant d'un système s, *Root* la constante interprétée de sorte *STMDD* encodant le STMDD de s, $last_{inst_i}$ le dernier index des tableaux order et prob pour l'instance $inst_i$ c'est-à-dire $nbAlt_{inst} - 1$, `atMostOne` un prédicat vrai si et seulement si au plus un littéral est vrai parmi un ensemble donné et $rReq, orderReq$ les contraintes sur la fiabilité et l'ordre de la solution du problème DSE. Alors le problème DSE consistant à trouver une architecture obtenue par substitution des instances de composant de s respectant les exigences de sûreté est encodé comme montré dans le listing 10.1.

Algorithme 4 Encodage du STMDD en MSFOL

```
function translateSTMDD(l) : List[ConstanteDefinition]
    l match
      case Nil ⇒ return Nil
      case Terminal(1) :: tail ⇒ return one : : translateSTMDD(tail)
      case Terminal(0) :: tail ⇒ return zero : : translateSTMDD(tail)
      case Terminal(⊥) :: tail ⇒ return bottom : : translateSTMDD(tail)
      case N :: tail ⇒
          var ←getSubVarOf(N)
          name ←getNodeName(N)
          sons ←mkSonList(N,getSonsOf(N))
          trNode ←defineConst(name, STMDD, node(var, sons))
          return trNode : : translateSTMDD(tail)
end function

function mkSonList(node, sons) : SonList
    sons match
      case Nil ⇒ getMSFOLNil()
      case tr ↦ son :: tail ⇒
          sonName ←getNodeName(son)
          order ←getOrder(node, tr)
          prob ←getProb(node, tr)
          return cons( mkLabel (order, prob, sonName), mkSonList(tail))
end function
```

```
;; variables de substitution
(declare-const sub_inst1 Sub) ... (declare-const sub_instn Sub)

;;contraintes de surete
(assert (isSafeR Root rReq))
(assert (isSafeOrder Root orderReq))

;;contraintes sur les variables de substitutions
(assert (atMostOne (choose sub_inst1 0) ... (choose sub_inst1 last_inst1)))
...
(assert (atMostOne (choose sub_instn 0) ... (choose sub_inst1 last_instn)))
```

Listing 10.1 – Encodage du problème DSE

Exemple 10.1 (Encodage problème DSE)**.** *Le listing 10.2 présente l'encodage du problème* DSE *de* Rosace *introduit dans l'exemple 9.14 où l'ordre de* Rosace *doit être supérieure à 2 et la fiabilité au bout de* 10^2 *heure doit être supérieure à 0.99.*

```
;; variables de substitution
(declare-const sub_FVa Sub)
(declare-const sub_CVa Sub)

;; traduction des noeuds du STMDD de Rosace
(define-const emptyOrder (Array Sub InfInt) ((as const (Array Sub InfInt)) (mkInfInt 0)))
(define-const emptyUnr (Array Sub Real) ((as const (Array Sub Real)) 1.0))
(define-const CVaNode STMDD (node
  sub_CVa
  (cons (mkLabel (store (store emptyOrder 0 (mkInfInt 1)) 1 (mkInfInt 1)) (store (store emptyUnr 0 0.01) 1  0.04) one)
   (cons (mkLabel (store (store emptyOrder 0 (mkInfInt 1)) 1 (mkInfInt 2)) (store (store emptyUnr 0 0.01) 1  0.0001) one)
    (cons (mkLabel (store (store emptyOrder 0 (mkInfInt 0)) 1 (mkInfInt 0)) (store (store emptyUnr 0 0.98) 1  0.9599) one) nil )))))

(define-const FVaNode STMDD (node
  sub_FVa
   (cons (mkLabel (store (store emptyOrder 0 (mkInfInt 1)) 1 (mkInfInt 1)) (store (store emptyUnr 0 0.01) 1  0.04) one)
   (cons (mkLabel (store (store emptyOrder 0 (mkInfInt 1)) 1 (mkInfInt 2)) (store (store emptyUnr 0 0.01) 1  0.0001) one)
    (cons (mkLabel (store (store emptyOrder 0 (mkInfInt 0)) 1 (mkInfInt 0)) (store (store emptyUnr 0 0.98) 1  0.9599) CVaNode) nil )))))

;; definition de la variable STMDD
(define-const Root STMDD FVaNode)

;;contraintes de surete
(assert (isSafeR Root 0.99))
(assert (isSafeOrder Root 2))

;;contraintes sur les variables de substitutions
(assert (atMostOne (choose sub_FVa 0) (choose sub_FVa 1) (choose sub_FVa 2)))
(assert (atMostOne (choose sub_CVa 0) (choose sub_CVa 1) (choose sub_CVa 2)))
```

Listing 10.2 – Encodage du problème DSE de Rosace

10.4 Solveur de la théorie Safety

Nous avons vu dans la section 10.2 que la théorie ne peut pas être définie par l'intégration de son axiomatique (basée sur les théories NIRA, la théorie des tableaux, la théorie des types algébriques et la théorie des fonctions récursives) dans un problème SMT. Ainsi, nous proposons de développer le solveur de la théorie Safety c'est-à-dire la procédure décidant de la satisfiabilité d'une conjonction de prédicats *isSafeR*, *isSafeOrder* et *choose*.

10.4.1 Implantation de la procédure de décision

Pour définir le solveur de la théorie Safety, nous donnons l'implantation de la fonction CHECKSAT(C :CUBE) :EITHER[MODEL,CLAUSE], décrite par l'algorithme 5, vérifiant si la conjonction des littéraux (représentant des prédicats de théorie) contenus dans un cube C est bien satisfiable modulo Safety. Cette dernière construit un choix de substitution M à partir des prédicats *choose* contenus dans C. Puis si un des prédicats de sûreté n'est pas satisfiable avec M alors un clause de conflit est calculée, autrement le modèle est renvoyé.

Algorithme 5 Implantation de CHECKSAT pour la théorie Safety

```
function checkSat(c) : Either[Model,Clause]
    M ← {var ↦ val|(choose var val) ∈ c}
    RChecks ← {(isSafeR req N) ∈ c}
    OrderChecks ← {(isSafeOrder req N) ∈ c}
    for (isSafeR req N) ∈ RChecks do
        if checkR(N, req, M) = Unsat then
            return conflict((isSafeR req N), M)
        end if
    end for
    for (isSafeOrder req N) ∈ OrderChecks do
        if checkOrder(N, req, M) = Unsat then
            return conflict((isSafeOrder req N), M)
        end if
    end for
    return M
end function
```

Focalisons nous sur l'implantation des fonctions CHECKR et CHECKORDER utilisées par CHECKSAT. Ces fonctions vérifient qu'un candidat obtenu par un ensemble de substitutions respecte bien une exigence de sûreté. Puisque le solveur de théorie est utilisé de manière paresseuse, celui-ci ne sera appelé que lorsque le solveur SMT a construit un modèle *complet* des choix de substitution c'est-à-dire un seul prédicat *choose* vrai par variable de substitution. Vérifier une exigence revient alors à calculer l'indicateur sur le STMDD du système à l'aide des équations 9.1 et 9.4 puis à comparer la valeur obtenue à la borne donnée par l'exigence.

Néanmoins pour générer et minimiser les clauses de conflits nous avons besoin de vérifier si un choix partiel de substitution (c'est-à-dire tel que, pour certains composants, au moins deux prédicats *choose* peuvent être vrais) satisfait une exigence. Plus précisément, considérons $\mathcal{I}$ l'ensemble des variables de substitutions du problème DSE, et $M = \{var \mapsto val | var \in \mathcal{I}, val \in \mathbb{N}^+\}$ un modèle de

ces variables, on dit alors que le modèle est *partiel* si $\exists var \in \mathcal{I}, \neg\exists var' \mapsto val \in M, var = var'$ on notera ce cas $M(var) = \texttt{undef}$.

Pour un modèle partiel, les règles de calcul des indicateurs doivent être adaptées. Nous proposons de calculer une sur-approximation de la fiabilité et de l'ordre à l'aide de l'arithmétique d'intervalle comme introduit par la définition 10.1. Bien évidemment la comparaison des intervalles aux exigences ne permet pas toujours d'établir la conformité d'un modèle partiel, dans ce cas le solveur de théorie renvoie UNKNOWN.

Définition 10.1 (Bornes)**.** *Soit s un système, M un modèle partiel des variables de substitution, $order^+$ (respectivement $order^-$) l'ordre maximal (respectivement minimal) contenu dans un tableau order, $prob^+$ (respectivement $prob^-$) la probabilité maximale (respectivement minimale) contenue dans un tableau prob et N la constante encodant le* STMDD *de s alors*

Si $N = (var, sons)$ nous calculons les bornes suivantes :

$$
\begin{aligned}
prob^+(N) &= \sum_{(prob,order,son)\in sons} prob^+(son) \times \begin{cases} prob(val) & si\ M(var) = val \\ prob^+ & sinon \end{cases} \\
prob^-(N) &= \sum_{(prob,order,son)\in sons} prob^-(son) \times \begin{cases} prob(val) & si\ M(var) = val \\ prob^- & sinon \end{cases} \\
order^+(N) &= \min_{(prob,order,son)\in sons} \left(order^+(son) + \begin{cases} order(val) & si\ M(var) = val \\ order^+ & sinon \end{cases} \right) \\
order^-(N) &= \min_{(prob,order,son)\in sons} \left(order^-(son) + \begin{cases} order(val) & si\ M(var) = val \\ order^- & sinon \end{cases} \right)
\end{aligned}
$$

Sinon si N est un terminal alors

$$order(N) = order^+(N) = order^-(N) \quad et \quad prob(N) = prob^+(N) = prob^-(N)$$

Propriété 10.1. *Soit s un système, M un modèle des variables de substitutions, mincard (respectivement R) l'ordre (respectivement la fiabilité) de s et N la racine du* STMDD *de s alors si M est partiel :*

$$order^-(N) \leq mincard \leq order^+(N) \quad et \quad 1 - prob^+(N) \leq R \leq 1 - prob^-(N)$$

sinon si M est complet :

$$order^-(N) = order^+(N) = mincard \quad et \quad 1 - prob^-(N) = 1 - prob^+(N) = R$$

Démonstration. Si M est complet alors les formules des bornes $order^-(N), order^+(N)$ (respectivement $prob^-(N), prob^+(N)$) sont celles de $order(N)$ (respectivement $prob(N)$) donc d'après les equations 9.3 et 9.5

$$order^-(N) = order^+(N) = mincard \quad \text{et} \quad 1 - prob^-(N) = 1 - prob^+(N) = R$$

Lorsque M est partiel, prouvons $order^-(N) \leq mincard$ par induction sur la structure de N.

Si $N = Terminal(0)$ ou $N = Terminal(1)$ ou $N = Terminal(\perp)$ alors par la définition 10.1 $order^-(N) = order(N)$ et par l'équation 9.5 $order(N) = mincard$, d'où

$$order^-(N) \leq mincard$$

Si $N = node(var, sons)$ posons l'hypothèse d'induction suivante :

$$\forall (order, prob, son) \in sons, order^-(son) \leq order(son)$$

Opérons par disjonction de cas sur $M(var)$:

- Si $M(var) = val$ alors

  $$order^-(N) = \min_{(prob,order,son)\in sons} (order^-(son) + order(val))$$

  en utilisant l'hypothèse de récurrence on obtient

  $$order^-(N) \leq \min_{(prob,order,son)\in sons} (order(son) + order(val))$$

  Donc $order^-(N) \leq order(N)$ or par la propriété 9.1 $order(N) = mincard$ d'où $order^-(N) \leq mincard$
- Si $M(var) =$ `undef` alors considérons tout choix possible $var \mapsto val, val \in [0, nbAlt_{var}[$. Par hypothèse d'induction on a

  $$\min_{(prob,card,son)\in sons} (order^-(son) + order(val)) \leq \min_{(prob,order,son)\in sons} (order(son) + order(val))$$

  or $order^- \leq order(val)$ donc

  $$\min_{(prob,order,son)\in sons} (order^-(son) + order^-) \leq \min_{(prob,order,son)\in sons} (order(son) + order(val))$$

  on a alors $order^-(N) \leq order(N)$ or par l'équation 9.5 $order(N) = mincard$ d'où

  $$order^-(N) \leq mincard$$

Les preuves pour les autres bornes sont faites de manière similaire. □

Exemple 10.2 (Calcul des bornes)**.** *Reprenons les calculs de l'exemple 9.15, lorsque F_{Va} est considéré comme un filtre mais qu'aucun choix n'est fait sur C_{Va}. Ceci revient à dire que seul le prédicat* (`choose` sub_{FVa} 0) *est vrai. Les bornes de mincard calculées à partir du* STMDD *de la figure 10.3 sont :*

$$\left.\begin{array}{l} order^-(N) = min(1, 0 + min(min(1,1), min(1,2), min(0+\infty, 0+\infty))) = 1 \\ order^+(N) = min(1, 0 + min(max(1,1), max(1,2), max(0+\infty, 0+\infty))) = 1 \end{array}\right\} \Rightarrow mincard = 1$$

La vérification des prédicats de sûreté est basée sur le calcul des bornes des indicateurs introduites par la définition 10.1. Or il n'est pas toujours nécessaire de parcourir l'ensemble du STMDD pour déterminer si un choix satisfait ou non un prédicat. En effet, si une des traces système représentée par le STMDD possède un ordre ou une probabilité ne respectant pas l'exigence alors quelles que soient les bornes calculées sur les autres traces, le choix n'est pas correct. Par conséquent, l'algorithme 6 (respectivement 7) vérifient la conformité d'un choix en calculant un *objectif* (respectivement un *budget*) lors du parcours du STMDD. Notons que, comme les indicateurs, cet objectif (respectivement budget) est défini par une sur-approximation obj^+ (respectivement bud^+) et sous-approximation obj^- (respectivement bud^-).

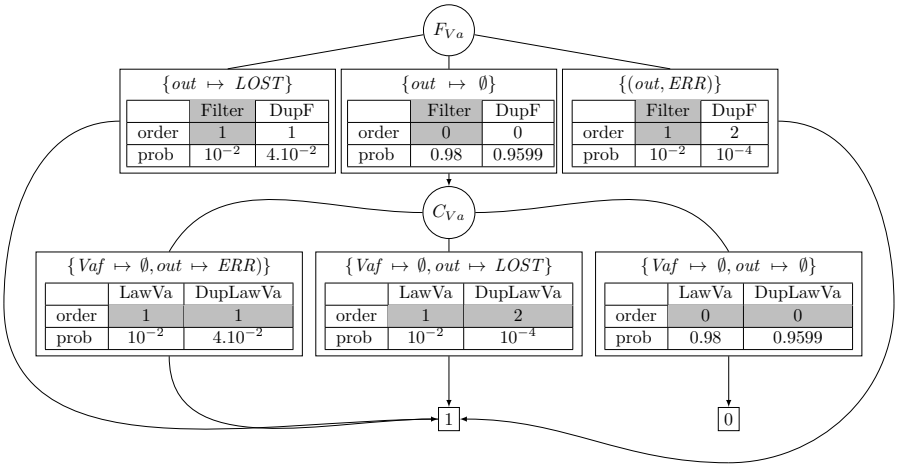


FIGURE 10.3 – STMDD de ROSACE

Pour la vérification de l'ordre présentée dans l'algorithme 6, les approximations obj^+ et obj^- de l'objectif représentent la contrainte d'ordre à laquelle est retirée l'ordre des traces étiquetées par les prédécesseurs du nœud N pour le choix M. Cet objectif est réévalué en considérant que la trace étiquetée par N se produit (ligne 5-6). Si celui-ci est atteint (ligne 7) alors le nombre d'événements nécessaires pour déclencher une trace système est supérieur à l'exigence d'ordre. Par contre, si le terminal 1 est accessible avec un objectif positif (ligne 17) alors la trace évaluée peut être déclenchée avec strictement moins d'événements que l'ordre donc le choix ne satisfait pas l'exigence. Dans les autres cas, il n'est pas possible de conclure.

Pour la vérification de la fiabilité présentée dans l'algorithme 7, les approximations bud^+ et bud^- du budget représentent l'exigence de défiabilité (c'est-à-dire $1 - req$) à laquelle la probabilité d'occurrence d'une partie des fils de N pour le choix M a été retirée. Ce budget est réévalué en retirant itérativement la probabilité qu'un fil se produise (ligne 8-10). Si celui-ci est épuisé (ligne 11) alors la probabilité qu'un sous-ensemble des fils de N se produise est supérieure à l'exigence de défiabilité donc le choix ne satisfait pas l'exigence de fiabilité. Par contre, si le budget est tenu jusqu'à la fin de l'évaluation (ligne 15) alors la probabilité de déclencher une trace de N est inférieure à l'exigence de défiabilité, le choix respecte bien l'exigence. Dans les autres cas, il n'est pas possible de conclure.

10.4.2 Génération des clauses de conflit

Intéressons nous maintenant à la fonction CONFLICT de génération des clauses de conflits. Lorsqu'un ensemble de choix de substitution entraîne un violation des exigences de sûreté, le solveur de théorie doit identifier le sous-ensemble des choix de substitution expliquant le conflit et générer une clause de conflit.

L'objectif principal est alors de rejeter le plus grand nombre possible de candidats afin de réduire l'espace de recherche et donc le temps d'exploration. Nous proposons un implantation de CONFLICT, présentée par l'algorithme 8, vérifiant que chaque choix m contenu dans un modèle M est nécessaire pour expliquer un conflit pour un prédicat de sûreté r c'est-à-dire que $M \setminus \{m\}$ ne génère plus de

Algorithme 6 Vérification de l'ordre

```
1: function checkOrder(N, obj+, obj−, M) : Status
2:     r match
3:         case node(sons, var) ⇒
4:             M(var) match
5:                 case val ⇒ (nObj+, nObj−) ← (obj+ − order(val), obj− − order(val))
6:                 case undef ⇒ (nObj+, nObj−) ← (obj+ − order−, obj− − order+)
7:             if obj+ < 0 then
8:                 return Sat
9:             else if ∃s ∈ sons,checkOrder(s, nObj+, nObj−, M)= Unsat then
10:                return Unsat
11:            else if ∃s ∈ sons,checkOrder(s, nObj+, nObj−, M)= Unknown then
12:                return Unknown
13:            else
14:                return Sat
15:            end if
16:        case one ⇒
17:            if obj− > 0 then return Unsat
18:            else if obj+ ≤ 0 then return Sat
19:            else return Unknown
20:            end if
21:        case _ ⇒ return Sat
22: end function
```

Algorithme 7 Vérification de la fiabilité

```
1: function checkR(N, req, M) : Status
2:     r match
3:         case node(sons, var) ⇒
4:             (bud+, bud−) ← (1 − req, 1 − req)
5:             for s ∈ sons do
6:                 M(var) match
7:                     case val ⇒
8:                         (bud+, bud−) ← (bud+ − prob(val)prob−(s), bud− − prob(val)prob+(s))
9:                     case undef ⇒
10:                        (bud+, bud−) ← (bud+ − prob−prob−(s), bud− − prob+prob+(s))
11:                if bud+ < 0 then
12:                    return Unsat
13:                end if
14:            end for
15:            if bud− ≥ 0 then
16:                return Sat
17:            else
18:                return Unknown
19:            end if
20:        case one ⇒ return Unsat
21:        case _ ⇒ return Sat
22: end function
```

conflit (ligne 6-10). Si ce n'est pas le cas alors le choix n'est pas nécessaire pour expliquer le conflit. Ce processus de simplification assure de générer des clauses de conflit minimales en faisant un nombre d'appel linéaire dans la taille du modèle à minimiser.

Algorithme 8 Génération des clauses de conflit

```
1: function CONFLICT(r, M) : Clause
2:     conflict ← M
3:     for m ∈ M do
4:         r match
5:             case (isSafeOrder N req) ⇒
6:                 if CHECKORDER(N, conflict \ {m}, req) = UNSAT then
7:                     conflict ← conflict \ {m}
8:                 end if
9:             case (isSafeR N req) ⇒
10:                if CHECKR(N, conflict \ {m}, req) = UNSAT then
11:                    conflict ← conflict \ {m}
12:                end if
13:    end for
14:    return {¬(choose var val)|var ↦ val ∈ conflict}
15: end function
```

10.5 Résumé

Nous avons introduit dans ce chapitre la théorie Safety nous permettant d'encoder le STMDD comme une constante interprétée et les contraintes de sûreté comme des prédicats de la MSFOL. Par la suite nous avons utilisé ces constantes et prédicats pour encoder le problème DSE comme un problème SMT par un solveur SMT augmenté de la théorie Safety. Finalement, nous avons présenté les algorithmes permettant d'implanter le solveur de la théorie Safety. Nous allons maintenant introduire l'outil KCR ANALYSER implantant les méthodes d'analyse et de résolution des problèmes DSE présentées dans ce document et évaluer l'efficacité de nos méthodes par rapport aux approches existantes sur un ensemble de cas d'étude.

Quatrième partie

Mise en œuvre de la solution proposée

Chapitre 11

Présentation de KCR Analyser

Ce chapitre introduit l'outil KCR Analyser implantant les méthodes d'analyse et de résolution des problèmes DSE présentées dans ce document. Pour cela, nous présentons, dans la section 11.1, un rapide aperçu de l'architecture de KCR Analyser. Celle-ci étant basée sur l'assemblage de modules d'analyse, nous présentons, dans la section 11.2, les différents modules ainsi que les dépendances entre eux. Puis, nous détaillons, dans la section 11.3, l'intégration du solveur de la théorie Safety et du solveur Z3 au sein d'un nouveau solveur capable de résoudre un problème SMT contenant des prédicats de la théorie Safety. Finalement, nous présentons succinctement l'éditeur KCR intégré à l'outil.

11.1 KCR Analyser : un analyseur modulaire

KCR Analyser [1] est un outil d'édition et d'analyse de modèles dysfonctionnels de systèmes décrits dans le langage KCR (cf chapitre 7). Nous avons choisi d'implanter KCR Analyser à l'aide du langage Scala car celui-ci permet :

- de créer un logiciel multi-plateforme (exécution par la JVM),
- une interopérabilité avec Java et ses librairies (notamment SWING pour l'interface graphique),
- une programmation objet et fonctionnelle,
- des éléments de programmation générique,
- un environnement de compilation complet (SBT).

KCR Analyser a été pensé comme une librairie de *modules* d'analyse que l'utilisateur compose pour créer un analyseur répondant à ses besoins. Cette librairie de modules est implantée à l'aide d'un patron de conception appelé *cake pattern* introduit par [70]. Plus précisément, les modules sont implantés comme des entités abstraites (des *traits* en Scala) soumises à des dépendances avec d'autres modules c'est-à-dire les services nécessaires au fonctionnement du module (un module est donc une *part* du *gâteau*). Chaque module peut être spécialisé pour proposer différentes implantations des services fournis (ces spécialisations forment les *couches* du *gâteau*). Ainsi la librairie de modules de KCR Analyser offre un ensemble de spécialisations des différentes techniques disponibles pour mener une analyse (par exemple énumération des coupes par SMT ou par BDD).

1. disponible sur `www.onera.fr/en/staff/kevin-delmas`

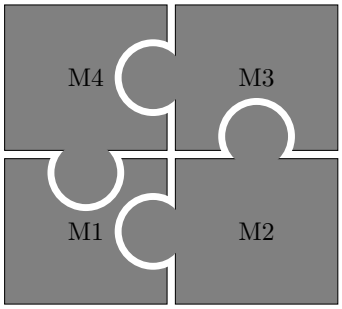


FIGURE 11.1 – Composition de modules

Comme le montre la figure 11.1, un module est similaire à une pièce d'un *puzzle* où les encoches sont les services demandés et les dentelures sont les services fournis par le module. En SCALA, les dépendances du module `M1` sont décrites à l'aide de la *self type annotation* qui impose que toute instanciation du module `M1` doit être obtenue par un *mélange* d'un module `M2` et `M4`, comme donné dans le code suivant :

```
trait M1 {
  /** dependances */
  this: M2 with M4 =>

  /** code */
}
```

Pour créer un analyseur il suffit alors de combiner les modules à l'aide de la fonction dite de *mélange* (*mixin* en anglais) assurant statiquement que les dépendances des modules utilisés sont satisfaites (vérification réductible à un problème de typage). Par exemple, l'analyseur de la figure 11.1 est construit comme suit :

```
val analyzer= new M1 with M2 with M3 with M4 {
  /** concretisation des membres abstraits */
}
```

Par ailleurs, chaque module d'analyse de KCR ANALYSER possède un ou plusieurs attributs *non modifiables* (*immutable* en anglais) stockant la résultat de l'analyse opérée par le module. Ces attributs sont initialisés/évalués de manière paresseuse c'est-à-dire que leur valeur n'est calculée que lorsque qu'un module tiers ou l'utilisateur souhaite accéder à la valeur de cet attribut. Par exemple, considérons que chaque module `Mi` possède un attribut `resultMi` calculé par une fonction `computeResMi` du module `Mi`. Alors l'attribut est déclaré comme suit :

```
trait Mi {
  /** dependances */
  this: Mx with My =>

  /** variable non modifiable calculee de maniere paresseuse*/
  lazy val resultMi : ResultType = computeResMi()

  /** code */
}
```

Considérons maintenant que les fonctions `computeResMi` utilisent les attributs `resultMx` des modules dont `Mi` dépend. Dans ce cas, lors de la création de l'analyseur `analyzer` introduite précédemment, aucune analyse n'est lancée puisque les valeurs des attributs ne sont pas exigées lors de l'initialisation. Par contre si l'utilisateur demande la valeur de `resultM3` alors la fonction `computeResM3` est exécutée. Or celle-ci à besoin de la valeur de `resultM2`, d'où l'exécution de `computeResM2`. Une fois cette requête effectuée, l'utilisateur demande maintenant la valeur de `resultM4`, ceci déclenche l'exécution de `computeResM4` et comme les valeurs `resultM2` et `resultM3` sont déjà disponibles, aucune action supplémentaire n'est nécessaire.

En résumé, la construction d'un analyseur par assemblage de modules interdépendants stockant les résultats d'analyse dans des attributs paresseux permet à l'utilisateur :

- de créer un analyseur ne contenant que les analyses souhaitées
- de ne mener que les analyses nécessaires pour obtenir la valeur d'une analyse donnée
- de ne mener chaque analyse qu'une seule fois
- tout en garantissant statiquement que l'assemblage de modules satisfait les dépendances.

11.2 Modules de KCR Analyser

La figure 11.2 présente les différents modules [2] disponibles pour construire un analyseur de systèmes KCR et les dépendances entre modules (où une flèche de M vers M' signifie que M dépend de M'). Notons que les modules implantent les fonctions présentées dans le processus de modélisation et d'analyse des systèmes introduit dans la section 6.2.

11.2.1 Dépendances externes

Outres les dépendances présentées par la figure 11.2, les modules contenus dans KCR Analyser reposent sur les librairies/outils suivants :

- JavaBDD [1] offre une API Java pour manipuler et construire efficacement des BDDs (possibilité d'utiliser les librairies CUDD, BuDDy ou CAL (écrites en C) pour mener les calculs). Cette performance justifie son utilisation notamment pour les modules utilisant intensément les BDDs comme l'interpréteur, le calculateur de fiabilité, ou encore le constructeur du STMDD.
- AntLR [79] est un outil de génération automatique de lexers et parsers à partir d'une description EBNF d'un langage. L'utilisation de cet outil nous assure une bonne confiance dans le lexer/parser de KCR, et nous offre une grande flexibilité dans la spécification du langage.
- Z3 est un solveur SMT développé par Microsoft que nous utilisons pour résoudre les problèmes SMT. Ainsi KCR Analyser ne fait que formuler les problèmes SMT et délègue la résolution à Z3, assurant ainsi une bonne confiance dans la validité des analyses et réduit la taille du code de KCR Analyser. Par ailleurs, Z3 est, à notre connaissance, le seul solveur capable de traiter des problèmes de la théorie UFBV avec quantificateurs dont nous avons besoin pour analyser les systèmes décrits en KCR.
- SharpCDCL [56] est un solveur Sat proposant des fonctions de projections et d'énumération de modèles que nous utilisons pour l'énumération de coupes respectivement pour projeter les clauses de conflit sur l'ensemble des événements de défaillance et pour énumérer les coupes minimales.

2. l'implantation de ce modules est donnée dans le répertoire `src/main/scala/analyzers` des fichiers sources de KCR Analyser disponibles sur `www.onera.fr/en/staff/kevin-delmas`

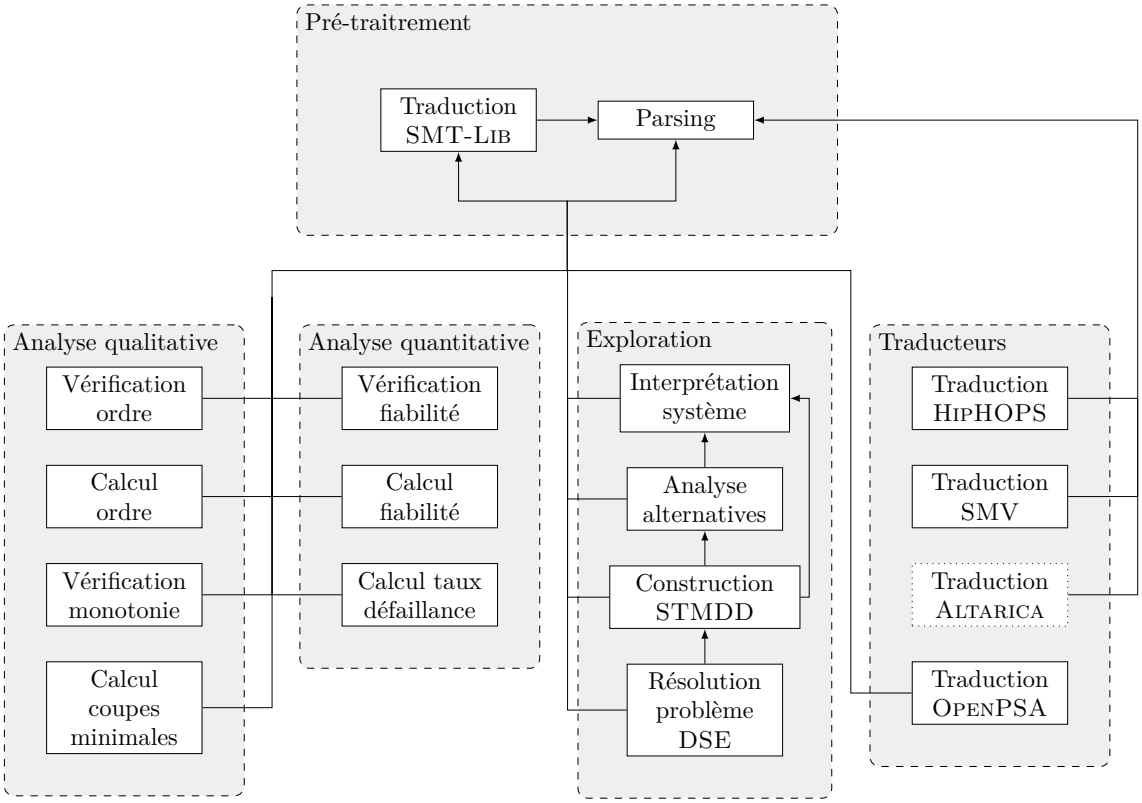


Figure 11.2 – Dépendances des modules de KCR Analyser

11.2.2 Modules de pré-traitement

Les modules de pré-traitement construisent les structures de données utilisées par les autres modules pour analyser le système. Par la suite, le nom du *trait* Scala implantant un module est donné entre parenthèses.

Parsing (`Parsing`) analyse les fichiers de modélisation KCR et extrait la modélisation du système et les données contenues dans la configuration du problème. Pour cela le parseur et le lexeur ont été générés à partir de la description AntLR de l'EBNF de KCR (voir chapitre 7). Puis le résultat du parsing est analysé afin d'assurer que :

- le typage des flots est correct ;
- les définitions de flots ne sont pas cycliques ;
- les identificateurs de flots, d'événements et composants sont tous définis ;
- les restrictions sur les définitions de composant sont respectées (système clos, composant déclaré comme atomique ne contient pas d'instanciations, etc).

Traduction SMT-Lib (`SMTTranslation`) se charge de traduire le modèle initial et génère la fonction de structure à l'aide du solveur Z3 (cf section 8.1) intégré à KCR grâce à son API Java.

11.2.3 Modules d'analyse

L'outil KCR Analyser fournit un ensemble de modules de calcul des indicateurs qualitatifs et quantitatifs de sûreté mais aussi des modules de vérification de la conformité du système initial par rapport aux exigences données par la configuration.

Vérification de l'ordre (`AcceptableCardAnalyser`) résout le problème SMT présenté dans la section 8.2 à l'aide de Z3. Si le problème est Unsat alors le système respecte l'exigence d'ordre donnée par la configuration. Sinon le solveur renvoie un contre-exemple sous forme d'un ensemble d'événements violant la contrainte.

Calcul de l'ordre (`CardinalityComputer`) calcule l'ordre du système par résolution itérative de problèmes SMT (cf section 8.2).

Vérification de la monotonie (`MonotonyAnalyser`) résout le problème SMT présenté dans la section 8.4 afin d'établir si la fonction de structure est monotone ou non. Si le problème est Unsat alors la fonction est monotone, sinon un contre-exemple est généré par le solveur.

Calcul des coupes minimales (`CutEnumerator`) énumère les coupes minimales d'un système jusqu'à une certaine borne k. Plusieurs spécialisations sont disponibles :

- calcul fondé sur la méthode itérative (`IterativeCutEnumerator`) présentée dans la section 8.3 et utilisant l'énumération de modèles et la fonction de projection du solveur SharpCDCL ;
- calcul fondé sur l'analyse du BDD (`BDDCutEnumerator`) de la fonction de structure (voir 2.2) à l'aide de la librairie JavaBDD ;
- calcul délégué directement aux outils HipHOPS (`HipHOPSCutEnumerator`), XFTA (`XFTACutEnumerator`) ou XSAP (`XSAPCutEnumerator`).

Calcul de la fiabilité (`ReliabilityComputer`) construit le BDD de la fonction de structure à l'aide de la librairie JavaBDD puis calcule la fiabilité par la méthode des BDDs introduite dans la section 2.1.

Vérification de la fiabilité (`AcceptableReliabilityAnalyser`) calcule la fiabilité et vérifie que celle-ci est bien supérieure à la borne donnée dans la configuration.

Calcul du taux de défaillance (`SymbolicFailureRateComputer`) calcule la formule symbolique de la fiabilité et génère un script Python calculant la formule du taux de défaillance comme décrit dans la section 8.5.

11.2.4 Modules d'exploration

L'outil KCR Analyser implante le processus de résolution présenté dans le chapitre 10 c'est-à-dire encodant et résolvant un problème DSE comme un problème SMT :

Interprétation système (`DSLInterpreter`) implante l'interpréteur de KCR présenté dans la section 9.2.

Analyse alternatives (`SubstitutionAnalyser`) fournit une double fonctionnalité :

- vérifie que les alternatives considérées dans le problème sont légales (voir section 9.3) ;
- caractérise les alternatives et construit les tableaux contenant la valeur des indicateurs locaux (voir section 9.3).

Construction du STMDD (`STMDDBuilder`) construit le STMDD du système à partir de l'interprétation des flots du système et des caractérisations des alternatives comme présenté dans le chapitre 9.

Résolution du problème DSE (`DSEProblemSolver`) résout le problème SMT à l'aide de Z3 combiné au solveur de la théorie Safety et fournit un choix de substitution si le problème est SAT. Il est possible d'exporter la solution comme un fichier KCR pour vérifier *a posteriori* que la solution respecte bien les exigences.

11.2.5 Modules de traduction

L'outil KCR ANALYSER offre des fonctions de traduction des modèles KCR vers les langages HIPHOPS (`HipHOPSExporter`), SMV (`SMVExporter`) et l'export de la fonction de structure en OPENPSA (`OpenPSAExporter`). Un export vers le langage ALTARICA est prévu mais n'est pas aujourd'hui implanté. Ceci permet d'utiliser les outils tiers HIPHOPS, XSAP et XFTA pour calculer les coupes minimales du système et HIPHOPS pour résoudre le problème d'exploration.

11.3 Intégration du solveur de Safety

Le solveur SMT augmenté de la théorie Safety [3] est basé sur une utilisation paresseuse du solveur de la théorie Safety (voir chapitre 10). La figure 11.3 détaille le fonctionnement du solveur SMT augmenté de la théorie Safety (que nous notons SMT $\cup$ Safety) :

Abstraction prédicat Safety remplace les prédicats de la théorie Safety par des propositions booléennes, le problème résultant ne fait donc plus appel à la théorie Safety ;

Z3 Check Sat Z3 est alors appelé sur ce nouveau problème pour vérifier si celui-ci est satisfiable, si non alors une preuve est renvoyée ;

Extraction modèle prédicats Safety si le problème SMT est SAT alors le modèle des propositions d'abstraction des prédicats de Safety est récupéré. Ce modèle est ensuite traduit comme un cube de prédicats de la théorie Safety.

Safety Check Sat le solveur Safety vérifie alors que le cube de prédicats est satisfiable ; si oui alors le modèle renvoyé par Z3 est complété par celui des variables de substitution ; sinon la clause de conflit générée par le solveur de Safety est ajoutée au problème SMT et le processus de résolution est relancé sur ce nouveau problème.

11.4 Interface utilisateur

KCR ANALYSER possède une interface classique en ligne de commande mais aussi une interface graphique (basée sur SWING) présentée sous la forme d'un éditeur de texte pour le code KCR comme montré sur la figure 11.4.

L'interface est séparée en trois parties : le menu, l'éditeur de code et la console de résultat. Les champs du menu sont :

File/Edit contenant les fonctions classiques d'édition de texte ;

Analysis permettant de lancer les analyses du système, c'est-à-dire la vérification d'exigences, le calcul d'indicateurs de sûreté et la résolution du problème d'exploration ;

View permet de visionner le BDD de la fonction de structure ou le STMDD du système étudié ;

3. `AugmentedSafetySolver` dans le répertoire `src/main/scala/util/analyzers` du code source de KCR ANALYSER

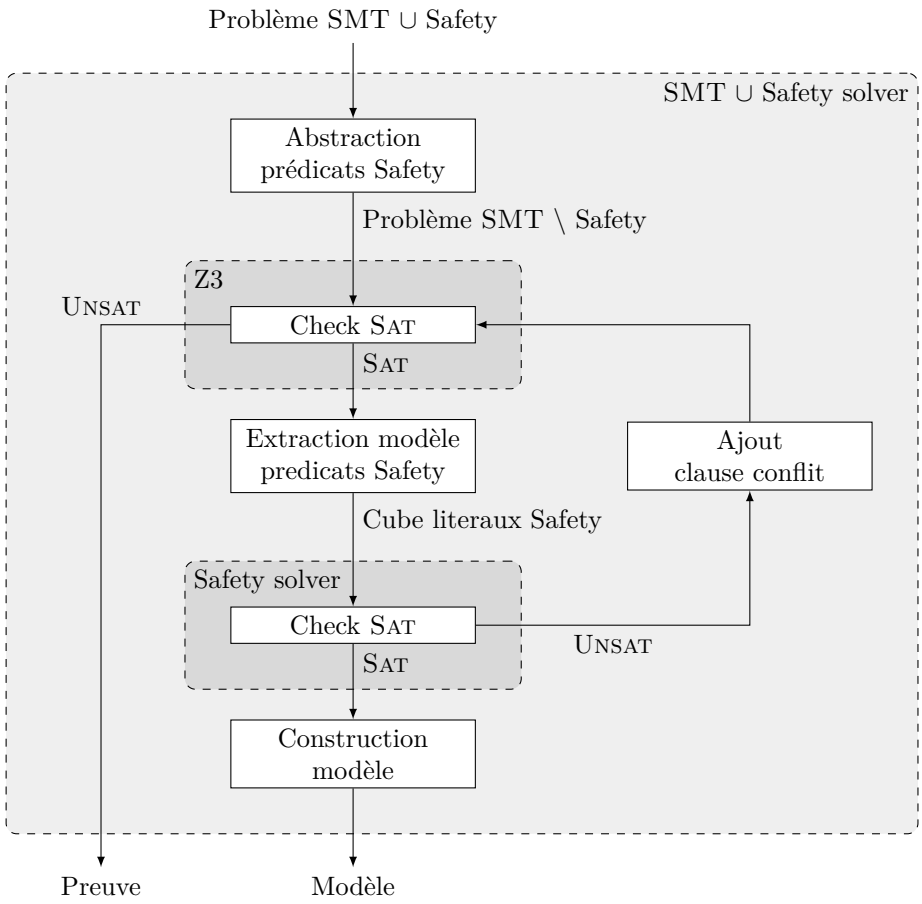


Figure 11.3 – Résolution SMT ∪ Safety paresseuse

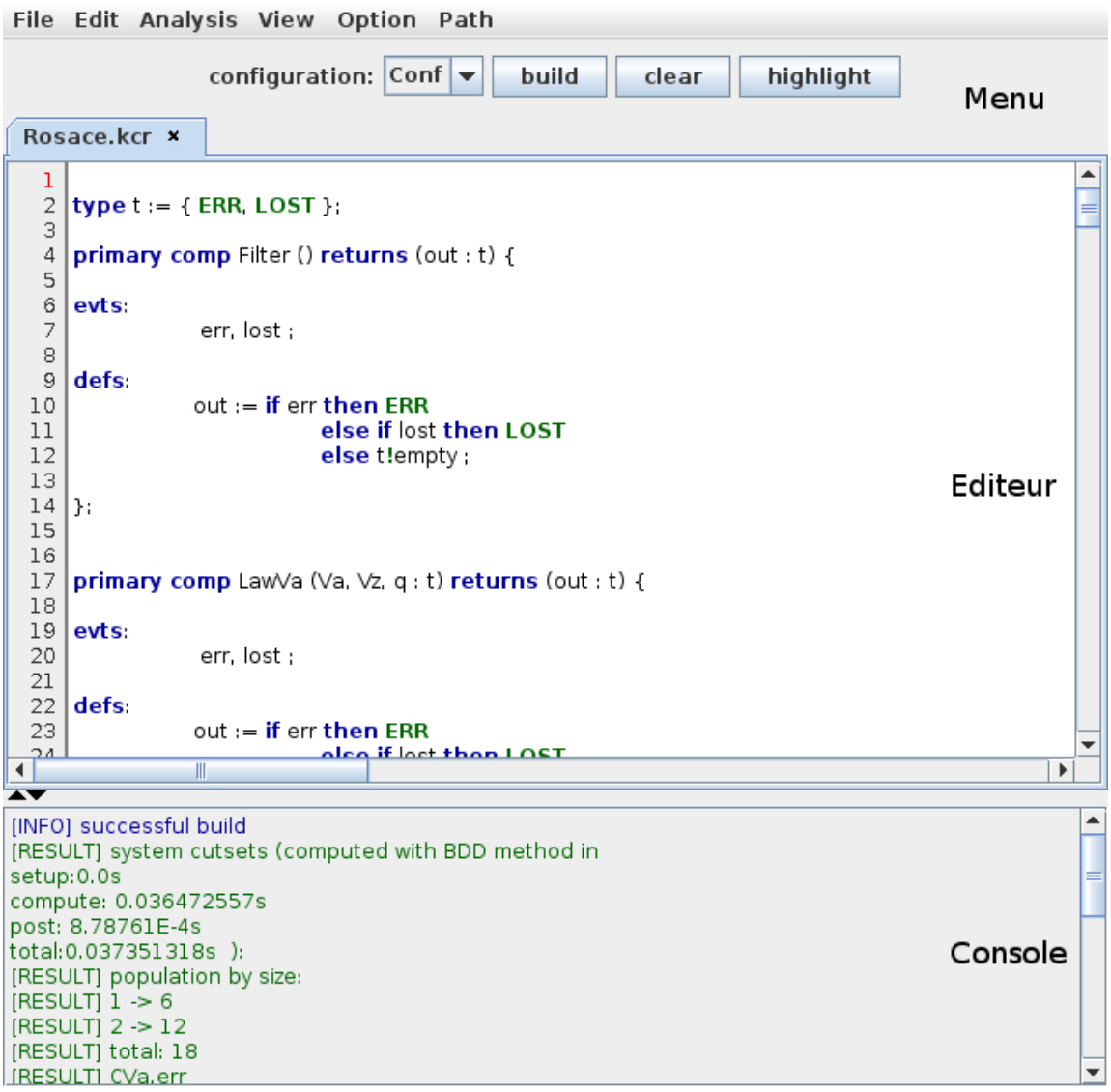


FIGURE 11.4 – Interface graphique que KCR ANALYSER

Option permet de configurer les analyses, notamment en choisissant la méthode de calcul des coupes (par KCR Analyser ou outil tiers), la méthode d'exploration (génétique ou SMT) et offre des services comme l'export du modèle vers d'autres langages ;

Path permet de préciser les chemins d'accès aux outils tiers ;

Configuration champs recherchant les configurations déclarées dans le fichier courant et propose à l'utilisateur de choisir celle qu'il souhaite utiliser ;

Build crée l'analyseur à partir de la configuration donnée par l'utilisateur ;

Clear nettoie la console ;

Highlight déclenche la coloration syntaxique du texte contenu dans l'éditeur.

11.5 Résumé

Nous avons présenté dans ce chapitre l'outil KCR Analyser, implantant les différentes méthodes d'analyse et de résolution du problème DSE. Par la suite, nous utilisons cet outil pour évaluer l'efficacité des méthodes d'analyse développées dans ce manuscrit par rapports aux méthodes existantes identifiées dans l'état de l'art.

Chapitre 12

Expérimentations

Ce chapitre compare les méthodes d'analyse et de résolution des problèmes DSE présentées dans ce document aux approches existantes présentées dans le chapitre 3. Pour cela, nous présentons, dans la section 12.1, les cas d'études considérés dans les expérimentations et comparons, dans la section 12.2, le temps d'analyse de KCR Analyser par rapport aux outils XSAP, XFTA, Grif et HipHOPS et le temps de résolution du problème DSE de KCR Analyser à celui de HipHOPS.

12.1 Cas d'étude

Présentons les systèmes et les patrons de sûreté utilisés pour comparer les outils de calcul des indicateurs de sûreté et de résolution du problème DSE.

12.1.1 Systèmes

Les systèmes considérés sont des modèles dysfonctionnels relativement simples mais décrivant le comportement de systèmes concrets. Ces modèles (hormis Grid) se différencient des systèmes classiquement étudiés (série-parallèle) sur les points suivants :

- plusieurs modes de défaillance ;
- une logique de génération des modes de défaillance reposant sur des opérateurs non-monotones (if-then-else).

Rosace est le contrôleur longitudinal de vol présenté dans la section 2.1.1 où l'événement redouté est « la production d'une commande moteur (δ_{ec}) ou ailerons (δ_{thc}) incorrecte ».

Fuel est un système de gestion du carburant [1] d'un véhicule sans redondance initiale des composants de la chaîne de traitement. L'événement redouté est « le carburant n'est pas distribué au moteur ».

1. disponible sur `hip-hops.eu`

HBS est un système de freinage [1] d'un véhicule contenant deux lignes de freinage redondantes et dissymétriques. L'événement redouté pour HBS est « aucune des chaînes de freinage ne déclenche le serrage des freins » en considérant, dans la modélisation, que la pédale de frein est enfoncée par le conducteur.

Quadcopter est un système de pilotage semi-automatique d'un drone contenant un mode de commande nominal (entièrement automatisé) et dégradé (demandant l'intervention d'un pilote au sol). L'événement redouté considéré pour QUADCOPTER est « la commande fournie par le contrôleur est erronée ».

Grid est une grille de 4×60 composants à état binaire (marche/panne) où :
- la première colonne est constituée de composants sources S ne possédant pas d'entrées et où la sortie est défaillante si l'événement $S.f$ survient ;
- les autres colonnes sont constituées de transmetteurs T possédant trois entrées (respectivement deux pour ceux positionnés sur la première et la dernière ligne) et où la sortie est défaillante si l'événement $T.f$ se produit ou si les trois entrées (respectivement les deux entrées) sont défaillantes ;
- la sortie des transmetteurs est envoyée à leur voisin est, nord-est et sud-est ;
- l'événement redouté pour ce système est « toutes les sorties des composants de la dernière colonne sont défaillantes ».

La modélisation KCR complète de ces systèmes est fournie dans l'annexe A.

12.1.2 Patrons de sûreté

Présentons maintenant les patrons extraits de [6] que nous utilisons comme alternatives pour les composants des cas d'étude.

COM-MON illustré par la figure 12.1 est une patron constitué d'un calculateur (COM) et d'un moniteur (MON) calculant une même sortie pour une même entrée. MON reçoit la valeur calculée par COM et la compare à celle qu'il a produit. Si les deux valeurs différent alors MON envoi un signal d'arrêt à COM qui cesse de transmettre sa sortie au reste du système. Si nous considérons que les flots de ce patron peuvent être *perdus* ou *erronés* alors :
- la sortie de COM est transmise si COM et MON produisent une sortie correcte ou s'ils produisent une sortie erronée (il peuvent produite la même valeur erronée) ;
- dans les autres cas, nous considérons que le signal du COM est soit perdu car COM est perdu ou bien coupé car MON détecte une incohérence.

La modélisation KCR de ce patron est donnée par le listing 12.1 .

Réplication illustré par la figure 12.2 est un patron très répandu consistant à répliquer un composant en n entités indépendantes puis à procéder à un vote sur les sorties de ces composants. Dans notre cas, nous considérerons un voteur m parmi n qui renvoi l'union des modes de défaillance observés sur au moins m instances parmi les n. Le listing 12.2 donne la modélisation KCR de ce patron pour les modes de défaillance erroné et perte.

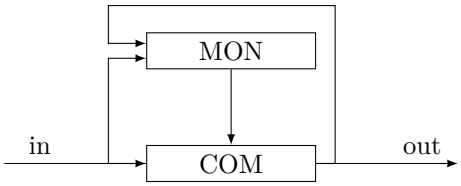


FIGURE 12.1 – Patron COM-MON

```
comp COMMON(in:t) returns (out:t){
  locs: com,mon:t;
  defs:
    com:= C @COM(in);
    mon:= C @MON(in);
    out:= if com = mon then com
          else LOST;
}
```

Listing 12.1 – Code KCR du patron COMMON

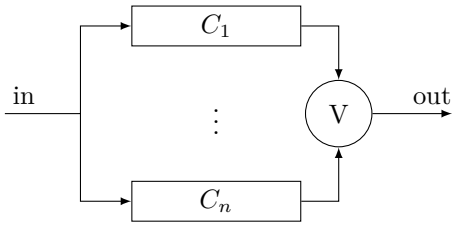


FIGURE 12.2 – Patron de réplication

```
comp RepMooN(in:t) returns (out:t){
  locs: c1,...,cn:t;
  defs:
    c1:= C @C1(in);
    ...
    cn:= C @CN(in);
    out:= VoteurMooN@V(c1,...,cn);
}
```

Listing 12.2 – Code KCR du patron de réplication

Récupération illustré par la figure 12.3, consiste à répliquer un composant en n instances indépendantes puis à vérifier tour à tour les instances à l'aide d'un test d'acceptation. La sortie de la première instance satisfaisant le test est alors renvoyée au reste du système par un sélecteur. Le test devant être simple, l'auteur de [6] considère qu'il peut produire des faux positifs et négatifs. Le listing 12.3 donne le modèle KCR de ce patron.

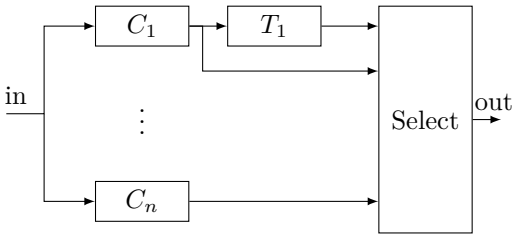


FIGURE 12.3 – Patron de récupération

```
type type2:={OK,KO};
comp RecoveryN(in:t) returns (out:t){
  locs: c1,...,cn:t;
        test1,...,testn-1:type2;
  defs:
    c1:= C @C1(in);
    t1:= Test@ T1(c1);
    ...
    cn:= C @CN(in);
    out:= if t1=OK then c1
          else if t2=OK c2
          ...
          else cn;
}
```

Listing 12.3 – Code KCR du patron récupération

Système	k	$\lvert MCS\rvert$	Outil			
			HipHOPS	XSAP	XFTA	KCR Analyser
Rosace	1	4	0.383	> 100	0.045	**0.035**
	2	18	0.426	> 100	**0.048**	0.119
	3	48	0.489	> 100	**0.047**	0.287
	7	15702	9.408	> 100	**0.298**	> 100
Fuel	1	9	> 100	> 100	**0.047**	0.062
	2	18	> 100	> 100	**0.063**	0.141
	3	36	> 100	> 100	**0.302**	0.314
	7	864	> 100	> 100	> 100	**13.404**
HBS	1	2	21.552	> 100	0.073	**0.044**
	2	48	23.236	> 100	**0.064**	0.322
	3	84	34.494	> 100	**0.154**	0.735
	7	3817	> 100	> 100	84.797	**59.786**
Quadcopter	1	1	17.511	> 100	0.039	**0.026**
	2	6	22.372	> 100	**0.038**	0.044
	3	15	37.088	> 100	**0.039**	0.076
	7	896	> 100	> 100	**0.038**	2.892

Table 12.1 – Temps de calcul (en s) des coupes minimales

12.2 Expérimentation : calcul des indicateurs

Nous comparons dans cette section le temps d'exécution des méthodes d'analyse du système implantées dans KCR Analyser par rapports aux outils XSAP, XFTA et HipHOPS. L'ensemble des expérimentations ont été menées sur un processeurs Intel XeonE5-2609v2.0@2.50GHz (4 cœurs) avec 64GB de RAM DDR3 et les temps de calcul sont donnés en secondes.

12.2.1 Coupes minimales

La première expérimentation consiste à calculer les coupes minimales des systèmes présentés précédemment où le patron de récupération est appliqué sur chaque composant du système. Ceci nous permet d'obtenir un système suffisamment complexe pour que le calcul des coupes ne soit pas trivial. Le temps de calcul des coupes avec KCR Analyser et les outils XSAP, XFTA et HipHOPS est présenté par la table 12.1 (où les temps de calcul les plus courts sont indiqués en gras) pour différentes bornes sur la taille des coupes minimales.

Comme attendu, les expérimentations de la table 12.1 démontrent que l'énumération des coupes d'un modèle SMV des systèmes précédents est plus lente que les approches statiques. En effet, l'énumération par exploration de l'espace d'état d'un système qui, en réalité est statique, amène à considérer l'ordre d'occurrence des événements alors que celui-ci n'a pas d'importance.

En ce qui concerne HipHOPS, les expérimentations mettent en avant les difficultés des méthodes d'analyse des arbres de défaillance à calculer les coupes minimales de systèmes non-monotone de taille réaliste. En effet, le temps de calcul de l'approche Sat implantée dans KCR Analyser est

		Outil				
k	$\|MCS\|$	Grif	HipHOPS	XSAP	XFTA	KCR Analyser
3	0	0.30	> 100	> 100	**0.063**	0.413
4	60	1.154	> 100	> 100	**0.097**	1.431
5	296	22.714	> 100	> 100	**0.143**	10.206
6	760	> 100	> 100	> 100	**0.293**	56.459

Table 12.2 – Temps de calcul (en s) des coupes minimales pour Grid

	k				
nombre threads	3	4	5	6	7
1 thread	0.413	1.431	10.206	56.459	**297.928**
4 threads	0.200	0.941	4.582	46.076	89.928
8 threads	0.205	0.740	3.398	32.026	68.845
36 threads	0.309	0.902	2.836	10.816	**20.395**

Table 12.3 – Effet de la parallélisation sur le temps de calcul (en s) des coupes de Grid

souvent de plusieurs ordres de grandeurs plus rapide que l'approche d'analyse MIC-SUP implantée dans HipHOPS. La différence peut notamment être dû à l'application de la loi de consensus qui semble être la phase la plus chronophage du processus de génération de HipHOPS.

Finalement, les expérimentations montrent que XFTA est l'outil le plus performant sur la plupart des systèmes. Nous pouvons expliquer cette avantage pour plusieurs raisons :

- XFTA utilise des optimisations comme la modularisation [38] qui permet de séparer le problème d'énumération en sous-problèmes plus simples et pouvant être résolus indépendamment ;
- l'algorithme de résolution *branch and deduce* de XFTA est proche des approches de résolution des solveurs Sat mais est néanmoins adapté exclusivement à l'énumération des coupes minimales et ne possède pas de méthode d'apprentissage de clauses ;

Néanmoins, la méthode d'énumération de KCR Analyser démontre son intérêt notamment sur les cas d'études HBS et Fuel. De plus les temps de calcul des coupes par KCR Analyser reste en général dans le même ordre de grandeur que celui de XFTA.

Afin de comparer Grif aux autres outils, nous avons mené le même calcul des coupes pour le système Grid modélisé en diagramme de fiabilité. Les résultats donnés par la table 12.2 confirment les résultats précédents et démontrent la limitation des méthodes de calcul par BDD, dû au coût de construction du BDD de la fonction de structure.

Finalement la table 12.3 illustre l'impact de la parallélisation [2] de la résolution des problèmes Sat sur le temps de calcul des coupes du cas d'étude Grid. Grâce à une parallélisation massive du calcul, le temps de calcul pour la borne $k = 7$ est réduite d'un ordre de grandeur par rapport à une résolution séquentielle. La parallélisation peut fournir une piste d'amélioration rapide de la méthode d'énumération implantée dans KCR Analyser.

2. exécuté sur Intel XeonE5-2699-v3@2.3GHz (36 cœurs)

Analyse	Système				
	Rosace	HBS	Quadcopter	Fuel	Grid
Calcul ordre	0.087(1)	0.021(1)	0.013(1)	0.066(1)	0.176(4)
Vérification ordre	0.023(faux)	0.115(faux)	0.044(faux)	0.025(faux)	0.261(vrai)
Monotonie	0.076(faux)	0.089(faux)	0.044(faux)	0.059(faux)	0.289(vrai)

Table 12.4 – Temps d'exécution (en s) des fonctions de vérification de l'ordre et de la monotonie

12.2.2 Ordre et monotonie

Évaluons maintenant le temps d'exécution des méthodes de calcul et de vérification de l'ordre d'un système et de vérification de la monotonie implantées dans KCR Analyser. Comme ces fonctionnalités ne sont pas implantées dans les outils existants nous ne pouvons pas comparer le temps d'analyse de KCR Analyser avec ces outils.

La table 12.4 fournit le temps d'exécution des fonctions de vérification de l'ordre et de la monotonie (le résultat est noté entre parenthèses). Concernant la vérification et le calcul de l'ordre, nous remarquons que le temps d'analyse est inférieur ou assez proche de celui nécessaire pour calculer les coupes d'ordre $k = \texttt{mincard}$. Le gain est surtout observable sur le cas d'étude Grid où la vérification et le calcul de l'ordre est 10 fois plus rapide que le temps nécessaire au calcul des coupes d'ordre $k = \texttt{mincard}$ avec l'énumération des coupes de KCR Analyser.

Par ailleurs, la vérification de la monotonie nous permet de savoir si le système est monotone ou non sans avoir à calculer les impliquants premiers de la fonction de structure. Les résultats nous indiquent que les systèmes que nous considérons sont non-monotones (hormis Grid). Par conséquent, les impliquants premiers de la fonction de structure sont différents de ses coupes minimales. Les indicateurs quantitatifs obtenus à partir des coupes minimales sont donc des approximations *pessimistes* (sous-approximation pour la fiabilité).

12.2.3 Fiabilité et taux de défaillance

Analysons maintenant le temps de calcul de la fiabilité et du taux de défaillance des cas d'étude. Comme les outils existants basent leur calcul des indicateurs sur les coupes minimales et que nos systèmes ne sont pas monotones, la fiabilité calculée par les autres outils est une approximation. Puisque le calcul mené par KCR Analyser est exacte, il ne serait pas pertinent de comparer les temps de calcul de la fiabilité.

La table 12.5 fournit les temps de calcul de la fiabilité pour différentes représentations des valeurs numériques (rationnelle ou double précision) et le temps de calcul du taux de défaillance au temps d'exposition donné (où le temps de génération du script est donné entre parenthèses).

Nous remarquons que pour les systèmes HBS, Quadcopter et Fuel, le calcul de la fiabilité est bien plus rapide que le temps de calcul des coupes minimales. En effet, le BDD encode la fonction de structure de manière compacte et facilite le calcul de la fiabilité. Néanmoins pour les systèmes Rosace et Grid, le calcul ne termine pas dans le temps imparti. L'explosion de la taille du BDD de la fonction de structure est la principale explication de l'explosion du temps de calcul.

Concernant le calcul du taux de défaillance, nous observons que le temps de génération du script reste raisonnable lorsque la fiabilité peut être évaluée (rappelons que le taux de défaillance est défini en fonction de la fiabilité). Néanmoins le temps de calcul par l'exécution du script à l'aide

Analyse	format calcul	Système				
		ROSACE	HBS	QUADCOPTER	FUEL	GRID
Calcul fiabilité	rationnel	> 600	6.63	0.239	2.917	> 600
	double précision	> 600	0.526	0.218	0.304	> 600
Calcul taux de défaillance	symbolique	> 600	> 600(0.530)	> 600(0.257)	> 600(0.232)	> 600

TABLE 12.5 – Temps d'exécution (en s) des analyses quantitatives

de la librairie SYMPY dépasse le temps imparti (600 secondes) sur l'ensemble des cas d'étude. Ces expérimentations illustrent la difficulté de calculer exactement le taux de défaillance.

12.3 Expérimentation : résolution du problème DSE

Comparons le temps de résolution de problèmes DSE avec l'outil KCR ANALYSER et l'outil HIPHOPS. Nous avons choisi l'outil HIPHOPS pour cette étape de comparaison car :

- à notre connaissance, HIPHOPS est le seul outil disponible et possédant une version d'évaluation gratuite permettant d'exprimer et résoudre des problèmes DSE pour un système statique quelconque ;
- HIPHOPS a été utilisé notamment dans [96, 3, 76] pour résoudre des problèmes concrets dans le domaine automobile ;
- la résolution des problèmes DSE implantée par HIPHOPS repose sur les algorithmes génétiques c'est-à-dire l'approche majoritairement employée.

12.3.1 Problèmes DSE

Les exigences de sûreté des problèmes DSE que nous considérons sont issues des exigences associés aux différents niveaux de criticité donnés par l'ARP4754 (cf section 2.1). Notons que l'exploration que nous avons présentée jusqu'ici est basée sur la fiabilité et non le taux de défaillance. Ainsi, dans les expérimentations, l'exigence concerne le taux de défaillance moyen ($\bar{\Lambda}$), or

$$\bar{\Lambda} = \frac{1}{T}\int_0^T \Lambda(t)dt \simeq \frac{\overline{R}(T)}{T}$$

Pour simplifier les calculs, nous considérons un temps d'opération T de une heure, ce qui revient à imposer une restriction sur la défiabilité du système. Les couples $(mincard, \overline{R})$ considérés sont donnés par la table 12.6.

Dans les problèmes DSE que nous considérons, les alternatives des composants sont obtenues en instanciant les patrons introduits dans la section précédente pour différentes valeurs des taux de défaillance des événements. Pour illustrer l'explosion combinatoire de l'espace des architectures des problèmes DSE, la table 12.7 indique le nombre d'instances de composant par système, les patrons utilisés pour créer les alternatives et la taille de l'espace des architectures résultant. Notons que le patron *initial* indique que les alternatives contiennent le composant initial pour différentes valeurs des taux de défaillance de ses événements.

Indicateur	Criticité			
	Mineure	Majeure	Dangereuse	Catastrophique
défiablité ($\overline{R}$)	10^{-3}	10^{-5}	10^{-7}	10^{-9}
ordre (*mincard*)	1	1	1	2

TABLE 12.6 – Exigences de sûreté considérées pour les problèmes DSE

Système	Nombre instances	Patrons utilisés						Nombre candidats
		Initial	Rep1oo2	Rep2oo3	Rep1oo3	COMMON	Recovery3	
ROSACE	7	✓	✓	✓	✓	✓	✓	4.10^8
FUEL	11	✓	✓	✓	✗	✗	✗	$1.7.10^{13}$
QUADCOPTER	7	✓	✓	✓	✓	✗	✓	$1.3.10^6$
HBS	10	✓	✗	✗	✗	✗	✗	$6.5.10^4$

TABLE 12.7 – Espace des architectures des problèmes DSE

12.3.2 Temps de résolution

Comparons le temps de résolution des problèmes DSE par les approches génétiques (HIPHOPS) et SMT (KCR ANALYSER). Pour cela, la table 12.8 indique le temps de résolution des problèmes DSE en donnant le système et les exigences utilisées pour définir le problème. Notons que HIPHOPS ne considère pas les exigences d'ordre, par conséquent la table 12.8 donne le temps d'exécution de KCR ANALYSER sans contrainte d'ordre. De plus, dans les cas UNSAT, l'algorithme génétique ne peut pas trouver de solution et énumère sans fin les candidats, nous fixons donc un nombre maximum de générations de 10.

Sur l'ensemble des problèmes DSE, la table 12.8 montre que le temps de résolution par SMT est bien inférieur (parfois de plusieurs ordres de grandeurs) à celui de HIPHOPS. Par ailleurs, le solveur parvient à prouver qu'il n'existe pas de solutions pour certains problèmes (cas UNSAT), ce

Outil	$\overline{R}$	Système			
		ROSACE	FUEL	QUADCOPTER	HBS
KCR ANALYSER	10^{-3}	7.686	0.602	0.557	0.617
	10^{-5}	3.512	0.145	0.349	3.373
	10^{-7}	4.442	0.366	0.565	0.738(UNSAT)
	10^{-9}	2.559	0.041(UNSAT)	0.510	0.744(UNSAT)
HIPHOPS	10^{-3}	> 600	37.431	8.230	> 600
	10^{-5}	> 600	40.142	7.997	> 600
	10^{-7}	> 600	54.024	7.769	> 600
	10^{-9}	> 600	25.276	7.729	> 600

TABLE 12.8 – Temps de résolution (en s) du problème DSE sans contrainte d'ordre

		Système			
Ordre	$\overline{R}$	Rosace	Fuel	Quadcopter	HBS
1	10^{-3}	3.838	0.552	0.326	0.599
1	10^{-5}	3.222	0.208	0.450	3.309
1	10^{-7}	4.504	0.416	0.582	0.797(Unsat)
2	10^{-9}	2.481	0.036(Unsat)	0.486	0.745(Unsat)

Table 12.9 – Temps de résolution (en s) du problème DSE avec contrainte d'ordre

que les algorithmes génétiques ne peuvent pas prouver.

Afin d'évaluer l'impact d'une résolution simultanée des deux exigences, nous donnons dans la table 12.9, le temps de résolution par KCR Analyser lorsque la contrainte de fiabilité et d'ordre sont prises en compte.

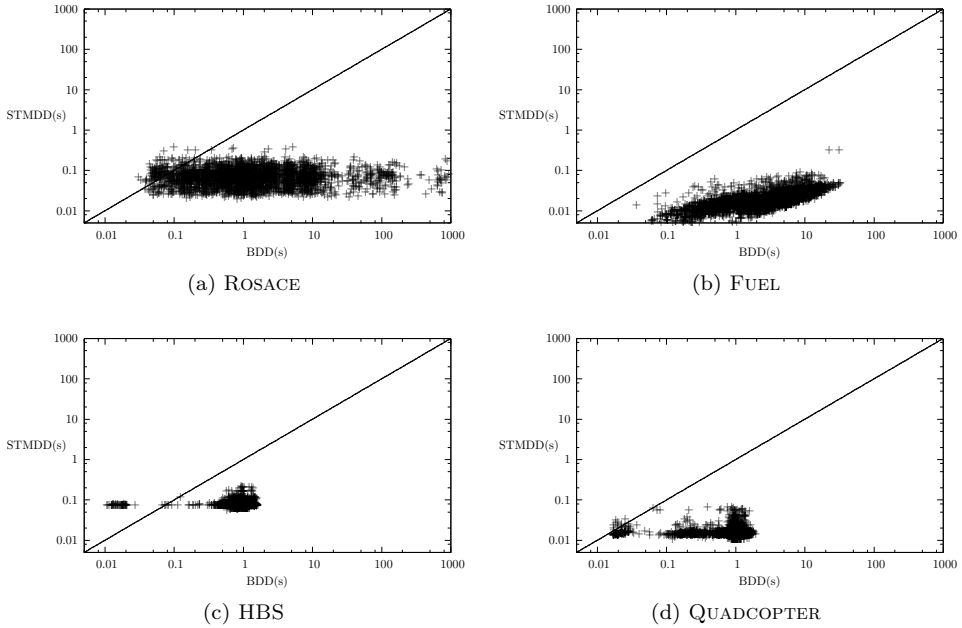
Comme le montre la table 12.9, ajouter une contrainte d'ordre permet à l'outil de réduire sensiblement le temps d'exploration. En effet, la théorie Safety peut utiliser les clauses de conflit générées par l'analyse de l'ordre afin de réduire plus efficacement l'espace de recherche.

Expliquons en quoi les mécanismes de génération des clauses de conflit de la théorie Safety et le calcul des indicateurs basé STMDD expliquent les écarts de temps de résolution observés durant les expérimentations.

12.3.3 Impact de la méthode d'analyse des candidats

La particularité de l'analyse de sûreté basée sur le STMDD est de calculer les indicateurs de tout candidat appartenant à l'espace des architectures d'un problème DSE en traversant le STMDD du système initial (comme montré dans la section 9.5). Par conséquent, le temps d'analyse des candidats est linéaire par rapport à la taille du STMDD et ne dépend pas des alternatives choisies. HipHOPS quant à lui, se base sur des techniques classiques d'évaluation de la sûreté basées sur l'énumération des coupes minimales. Bien entendu, le temps de calcul des coupes dépends de la complexité de la fonction de structure du système et donc des alternatives choisies.

La figure 12.4 illustre l'intérêt du STMDD en représentant le temps de calcul de la fiabilité (sur un Intel Xeon E5-2699 @2.30GHz (36 cœurs) 500GB RAM) par la méthode du STMDD et BDD pour les systèmes Rosace, Fuel, HBS et Quadcopter pour 5000 candidats choisis aléatoirement (pour indication la droite x=y est donné sur la figure). Afin d'obtenir des temps de calcul non négligeables, ces candidats font partie du sous-ensemble de l'espace des architectures dont l'analyse est à priori *difficile* c'est-à-dire en réduisant, si possible, les choix de substitutions aux patrons de triplication et de récupération. Celle-ci montre bien que le temps d'analyse par la méthode STMDD ne dépend pas du candidat, contrairement aux méthodes classiques (ici analyse BDD). Par ailleurs, le temps d'analyse d'un système par STMDD est généralement bien inférieur à celui de la construction du BDD puis de son analyse. Le facteur d'accélération moyen du temps de calcul par rapport à la méthode BDD est de : 254 pour Rosace, 208 pour Fuel, 60 pour Quadcopter et 12 pour HBS. Notons que HBS apporte une accélération relativement faible par rapport aux autres systèmes. En effet, puisque nous ne considérons, dans ce problème, que des alternatives où seuls les taux de défaillance sont changés (cf table 12.7), la fonction de structure du système (et donc son BDD) est toujours la même. Par conséquent, le temps d'analyse par BDD ne



(a) Rosace (b) Fuel

(c) HBS (d) Quadcopter

Figure 12.4 – Temps de calcul de la fiabilité par méthode BDD et STMDD

dépend plus des substitutions choisies, limitant ainsi l'intérêt d'utiliser un STMDD. En résumé, l'utilisation du STMDD permet de décider rapidement si un candidat respecte les exigences et explique en partie l'efficacité de l'approche d'exploration implantée par KCR Analyser.

12.3.4 Impact des clauses de conflit

Après analyse d'un candidat ne respectant pas les exigences, la théorie Safety génère une clause de conflit réduisant l'espace de recherche. Afin d'illustrer l'importance de la minimisation de ses clauses, la table 12.10 donne le temps de résolution des problèmes d'exploration sans utiliser la minimisation de clause c'est-à-dire en bloquant le modèle analysé.

Retirer la minimisation augmente mécaniquement le temps de résolution de plusieurs ordre de grandeurs. En effet, puisque la théorie est utilisée de manière paresseuse, le solveur SMT calcule un modèle des prédicats de la théorie Safety avant de vérifier si la conjonction de littéraux construite à partir de ce modèle est satisfiable modulo Safety. Ceci revient à évaluer la conformité d'un unique candidat de l'espace des architecture vis-à-vis des exigences de sûreté. Sans minimisation de la clause de conflit, seul ce candidat est rejeté, la résolution devient alors une énumération exhaustive donc inefficace.

		Système			
Minimisation	$\overline{R}$	Rosace	Fuel	Quadcopter	HBS
Inactif	10^{-3}	8.371	1.971	> 100	> 100
	10^{-5}	13.534	2.094	> 100	> 100
	10^{-7}	> 100	> 100	> 100	> 100
	10^{-9}	> 100	> 100	> 100	> 100
Actif	10^{-3}	7.686	0.602	0.557	0.617
	10^{-5}	3.512	0.145	0.349	3.373
	10^{-7}	4.442	0.366	0.565	0.738(Unsat)
	10^{-9}	2.559	0.041(Unsat)	0.510	0.744(Unsat)

Table 12.10 – Temps de résolution (en s) du problème DSE avec et sans minimisation des clauses de conflit

12.4 Résumé

Nous avons utilisé l'outil KCR Analyser pour évaluer et comparer le temps de calcul des indicateurs et de résolution de problèmes DSE par rapport aux approches et outils introduits dans l'état de l'art. Nous avons pu en conclure que l'approche SMT, tant sur le plan du calcul des indicateurs que sur celui de la résolution de problèmes DSE, permet souvent de réduire le temps de calcul.

Cinquième partie

Conclusion

Chapitre 13

Conclusion

Ces travaux ont adressé la synthèse d'architectures de systèmes critiques devant respecter des exigences de sûreté de fonctionnement. Nous avons formalisé cette synthèse comme un problème d'exploration de l'espace des architectures (problème DSE). Nous avons alors proposé un processus de modélisation, d'analyse et de résolution de ce problème :

1. Assurant formellement le respect des exigences de sûreté de fonctionnement ;
2. Assurant de trouver une solution si et seulement si celle-ci existe c'est-à-dire un processus complet et correct ;
3. Applicable à tout système statique sans restriction sur sa structure ;
4. Considérant des contraintes de sûreté sur des indicateurs classiquement utilisés pour évaluer la sûreté des systèmes.

Nous récapitulons, dans ce chapitre, les principales contributions présentées dans ce manuscrit ayant permis de répondre à la problématique et discuterons de leurs limitations et des perspectives d'amélioration.

13.1 Modélisation du problème

13.1.1 Contributions

Modélisation avec KCR Afin de faciliter la formulation des problèmes DSE pour des systèmes statiques, nous avons introduit le langage de modélisation KCR publié dans [36]

- permettant de décrire le modèle dysfonctionnel du système initial, de manière modulaire et hiérarchique afin de faciliter la description et le maintien du modèle,
- offrant la capacité de formuler le problème DSE en spécifiant les exigences de sûreté de fonctionnement et les mécanismes de sûreté dans une configuration. Cette séparation du modèle et des exigences facilite la réutilisation d'un même modèle dans différents problèmes.

Sémantique de KCR La sémantique de KCR est définie formellement par une traduction vers la MSFOL, en particulier dans le langage SMT-Lib, standard reconnu par de nombreux solveurs pour formuler des problèmes SMT. Cette traduction facilite l'intégration des solveurs SMT dans le

processus d'analyse de la sûreté en fournissant un modèle dysfonctionnel décrit comme un ensemble de fonctions et prédicats de la théorie UFBV.

13.1.2 Limitations

Modélisation des patrons Le langage KCR n'intègre pas la notion de patron de sûreté. Autrement dit, l'utilisateur ne peut pas définir des bibliothèques de patrons bien formés, l'obligeant alors à instancier manuellement les patrons pour chaque type de composants. Ceci engendre une réplication fastidieuse de code KCR, difficilement gérable pour des systèmes contenant de nombreux types de composant et où de nombreux patrons sont disponibles. Par ailleurs, puisqu'il n'existe pas de description formelle du patron, il est impossible de vérifier si une instance donnée est bien une instance du patron que l'utilisateur souhaite considérer.

Densités de probabilité En KCR, la seule distribution de probabilité disponible pour les événements de défaillance est une distribution exponentielle. Cette loi est certes très utilisée dans les calculs de défaillance mais n'est pas la seule. Nous pouvons notamment citer la loi de Weidbull permettant de modéliser des distributions avec un taux de défaillance variant au cours du temps.

13.1.3 Perspectives

Modélisation des patrons Pour répondre au problème de modélisation des patrons, une première perspective consisterait à ajouter des éléments de programmation générique dans KCR. Plus précisément, pour modéliser un patron il faut être capable d'instancier n'importe quel composant un nombre arbitraire de fois et spécifier les règles de consolidation sur l'ensemble des sorties produites par les composants. Par conséquent, la modélisation doit être générique par rapport aux types des entrées et sorties des composants (généricité de type) et générique par rapport au nombre d'entrées et sorties des composants (généricité d'arité). Les figures 13.1a et 13.1b illustrent ces deux types de généricité séparément. L'intégration de cette généricité au sein du langage KCR constituerait un travail considérable car complexifie fortement le typage des expressions. Aussi nous proposons d'embarquer KCR comme un DSL intégré dans le langage Scala. Ce dernier offre les mécanismes de programmation générique notamment à l'aide des *listes hétérogènes* (notées HList) et des *fonctions polymorphes* inspirées de [55]. Une liste hétérogène représenterait la liste des entrées et sorties du composant tout en spécifiant le type de chaque flot. Comme montré dans la figure 13.1c, les composants et mécanismes de consolidation pourraient être modélisés comme des fonctions sur ces listes. Par exemple un composant prenant $in_1 : I_1, in_2 : I_2$ et retournant $o_1 : O_1, o_2 : O_2$ serait vu comme une fonction de $\mathcal{F}_{I_1::I_2::HNil \to O_1::O_2::HNil}$. Les définitions de flots seraient alors des opérations sur les listes hétérogènes (effectuées par des fonctions d'ordre supérieur comme *map* et *fold*) nous assurant la généricité de type et d'arité.

Densité de probabilité Les méthodes de calcul des indicateurs présentées dans ce manuscrit n'utilise aucune propriété particulière de la loi exponentielle. Ces méthodes peuvent donc être utilisées avec n'importe quelle distribution, la seule limitation étant de pouvoir les spécifier dans le langage KCR. Une perspective à court terme consisterait à étendre le langage avec un champ au sein de la configuration permettant de choisir la distribution parmi une bibliothèque de distributions connues et de spécifier ses paramètres.

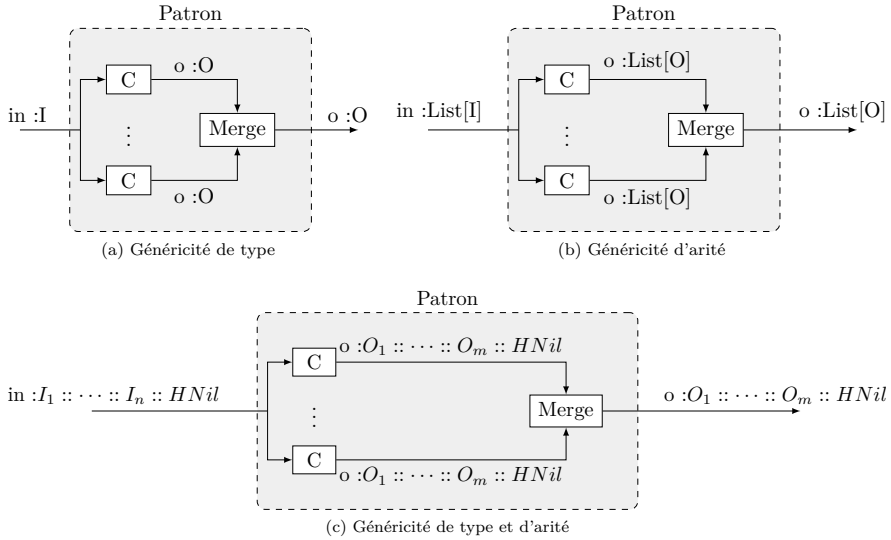


FIGURE 13.1 – Généricité pour la modélisation des patrons

13.2 Analyse des systèmes

13.2.1 Contributions

Analyses basées SMT Grâce à la traduction des modèles KCR en SMT-LIB, nous avons formalisé le calcul des principaux indicateurs qualitatifs de sûreté (coupes minimales, ordre, monotonie) comme la résolution de problèmes SMT. Le calcul des indicateurs peut alors :

- utiliser des solveurs SMT matures pour mener les analyses assurant ainsi une meilleure confiance dans les analyses que le développement d'algorithmes spécifiques pour chaque indicateur ;
- choisir le solveur le mieux adapté à la résolution de chaque problème et bénéficier des améliorations des solveurs SAT et SMT.

Ces méthodes de calcul évaluent généralement plus rapidement les indicateurs de sûreté que celles implantées dans les outils HIPHOPS, XSAP et GRIF pour un ensemble de systèmes.

STMDD Nous avons montré que les méthodes classiques de calcul des indicateurs de sûreté devaient être menées systématiquement et complètement après chaque modification de l'architecture initiale. Pour éviter cela, nous avons développé un processus d'analyse, publié dans [37], produisant une structure de donnée appelée STMDD. Nous avons alors introduit des méthodes de calcul des indicateurs de sûreté basée sur la traversée du STMDD. Ces méthodes sont capables de fournir, en un temps linéaire dans la taille du STMDD :

- la valeur exacte des indicateurs pour un candidat donné de l'espace des architectures ;
- un estimation des indicateurs (sous forme d'intervalles) pour un choix partiel des substitutions opérées sur le système initial.

Nous avons alors montré, sur un ensemble de systèmes, que le calcul par STMDD est bien plus efficace que les méthodes classiques pour évaluer un grand nombre de candidats.

13.2.2 Limitations

Performance Les méthodes d'analyse basées sur la résolution de problème SMT ne parviennent pas à rivaliser très nettement avec l'approche implantée par XFTA sur tous les cas d'étude. Nous avons notamment identifiés des faiblesses de la méthode d'énumération des coupes minimales dont le calcul pourrait profiter d'optimisation classiques comme la modularisation. Par ailleurs nous avons constaté que le calcul symbolique du taux de défaillance n'est pas une solution viable pour des systèmes de grande taille.

Hypothèse d'inclusion Pour construire le STMDD d'un système, nous devons assurer que les alternatives respectent l'hypothèse d'inclusion présentée dans le chapitre 9. Même si nous avons argumenté que la plupart des alternatives issues de l'application d'un patron de sûreté respectent cette hypothèse, cette restriction limite le champ d'application de notre approche.

13.2.3 Perspectives

Suppression de l'hypothèse d'inclusion La restriction de l'analyse aux alternatives acceptables permet d'assurer que l'ensemble des traces du système initial contient celui de tout candidat appartenant à l'espace des architectures. Cette propriété est indispensable pour calculer les indicateurs de sûreté des candidats à partir du STMDD du système initial. Pour supprimer cette restriction, la construction du STMDD devra prendre en compte le fait que les traces possibles d'une instance de composant du système ne sont pas celles du composant initial mais l'union des traces composant des alternatives possibles pour cette instance. L'utilisation du simulateur pour construire le STMDD serait alors obsolète puisque les traces de composant liées à une instance ne correspondent pas à un composant en particulier. Une piste de construction serait alors de voir les instances de composant du système comme des ensembles de traces composant c'est-à-dire une abstraction de l'instance. Les traces système seraient alors obtenues en assemblant les traces de composant pouvant se produire simultanément c'est-à-dire telles que les contraintes de connexions entre composants soient vérifiées. Nous obtenons alors les traces système de tous les candidats possibles et nous retombons dans le cadre d'application du STMDD développé dans ce travail.

Analyses de sûreté incrémentales Le STMDD permet de séparer le système et l'implantation de ses composants atomiques. En effet, le STMDD représente les traces système connaissant les traces composant que peuvent produire les instances de composant du système. Ces traces de composant sont donc une abstraction du composant concret et l'hypothèse d'inclusion définit un *contrat* sur les traces que peut fournir l'implantation de ce composant. Si ce contrat est respecté alors une nouvelle implantation peut être fournie au composant initial. La seule modification à opérer sur le STMDD est d'ajouter les indicateurs locaux des traces de la nouvelle implantation dans les tableaux order et prob. Le STMDD permettrait alors de mener des analyses incrémentales du système. Plus précisément, si la structure du système est stable mais que les implantations possibles pour les composants atomiques ne sont pas décidées alors il est possible de les ajouter incrémentalement et d'enrichir les tableaux order et prob contenus dans les nœuds du STMDD. Si une nouvelle implantation est ajoutée, il est possible de réévaluer les indicateurs de sûreté en parcourant simplement le STMDD. L'utilisation du STMDD semble donc fournir une piste prometteuse pour évaluer les indicateurs de sûreté de systèmes de grande taille au fil des évolutions des composants atomiques.

13.3 Formalisation et résolution du problème DSE

13.3.1 Contributions

Encodage à l'aide de la théorie Safety Pour résoudre automatiquement les problèmes DSE, nous avons développé :

- d'une part la théorie Safety formalisant les contraintes de sûreté comme des prédicats de la MSFOL ;
- d'autre part, le solveur de la théorie Safety, utilisant les analyses du STMDD pour décider si un ensemble de contraintes de sûreté est satisfiable. Nous avons notamment établi la procédure décidant si un choix complet ou non de substitutions satisfait les exigences de sûreté. Par ailleurs, nous avons détaillé les mécanismes de génération et de minimisation des clauses de conflit qui réduisent l'espace de recherche au fil de l'exploration.

Grace à la théorie Safety, nous avons fourni une traduction automatique des problèmes DSE en KCR vers un problème SMT. Puis nous avons utilisé un solveur SMT combiné au solveur de Safety pour résoudre le problème DSE. La résolution du problème garantit alors de fournir une solution respectant formellement les exigences de sûreté si et seulement si celle-ci existe (correction et complétude).

Performance Nous avons confronté notre méthode aux approches génétiques implantées dans l'outil HipHOPS sur un ensemble de problèmes DSE générés à partir de systèmes, d'exigences et d'alternatives issus de cas d'étude dans le domaine aéronautique et automobile. La confrontation démontre alors que la méthode SMT résout davantage de problèmes et plus rapidement que les approches génétiques. Pour expliquer cette différence, nous avons démontré l'impact prépondérant des mécanismes de génération des clauses de conflit et d'analyse des candidats sur le temps de résolution des problèmes DSE.

13.3.2 Limitations

Indicateurs de sûreté Les contraintes de sûreté exprimées dans les problèmes DSE portent sur la fiabilité et non sur le taux de défaillance, qui est l'indicateur utilisé dans les normes aéronautiques. Cette limitation, partagée par l'ensemble des approches existantes, est motivée par la difficulté de calculer le taux de défaillance et surtout de sa non-monotonie. En effet, pour vérifier une exigence de fiabilité, il suffit de vérifier que la fiabilité au temps d'exposition est bien supérieure à la borne car la fiabilité est décroissante. Pour le taux de défaillance il faut être capable de calculer l'instant, sur l'intervalle de temps considéré, où le taux est maximal. Ces calculs supplémentaires auraient sûrement un fort impact sur la performance de résolution du problème DSE pour les systèmes considérés dans ce manuscrit.

Coûts et préférences Dans KCR, la notion de coût ou de préférence de sélection d'une alternative n'existe pas. Les seules contraintes exprimables portent sur les indicateurs de sûreté. Or l'optimisation de l'architecture pour un ensemble de coût est une dimension importante du problème DSE. D'ailleurs la plupart des approches de résolution existantes optimisent l'architecture selon un ou plusieurs critères. La limitation de l'approche actuelle aux seules contraintes de sûreté est l'une des principales limitations de ce travail.

Intégration de Safety dans SMT Pour nous interfacer avec le solveur Z3, nous avons choisi d'utiliser la théorie Safety de manière *paresseuse* c'est-à-dire que le solveur de la théorie Safety n'est appelé que lorsque Z3 a trouvé un modèle de l'abstraction propositionnelle du problème SMT issue de l'encodage du problème DSE. Or appeler le solveur de théorie en fin de résolution peut nuire à la résolution du problème général puisque Z3 applique les règles de décision et de propagation sans savoir si ces choix ont un sens pour la théorie Safety. En effet, ce n'est qu'à la fin du processus de résolution que la théorie Safety peut signaler un conflit à Z3 qui aurait potentiellement pu être détecté plus tôt. Par ailleurs, la procédure décidant la satisfiabilité d'une conjonction de prédicat de Safety est relativement efficace (coût linéaire dans la taille du STMDD). De plus, le solveur de théorie peut être utilisée pour évaluer des modèles incomplets c'est-à-dire sans demander de connaître le choix de substitution pour chaque composant du système. Par conséquent, une intégration paresseuse est une intégration certes simple mais dégradant probablement les performances générales de la méthode de résolution.

13.3.3 Perspectives

Indicateurs de sûreté Au lieu de considérer le taux de défaillance instantané, difficile à calculer, nous pourrions étendre notre approche pour traiter des contraintes sur le taux de défaillance moyen ($\bar{\Lambda}$). En effet, puisque $\bar{\Lambda}$ peut être approximé par la défiabilité (section 12.3), il est possible d'adapter notre exploration basée sur la fiabilité pour considérer des exigences sur $\bar{\Lambda}$. En effet, soit T le temps d'exposition, assurer une exigence de la forme $\bar{\Lambda} \leq x$ revient à assurer que $R(T) \geq 1 - xT$. Cette adaptation pourrait être ajoutée via un nouveau prédicat `isSafeRate`(stt, x) qui serait traduit comme le prédicat de fiabilité `isSafeR`$(stt, (- \ 1 \ (* \ x \ T)))$.

Coûts et préférences Une des limitations principales de notre approche est l'absence de la gestion du coût de la solution. Pour pallier ce problème, nous proposons d'ajouter des contraintes sur le coût total du système exprimé comme la somme des coûts des alternatives choisies pour les instances de composant. Cette intégration nécessite d'étendre le langage KCR pour que l'utilisateur puisse fournir un vecteur de coûts à chaque alternative ainsi que des contraintes sur les coûts du système. Les contraintes de coût pourront alors être traitées comme des contraintes *dures* c'est-à-dire qui doivent être respectées ou *molles* qui peuvent être falsifiées. Dans le second cas, nous pourrons utiliser le formalisme max-SMT [68] pour résoudre le nouveau problème en maximisant le nombre de contraintes de coût.

Par ailleurs, nous proposons de spécifier des préférences qualitatives sur la sélection des alternatives c'est-à-dire une préférence de la forme « lorsque l'alternative A est choisie pour B alors l'alternative C est préférable à l'alternative D pour E ». Ces préférences peuvent capturées une tactique de substitution considérée comme pertinente par l'utilisateur. Ces exigences peuvent être décrites de manière compacte à l'aide des graphes de préférence (CP-nets en anglais) décrits dans [17]. Ceux-ci sont compilables en une formule qui peut être intégrée au problème initial. La combinaison de ces deux perspectives permettrait alors d'assurer formellement de trouver un candidat respectant un ensemble de contraintes de sûreté et de coût tout en respectant les préférences qualitatives de substitution données par l'utilisateur.

Intégration de Safety dans SMT L'intégration paresseuse du solveur de Safety prive le solveur SMT d'informations utiles pour la résolution du problème DSE. Une piste d'amélioration consisterait à intégrer le solveur de Safety au plus près du solveur SMT en implantant les méthodes

$\mathcal{T}$-*Conflit*, $\mathcal{T}$-*Propagation* et $\mathcal{T}$-*Decision* et en les appliquant en priorité.

La règle $\mathcal{T}$-*Conflit* c'est-à-dire de génération de clauses de conflit, serait appelée dés que le solveur SMT ajoute une prédicat de Safety (ou sa négation) dans sa base d'assertions. Ainsi, le solveur de Safety vérifie en permanence que la conjonction des décisions opérées par le solveur SMT est bien satisfiable modulo Safety. Cette utilisation du solveur de théorie permet de détecter au plus tôt les mauvaises décisions.

La règle $\mathcal{T}$-*Propagation* calculerait à partir du STMDD et d'un modèle courant, les choix de substitution essentiels pour que les exigences de sûreté soient respectées. Ceci évite de considérer inutilement des décisions que ne mènent pas à une solution du problème.

Finalement la règle de $\mathcal{T}$-*Decision* permet de développer des heuristiques de décision des choix de substitution basées sur l'analyse du STMDD. En effet, nous pourrions développer une heuristique de choix basée sur le calcul et l'analyse de facteurs d'importance comme celui de Birnbaum [14] calculés sur le STMDD.

Combinaisons des méthodes basées heuristiques et contraintes Dans ce travail nous nous sommes placés dans le cadre des méthodes de résolution basées contraintes pour résoudre le problème d'exploration. Or il existe des approches combinant les méthodes basées contraintes et heuristiques. Ainsi pour un problème DSE, le problème d'optimisation de l'architecture suivant un ensemble de coûts serait traité par une méthode basée heuristique et le respect des contraintes de sûreté serait traité par une méthode basée contrainte. Parmi ces approches, nous pouvons considérer la recherche locale basée contrainte (CBLS en anglais) [50]. Dans ces approches, un opérateur d'évaluation de respect des contraintes du problème indique à l'algorithme de recherche locale les voisins ne respectant pas les exigences. Or pour vérifier les contraintes de sûreté le STMDD semble être un candidat tout indiqué car ce dernier permet de vérifier le respect des exigences pour n'importe quel candidat en temps constant en fonction des substitutions choisies. La recherche locale bénéficierait alors d'un moyen efficace de vérifier la conformité d'un ensemble de voisins et utiliserait cette information pour mener au mieux l'optimisation. Bien entendu, ces avantages seront mitigés par la perte de la complétude qu'induit mécaniquement l'utilisation de la recherche locale.

Synthèse sous des contraintes multi-domaines Nous nous sommes intéressés au problème de synthèse d'architectures sous des contraintes de sûreté de fonctionnement. Néanmoins, la synthèse d'une architecture d'un système critique est généralement soumise à bien d'autres contraintes issues de domaines différents (sécurité, ordonnancement, réseau, etc). Or il est difficile d'intégrer l'ensemble de ces contraintes au sein d'un même problème car les techniques de résolution de ces contraintes sont généralement propres au domaine considéré. Pour pallier ce problème, il parait judicieux d'utiliser le formalisme SMT pour formaliser et résoudre des contraintes conjointement. En effet, des travaux comme [49] proposent des théories *métier* qui, au travers de leurs prédicats et fonctions, permettent de modéliser des contraintes métier comme des formules de la MSFOL et de traiter ces contraintes avec des techniques dédiées, implantées dans le solveur de la théorie. Cette perspective revient alors à développer un écosystème de théories métier composables autour d'un solveur SMT cœur qui, augmenté de ces théories, pourraient traiter des problèmes de synthèse contenant des contraintes multi-domaines.

Bibliographie

[1] The JavaBDD library. `javabdd.sourceforge.net/`.

[2] Hip-hops : Automated fault tree, fmea and optimisation tool user manual, july 2013.

[3] Masakazu Adachi, Yiannis Papadopoulos, Septavera Sharvia, David Parker, and Tetsuya Tohdo. An approach to optimization of fault tolerant architectures using hip-hops. *Software : Practice and Experience*, 41(11) :1303–1327, 2011.

[4] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. Software architecture optimization methods : A systematic literature review. *Software Engineering, IEEE Transactions on*, 39(5) :658–683, 2013.

[5] Suprasad V Amari and Glenn Dill. Redundancy optimization problem with warm-standby redundancy. In *Reliability and Maintainability Symposium (RAMS), 2010 Proceedings-Annual*, pages 1–6. IEEE, 2010.

[6] Ashraf Armoush. *Design patterns for safety-critical embedded systems.* PhD thesis, RWTH Aachen University, 2010.

[7] André Arnold, Gérald Point, Alain Griffault, and Antoine Rauzy. The altarica formalism for describing concurrent systems. *Fundamanta Informaticae*, 40(2-3) :109–124, 1999.

[8] Sanghamitra Bandyopadhyay, Sriparna Saha, Ujjwal Maulik, and Kalyanmoy Deb. A simulated annealing-based multiobjective optimization algorithm : Amosa. *IEEE transactions on evolutionary computation*, 12(3) :269–283, 2008.

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The smt-lib standard—version 2.5, 2010.

[10] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185 :825–885, 2009.

[11] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3) :59–6, 2010.

[12] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[13] Alain Billionnet. Redundancy allocation for series-parallel systems using integer linear programming. *Reliability, IEEE Transactions on*, 57(3) :507–516, 2008.

[14] Zygmund William Birnbaum. On the importance of different components in a multicomponent system. Technical report, DTIC Document, 1968.

[15] Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xsap safety analysis platform. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 533–539. Springer, 2016.

[16] Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, and Gianni Zampedri. Automated verification and tightening of failure propagation models. In *AAAI*, pages 907–913, 2016.

[17] Craig Boutilier, Ronen I Brafman, Carmel Domshlak, Holger H Hoos, and David Poole. Preference-based constrained optimization with cp-nets. *Computational Intelligence*, 20(2) :137–157, 2004.

[18] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. Efficient theory combination via boolean search. *Information and Computation*, 204(10) :1493–1525, 2006.

[19] Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. Symbolic fault tree analysis for reactive systems. In *International Symposium on Automated Technology for Verification and Analysis*, pages 162–176. Springer, 2007.

[20] Marco Bozzano, Adolfo Villafiorita, Ove Åkerlund, Pierre Bieber, Christian Bougnol, Eckard Böde, Matthias Bretschneider, Antonella Cavallo, C Castel, M Cifaldi, et al. Esacs : an integrated methodology for design and safety analysis of complex systems. In *Proc. ESREL*, pages 237–245, 2003.

[21] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3) :293–318, 1992.

[22] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.

[23] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. *NuSMV 2.6 User Manual*. FBK, 2010.

[24] Adrien Champion. *Collaboration de techniques formelles pour la vérification de propriétés de sûreté sur des systèmes de transition*. PhD thesis, Toulouse, ISAE, 2014.

[25] Vasek Chvatal. *Linear programming*. Macmillan, 1983.

[26] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. Ocra : A tool for checking the refinement of temporal contracts. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 702–705. IEEE, 2013.

[27] David W Coit and Abdullah Konak. Multiple weighted objectives heuristic for the redundancy allocation problem. *Reliability, IEEE Transactions on*, 55(3) :551–558, 2006.

[28] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[29] Dassault. *Cecilia OCAS framework*, 2014.

[30] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7) :394–397, 1962.

[31] Leonardo De Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–12. Springer, 2013.

[32] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3 : an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference,*

TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, pages 337–340, 2008.

[33] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i : Solving problems with box constraints. *IEEE Trans. Evolutionary Computation*, 18(4) :577–601, 2014.

[34] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm : Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2) :182–197, 2002.

[35] Kevin Delmas, Rémi Delmas, and Claire Pagetti. Automatic architecture hardening using safety patterns. In Floor Koornneef and Coen van Gulijk, editors, *Computer Safety, Reliability, and Security : 34th International Conference, SAFECOMP 2015, Delft, The Netherlands, September 23-25, 2015, Proceedings*, pages 283–296, Cham, 2015. Springer International Publishing.

[36] Kevin Delmas, Rémi Delmas, and Claire Pagetti. Smt-based architecture modelling. In *12th IEEE International Symposium on Industrial Embedded Systems, SIES*. Springer, 2017.

[37] Kevin Delmas, Rémi Delmas, and Claire Pagetti. Smt-based synthesis of fault-tolerant architectures. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 2017.

[38] Yves Dutuit and Antoine Rauzy. A linear-time algorithm to find modules of fault trees. *IEEE Transactions on Reliability*, 45(3) :422–425, 1996.

[39] JD Esary and F Proschan. Coherent structures of non-identical components. *Technometrics*, 5(2) :191–209, 1963.

[40] Alan M Frisch and Paul A Giannaros. Sat encodings of the at-most-k constraint. some old, some new, some fast, some slow. In *Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation*, 2010.

[41] David E Fyffe, William W Hines, and Nam Kee Lee. System reliability allocation and a computational algorithm. *IEEE Transactions on Reliability*, 2 :64–69, 1968.

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns : Elements of reusable software components. 1995.

[43] Marco Gario, Alessandro Cimatti, Cristian Mattarei, Stefano Tonetta, and Kristin Yvonne Rozier. Model checking at scale : automated air traffic control design space exploration. In *International Conference on Computer Aided Verification*, pages 3–22. Springer, 2016.

[44] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control : Preliminary report. *CoRR*, abs/1405.3694, 2014.

[45] Alain Griffault, Gérald Point, Fabien Kuntz, and Aymeric Vincent. Symbolic computation of minimal cuts for altarica models. 2011.

[46] Lars Grunske, Peter Lindsay, Egor Bondarev, Yiannis Papadopoulos, and David Parker. An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems. In *Architecting dependable systems IV*, pages 188–209. Springer, 2007.

[47] Walter J Gutjahr. A graph-based ant system and its convergence. *Future generation computer systems*, 16(8) :873–888, 2000.

[48] Bruce Hajek. Optimization by simulated annealing : a necessary and sufficient condition for convergence. *Lecture Notes-Monograph Series*, pages 417–427, 1986.

[49] Christine Hang, Panagiotis Manolios, and Vasilis Papavasileiou. Synthesizing cyber-physical architectural models with real-time constraints. In *CAV*, pages 441–456. Springer, 2011.

[50] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. The MIT press, 2009.

[51] Steffen Iredi, Daniel Merkle, and Martin Middendorf. Bi-criterion optimization with multi colony ant algorithms. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 359–372. Springer, 2001.

[52] Elkhatib Kamal, Abdel Aitouche, and Mireille Bayart. Fault tolerant control of WES parametric uncertainties. In *2nd International Conference on Systems and Computer Science, ICSCS 2013, Villeneuve d'Ascq, France, August 26-27, 2013*, pages 150–155, 2013.

[53] Christophe Kehren. *Motifs formels d'architectures de systèmes pour la sûreté de fonctionnement*. PhD thesis, Ecole nationale superieure de l'aeronautique et de l'espace, 2005.

[54] HO-GYUN KIM, CHANG-OK BAE, and SUNG-YOUNG PARK. Simulated annealing algorithm for redundancy optimization with multiple component choices. In *Advanced Reliability Modeling*, pages 237–244. World Scientific, 2004.

[55] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM, 2004.

[56] Vladimir Klebanov, Norbert Manthey, and Christian J. Muise. Sat-based analysis and quantification of information flow in programs. In *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, pages 177–192, 2013.

[57] Jan Kühn, Pierre Schoonbrood, André Stollenwerk, Christian Brendle, Nabil Wardeh, Marian Walter, Rolf Rossaint, Steffen Leonhardt, Stefan Kowalewski, and Rüdiger Kopp. Safety conflict analysis in medical cyber-physical systems using an smt-solver. In *Software Engineering (Workshops)*, pages 19–23, 2015.

[58] Sadan Kulturel-Konak, Bryan A Norman, David W Coit, and Alice E Smith. Exploiting tabu search memory in constrained problems. *INFORMS Journal on Computing*, 16(3) :241–254, 2004.

[59] Sadan Kulturel-Konak, Alice E Smith, and Bryan A Norman. Multi-objective tabu search using a multinomial probability mass function. *European Journal of Operational Research*, 169(3) :918–931, 2006.

[60] Yun-Chia Liang and Alice E Smith. An ant colony optimization algorithm for the redundancy allocation problem (rap). *Reliability, IEEE Transactions on*, 53(3) :417–423, 2004.

[61] David G Luenberger and Yinyu Ye. *Linear and nonlinear programming*, volume 228. Springer, 2015.

[62] Norbert Manthey and Sibylle Möhle. Better evaluations by analyzing benchmark structure.

[63] Florence Maraninchi and Yann Rémond. *Mode-automata : About modes and states for reactive systems*, pages 185–199. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[64] Daniel D McCracken and Edwin D Reilly. Backus-naur form (bnf). 2003.

[65] Safa Messaoud. *Optimal Architecture Synthesis for Aircraft Electrical Power Systems*. PhD thesis, University of California Berkeley, 2013.

[66] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2) :245–257, 1979.

[67] Greg Nelson and Derek C Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2) :356–364, 1980.

[68] Robert Nieuwenhuis and Albert Oliveras. On sat modulo theories and optimization problems. *SAT*, 4121 :156–169, 2006.

[69] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. *Abstract DPLL and Abstract DPLL Modulo Theories*, pages 36–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[70] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *ACM Sigplan Notices*, volume 40, pages 41–57. ACM, 2005.

[71] Frank Ortmeier and Wolfgang Reif. Safety optimization : A combination of fault tree analysis and optimization techniques. In *Dependable Systems and Networks, 2004 International Conference on*, pages 651–658. IEEE, 2004.

[72] Mohamed Ouzineb, Mustapha Nourelfath, and Michel Gendreau. Tabu search for the redundancy allocation problem of homogenous series–parallel multi-state systems. *Reliability Engineering & System Safety*, 93(8) :1257–1272, 2008.

[73] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1) :60–100, 1991.

[74] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The rosace case study : From simulink specification to multi/many-core execution. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 309–318. IEEE, 2014.

[75] Pradip K Pande, Michael E Spector, and Purnendu Chatterjee. Computerized fault tree analysis : Treel and micsup. Technical report, DTIC Document, 1975.

[76] Yiannis Papadopoulos and Christian Grante. Evolving car designs using model-based automated safety analysis and optimisation techniques. *Journal of Systems and Software*, 76(1) :77–89, 2005.

[77] Yiannis Papadopoulos and John A McDermid. Hierarchically performed hazard origin and propagation studies. In *Computer Safety, Reliability and Security*, pages 139–152. Springer, 1999.

[78] David Parker. *Multi-objective optimisation of safety-critical hierarchical systems*. PhD thesis, University of Hull, Kingston upon Hull, UK, 2010.

[79] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[80] Steffen Peter and Tony Givargis. Component-based synthesis of embedded systems using satisfiability modulo theories. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(4) :49, 2015.

[81] Christopher Preschern, Nermin Kajtazovic, Christian Kreiner, et al. Catalog of safety tactics in the light of the IEC 61508 safety lifecycle. In *Proceedings of VikingPLoP 2013 Conference*, page 79, 2013.

[82] Tatiana Prosvirnova. *AltaRica 3.0 : a Model-Based approach for Safety Analyses*. PhD thesis, Ecole Polytechnique, 2014.

[83] A Rauzy. Xfta : pour que cent arbres de défaillance fleurissent au printemps. *Actes du Congrès Lambda-Mu*, 18, 2012.

[84] Antoine Rauzy. Mathematical foundations of minimal cutsets. *Reliability, IEEE Transactions on*, 50(4) :389–396, 2001.

[85] Antoine Rauzy. Mode automata and their compilation into fault trees. *Rel. Eng. & Sys. Safety*, 78(1) :1–12, 2002.

[86] Antoine Rauzy. Binary decision diagrams for reliability studies. In *Handbook of Performability Engineering*, pages 381–396. Springer, 2008.

[87] Ana-Elena Rugina. *Modélisation et évaluation de la sûreté de fonctionnement-De AADL vers les réseaux de Pétri stochastiques*. PhD thesis, Institut National Polytechnique de Toulouse-INPT, 2007.

[88] SAE. Aerospace Recommended Practices 4754a - Development of Civil Aircraft and Systems, 2010.

[89] Arvind Srinivasan, Timothy Ham, Sharad Malik, and Robert K Brayton. Algorithms for discrete function manipulation. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 92–95. IEEE, 1990.

[90] Epstein Steven and Rauzy Antoine. *Open-PSA Model Exchange Format*, February 2017.

[91] SymPy Development Team. *SymPy Documentation Release 1.1rc1*, 2017.

[92] Total. *GRIF 2016 Fault Tree User Manual*, January 2016.

[93] B Trahktenbrot. Impossibility of an algorithm for the decision problem in finite classes. 1963.

[94] G Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[95] Alain Villemeur. *Reliability, availability, maintainability and safety assessment*. John Wiley & Sons, 1992.

[96] Martin Walker, Mark-Oliver Reiser, Sara Tucci-Piergiovanni, Yiannis Papadopoulos, Henrik Lönn, Chokri Mraidha, David Parker, DeJiu Chen, and David Servat. Automatic optimisation of system architectures using east-adl. *Journal of Systems and Software*, 86(10) :2467–2487, 2013.

[97] Ian Philip Wolforth, Martin Walker, and Yiannis Papadopoulos. A language for failure patterns and application in safety analysis. In *Third International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX 2008, June 26-28, 2008, Szklarska Poreba, Poland*, pages 47–54, 2008.

[98] Peng-Sheng You and Ta-Cheng Chen. An efficient heuristic for series–parallel redundant reliability problems. *Computers & operations research*, 32(8) :2117–2127, 2005.

[99] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.

Sixième partie

Annexes

Annexe A

Modélisation des cas d'études

A.1 Rosace

A.1.1 Code du système principal

```
//modes de defaillance des composants de Rosace
//ERR: errone, LOST: perte
type t={ERR,LOST}

//modele des filtres de Rosace
primary comp Filter() returns (out:t){
    evts: e,l;
        defs:
        out:= if e then ERR else if l then LOST else t!empty;
}

//modele de la loi de regulation de Va
primary comp LawVa(Va,Vz,q:t) returns (out:t){
    evts: err,lost;
    defs:
    out:= if err then ERR
            else if lost then LOST
            else if Va = ERR then ERR
            else if Va = LOST then LOST
            else if count(Vz=ERR,q=ERR) >=2 then ERR
            else if count(Vz=LOST,q=LOST) >=2 then LOST
            else t!empty;
}


//modele de la loi de regulation de Va
primary comp LawVz(h,az,Vz,q:t) returns (out:t){
    evts: err,lost;
    defs:
    out:= if err then ERR
            else if lost then LOST
            else if count(h=ERR, az=ERR, Vz=ERR, q=ERR) >=2 then ERR
            else if count(h=LOST, az=LOST, Vz=LOST, q=LOST) >=2 then LOST
            else t!empty;
```

```
}

//interconnection des composants Rosace produisant les ordres
//moteurs (delta_ec) et ailerons (delta_thc)
comp Rosace() returns (delta_ec, delta_thc: t) {
  locs: Vaf,Vzf,qf, hf, azf: t;
  defs:
    Vaf := Filter@FVa();
    hf := Filter@Fh();
    azf := Filter@Faz();
    Vzf := Filter@FVz();
    qf := Filter@Fq();

    delta_ec := LawVa@CVa(Vaf,Vzf,qf);
    delta_thc := LawVz@CVz(hf,azf,Vzf,qf);
};
```

A.2 HBS

A.2.1 Code du système principal

```
//modes de defaillance des composant de HBS
//OM: omission d'une donnee => omission de freinage,
//VAL: erreur sur la donnee => freinage intempestif
type t:={OM,VAL};

//composant modelisant la pedale de frein avec un
//systeme integre de redondance
primary comp Pedal() returns (out1,out2:t){
    evts: om1, val1, om2, val2;
   locs: l11,l12,l21,l22:t;
    defs:
    l11:= if(om1 ) then OM else t!empty;
    l12:= if(val1 ) then VAL else t!empty;
    l21:= if(om2 ) then OM else t!empty;
    l22:= if(val2 ) then VAL else t!empty;
    out1:=  l11 union l12;
    out2:= l21 union l22;
}

//Bus de communication entre la pedale et le gestionnaire de freinage
//omission si les deux entree sont omises ou si l'evenement om survient
//erreur de valeur si erreur de in1 ou si in1 omise et erreur de in2
primary comp Bus(in1,in2:t) returns (out:t){
    evts: om;
    defs:
    out:= (if(om # ((OM ? in1) &  (OM ? in2))) then OM else t!empty)
          union (if(((OM ? in1) & (VAL ? in2)) # (VAL ? in1)) then VAL else t!empty
    );
}

//Controleur de freinage prenant les ordres redondes issu du bus
//Envoi redonde de l'ordre de freinage, dans les deux cas
//omis si omission de in1 et in2 ou si om survient
//erreur en valeur si in1 omis et erreur sur in2 ou erreur sur in1 ou val
```

```
34 primary comp WheelController(in1,in2:t) returns (out1,out2:t){
35     evts: om1, om2, val1 , val2;
36     locs: l11,l12, l21,l22:t;
37     defs:
38     l11:= if(om1 # ((OM ? in1) &  (OM ? in2))) then OM
39               else t!empty;
40     l12:= if(((OM ? in1) & (VAL ? in2)) # (VAL ? in1) # val1) then VAL
41               else t!empty;
42     out1:= l11 union l12;
43     l21:= if(om2 # ((OM ? in1) &  (OM ? in2)) then OM
44               else t!empty;
45     l22:= if(((OM ? in1) & (VAL ? in2)) # (VAL ? in1) # val2) then VAL
46               else t!empty;
47     out2:= l21 union l22;
48 }
49
50 //Batterie d'alimentation des systemes de freinage
51 //possible perte d'alimentation
52 primary comp Battery() returns (out:t){
53     evts: om;
54     defs: out:= if om then OM else t!empty;
55 }
56
57 //System d'alimentation des freins
58 //omission de freinage si omission de commande (cmd) ou
59 //d'alimentation (power) ou evenement fail
60 //erreur de valeur si erreur de commande
61 primary comp PowerManager(cmd,power:t) returns (out:t){
62     evts: fail;
63     defs:
64     out:= if( (OM ? power) # (OM ? cmd) # fail) then OM
65               else if(VAL ? cmd) then VAL
66               else t!empty;
67 }
68
69 //premier frein recevant la commande in
70 //omission du freinage si om ou omission sur in
71 //freinage intempestif si erreur sur in
72 primary comp EMB(in:t) returns (out:t){
73     evts: om;
74     defs:
75     out:= if (om # OM ? in) then OM
76               else if (VAL ? in) then VAL
77               else t!empty;
78 }
79
80 //second systeme de frein, cmd est la commande recu du systeme
81 //power est l'alimentation
82 //omission du freinage si om ou omission sur OM ou omission sur cmd
83 //freinage intempestif si erreur sur cmd
84 primary comp IMW(cmd,power:t) returns (out:t){
85     evts: om;
86     defs:
87     out:= if (om # (OM ? cmd) # (OM ? power)) then OM
88               else if (VAL ? cmd) then VAL
89               else t!empty;
90 }
91
```

```
92 //Le systeme contient un pedale (P) envoyant les ordres redondes a deux bus
93 //independants (Bus1 et Bus2) qui transmettent les ordres au gestionnaire de
94 //freinage (Controller). Celui-ci produit deux ordres de freinage envoyes aux deux
95 //systemes d'alimentation (EMBPw) et (IMWPw) eux memes alimentes par deux batteries
96 // AuxBat et PowTrainBat. Finalement les gestionnaires d'alimentation envoient
97 //l'ordre de freinage aux systemes dissymetriques EMB et IMW dont les sorties
98 //donnent les modes de defaillance observes sur les deux chaines de freinage
99 comp System() returns (out1,out2:t){
100     locs:f1,f2,f3,f4,f5,f6,f7,f8,f9,f10:t;
101     defs:
102     f1,f2:= Pedal@P();
103     f3:= Bus@Bus1(f1,f2);
104     f4:= Bus@Bus2(f1,f2);
105     f5,f6:= WheelController@Controller(f3,f4);
106     f7:= Battery@AuxBat();
107     f8:= Battery@PowTrainBat();
108     f9:= PowerManager@EMBPw(f5,f7);
109     f10:= PowerManager@IMWPw(f6,f8);
110     out1:=EMB@EMB(f9);
111     out2:=IMW@IMW(f10,f8);
112 }
```

A.3 Fuel

```
1  //modes de defaillance des composants de fuel
2  //OF: omission de donnee
3  type t := {OF};
4
5  //composant generique possedant un entree et renvoyant
6  //une omission si l'entree in est omise ou si fail se produit
7  primary comp GenComp(in:t) returns (out:t){
8    evts: fail;
9    defs:
10     out:= if in=OF # fail then OF else t!empty;
11 };
12
13 //composant modelisant le reservoir d'essence
14 //possibilite de ne pas fournir de carburant
15 // c'est a dire omission de out lorsque fail se produit
16 primary comp ServiceTank() returns (out:t){
17   evts: fail;
18   defs:
19     out:= if fail then OF else t!empty;
20 };
21
22 //le carburant est du reservoir serv est chauffe par heat puis est
23 //transporte par circ dans un systeme de brassage mix. Le debit en sortie
24 // du systeme de brassage est mesure par flow, puis le melange est filtre par
25 //filter. Le carburant est alors envoye au booster (boost), sa viscosite est
26 //mesuree (visc), puis le carburant est envoye au moteur (ind).
27 comp circuit() returns (out : t){
28   locs: mixOut, servOut, heatOut, circOut, flowOut, filterOut, boosterOut, viscOut:t
       ;
29   defs :
30     servOut:= ServiceTank@serv();
```

```
    heatOut:= GenComp@heat(servOut);
    circOut:= GenComp@circ(heatOut);
    mixOut:= GenComp@mix(circOut);
    flowOut:=GenComp@flow(mixOut);
    filterOut:= GenComp@filter(flowOut);
    boosterOut:=GenComp@boost(filterOut);
    viscOut:=GenComp@visc(boosterOut);
    out:=GenComp@ind(viscOut);
};
```

A.4 Quadcopter

```
//modes de defaillance des composant de Quadcopter
//ERR: errone, LOST:perte
type t:={ERR,LOST};

//valeurs pour les alarmes et type de commande
type a:={ALARM};
type cmd:= {AUTO,MANUAL,CRASH}

//composant estimant les parametres (out) de vol du drone
//parametres errones si err survient, perdu si lost survient
primary comp EstimateFlightParameter () returns (out:t) {
  evts: err, l;
  defs:
    out:= if err then ERR
              else if l then LOST
              else t!empty;
};

//composant modelisant les ordres (out) du pilote
//ordres incoherents => out errone si err survient
//perte des ordres => out perdu si l survient
primary comp PilotOrder () returns (out:t) {
  evts: err, l;
  defs:
    out:= if err then ERR
              else if l then LOST
              else t!empty;
};

//systeme de calcul automatique des commandes a envoyer au drone (out) a partir de
    deux
//parametres de vols estime (in1,in2). Ce systeme contient nativement un patron
//COM-MON et propage une alarme (a) si un probleme est detecter. com et mon
//produise une commande erronee si comE survient ou in1 ou in2 erronees de meme
//pour la perte
primary comp ComputeCommandAuto (in1, in2:t) returns (out:t; alarm:a ) {
  evts: comE, comL, monE, monL;
  locs: com,mon: t;
  defs:
  com:= if comE then ERR
    else if comL then LOST
    else if (in1=LOST # in2=LOST) then LOST
    else if (in1=ERR # in2=ERR) then ERR
```

```
43      else t!empty;
44
45    mon := if monE then ERR
46      else if monL then LOST
47      else if (in1=LOST # in2=LOST) then LOST
48      else if (in1=ERR # in2=ERR) then ERR
49      else t!empty;
50
51    out:= if mon = com then com else LOST;
52    alarm:= if  com=mon then a!empty else ALARM;
53  };
54
55  //systeme de calcul manuel des commandes a envoyer au drone (out) a partir de deux
56  //parametres de vols estime (flightP1,flightP2) et de la commande pilote
57  //(pilotOrder). Ce systeme contient nativement un patron
58  //COM-MON et propage une alarme (a) si un probleme est detecter. com et mon
59  //produise une commande erronee si comE survient ou flightP1 ou flightP2 erronees.
60  //Par contre perte si flightP1 et flightP2 perdues ou si comL survient
61  primary comp ComputeCommandManual (flightP1,flightP2, pilotOrder:t) returns (out:t;
        alarm:a ) {
62    evts: comE, comL, monE, monL;
63    locs: com,mon:t;
64    defs:
65      com:=if comE then ERR
66        else if comL then LOST
67        else if ( (flightP1=LOST & flightP2=LOST) # pilotOrder=LOST) then LOST
68        else if ( (flightP1=ERR # flightP2=ERR) # pilotOrder=ERR) then ERR
69        else t!empty;
70      mon:= if monE then ERR
71        else if monL then LOST
72        else if ( (flightP1=LOST & flightP2=LOST) # pilotOrder=LOST) then LOST
73        else if ( (flightP1=ERR # flightP2=ERR) # pilotOrder=ERR) then ERR
74        else t!empty;
75
76      out:= if com=mon then com else LOST;
77      alarm:= if com=mon then a!empty else ALARM;
78  };
79
80  //systeme de verification des ordres redondes (in1,in2) a envoyer au drone avec
81  //reception des alarmes renvoyees par les systemes de generation des ordres
82  //(alarmAuto,alarmManual). Le systeme renvoi une alarme (outAlarm) pour signaler
83  //la defaillance d'un systeme de calcul des ordres et une alarme (outputCrashAlarm)
84  //lors d'un risque de crash du drone.
85  //=> outAlarm est active si err survient (faux
86  //positif), desactive si l survient (faux negatif) et normalement active si l'un
87  //des systemes de guidage est perdu ou si l'alarme du system automatique est
88  //observe.
89  //=> outputCrashAlarm est active si err survient (faux
90  //positif), desactive si l survient (faux negatif) et normalement active si les deux
91  // systemes de guidage sont perdus ou si l'alarme du system manuel est
92  //observe.
93  primary comp ControlFlightParam (in1,in2:t;alarmAuto, alarmManual:a) returns (
        outAlarm:a ; outputCrashAlarm:a) {
94    evts: err, l ;
95    defs:
96      outAlarm:= if err then ALARM
97        else if l then a!empty
98        else  if (alarmAuto=ALARM # (in1=LOST) # (in2= LOST)) then ALARM
```

```
99        else a!empty;
100
101     outputCrashAlarm:= if err then ALARM
102       else if l then a!empty
103       else  if alarmManual=ALARM # ((in1=LOST) & (in2= LOST)) then ALARM
104       else a!empty;
105 };
106
107 //systeme choisissant le systeme de guidage (manuel ou automatique) a utiliser
108 //(inAuto, inManual) en fonction des alarmes recues (inputAlarm, inputCrashAlarm).
109 //outMode=> Si l'alarme inputCrashAlarm est declenchee alors toutes les options de
        guidage
110 //sont defaillantes donc mode CRASH actif.  Sinon si inputAlarm est active alors le
111 //systeme automatique est perdu donc passage en manuel, sinon mode automatique.
112 //out=> Si mode auto alors renvoi des ordres calcule par le systeme auto, sinon si
113 //mode manuel envoi des ordres du systeme manuel, sinon mode CRASH donc pas d'ordre
114 //envoyes
115 primary comp ManagerFlightMode(inAuto, inManual:t; inputAlarm, inputCrashAlarm:a )
        returns (out:t; outMode:cmd){
116   evts: goManual, goCrash;
117   defs:
118     outMode:= if inputCrashAlarm=ALARM then CRASH
119       else if inputAlarm=ALARM then MANUAL
120       else AUTO;
121     out:= if (outMode= AUTO) then inAuto
122       else if (outMode=MANUAL) then inManual
123       else LOST;
124 };
125
126 //Systeme complet instanciant deux estimateurs (E1 et E2) des parametres de vol du
127 //drone et un composant modelisant les ordres du pilote (PO). Ces donnees sont
128 //utilisees pour calculer les ordres de vol en automatique (CA) et manuel (CM)
129 //ainsi que les alarmes correspondantes. Le controleur CP verifie alors ces ordre
130 //et produit les alarmes outAlarm et  outCrashAlarm. Finalement aiguillage des
131 //ordres a envoyer par MF en fonction des alarmes recues.
132 comp System () returns (out:t; outMode:cmd) {
133   locs: estimateParam1, estimateParam2, pilotOrder, outAuto, outManual :t;
134           alarmAuto, alarmManual, outAlarm, outCrashAlarm : a ;
135   defs:
136     estimateParam1:= EstimateFlightParameter@E1();
137     estimateParam2:= EstimateFlightParameter@E2();
138     pilotOrder:= PilotOrder@PO();
139
140     outAuto, alarmAuto:= ComputeCommandAuto@CA(estimateParam1,estimateParam2);
141     outManual, alarmManual:= ComputeCommandManual@CM(estimateParam1,
142                                                      estimateParam2,
143                                                      pilotOrder);
144
145     outAlarm, outCrashAlarm:= ControlFlightParam@CP(outAuto,
146                                                     outManual,
147                                                     alarmAuto,
148                                                     alarmManual);
149
150     out, outMode:= ManagerFlightMode@MF(estimateParam1,
151                                         estimateParam2,
152                                         outAlarm,
153                                         outCrashAlarm);
154 };
```

Résumé

La sûreté de fonctionnement occupe une place prépondérante dans la conception de systèmes critiques, puisqu'un dysfonctionnement peut être dangereux pour les utilisateurs ou l'environnement. Dans certains domaines, comme l'aéronautique, les concepteurs doivent également démontrer aux autorités de certification que les risques encourus sont acceptables. Des standards, comme l'ARP4754 dans le domaine aéronautique, décrivent des processus de développement intégrant la sûreté de fonctionnement et facilitant les activités de certification. La phase de définition d'une architecture est une étape importante de ces recommandations. Le concepteur doit définir une architecture contenant un ensemble de mécanismes de sûreté permettant de mitiger ou tout du moins limiter la probabilité d'occurrence des risques identifiés.

L'objectif de ce travail est de développer une méthode automatique et générique de synthèse d'architecture assurant formellement le respect d'exigences de sûreté. Cette activité de synthèse est alors formalisée comme un problème d'exploration de l'espace des architectures c'est-à-dire trouver un candidat appartenant à un espace de recherche fini, respectant les exigences de sûreté. Ainsi nous proposons dans ce manuscrit un processus de résolution complet et correct des problèmes d'exploration basé sur l'utilisation des solveurs SMT. Les contributions principales sont : 1. la formalisation de la synthèse comme un problème de Satisfiabilité Modulo Théorie (SMT) afin d'utiliser les solveurs existants pour générer automatiquement une solution assurant formellement le respect des exigences ; 2. le développement de méthodes d'analyse spécialement conçues pour évaluer efficacement la conformité d'une architecture vis-à-vis d'un ensemble d'exigences ; 3. la définition d'un langage KCR permettant de formuler les problèmes d'exploration et l'implantation des méthodes de résolution présentées dans ce travail au sein de l'outil KCR Analyser.

L'approche est évaluée sur un ensemble de cas d'étude et les résultats illustrent l'efficacité de la méthode de résolution basée sur SMT qui résous plus rapidement et avec plus de garanties les problèmes d'exploration considérés par rapports aux approches existantes.

Mots clés : méthodes formelles, sûreté de fonctionnement, synthèse d'architectures, SMT

Abstract

Safety is a major issue in the design of critical systems since any failure can be hazardous to the users or the environment of such systems. In some areas, such as aeronautics, designers must also demonstrate to the certification authorities that the risks are acceptable. Some guidelines, such as the ARP4754 in the aeronautical field, help the designer to integrate safety related assessment in the development processes in order to ease the certification activities. The architecture design is an important step in these recommendations. The designer must design an architecture containing a set of security mechanisms to mitigate or at least limit the probability of occurrence of the identified risks.

The objective of this work is to develop an automatic and generic method of architectural synthesis which formally ensures compliance with the safety requirements. This synthesis activity is then formalized as a design space exploration problem, c'est-à-dire find a candidate belonging to a finite set of architectures, fulfilling the safety requirements. Thus, we propose in this document a complete and correct resolution process of the design space exploration problem based on the use of SMT solvers.

The main contributions are : 1. the formalization of the synthesis as a problem of Satisfiability Modulo Theory (SMT) in order to use existing solvers to automatically generate a solution formally ensuring safety requirements ; 2. the development of analytic methods specially designed to efficiently assess the conformity of an architecture with respect to a set of safety requirements ; 3. the definition of a language named, KCR, allowing to formulate the design space exploration problem and the implementation of the methods of resolution presented in this work within the tool KCR Analyser.

The approach is evaluated on a set of case studies and the results show that the SMT-based resolution method can resolve more quickly and with more guarantees a bench of exploration problems compared to the existing approaches.

Keywords : formal methods, safety, architecture synthesis, SMT