Thèse de doctorat de
L'Université de Bretagne Occidentale

École doctorale numéro 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication

Spécialité : *Informatique*

"Validation par parties et non-intrusive de systèmes embarqués"

Luka Le Roux

Thèse présentée et soutenue le 30 Nomvembre 2018 à l'ENSTA Bretagne
Unité de recherche : Lab-STICC UMR CNRS6285

	Rapporteurs avant soutenance		
	M. AIT AMEUR Yamine	Professeur des universités, INPT-ENSEIIHT, Laboratoire IRIT	Toulouse
	M. MONSUEZ Bruno	Profeseur, ENSTA Paris-Tech	Paris
	Composition du jury		
Président	M. PERSON Christian	Professeur, Directeur-adjoint de l'ED MATHSTIC, IMT Atlantique	Brest
Examinateurs	M. AIT AMEUR Yamine	Professeur des universités, INPT-ENSEIIHT, Laboratoire IRIT	Toulouse
	M. MONSUEZ Bruno	Profeseur, ENSTA Paris-Tech	Paris
	M. BONIOL Frédéric	Professeur des universités, Directeur, ONERA	Palaiseau
	M. PLANTEC Alain	Professeur des universités, UBO	Brest
	M. MAMOUNE Abdeslam	Vice-président "formation tout au long de la vie" CFVU	Brest
Invités	M. DHAUSSY Philippe	Enseignant-Chercheur, HDR, ENSTA Bretagne	Brest
	M. TEODOROV Ciprian	Enseignant-Chercheur, ENSTA Bretagne	Brest
	Mme RIBAULT Virginie	Bureau REVA, UBO	Brest
	Référents Validation des Acquis et de l'Expérience		
Recherche	M. PLANTEC Alain	Professeur des universités, UBO	Brest
Extra-recherche	Mme RIBAULT Virginie	Bureau REVA, UBO	Brest

Cette page est laissée intentionnellement blanche.

Remerciements

L'**UBO** et l'**ENSTA Bretagne**. Merci pour m'avoir fait confiance dans cette démarche de Validation des Acquis et de l'expérience. Sans cette opportunité, ce travail n'aurait jamais vu le jour.

Philippe Dhaussy de l'ENSTA Bretagne à Brest. Merci à toi de m'avoir ouvert une porte et m'avoir fait confiance malgré une situation très compliquée. J'espère continuer à porter les problématiques de recherche nous tiennent tout les deux à cœur pour encore quelques années. **Ciprian Teodorov** de l'ENSTA Bretagne à Brest. Merci à toi pour m'avoir encadré au quotidien et en particulier pour m'avoir fait partager ton expérience dans l'écriture et la lecture.

Le jury. Merci d'avoir lu le présent manuscrit et d'avoir fait le déplacement pour m'écouter parler de ces travaux. Merci aussi pour vos retours, questions et perspectives. J'attendais cette soutenance avec impatience et nos interactions sont allées au delà de mes espérances.

Zoé Drey, Merci à toi d'avoir pris le temps de lire l'ensemble du manuscrit avant sa soumission. **Joël Champeau** de l'ENSTA Bretagne à Brest. Merci à toi pour avoir pris le taureau par les cornes et ainsi réglé de nombreux problèmes que tu aurais tout à fait pu choisir d'ignorer. **Annick Billon-Coat** et **Michèle Hofmann** de l'ENSTA Bretagne à Brest. Merci à vous deux, nos super secrétaires.

Thomas Genet et **Thomas Jensen** de l'IRISA à Rennes. Merci à vous pour m'avoir initié à la recherche, m'avoir transmis votre goût pour l'élégance dans l'approche et les méthodes formelles. **Les angevins de l'ESEO**. Jérôme Delatour, Frédéric Jouault, Camille Constant, Mathias Brun et Valentin Besnard. Merci à vous ! On travaille bien ensemble et j'espère bien que ça continue.

Les collègues, ma famille, mes amis ... J'ai vraiment énormément de chance de vous avoir tous autour de moi. Dans le désordre : Loïc Lagadec, Jean-Cristophe Lelann, Vincent Ribault, Vincent Leidle, Jean-Charles Roger, Annick Le Roux (a.k.a. Maman), Jean-François Samain, Pierre Lavoie, José Le Roux, Yvon Kerebel, Glen Kerebel, Fleur Cochon, Olivier Cochon, Frédéric Mouden, Thierry Lepors, Philippe Rabin, les cousins et cousines de Savoie, les *fous* de Torchamp. **Et tous ceux que j'oublie**

Table des matières

Chapitre 1

Introduction

1.1 Contexte de l'étude

L'évolution des technologies permet la production de systèmes embarqués toujours plus petits, performants et aux possibilités étendues. Cependant, ces avancées impliquent aussi une complexité grandissante et un processus de certification de plus en plus coûteux. Il s'agit d'un point particulièrement sensible pour les systèmes dont les responsabilités sont critiques pour la sûreté et donc avec un fort besoin de validation.

Parmi les méthodes formelles utilisées pour la vérification de systèmes critiques, on peut citer la preuve de théorème, l'analyse statique et le model-checking. Quelque soit l'approche retenue, il est nécessaire de fournir une spécification formelle et adaptée des comportements à vérifier en amont ; ainsi que de respecter les éventuelles hypothèses posées lors de l'analyse en aval (déploiement). D'une approche à une autre, ces contraintes varient de nature mais impactent invariablement le processus de conception et en augmentent le coût.

Il existe de nombreuses familles de systèmes embarqués. Entre autres : le domaine métier (avionique, santé, automobile, ferroviaire, ...), la nature des exigences, les spécificités de la cible envisagée pour le déploiement et les possibilités d'abstraction sont autant de variables caractérisant le sujet de l'étude. La diversité est telle qu'il est souvent nécessaire d'affiner la technique retenue pour la vérification au cas-par-cas et de proposer une solution *sur-mesure*.

Une approche pour la validation de systèmes embarqués peut être caractérisée sur ces deux vecteurs : la méthode formelle dont elle est issue et la famille de systèmes sur laquelle elle s'applique. Les travaux présentés ici se focalisent sur le model-checking appliqué aux systèmes embarqués pour lesquels l'environnement peut être clairement distingué des comportements à vérifier.

1.2 Problématiques adressées

Explosion de l'espace d'états Le model-checking [30, 14] est une technique basée sur l'énumération de l'espace d'états, c'est à dire l'exploration exhaustive de l'ensemble des configurations possibles du système lors de l'exécution (i.e. les états atteignables). Pour garantir la terminaison de l'algorithme, il est nécessaire de conserver en mémoire l'ensemble des états déjà explorés et ainsi de pouvoir détecter la présence de cycles. La taille de cet ensemble grandit exponentiellement avec la complexité du système à vérifier et est donc rapidement prohibitive pour le passage à l'échelle de la vérification. Il s'agit d'un problème intrinsèque au model-checking, communément appelé explosion de l'espace d'états [37].

Mise en pratique Pour contourner ce problème, de nombreuses approches sont proposées dans la littérature [4, 36, 10, 12]. Cependant, la grande diversité existante des systèmes embarqués, leurs complexités et le coût induit sur le processus de développement sont autant de problèmes déjà évoqués pour leurs mises en pratique.

Pour permettre l'utilisation de ces techniques spécialisées basées sur le model-checking, il est important d'en illustrer la mise-en-œuvre à travers des cas d'étude de dimension industrielle. L'enjeu est de mettre en avant comment certaines spécificités du système peuvent être exploitées pour le passage à l'échelle de la vérification. Il s'agit à la fois de clarifier les concepts clés de l'approche, d'identifier les prérequis à son algorithmique dans un cadre concret et de proposer une méthodologie adaptée.

Validation par parties Pour valider un système complexe, ces techniques peuvent ne pas suffire et il peut être nécessaire de diviser l'effort de vérification [20, 2]. C'est à dire d'extraire de la spécification en entrée plusieurs problèmes *plus petits* de sorte que en vérifier l'ensemble est équivalent à certifier le modèle d'origine. Ces sous-problèmes sont appelés unités de preuve par la suite.

Il s'agit, dans un premier temps, d'identifier les sous-parties des exigences pouvant être traitées séparément. Pour chacune, le modèle est ensuite adapté pour une vérification spécialisée. Chaque couple d'un ensemble de propriétés et d'un modèle pertinent constitue une unité de preuve.

Cette division en amont de la validation vise à en réduire la complexité. Il devient aussi possible d'appliquer, pour chaque unité de preuve, la technique la mieux adaptée.

Cependant, cette démarche implique plusieurs défis conséquents. Comment séparer les exigences en sous-parties pertinentes ? Comment produire et maintenir, à partir du système en entrée, les modèles spécialisés pour chacun de ces sous-ensembles ? Comment justifier de la complétude de l'analyse ? Comment extraire un modèle unique pour le déploiement des différentes unités de preuve ?

Capture de l'environnement Comme introduit section 1.1, les travaux présentés ici supposent qu'il est possible de distinguer l'environnement du système à vérifier. Il s'agit d'une caractéristique importante des cibles potentielles de la vérification qui peut être exploitée pour apporter des éléments de réponse aux questions évoquées fin du paragraphe précédent.

Intuitivement, les comportements à vérifier peuvent alors constituer une partie invariante des unités de preuve. Corollairement, la sous-partie des exigences considérée et l'environnement lui étant pertinent pour la vérification en constituent la partie variable. L'unicité de la modélisation du système ouvert (sans son environnement) peut faciliter la production et la maintenance des unités de preuve, réduire l'effort nécessaire pour justifier de la complétude de l'analyse et être le point d'entrée au déploiement.

Pour y parvenir, il est nécessaire de capturer formellement l'environnement. Il s'agit d'un point sensible dont les impacts sur la modélisation du système ouvert et les techniques utilisées en aval doivent être maîtrisés. Il s'agit de minimiser les contraintes induites sur la modélisation des comportements et de conserver un spectre satisfaisant de techniques pour la validation des unités de preuve générées. De plus, selon l'approche retenue, il peut être possible de proposer des algorithmes spécialisés et performants.

1.3 Contributions

Le présent manuscrit propose un cadre formel pour la capture de l'environnement du système étudié et en permettre une validation par parties.

Formalisation de l'hypothèse de distinction En accord avec le contexte présenté section 1.1, l'approche proposée repose sur la capacité de distinguer les transitions appartenant au système à vérifier de celles dénotant une interaction avec l'environnement.

Cette distinction est formalisée à travers une fonction d'étiquetage des transitions. Si la transition considérée est un comportement de l'environnement, une étiquette lui est attribuée dans un alphabet d'actions observables. A l'inverse, si il s'agit d'un comportement propre au système à vérifier, la transition est étiquetée par τ dénotant l'absence d'action observable.

Cette formalisation est non-intrusive et ne contraint donc pas la modélisation du système.

Un langage dédié Cette fonction d'étiquetage est le premier pas de l'approche proposée pour la définition d'un environnement pertinent à la vérification. Le suivant en est la proposition d'un langage dédié : xGDL.

Ce formalisme, au centre des contributions du présent manuscrit, est similaire aux expressions régulières étendues pour le parallélisme et les permutations. Son alphabet d'actions atomiques correspond aux étiquettes de la fonction d'étiquetage déjà évoquée. Le couplage du système et de son environnement xGDL est obtenu par composition synchrone.

Dans ce cadre et pour la vérification par parties, une unité de preuve peut alors être définie par le système fermé (incluant l'ensemble des comportements du systèmes et de l'environnement), un guide xGDL et la sous-partie des exigences à vérifier.

Dépliage borné Pour permettre l'application en aval de techniques nécessitant une composante acyclique, un algorithme de dépliage borné du guide xGDL est proposé.

Il s'agit d'extraire, à partir d'une modélisation xGDL potentiellement cyclique, un équivalent acyclique pour une borne donnée. Le résultat de la composition du système avec un guide ainsi déplié présente des similitudes avec le model-checking borné [11]. Cette technique est donc introduite par la suite et de potentiels liens sont évoqués et discutés.

Past-Free[ze] Le Past-Free[ze] est une exploration spécialisée pour la composition du modèle à vérifier avec un guide xGDL. L'algorithme exploite l'acyclicité du guide pour identifier des états déjà atteints par l'exploration pouvant être *oubliés* et donc déchargés de la mémoire. Il permet ainsi de réduire la charge mémoire lors de l'analyse et ainsi en favoriser le passage à l'échelle.

Split Cet algorithme de décomposition de l'espace d'états exploite la structure du guide xGDL pour la génération automatique d'unités de preuve. L'approche nominale est paramétrée par la profondeur à laquelle le guide doit être divisé, mais d'autres approches sont évoquées et illustrées à travers les cas d'études.

Application de l'approche Pour valider expérimentalement ces contributions dans un cadre pertinent, trois cas d'études ont été retenus. Un train d'atterrissage (*LGS* [8]), issu du domaine de l'avionique, a été originalement soumis comme point d'entrée commun de la quatrième conférence internationale ABZ (ABZ'14 [3]). Il s'agit aussi de l'exemple fil rouge utilisé pour différentes illustrations tout le long du manuscrit. Un *Pacemaker* [1], issu du domaine de la santé, est l'exemple au centre d'un ouvrage dédié à la vérification de modèles [7]. Un régulateur de vitesse (*CCS*), issue du domaine de l'automobile, est un cas d'étude de complexité conséquente et représentatif d'une famille répandu de systèmes embarqués.

1.4 Plan du mémoire

Cette section présente le plan du présent document accompagné d'une description succincte de chacun des chapitres.

Chapitre 2 : Model-checking et le problème d'explosion Il s'agit d'une présentation des concepts nécessaires à la compréhension des chapitres suivants. Le model-checking y est introduit de manière générique via les notions de systèmes de transitions étiquetées, de composition synchrone et d'exploration. Parmi ses possibles raffinements, le model-checking borné et la vérification guidée par le contexte sont décrits. Enfin, le cas d'étude en fil rouge (le train d'atterrissage) est présenté.

Chapitre 3 : Vérification dirigée par le contexte Ce chapitre regroupe les principales contributions de ce manuscrit. Il s'agit d'une généralisation des travaux autour de la vérification guidée par le contexte à travers une approche revue, un nouveau formalisme pour la capture de l'environnement, différents algorithmes dédiés et une discussion sur les liens avec le model-checking borné (BMC [11]).

Chapitre 4 : Validation de l'approche Les différents cas d'étude retenus (LGS, $Pacemaker$ et CCS) font l'objet d'une étude à travers la mise en pratique de l'approche proposée. L'accent est mis sur comment les différentes spécificités des familles de systèmes embarqués dont ils sont représentatifs peuvent être exploitées pour le passage à l'échelle.

Chapitre 5 : Conclusion et perspectives Ce chapitre présente une synthèse des contributions proposées dans ce manuscrit. Elles sont alors organisées en trois niveaux : les aspects de formalisation, l'algorithmique et les résultats obtenus par application sur les différents cas d'études. Enfin, deux perspectives de recherche sont évoquées.

Chapitre 2

Model-checking et le problème d'explosion

2.1 Introduction

Les contributions du présent document s'inscrivent dans le cadre du model-checking et, plus particulièrement, de la vérification guidée par le contexte.

La section 2.2 introduit le model-checking : son cadre d'application, le problème du passage à l'échelle (explosion de l'espace d'état) et quelques axes et techniques l'adressant.

La section 2.3 pose quelques prérequis généraux nécessaires à la compréhension de la suite de ce document : les systèmes de transitions, leur composition et l'algorithmie d'exploration (énumération de l'espace d'état).

Les sections 2.4 et 2.5 présentent plus avant le model-checking borné et la vérification guidée par les contexte, deux techniques basées sur le model-checking.

La section 2.6 présente le cas d'étude du train d'atterrissage (LGS), l'exemple fil rouge des illustrations du chapitre 3 et sujet de l'étude traitée section 4.2.

2.2 Vérification de systèmes critiques

Dans le cycle de conception d'un système critique, la vérification des exigences (certification) est une phase coûteuse. Sa complexité augmente rapidement avec celle du système et le nombre d'exigences. Les constructeurs de systèmes industriels investissent donc énormément d'efforts, souvent vers des méthodes basées sur le test et la simulation, pour assurer la qualité de ces systèmes et leurs certifications.

Les méthodes formelles offrent une alternative aux techniques traditionnelles de développement guidées par le test. Cependant leurs applications dans un cadre industriel posent un grand nombre de défis.

Le model-checking, inventée dans les années 80s [30, 14], est l'une de ces méthodes formelles. Il s'agit d'une technique de preuve de basée sur l'énumération et l'analyse exhaustive des comportements. Cette technique suppose une modélisation finie des comportements en

entrée et en permet la validation automatique vis-à-vis d'une spécification temporelle des exigences. Si une faille est détectée, l'approche exhibe un contre-exemple menant à l'anomalie, une aide précieuse au diagnostique.

De nombreux outils ont été développés mettant en œuvre cette technique [25, 5] et elle a fait ses preuves à de nombreuses reprises, tant dans des contextes académiques qu'industriels [28, 10].

Cependant, elle souffre d'un problème intrinsèque, communément appelé *l'explosion de l'espace d'états* [37]. Intuitivement, un algorithme basé sur l'énumération exhaustive des comportements implique la construction de l'espace d'états. Cet ensemble permet de déterminer si un état a déjà été analysé ou non et ainsi garantir la terminaison de l'algorithme. On parle d'explosion lorsque la taille de cet espace d'états atteignables dépasse la capacité mémoire de la machine sur laquelle l'algorithme est exécuté.

L'explosion de l'espace d'états est intrinsèque au model-checking et, dans le cas de systèmes complexes, peut être prohibitive. Néanmoins, depuis l'introduction de cette approche, de nombreux efforts de recherche se sont focalisés sur la réduction de l'impact de ce problème dans un contexte industriel et ainsi en aider le passage à l'échelle. Dans sa présentation pour le prix Turing [13], E. Clarke cite quatre techniques comme étant les progrès les plus conséquents dans ce sens :

— le model-checking symbolique ;

— la réduction d'ordre partiel ;

— le model-checking borné ;

— le *counter-exemple guided abstraction refinement* (CEGAR).

Au delà de ces techniques on peut citer également les progrès algorithmiques basés, notamment, sur l'exploitation du parallélisme et des ressources de stockage externes, ainsi que les techniques de décomposition du problème telles que la vérification compositionnelle [2] et la vérification guidée par le contexte [20].

Dans la suite nous introduiront les bases théoriques du model-checking, nous présenterons le model-checking borné, ainsi que la vérification guidée par le contexte.

L'ensemble des autres techniques sont décrites par ailleurs, et la littérature est riche en sources. Le lecteur intéressé est invité à consulter :

— [4] pour une introduction au techniques de model-checking ;

— [36, 24, 29] pour des techniques basées sur la réduction d'ordre partiel ;

— [10] pour une description détaillé des techniques symboliques ;

— [12] pour les fondations de *CEGAR*.

2.3 Prérequis

Cette section présentent des prérequis généraux à la compréhension de la suite du présent document.

La section 2.3.1 introduit quelques concepts pour raisonner sur les systèmes d'états et transitions, sur les graphes étiquetés ou non.

La section 2.3.2 défini la composition synchrone de deux systèmes de transitions.

La section 2.3.3 présente l'algorithmique d'exploration.

2.3.1 Systèmes de transitions étiquetées

Cette section introduit le concept de système de transitions étiquetées (LTS). Dans un premier temps, les systèmes de transitions (sans étiquettes, définition 1) sont présentés. Les notions de chemins, d'atteignabilité et d'espace d'états sont exposées. Cette première formulation est ensuite étendue à celle de LTS (définition 2). Sur celle-ci, le déterminisme et les notions de traces et langages sont définis.

Définition 1. ***Transition System***

Un système de transition est défini par un triplé $T = (S, S_0, \delta)$ avec :

- *S, un ensemble fini d'états ;*
- *$S_0 \subseteq S$, un ensemble fini d'états initiaux ;*
- *$\delta \subseteq S \times S$, une relation de transition.*

Dans la suite, la notation $s \to s' \in \delta$ dénote le couple $(s, s') \in \delta$ ou, en langage naturel, une transition dans T de s à s'.

Chemin Un chemin de s_0 à s_n dans un système de transition est une séquence de transitions chaînées $s_0 \to s_1 \to s_2 \to \ \ldots\ s_{n-1} \to s_n$ avec $\forall i, 0 \leq i < n, s_i \to s_{i+1} \in \delta$. Sauf si explicitement précisé autrement, le premier état d'un chemin dans un système de transition est un état initial ($s_0 \in S_0$).

Atteignabilité Un état $s' \in S$ est dit atteignable à partir de $s \in S$, noté $s \xrightarrow{*} s'$, si $s = s'$ ou si il existe un chemin de s à s' dans le système de transition.

Un état $s' \in S$ est dit atteignable dans un système de transition T si il existe $s_0 \in S_0$ tel que $s_0 \xrightarrow{*} s'$. On peut noter que, par définition, tous les états initiaux de T sont atteignables dans T.

Espace d'états Dans la suite, l'espace d'états d'un système de transition T dénote son ensemble d'états atteignables : $\{s \in S \mid \exists s_0 \in S_0, s_0 \xrightarrow{*} s\}$.

Définition 2. ***Labeled Transition System (LTS)***

Un système de transitions étiquetées repose sur la notion d'action observable. Dans la suite :

- *A dénote l'ensemble universel des actions observables ;*
- *τ dénote l'absence d'action observable.*

Un système de transitions étiquetées est défini par un quadruple $L = (S, S_0, A_L, \delta)$ avec :

- *S un ensemble fini d'états ;*
- *$S_0 \subseteq S$ l'ensemble des états initiaux ;*
- *$A_L \subseteq A$ un ensemble fini d'actions observables, l'alphabet de L ;*
- *$\delta \subseteq S \times (A_L \cup \{\tau\}) \times S$ une relation de transition.*

.

Dans la suite, la notation $s \xrightarrow{a} s'$ dénote le triplé $(s, a, s') \in \delta$ ou, en langage naturel, une transition dans L de l'état s à l'état s' étiquetée par l'action $a \in A_L \cup \{\tau\}$.

Déterminisme Un LTS est déterministe si :

- son ensemble d'états initiaux contient au plus un état ;
- toutes ses transitions sont étiquetées par des actions observables (pas de τ-transitions) ;
- $\forall s, s', s'' \in S,\ \forall a \in A_L,\ s \xrightarrow{a} s' \in \delta \wedge s \xrightarrow{a} s'' \in \delta \ \Rightarrow\ s' = s''$.

Cette définition est suffisante mais pas strictement nécessaire. Il est en effet possible de définir des automates *déterministes* avec plusieurs états initiaux et/ou des τ-transitions.

Cependant, il est toujours possible de transformer un LTS quelconque vers un équivalent déterministe tel que défini plus haut (un seul état initial, pas de τ-transitions).

Traces et langage Une **trace** est une séquence d'actions observables.

La **trace d'un chemin** $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ ...\ s_{n-1} \xrightarrow{a_{n-1}} s_n$ est la séquence d'action observables ($a_i \in A_L$) obtenue en retirant toutes les occurrences de τ dans $a_0; a_1;\ ...\ ; a_{n-1}$.

Le **langage** d'un LTS L, dénoté $langage(L)$, est l'ensemble des traces dans L débutant par un état initial.

On peut noter que cette définition est **fermée par préfixe**. C'est à dire que pour un LTS L donné, si $w \in langage(L)$, pour tout w_{pre} préfixe de w, on a $w_{pre} \in langage(L)$.

Illustrations Dans la suite de ce document différents formalismes sont utilisés pour capturer ou représenter des comportements. L'ensemble de ces modèles peuvent être ramenés à des LTS.

La figure 2.1 présente plusieurs exemples d'automates et de graphes. Tous représentent un compteur d'incrément 1 modulo 4 initié à 0.

Construire les LTS correspondant implique d'identifier un ensemble d'états, les états initiaux, les actions observables et la relation de transition.

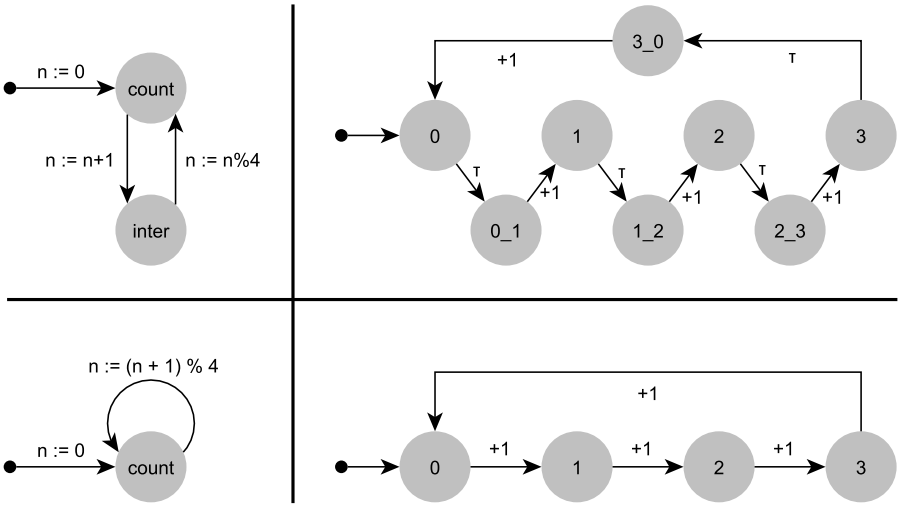

FIGURE 2.1 – LTS : Exemples d'automates et de graphes

Pour les ensembles d'états, dans le cas des automates, il est important de capturer la valuation des variables d'états. Ainsi pour les exemples à gauche de la figure 2.1, l'ensemble des états est $\{count,\ inter\} \times N$ en haut et $\{count\} \times N$ en bas. Dans les deux cas, $(count, 0)$ est le seul état initial.

Pour l'alphabet d'actions observables on défini ici un alphabet commun : $A_L = \{+1\}$. Les étiquettes des transitions, par définition, sont donc soit $+1$ (dénotant l'incrément modulo 4) soit τ (utilisé ici pour dénoter une action intermédiaire). Il est important de noter que le choix de l'alphabet impacte grandement les *observations* possibles en aval.

L'automate en haut à gauche de la figure 2.1 peut être alors capturé par le LTS suivant :

— $S = \{count,\ inter\} \times N$; $S_0 = \{(count, 0)\}$; $A_L = \{+1\}$;

— $\delta = \{(count, n) \xrightarrow{\tau} (inter, n+1), (inter, n) \xrightarrow{+1} (count, n\%4\})$.

Le graphe en haut à droite en est une version *dépliée* (variables d'états mises à plat). Il peut être capturé par le LTS suivant :

— $S = \{0, 0\_1, 1, 1\_2, 2, 2\_3, 3, 3\_0\} \times N$; $S_0 = \{0\}$; $A_L = \{+1\}$;

— $\delta = \{i \xrightarrow{\tau} i\_j,\ i\_j \xrightarrow{+1} j \mid (i, j) \in \{(0,1), (1,2), (2,3), (3,0)\}\}$.

Les exemples en bas de la figure 2.1 correspondent aux équivalents déterministes des exemples en haut.

2.3.2 Composition

Cette section décrit et illustre la composition de deux LTS (voir définition 2) telle que utilisée dans ce document. Il s'agit d'une définition synchrone *avec transitions de bégaiement.*

Définition 3. ***Synchronous composition with stuttering transitions***

La composition synchrone avec transitions de bégaiement de deux LTS L^1 $(S^1, S_0^1, A_L^1, \delta^1)$ *et* L^2 $(S^2, S_0^2, A_L^2, \delta^2)$*, notée* $L^1 \times L^2$*, est définie par le LTS* (S, S_0, A_L, δ) *où :*

- $S = S^1 \times S^2$;
- $S_0 = S_0^1 \times S_0^2$;
- $A_L = A_L^1 \cup A_L^2$;

La relation de transition de $L^1 \times L^2$*,* $\delta \subseteq S \times A_L \cup \{\tau\} \times S$*, est définie en fonction de* L^1 *et* L^2 *par les équivalences suivantes :*

$$\begin{array}{rll}
\forall a \in A_L^1 \cap A_L^2, & (s^1, s^2) \xrightarrow{a} (s'^1, s'^2) \in \delta & \Leftrightarrow \quad s^1 \xrightarrow{a} s'^1 \in \delta^1 \wedge s^2 \xrightarrow{a} s'^2 \in \delta^2 \\
\forall a \in A_L^1 \setminus A_L^2, & (s^1, s^2) \xrightarrow{a} (s'^1, s'^2) \in \delta & \Leftrightarrow \quad s^2 = s'^2 \wedge s^1 \xrightarrow{a} s'^1 \in \delta^1 \\
\forall a \in A_L^2 \setminus A_L^1, & (s^1, s^2) \xrightarrow{a} (s'^1, s'^2) \in \delta & \Leftrightarrow \quad s^1 = s'^1 \wedge s^2 \xrightarrow{a} s'^2 \in \delta^2 \\
a = \tau, & (s^1, s^2) \xrightarrow{a} (s'^1, s'^2) \in \delta & \Leftrightarrow \quad s^2 = s'^2 \wedge s^1 \xrightarrow{\tau} s'^1 \in \delta^1 \\
& & \vee \quad s^1 = s'^1 \wedge s^2 \xrightarrow{\tau} s'^2 \in \delta^1
\end{array}$$

Respectivement : les transitions observables synchrones $(a \in A_L^1 \cap A_L^2)$*, les transitions observables propres à* L^1 $(a \in A_L^1 \setminus A_L^2)$*, les transitions observables propres à* L^2 $(a \in A_L^2 \setminus A_L^1)$ *et les transitions de bégaiement* $(a = \tau)$*.*

On se propose de revenir sur la définition 3 pour en préciser les différentes notions. Similairement, on suppose deux LTS L^1 $(S^1, S_0^1, A_L^1, \delta^1)$ et L^2 $(S^2, S_0^2, A_L^2, \delta^2)$.

Leur composition, notée $\mathbf{L^1 \times L^2}$, est elle-même un LTS avec $L^1 \times L^2 = (S, S_0, A_L, \delta)$.

États composites L'ensemble d'états de la composition $L^1 \times L^2$ est le produit cartésien des ensemble d'états de ces LTS, $\mathbf{S = S^1 \times S^2}$. En d'autres mots, il s'agit de l'ensemble des couples (s^1, s^2) avec $s^1 \in S^1$ et $s^2 \in S^2$.

Similairement, l'ensemble des états initiaux de $L^1 \times L^2$ est le produit cartésiens des ensembles d'états initiaux de L^1 et L^2, $\mathbf{S_0 = S_0^1 \times S_0^2}$

Alphabet L'alphabet de $L^1 \times L^2$ est l'ensemble d'actions observables obtenu par union des alphabets de L^1 et L^2, $\mathbf{A_L = A_L^1 \cup A_L^2}$.

Étiquettes Par définition d'un LTS (voir définition 2), les transitions de $L^1 \times L^2$ sont étiquetées dans $A_L \cup \{\tau\} = A_L^1 \cup A_L^2 \cup \{\tau\}$.

Relation de transition Pour une action observable $a \in A$ donnée (et donc $a \neq \tau$), c'est l'appartenance de cette étiquette aux vocabulaires des LTS composés qui décident des participants à la synchronisation. La transition correspondante n'est possible dans la composition que si une transition étiquetée par a existe chez tous les participants ainsi identifiés.

La relation de transition de $L^1 \times L^2$ distingue quatre cas (correspondants respectivement aux équivalences en fin de la définition 3) :

— les transitions observables dans l'intersection des alphabets ($a \in A_L^1 \cap A_L^2$) ;

— les transitions observables propres à L^1 ($a \in A_L^1 \setminus A_L^2$) ;

— les transitions observables propres à L^2 ($a \in A_L^2 \setminus A_L^1$) ;

— les transitions non-observables ($a = \tau$).

Toutes les transitions étiquetées dans $A_L = A_L^1 \cup A_L^2$ (les trois premiers cas) sont dites observables et synchrones. Seules les transitions étiquetées dans $A_L^1 \cap A_L^2$ (le premier cas) forcent la synchronisation des deux LTS. Dans le cas des transitions propres à L^1 ou L^2 (deuxième et troisième cas) il s'agit de synchronisations avec un seul participant. Enfin, les transitions non-observables (étiquetées par τ, le dernier cas) sont dites de bégaiement et traitées comme des transitions asynchrones.

On a donc $(s^1, s^2) \xrightarrow{a} (s'^1, s'^2) \in \delta$ si et seulement si (disjonction) :

— $a \in A_L^1 \cap A_L^2$ et les transitions $s^1 \xrightarrow{a} s'^1$, $s^2 \xrightarrow{a} s'^2$ existent dans δ^1, δ^2 ;

— $a \in A_L^1 \setminus A_L^2$, $s^2 = s'^2$, et la transition $s^1 \xrightarrow{a} s'^1$ existe dans δ^1 ;

— $a \in A_L^2 \setminus A_L^1$, $s^1 = s'^1$, et la transition $s^2 \xrightarrow{a} s'^2$ existe dans δ^2 ;

— $a = \tau$ et soit $s^2 = s'^2 \wedge s^1 \xrightarrow{\tau} s'^1 \in \delta^1$, soit $s^1 = s'^1 \wedge s^2 \xrightarrow{\tau} s'^2 \in \delta^2$.

On peut noter que, pour une transitions dans $L^1 \times L^2$ donnée, il est toujours possible d'identifier sa nature en fonction de l'étiquette (synchronisation de L^1 et L^2, synchronisation avec L^1 pour seul participant, synchronisation avec L^2 pour seul participant, transition non-observable).

De plus, si la transition est observable (étiquetée dans A_L), il est aussi possible d'inférer les transitions correspondantes dans L^1 et L^2.

A l'inverse, si la transition n'est pas observable (étiqueté par τ), cela n'est pas toujours possible. Dans le cas où $s^1 = s'^1 \wedge s^2 = s'^2$, on a soit $s^1 \xrightarrow{\tau} s^1$, soit $s^2 \xrightarrow{\tau} s^2$.

Atteignabilité et espaces d'états Par définition de l'atteignabilité dans un LTS, un état (s^1, s^2) est atteignable dans $L^1 \times L^2$ si et seulement si il existe un état initial (s_0^1, s_0^2) de $L^1 \times L^2$ tel que $(s_0^1, s_0^2) \xrightarrow{*} (s^1, s^2)$. Cela implique $s_0^1 \xrightarrow{*} s^1$ dans L^1 et $s_0^2 \xrightarrow{*} s^2$ dans L^2.

On a donc, par définition de l'espace d'états d'un LTS, (s^1, s^2) dans l'espace d'états de $L^1 \times L^2$ implique s^1 (resp. s^2) dans l'espace d'états de L^1 (resp. L^2). En d'autres mots, l'espace d'états de $L^1 \times L^2$ est inclus dans le produit cartésien des espaces d'états de L^1 et L^2.

De plus, pour tous Sys, G et G' trois LTS quelconques, si $langage(G) = langage(G')$ et si G et G' sont définis sur le même alphabet, alors les ensemble d'états du LTS Sys atteints par composition avec G ou G' sont les même :

$$\begin{aligned} & \{s_{sys} \mid \exists (s_{sys}, \_) \text{ atteignable dans } Sys \times G\} \\ = \; & \{s_{sys} \mid \exists (s_{sys}, \_) \text{ atteignable dans } Sys \times G'\} \end{aligned}$$

Langage de la composition Telle que la composition est définie, on a :

$$langage(L^1) \cap langage(L^2) \subseteq langage(L^1 \times L^2)$$

C'est à dire que si il existe un chemin dans L^1 et un chemin dans L^2 portant la même trace, alors il existe une chemin dans $L^1 \times L^2$ portant cette trace.

La preuve peut en être faite en exploitant la définition de la composition en conjonction avec le fait que si une trace appartient à la fois à $langage(L^1)$ et à $langage(L^2)$, alors toutes ses actions observables appartiennent à l'intersection des alphabets $A_L^1 \cap A_L^2$ (pas d'action observable propre uniquement à L^1 ou L^2).

Par extension, on a aussi :

$$A_L^1 = A_L^2 \Rightarrow langage(L^1) \cap langage(L^2) = langage(L^1 \times L^2)$$

C'est à dire que si les alphabets de L^1 et de L^2 sont les même, alors un chemin existe dans $L^1 \times L^2$ si et seulement si il existe un chemin dans L^1 et il existe un chemin dans L^2 tels que tous ces chemins portent la même trace.

2.3.3 Algorithmie d'exploration

Algorithm 1 Generic exploration algorithm

```
function REACHABILITY(LTS : Sys)
    let (S, S_0, _, δ) ← Sys
    let K, Q ← S_0
    while Q ≠ ∅ do
        let s ← oneOf(Q)
        Q ← Q \ {s}
        let R ← {s' ∈ S | s → s' ∈ δ}
        for all s' ∈ R \ K do
            K ← K ∪ {s'}
            Q ← Q ∪ {s'}
        end for
    end while
    return K
end function
```

L'exploration d'un système de transition consiste à en expliciter l'espace des états atteignables. L'algorithme 1 en présente une mise-en-œuvre générique.

Algorithmie Les éventuelles étiquettes du LTS en entrée n'ont pas d'impact et sont ignorées ($(S, S_0, \_, \delta) \leftarrow Sys$). L'algorithme peut donc être appliqué à un système de transition non-étiquetées en entrée.

K correspond à l'ensemble des états atteints. Q correspond à l'ensemble des états atteints dont les transitions sortantes n'ont pas encore été parcourues. Ces deux ensembles sont initialisé à S_0. Tant que $Q \neq \emptyset$, pour chaque pas d'exploration :

— un nouvel état à explorer s est choisi dans Q ($s \leftarrow oneOf(Q)$) ;

— s est retiré de Q ($Q \leftarrow Q \setminus \{s\}$) ;

— l'ensemble des états s' dont il existe une transition de s vers s' est construit ($R = \{s' \in S | s \rightarrow s' \in \delta\}$) ;

— tous les états de R qui ne sont pas déjà dans K ($\forall s' \in R \setminus K$) sont ajoutés à K puis Q.

Terminaison Ainsi, l'ajout d'un état $s' \in R \setminus K$ à K et Q dénote la découverte d'un nouvel état atteignable. Retirer un état $s \in Q$ de Q en dénote l'*exploration* (l'ensemble de ses transitions sortantes sont parcourues).

On peut noter qu'un état ne peut être découvert qu'une seule et unique fois :

— $s' \in R \setminus K \Rightarrow \mathbf{s' \notin K}$ (garde pour la découverte de s') ;

— $K \leftarrow K \cup \{s'\} \Rightarrow \mathbf{s' \in K}$ (après découverte de s') ;

— pas d'état retiré de K.

En d'autres mots, un état n'est ajouté à Q (et ultérieurement retiré) qu'une seule et unique fois.

Cette dernière observation et le fait que l'ensemble des états du système (S) est fini assurent la terminaison de l'algorithme.

Généricité L'algorithme, tel que présenté, peut être raffiné pour obtenir différentes variations telles que une exploration en largeur d'abord (BFS) ou en profondeur d'abord (DFS).

Dans ces deux cas, Q est alors considérée comme une liste pour laquelle tout élément ajouté l'est à la fin. Pour BFS, $oneOf(Q)$ renvoie le premier élément ; pour DFS, le dernier.

Selon la représentation choisie pour K, Q et la définition de $oneOf(Q)$ il est possible de définir d'autres variations.

2.4 Model-checking borné

Intuitivement, le model-checking borné (BMC, [11]) énumère l'espace des états atteignables par un chemin de longueur égale ou inférieure à une borne donnée.

SAT Cette technique est classiquement présentée comme un problème de satisfaisabilité booléenne (SAT [6]). Il s'agit de déterminer si, pour une propriété à vérifier et une borne n données, il existe (ou non) un contre-exemple de taille n ou inférieure (i.e. un état atteignable par un chemin de taille n ou inférieure).

La première étape consiste donc à capturer le problème initial sous la forme du problème de la satisfaisabilité d'une proposition booléenne. Pour ce faire, on introduit quelques notations :

— Un état est un vecteur de variable booléenne : $\overrightarrow{s}$. Les variables d'états sont donc représentées sous forme binaire (ex : un entier représenté sur 8 *bits* requière 8 variables).

— Les états initiaux sont ceux satisfaisant une proposition dédiée notée I. Il s'agit donc de l'ensemble des $\overrightarrow{s_0}$ tel que la proposition $I(\overrightarrow{s_0})$ est satisfaite.

— La relation de transition est une relation sur les états notée R_δ. Une transition entre $\overrightarrow{s_1}$ et $\overrightarrow{s_2}$ existe si la proposition $R_\delta(\overrightarrow{s_1}, \overrightarrow{s_2})$ est satisfaite.

Sur ces bases, décider si il existe un chemin de taille n entre un état initial $\overrightarrow{s_0}$ et un état $\overrightarrow{s_{n-1}}$ est équivalent à déterminer l'existence d'une valuation satisfaisant la proposition suivante :

$$I(\overrightarrow{s_0}) \wedge R_\delta(\overrightarrow{s_0}, \overrightarrow{s_1}) \wedge \ \ldots \ \wedge R_\delta(\overrightarrow{s_{n-2}}, \overrightarrow{s_{n-1}})$$

Pour un prédicat sur les états P donné, on peut alors capturer les chemins de taille n menant à un contre exemple ainsi :

$$I(\overrightarrow{s_0}) \wedge R_\delta(\overrightarrow{s_0}, \overrightarrow{s_1}) \wedge \ \ldots \ \wedge R_\delta(\overrightarrow{s_{n-2}}, \overrightarrow{s_{n-1}}) \wedge \neg \mathbf{P}(\overrightarrow{\mathbf{s_{n-1}}})$$

Si cette proposition booléenne ne peut pas être satisfaite, P est vérifié sur l'ensemble des états atteignables par un chemin de taille n. Si une valuation des variables satisfaisant cette proposition booléenne existe, il s'agit d'un contre-exemple à la vérification de P.

Système de transition La suite de cette section en présente deux définitions équivalentes reposant sur les notions de système de transitions et de composition synchrone déjà introduites dans ce document

On considère en entrée un système de transition $Sys = (S, S_0, \delta)$ et une borne n. Le model-checking de Sys borné à n consiste à restreindre l'énumération des états atteignables de Sys à ceux atteignables par des chemins dont la taille est égale ou inférieure à n.

Par construction Le model-checking borné à n de Sys revient au model-checking du système de transition $Sys^n = (S^n, S_0^n, \delta^n)$ défini ainsi :

— $S^n = S \times [0, n]$;
— $S_0^n = S_0 \times \{0\}$;
— $\delta^n = \{(s, i) \to (s', i+1) \mid s \to s' \in \delta \wedge i < n\}$.

En langage naturel, l'ensemble d'états de Sys^n est l'ensemble des couples composés d'un état de Sys et d'un entier entre 0 et n (le produit cartésien $S^n = S \times [0, n]$). Cet entier dénote la profondeur à laquelle l'état est atteint. Par définition, la profondeur des états initiaux est 0 ($S_0^n = S_0 \times \{0\}$).

Une transition de l'état (s, i) à l'état $(s, i+1)$ existe dans δ^n si et seulement si la transition de s à s' existe dans δ et si $i < n$. La profondeur de l'état cible d'une transition est donc égale à la profondeur de l'état source incrémentée de 1.

Par composition synchrone De manière équivalente, le model-checking borné à n peut être exprimé comme une composition synchrone de deux systèmes de transitions, Sys et un compteur borné à n.

Telle que présentée section 2.3.2, définition 3, la composition synchrone porte sur des LTS. Dans le cas de systèmes de transitions non-étiquetés, on considère que toutes les transitions sont étiquetées par une action observable unique. On force ainsi la synchronisation des deux participants sur l'ensemble de leurs transitions (pas de bégaiement ni de synchronisation avec un seul participant).

Ainsi définie, les ensembles d'états et d'états initiaux de la composition synchrone de Sys et d'un compteur borné à n correspondent exactement à ceux obtenus par construction plus haut. La relation de transition est définie par l'équivalence suivante (extraite de la définition 3, section 2.3.2) :

$\forall a \in A_L^1 \cap A_L^2,\ (s^1, s^2) \xrightarrow{a} (s'^1, s'^2) \in \delta \ \Leftrightarrow\ s^1 \xrightarrow{a} s'^1 \in \delta^1 \wedge s^2 \xrightarrow{a} s'^2 \in \delta^2$

Avec $A_L^1 = A_L^2 = \{a\}$ (action observable unique), $L^1 = Sys = (S, S_0, \delta_{sys})$, $L^2 = ([0, n], \{0\}, \{(i, j) \in [0, n] \times [0, n] | j = i + 1\})$ le compteur borné à n, cette équivalence peut être réécrite dans la suivante :

$(s, i) \to (s', j) \in \delta \ \Leftrightarrow\ s \to s' \in \delta_{sys} \wedge (i, j) \in [0, n] \times [0, n] \wedge j = i + 1$

L'équivalence avec la définition ensembliste (par construction) présentée plus haut est alors triviale.

Sous-approximation L'espace d'états énuméré par le model-checking borné àn correspond à ceux atteignables par un chemin de longueur égale ou inférieure à n. Il s'agit donc d'une sous-approximation de l'espace d'états.

Cela implique que tous les états atteints par l'énumération bornée sont atteignables dans le cas non-borné. Par contre la réciproque n'est pas vraie dans le cas général, si un état est atteignable par un chemin de taille strictement supérieure à n, il n'est potentiellement pas atteint par un chemin de taille n ou inférieure.

Pour la vérification de propriétés, il convient donc de fournir une justification attestant que la borne retenue est suffisante.

Seuil d'exhaustivité Pour un système de transition donné, il existe une borne nécessaire et suffisante pour laquelle tous les états sont atteints. La preuve peut en être fait du fait que :

— l'espace d'états est, par définition, fini ;

— si un état est atteignable, alors il existe un chemin fini (i.e. dont la taille peut être déterminée) y menant.

La borne minimale est appelée le seuil d'exhaustivité. Fournir la preuve que la borne retenue est supérieure ou égale à ce seuil d'exhaustivité permet de garantir la validité du processus de vérification. Pour les vérifications de sûreté, ce seuil peut être obtenu à partir du diamètre d'atteignabilité [27].

Preuve par k-induction Pour justifier de l'exhaustivité du model-checking borné à n, une approche est de fournir la preuve que pour tous les états atteignables il existe un chemin y menant de taille inférieure ou égale à n.

Par induction, il s'agit de montrer que pour tout $k \geq n$, si un état est atteint par un chemin de taille $k+1$ alors il existe une chemin de taille inférieure ou égale à k y menant aussi.

Cette approche, communément appelée la k-induction [31], permet aussi la vérification d'invariants.

2.5 Vérification guidée par les contextes

La vérification guidée par le contexte (CaV, [20]) est une technique de model-checking axée sur la décomposition du problème. Cette section en présente le cadre général, une mise-en-œuvre de l'approche et liste quelques problèmes liés à celle-ci.

2.5.1 Cadre général

Cibler l'effort de vérification Pour contourner l'explosion de l'espace d'états lors du model-checking d'un système, les techniques classiques évoquées section 2.2 peuvent ne plus suffirent.

La complexité de ces études étant en partie due à la composition du système avec une représentation formelle des exigences, il est souvent nécessaire de cibler l'effort de vérification sur des sous-parties de l'ensemble de ces exigences.

Cela permet aussi de spécialiser le système en fonction des sous-parties considérées en élaguant les comportements neutres vis-à-vis des exigences considérées.

Cette décomposition du problème revient à spécifier plusieurs unités de preuve, chacune de complexité réduite vis-à-vis du problème initial.

Problématique Cependant, les travaux sur le model-checking présentés ou évoqués jusqu'ici supposent un système fermé en entrée. C'est à dire une modélisation contenant à la fois les comportements à vérifier (le système ouvert), le cadre d'utilisation (l'environnement) et parfois les exigences.

Sans une approche adaptée, produire les unités de preuve, les maintenir (entre elles et vis-à-vis de la spécification) et justifier de la couverture est un processus laborieux et source d'erreurs.

Context-aware verification Dans le cadre de la vérification de systèmes-embarqués, il est souvent aisé de distinguer le système-ouvert de son environnement. La vérification guidée par les contextes (CaV, [20]) permet d'exploiter cette observation pour faciliter la modélisation des unités de preuve et leurs maintenances.

Dans sa forme la plus générale, il s'agit de distinguer les comportements à vérifier (le système ouvert) de ceux correspondant à des interactions avec l'environnement (la fermeture du système).

Le système ouvert constitue alors une partie invariante des unités de preuves. La fermeture du système, modélisée indépendamment, en constitue (avec les exigences pertinentes) la partie variable.

Sur cette base, l'approche apporte des réponses aux problèmes liés à la production et la maintenabilité de ces unités de preuve. Bien qu'elle n'adresse pas directement le problème de leur couverture, l'unicité de la modélisation des comportements à vérifier permet de réduire l'effort nécessaire.

Il est important de noter que l'impact de l'approche par CaV peut se limiter à une phase en amont du model-checking, chaque unité de preuve pouvant être vue comme un système

fermé spécialisé. Dans sa forme la plus générale, elle est donc applicable indifféremment des algorithmes de vérification utilisés en aval.

La section suivante présente une mise en œuvre de la CaV via un formalisme dédié.

2.5.2 Mise en œuvre de l'approche via CDL

Comme défini précédemment, la CaV est basée sur la distinction entre les comportements à vérifier (le système ouvert) et les interactions avec l'environnement (la fermeture du système).

Contexte de preuve CDL Dans le cadre de la mise en œuvre avec CDL [18, 21], cette distinction est réalisée par une **séparation** formelle, méthodique et explicite du système ouvert, de son environnement et des exigences.

Une spécification CDL de l'environnement est appelée le **contexte de preuve**.

Un contexte de preuve CDL permet de capturer le langage des interactions avec l'environnement via des événements. Il s'agit de **communications asynchrones**, émissions et/ou réceptions, sur un ensemble fini de messages.

Ces événements constituent l'alphabet d'une spécification CDL. La sémantique en permet la combinaison via divers opérateurs (séquence, alternative, répétition bornée, parallélisme). On peut noter qu'il s'agit d'une définition **acyclique** de la sémantique, c'est à dire qu'il n'est pas possible de spécifier une suite d'événements infinie.

CDL permet aussi la spécification de propriétés à vérifier sous la forme d'invariants ou d'automates observateurs.

Spécification du système ouvert Pour permettre la *composition* d'un contexte de preuve CDL avec le système ouvert, ce dernier doit exposer des **variables partagées** (files de messages).

Le formalisme **Fiacre** [23] est utilisé pour capturer le système ouvert et offrir l'interface de communications asynchrones entre ce dernier et l'environnement. Il s'agit aussi du formalisme utilisé pour la déclaration des messages référencés par les événements CDL.

Outillage La figure 2.2 présente les entrées et sorties d'*Observer-Based Prover* (OBP [21]) pour la vérification d'un système ouvert dans un contexte de preuve donné.

On retrouve le système ouvert exprimé via Fiacre, le contexte de preuve et les propriétés à vérifier exprimés via CDL en entrée.

L'exploration exhaustive de la composition réalisée par OBP énumère l'ensemble des états atteignables (l'espace d'états) à travers un *LTS* (voir section 2.3.1, définition 2).

Le résultat renvoyé par OBP est soit un constat d'explosion de l'espace d'état, soit, pour chaque propriété en entré, la validation de cette propriété ou un contre-exemple dans le cas contraire.

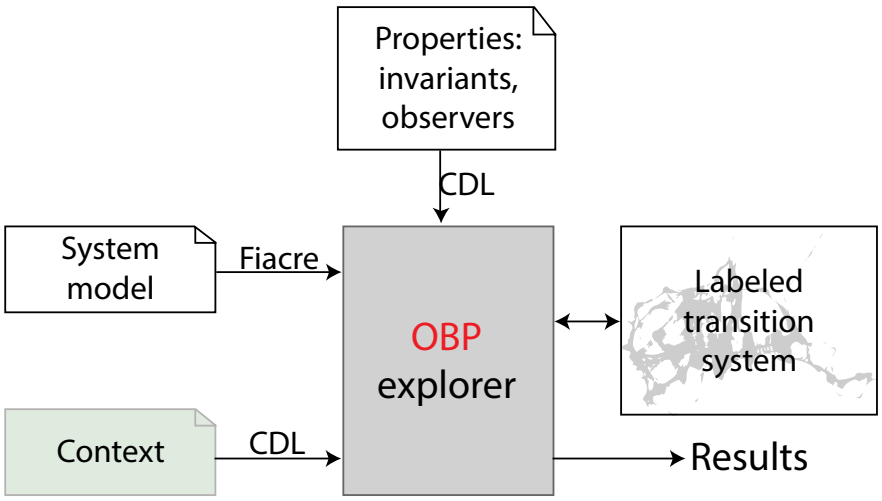

FIGURE 2.2 – Context-aware verification via CDL and OBP

Résultats L'approche ainsi mise-en-œuvre nous a permis d'obtenir des résultats encourageants sur des cas d'études de dimensions industrielles [9, 7].

Outre le gain obtenu par la division du problème en amont, la modélisation des interactions de l'environnement par un formalisme dédié a permis la mise en place d'algorithmes performants spécifiques à l'approche [34, 19].

2.5.3 Limites et opportunités

Cette section revient sur certains des points spécifiques de CDL. Il s'agit de limites identifiés de cette mise en œuvre de l'approche par CaV.

Pas de modèle fermé de référence Le système fermé *d'origine* n'est pas explicitement défini. Il peut être obtenu par composition du système ouvert avec l'union des comportements définis par les différents contextes de preuve CDL. Mais cette représentation varie donc avec les contextes, est peu maintenable et ne peut servir de modèle de référence.

De plus, il s'agit d'un point de divergence avec un grand nombre de techniques, ce qui impacte négativement la complémentarité de la CaV vis-à-vis de celles-ci.

Limité à des communications asynchrones pour la modélisation des interactions Le système ouvert doit exposer des variables partagées pour permettre la communication avec l'environnement. En pratique, de nombreux cas d'études se prêtent bien à l'exercice et il est souvent possible d'instrumenter d'autres approches à travers ces communications asynchrones [9, 7].

Il s'agit néanmoins d'un manque de flexibilité dans la modélisation et d'un frein à l'application de l'approche à des formalismes autres que Fiacre.

Acyclicité de la sémantique De plus, l'acyclicité forcée par la sémantique de CDL restreint l'expressivité à des chemins finis. Il s'agit d'une limitation forte mais qui peut être exploitée pour contourner le problème d'explosion de l'espace d'états.

Dans un processus de développement, l'environnement est classiquement illustré à travers des diagrammes de séquence acycliques. De ce fait, en pratique, il est souvent possible de fournir des éléments garantissant l'exhaustivité de l'analyse malgré l'acyclicité de l'environnement.

Cependant, lorsque ce n'est pas le cas, il n'est plus possible de formellement valider le système. Le coût induit par le model-checking sur le développement est alors prohibitif pour l'application de l'approche.

Ces différents points posent problème pour la robustesse du cycle de conception. Cependant, ils s'agit de spécificités propres à CDL et non au cadre général de la CaV.

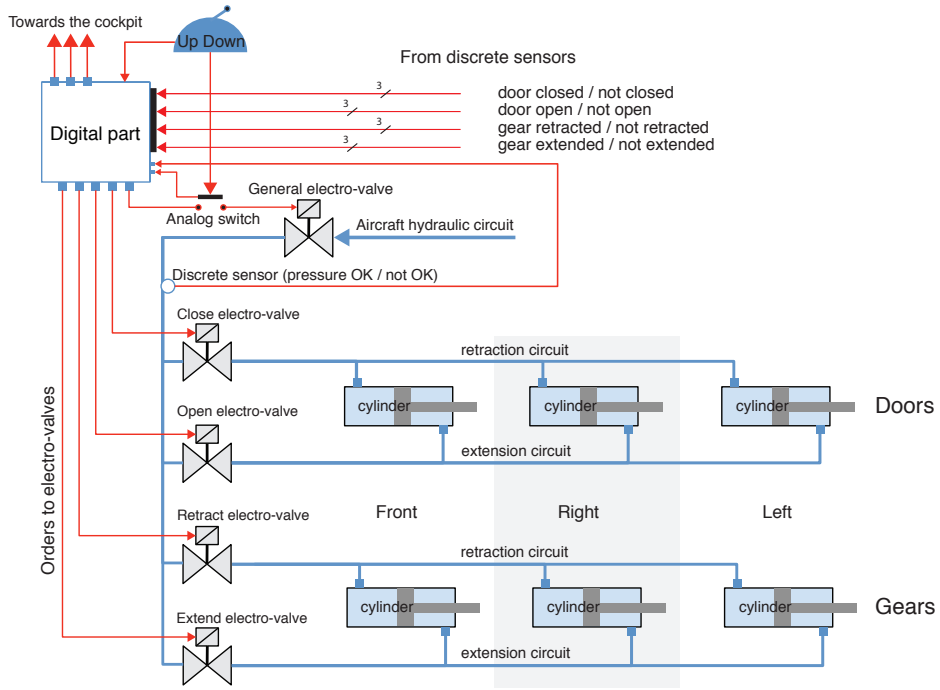

FIGURE 2.3 – *LGS* : Architecture

2.6 Cas d'étude : le train d'atterrissage (LGS - Avionique)

Le cas d'étude du train d'atterrissage présenté ici a été originalement soumis comme point d'entrée commun de la quatrième conférence internationale ABZ (ABZ'14 [3]) : *The Landing Gear System Case Study* [8] (Frédéric Boniol et Virginie Wiels).

Cette section constitue un résumé de la spécification [8] se focalisant sur les points nécessaires à la compréhension du chapitre 3 et de la section 4.2.

Généralités Le LGS est composé de trois trains d'atterrissage (à gauche, à droite et à l'avant de l'avion). Ceux-ci sont actionnés conjointement.

Physiquement, un système hydraulique (électrovannes, vérins) permet de lever et abaisser ces trains d'atterrissage.

La partie logicielle du cas d'étude porte deux responsabilités : le contrôle des parties physiques lors de la levée et de l'abaissement des trains ; la détection de fautes (porte, vérin ou électrovanne bloqué).

La figure 2.3 présente les différents composants du *LGS*, physiques et logiciels.

Interface pilote Le pilote peut lever et abaisser les trains d'atterrissages à tous moment via un levier de commande unique (en haut de la figure 2.3).

Ces actions ne sont pas instantanées et il doit être possible d'inverser la séquence en cours.

Le pilote est tenu à jour de l'état courant et des possibles fautes détectées via des indicateurs lumineux (signaux *towards the cockpit*, figure 2.3).

Composants physiques Comme présenté figure 2.3, le système comporte six vérins hydrauliques correspondants aux trains d'atterrissage et à leurs portes respectives. Ceux correspondants aux portes sont connectés via les même électrovannes d'ouverture et de fermeture. Il en va de même pour les vérins actionnant les trains d'atterrissage.

De plus, l'ensemble du circuit hydraulique est connecté à une électrovanne générale.

L'ensemble de ces composants peuvent se bloquer à tous moment.

Composants logiciels Le système comporte de l'informatique embarqué (*Digital part* sur la figure 2.3) pour le contrôle et le *monitoring* des parties physiques.

La partie de contrôle actionnent les différentes électrovannes pour réaliser les séquences d'abaissement et de relèvement des trains d'atterrissage.

La partie de *monitoring* synthétise la lecture des capteurs en termes d'indicateurs lumineux pour tenir le pilote informé des possibles dysfonctionnement.

Exigences Cette dernière partie logicielle (*monitoring*) est la cible de la validation attendue. Il s'agit de vérifier que les fautes éventuelles sont correctement détectées et que les indicateurs lumineux sont tenus à jour.

2.7 Conclusion

Ce chapitre a introduit le cadre dans lequel les contributions du présent document s'inscrivent, à savoir :

— le model-checking (section 2.2) ;

— la vérification guidée par le contexte (section 2.5).

De plus, des prérequis à la compréhension de la suite on été présenté :

— les systèmes de transitions et concepts connexes (section 2.3) ;

— le model-checking borné (section 2.4) ;

— le cas d'étude du train d'atterrissage (LGS, section 2.6).

Chapitre 3

Vérification par parties et non-intrusive

3.1 Introduction

Ce chapitre s'inscrit dans le cadre de la CaV [20], introduite section 2.5. Une description générale de l'approche et une mise-en-œuvre à travers le formalisme dédié CDL [18, 21] ont été alors présentés.

Des résultats encourageants ont été obtenus, illustrant la pertinence de l'approche pour le passage à l'échelle de cas d'études de tailles conséquentes [9, 7]. Cela est rendu possible par une décomposition du problème en amont de la vérification propre à la CaV, et par des algorithmes exploitant les particularités de CDL [34, 19].

Cependant, en fin de section 2.5, plusieurs limites spécifiques à CDL sont identifiés :

— pas de modèle fermé de référence (incluant les comportements à vérifier ainsi qu'une formalisation exhaustives de l'environnement) ;

— composition uniquement à travers des variables partagées modélisant des communications asynchrones ;

— acyclicité de la sémantique.

Pour les adresser, la section 2.5 constitue un premier pas important en proposant une définition plus générale de l'approche. En particulier, la **séparation** entre le système à vérifier et son environnement, hypothèse centrale à l'approche, est reformulée comme une **distinction**. C'est à dire, la capacité à distinguer les transitions dénotant une interaction avec l'environnement de celles représentant un comportement *interne* au système.

Sur cette base, il est alors possible de proposer une nouvelle mise-en-œuvre de l'approche.

Généralisation de la CaV Cette section propose donc une généralisation des concepts mis en jeu dans par la CaV. L'enjeu est d'offrir un cadre de modélisation plus flexible et robuste tout en conservant les bons résultats obtenus via CDL.

La section 3.2 introduit xGDL, le formalisme au cœur de cette nouvelle itération de l'approche. Plutôt que de se baser sur une séparation explicite, on suppose en entrée le système fermé et exhaustif que l'on se propose de restreindre pour la définition des unités de preuve.

La section 3.3.1 présente la compilation et le dépliage borné d'une spécification xGDL. Ce dernier permet, lorsque nécessaire, de déplier, pour une profondeur donnée, une spécification xGDL cyclique dans un graphe acyclique.

Les sections 3.4 et 3.5 présentent deux importantes contributions, le Past-Free[ze] et le Split. Il s'agit d'algorithmes exploitant les particularités de la CaV.

La section 3.6 ouvre une fenêtre sur le model-checking borné et discute d'une possible solution au problème de la couverture de l'approche après dépliage borné d'une spécification xGDL.

En fil rouge, le cas d'étude du LGS [8] présenté section 2.6 sert d'illustration pour les concepts introduits.

3.2 Vers un langage de description de contextes

Cette section introduit xGDL, un langage dédié à la capture de l'environnement du système à vérifier. Vis-à-vis de CDL [18, 21], ce nouveau formalisme lève la contrainte d'acyclicité et offre une approche plus flexible et générale de la composition avec le système (jusqu'alors limitée à des communications asynchrones).

La syntaxe et la sémantique de xGDL sont définis indépendamment du langage utilisé pour spécifier le système à vérifier. Le lien est réalisé via des actions atomiques exprimées de manière externe, référencées par xGDL et exécutables sur une configuration système. Un guide de vérification peut alors être vu comme un processus interagissant avec le système via un alphabet fini d'actions atomiques.

Pour l'exemple, dans le cas d'étude du LGS les actions possibles sont lever, abaisser le train d'atterrissage et injecter l'une des possibles fautes (comme bloquer un vérin dans une position).

La syntaxe et la sémantique de xGDL sont définies sections 3.2.1 et 3.2.2. La notion d'alphabet d'interaction, est précisée section 3.2.3.

3.2.1 Syntaxe abstraite

La syntaxe de xGDL est exprimée par la grammaire de style BNF suivante :

$$C \ ::= a \mid C;C \mid C \square C \mid C \| C \mid C? \mid C+ \mid C* \mid C\{i,j\} \mid \{i,j\} of [C_1, ..., C_n]$$

Avec $C \in \mathcal{E}$, l'ensemble des termes de xGDL ; $a \in A$, l'alphabet d'actions atomique ; $i, j \in \mathbb{N}$ avec $i \leq j$.

En langage naturel, un terme C dans un guide de vérification xGDL correspond soit à : *1)* une action atomique ; *2)* une séquence de deux termes ; *3)* un choix non-déterministe entre deux termes ; *4)* une composition parallèle (entrelacement) de deux termes ; *5)* un terme optionnel ; *6)* une répétition non-bornée d'un terme avec au moins une occurrence ; *7)* une répétition non-bornée d'un terme (potentiellement zéro) ; *8)* une répétition bornée d'un terme

avec au moins i et au plus j occurrences ; *9)* l'ensemble des permutations de tailles i à j de l'ensemble $[C_1, ..., C_n]$.

Ce dernier opérateur est particulièrement utile pour l'injection de faute. Par exemple, $\{2, 2\}of[f_1, ..., f_n]$ correspond à l'injection de deux fautes différentes parmi les n possibles.

Exemple Dans le cas du LGS, si a_l et a_b correspondent respectivement à lever et abaisser le train d'atterrissage, alors $(a_l\ ;\ a_b) *$ spécifie une répétition possiblement infinie de ces deux actions alternées.

3.2.2 Sémantique opérationnelle

La sémantique opérationnelle est basée sur la relation de transition représentée par l'ensemble $T = \{C \xrightarrow{a} C' | a \in A^+ \ \wedge \ C, C' \in \mathcal{E}^+\}$, où $A^+ = A \cup \{\tau\}$ avec $\tau \notin A$ et $\mathcal{E}^+ = \mathcal{E} \cup \{\bot\}$. C'est à dire que pour deux termes C et C', si la transition $C \xrightarrow{a} C'$ existe, alors le terme C peut être transformé dans le terme C' en exécutant l'action a. L'alphabet d'actions atomiques A est étendu avec l'action invisible τ pour en simplifier l'expression. Le terme $\bot$ correspond au terme final tel que $\forall\ C \in \mathcal{E}^+, \forall\ a \in A^+, \bot \xrightarrow{a} C \notin T$ (pas de transition sortante). Ce terme est utilisé pour définir la terminaison.

Les règles définissant la sémantique de xGDL sont présentées ici en trois sous-parties. Le cœur du langage, c'est à dire un sous-ensemble d'opérateurs vers lequel les autres peuvent être mis à plat (séquence, alternative et parallélisme) ; les différentes répétitions (terme optionnel, répétitions non-bornée et bornée) et l'opérateur de permutation.

Séquence, alternative et parallélisme

$$\frac{a \in A^+}{a \xrightarrow{a} \bot}\ [\text{atom}] \qquad \frac{a \in A^+}{a; C \xrightarrow{a} C}\ [\text{seq}_1] \qquad \frac{C_1 \xrightarrow{a} C_1' \wedge C_1 \neq a}{C_1; C_2 \xrightarrow{a} C_1'; C_2}\ [\text{seq}_2]$$

$$\frac{}{C_1 \square C_2 \xrightarrow{\tau} C_1}\ [\text{alt}_1] \qquad \frac{}{C_1 \square C_2 \xrightarrow{\tau} C_2}\ [\text{alt}_2] \qquad \frac{C_1 \xrightarrow{a} C_1'}{C_1 \| C_2 \xrightarrow{a} C_1' \| C_2}\ [\text{par}_1]$$

$$\frac{C_2 \xrightarrow{a} C_2'}{C_1 \| C_2 \xrightarrow{a} C_1 \| C_2'}\ [\text{par}_2] \qquad \frac{}{\bot \| C \xrightarrow{\tau} C}\ [\text{par}_3] \qquad \frac{}{C \| \bot \xrightarrow{\tau} C}\ [\text{par}_4]$$

En langage naturel, si le terme à exécuté n'est composé que d'une **action atomique**, celle-ci est exécuté et le terme est transformé dans le terme final $\bot$ notifiant la terminaison ($a \xrightarrow{a} \bot$, règle *atom*).

Si le terme est une **séquence** débutant par une action atomique, l'action est exécutée et le terme transformé dans la partie droite de la séquence considérée ($a; C \xrightarrow{a} C$, règle *seq*$_1$). Si le terme est une séquence dont la partie gauche est un terme composé C_1 tel que $C_1 \xrightarrow{a} C_1'$, alors l'action a est exécutée, le terme C_1 est transformé dans le terme C_1' et le membre droit de la séquence est conservé ($C_1; C_2 \xrightarrow{a} C_1'; C_2$, règle *seq*$_2$).

Si le terme est un **choix non-déterministe** ($C_1 \square C_2$), il peut évoluer soit dans son membre gauche ($C_1 \square C_2 \xrightarrow{\tau} C_1$, règle alt_1) soit dans son membre droit ($C_1 \square C_2 \xrightarrow{\tau} C_2$, règle alt_2). Dans les deux cas, aucune action n'est exécutée (τ représentant l'action nulle).

Enfin, si le terme est une **composition parallèle** ($C_1 \| C_2$) il peut évoluer soit sur sa partie gauche ($C_1 \| C_2 \xrightarrow{a} C_1' \| C_2$, règle par_1), soit sur sa partie droite ($C_1 \| C_2 \xrightarrow{a} C_1 \| C_2'$, règle par_2) en exécutant l'action associée. Si l'une de ses parties est le terme final $\perp$, il est transformé dans le terme restant (règles par_3 et par_4)

Répétitions Les opérateurs de répétitions sont définis comme suis :

$$\frac{}{C? \xrightarrow{\tau} \perp \square C}\ [\text{opt}] \qquad \frac{}{C* \xrightarrow{\tau} (C; C*)?}\ [\text{star}]$$

$$\frac{}{C+ \xrightarrow{\tau} C; C*}\ [\text{plus}] \qquad \frac{0 < i \leq j}{C\{i,j\} \xrightarrow{\tau} C; C\{i-1, j-1\}}\ [\text{rep}_1]$$

$$\frac{i = 0 \wedge j > 0}{C\{i,j\} \xrightarrow{\tau} (C; C\{0, j-1\})?}\ [\text{rep}_2] \qquad \frac{i = j = 0}{C\{i,j\} \xrightarrow{\tau} \perp}\ [\text{rep}_3]$$

Un **terme optionnel** ($C?$) est équivalent au choix non-déterministe entre $\perp$ et C ($\perp \square C$, règle opt). Soit l'argument est ignoré, soit il est évalué.

La **répétition non-bornée** ($C*$) est définie récursivement par la séquence optionnelle composée de C et de $C*$ ($(C; C*)?$, règle $star$). Après un pas de résolution, soit la répétition termine ($\perp$), soit l'argument est évalué une fois et la répétition continue ($C; C*$).

La **répétition non-bornée avec au moins une occurrence** ($C+$) est équivalente à la séquence composée de C et de $C*$ ($C; C*$, règle plus).

La **répétition bornée** entre i et j ($C\{i,j\}$) est définie via les règles rep_1, rep_2 et rep_3. Tant que $0 < i < j$, la première règle (rep_1) évalue C et décrémente les bornes ($C; C\{i-1, j-1\}$). Pour $i = 0$ et $0 < j$, la seconde règle (rep_2) soit termine la répétition (terme optionnel), soit évalue C et décrémente la borne supérieure j ($(C; C\{0, j-1\})?$). Enfin, pour $0 = i = j$, la dernière règle (rep_3) transforme la répétition dans le terme vide $\perp$ (terminaison). En d'autres mots, C est évalué séquentiellement au moins i fois et au plus j fois.

Permutations Enfin, l'opérateur de permutation est défini par les règles suivantes :

$$\frac{0 < i \leq j \leq n \wedge \forall k, 1 \leq k \leq n}{\{i,j\}\ of\ [C_1, ..., C_n] \xrightarrow{\tau} C_k; \{i-1, j-1\}\ of\ [C_1, ..., C_{k-1}, C_{k+1}, ..., C_n]}\ [\text{perm}_1]$$

$$\frac{\mathbf{0 = i < j} \leq n \wedge \forall k, 1 \leq k \leq n}{\{i,j\}\ of\ [C_1, ..., C_n] \xrightarrow{\tau} (C_k; \{0, j-1\}\ of\ [C_1, ..., C_{k-1}, C_{k+1}, ..., C_n])\ \mathbf{?}}\ [\text{perm}_2]$$

$$\frac{0 = i = j}{\{i,j\}\ of\ [C_1, ..., C_n] \xrightarrow{\tau} \perp}\ [\text{perm}_3] \qquad \frac{}{\{i,j\}\ of\ [] \xrightarrow{\tau} \perp}\ [\text{perm}_4]$$

La **permutation** permet de spécifier l'ensemble des arrangements en séquences de taille i à j entre les termes d'un ensemble de taille n dénoté $[C_1, ..., C_n]$. Le sous-terme

$[C_1, ..., C_{k-1}, C_{k+1}, ..., C_n]$ correspond à ce même ensemble minus le terme C_k avec $1 \leq k \leq n$. La règle $perm_1$ (cas $0 < i \leq n$) correspond au choix non-déterministe d'un terme C_k, lequel est composé en séquence avec la suite de l'arrangement $(\{i-1, j-1\} of [C_1, ..., C_{k-1}, C_{k+1}, ..., C_n])$. La règle $perm_2$ (cas $0 = i < j \leq n$) est similaire mais son membre droit (le terme transformé) est optionnel, la taille de l'arrangement attendu étant possiblement 0. Les règles $perm_3$ et $perm_4$ correspondent à la terminaison dans les cas où $j = 0$ et où l'ensemble de termes en argument est vide, respectivement.

Sémantique fermée par préfixe Décrite ainsi, la sémantique de xGDL correspond à celle des expressions régulières étendues pour le parallélisme et les permutations. Cependant, un terme accepté dénote une séquence possible d'interactions. Par définition, **l'ensemble des préfixes d'un terme accepté, dont $\perp$, sont aussi des séquences possibles** et doivent donc être acceptés. La sémantique de xGDL est donc dite fermée par préfixe.

3.2.3 Alphabet d'interactions

Système de transitions On considère les systèmes pouvant être exprimés (nativement ou implicitement) par un système de transition (S, S_0, δ) (voir section 2.3.1, définition 1).

Les transitions d'un tel système (δ) peuvent être vu comme des comportements atomiques. Pour permettre la modélisation de guides de vérification, il est nécessaire d'identifier au préalable celles correspondantes à des comportement *internes* (δ_o, le système ouvert) de celles dénotant une interaction avec l'environnement (δ_f, la fermeture du système).

Cette distinction doit être totale ($\delta = \delta_o \cup \delta_f$) et exclusive ($\delta_o \cap \delta_f = \emptyset$).

Fonction d'étiquetage et LTS On peut alors décider arbitrairement d'un alphabet d'actions observables A_f correspondant aux actions atomiques qui pourront être référencées par la suite par une spécification xGDL.

Le lien est réalisé par une fonction d'étiquetage $f_e : \delta \to A_f \cup \{\tau\}$ tel que :

— $\forall t \in \delta_o, f_e(t) = \tau$, toutes les transitions du système ouvert sont non-observables (étiquetées par τ) ;

— $\forall t \in \delta_f, f_e(t) \in A_f$, toutes les transitions de la fermeture du système sont observables (étiquetées dans A_f).

Via cette fonction d'étiquetage, le système de transitions représentant le système fermé peut maintenant être vu et manipulé comme un système de transitions étiquetées (LTS, voir section 2.3.1, définition 2).

Exemple Dans le cas du LGS, les entités de l'environnement sont le pilote et celle simulant l'injection de faute. Une transition système appartient à sa fermeture si elle correspond à une action de l'une d'elles. Corollairement, les transitions du système ouvert sont celles proposées par les autres entités (ex : ouverture ou fermeture des différentes vannes hydrauliques).

La fonction d'étiquetage des transitions de la fermeture renvoie a_l si la transition en argument déclenche la levée du train d'atterrissage, a_b pour l'abaissement, f_i si elle injecte la

faute indexée par i. Par définition, ces étiquettes (a_l, a_b, f_1, ..., f_i) composent donc l'alphabet d'interactions disponible lors de la modélisation de guides de vérification xGDL pour le LGS.

Propriétés de la fonction d'étiquetage La suite de cette section discute les différentes propriétés mathématiques de la fonction d'étiquetage. Les deux premiers points portent sur la pertinence d'une définition par fonction. Formellement, il s'agit d'une relation implicitement déterministe et totale.

Déterministe Si on autorise que la relation ne soit pas déterministe il peut alors exister une transition étiquetée par plusieurs étiquettes différentes. On peut alors revenir au cas déterministe en dupliquant la transition concernée autant de fois que nécessaire.

Totale Si on autorise que la relation ne soit pas totale, il peut alors exister des transitions sans étiquette. Lors de la composition avec le guide xGDL ces transitions ne sont donc pas contraintes et peuvent être vues comme comme des actions internes au système. Pour revenir à une relation totale, ces transitions peuvent être interprétées comme appartenant au système ouvert (étiquetées par τ) et non à sa fermeture.

On peut donc considérer, dans la suite de ce document, que la fonction d'étiquetage est bien une fonction au sens mathématique du terme.

Les points suivants discutent de l'injectivité et de la surjectivité.

Injective Toutes les transitions du système ouvert (δ_o) étant étiquetées par la même étiquette (τ), la fonction d'étiquetage ne peut être considérée injective dans le cas général.

Il est possible de la considérer injective sur les transitions de la fermeture (δ_f). Si ce n'est pas le cas, il existe plusieurs transitions t_1, ..., $t_n \in \delta_f$ différentes et une étiquette a tel que $\forall i, f_e(t_i) = a$. Il est alors possible d'introduire les étiquettes de a_1 à a_n tel que $f_e(t_i) = a_i$ et de remplacer les occurrences de a dans xGDL par le choix non-déterministe $a_1 \Box ... \Box a_n$ pour revenir au cas où la fonction est injective sur δ_f.

Cependant, une définition injective implique un alphabet de taille au moins égale à celle de δ_f ($|\delta_f| \leq |A_f|$) et donc peu raisonnable en pratique.

Surjective Si la fonction n'est pas surjective, il existe des étiquettes non référencées par les transitions du système ($\exists a \in A_f, \forall t \in \delta_f, f_e(t) \neq a$). Dans le cadre de la composition avec un guide xGDL, les transitions référençant ces actions observables ne peuvent donc jamais être *tirées*.

Revenir à une définition surjective implique soit d'augmenter l'ensemble des transitions du système par des transitions factices, soit de retirer ces étiquettes de l'alphabet (et donc, lors de la compilation, de retirer aussi les transitions correspondantes du guide xGDL).

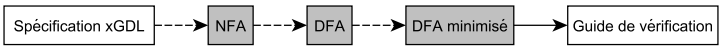

FIGURE 3.1 – Flot de compilation xGDL.

3.3 Compilation et transformations

Cette section présente les algorithmes liés à l'approche proposée et intervenant en amont de l'exploration proprement dite. La compilation, section 3.3.1, permet de passer d'une spécification xGDL vers un automate fini déterministe (DFA) minimal. Le dépliage borné, section 3.3.2, transforme le résultat de cette compilation vers un automate acyclique équivalent pour les termes de longueur égale ou inférieure à une borne donnée.

3.3.1 Compilation

Une spécification xGDL est transformée dans un automate fini déterministe (DFA) minimal en suivant le flot présenté figure 3.1 composé de trois étapes (flèches en tirets).

La première consiste à générer l'automate fini non-déterministe (NFA) correspondant aux entrées. Si la présence dans la sémantique des transitions silencieuses (étiquetées par τ) permet d'en simplifier l'expression et la lecture, laissées ainsi elles augmentent considérablement la taille du NFA obtenu.

Pour résoudre ce problème, ces transitions superflues sont retirées lors de la seconde étape par transformation vers un équivalent déterministe (DFA).

Enfin, le DFA obtenu est minimisé. Le résultat correspond (flèche pleine) au guide de vérification compilé. La suite de cette section détaille et illustre ces étapes.

Génération du NFA La première étape consiste à dériver de la spécification un automate fini non-déterministe (NFA). Celui ci peut être obtenu par exploration du système dont l'état initial est le terme correspondant à la spécification xGDL et la fonction de transition est celle de la sémantique opérationnelle définie section 3.2.2. L'approche utilisée par l'outillage est une construction Thompson-McNaughton-Yamada[35] étendue pour les opérateurs de parallélisme et de permutation.

Transformation vers DFA L'étape suivante consiste à transformer cet automate NFA obtenu dans son équivalent déterministe (DFA). Durant cette phase, les transitions étiquetées τ sont interprétées comme des ϵ-transitions (silencieuses) et retirées[26].

Minimisation du DFA Enfin, le DFA obtenu après transformation est minimisé. De nombreux algorithmes différents existent pour accomplir cette tâche[26]. L'automate résultant représente la forme compilée du guide de vérification xGDL, lequel sera utilisé pour contraindre la fermeture du système (en pratique : l'environnement) lors de la vérification. L'équivalence entre la spécification xGDL et le DFA obtenu par ce flot de compilation est assurée par les différents algorithmes utilisés [35, 26].

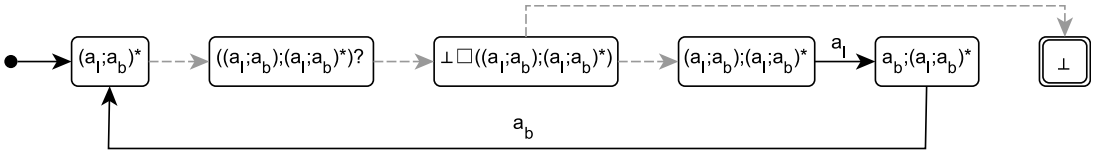

FIGURE 3.2 – $(a_l; a_b)*$ vers NFA.

Notes sur les états d'acceptation Comme mentionné section 3.2.2, la sémantique de xGDL est fermée par préfixe (si un terme est accepté, tous ses préfixes le sont aussi). En pratique, cela implique que lors de la composition avec le système tous les états de l'automate obtenu par le flot de compilation décrit ci-dessus sont des états d'acceptation.

Cependant, les états d'acceptation (au sens des expressions régulières) sont conservés pour dénoter une terminaison possible. Cela est utile pour conserver la correspondances entre la spécification en entrée et sa forme compilée vis à vis de compositions ultérieures. Dans la suite il sont représentés sur les différentes figures par un cerclage noir.

Exemple : spécifications xGDL Dans le cas du LGS, on considère la spécification xGDL suivante : $(a_l; a_b)*\|(f_1 \Box f_2)$. En langage naturel, il s'agit d'une alternance possiblement infinie entre lever et baisser le train d'atterrissage (spécifiée par une séquence répétée de manière non-bornée) avec une faute injectée arbitrairement. Pour la simplicité de cette illustration, l'injection de faute est modélisée par un choix non-déterministe. Cependant, pour plusieurs fautes parmi un ensemble plus grand, l'opérateur de permutation correspondrait mieux. Ici, $f_1 \Box f_2$ est équivalent à $\{1, 1\} of [f_1, f_2]$ (une et une seule faute parmi l'ensemble $[f_1, f_2]$).

Exemple : génération du NFA La première étape consiste à dériver de ce guide de vérification et de la sémantique de xGDL un NFA correspondant. Voici les états (termes) et transitions obtenus via exploration avec $(a_l; a_b)*$ (pas d'injection de faute) comme état initial :

$$
\begin{array}{rcll}
(a_l; a_b)* & \xrightarrow{\tau} & ((a_l; a_b); (a_l; a_b)*)? & (\text{règle } \textit{star}) \\
((a_l; a_b); (a_l; a_b)*)? & \xrightarrow{\tau} & \bot \Box (a_l; a_b); (a_l; a_b)* & (\text{règle } \textit{opt}) \\
\bot \Box (a_l; a_b); (a_l; a_b)* & \xrightarrow{\tau} & \bot & (\text{règle } \textit{alt}_1) \\
\bot \Box (a_l; a_b); (a_l; a_b)* & \xrightarrow{\tau} & (a_l; a_b); (a_l; a_b)* & (\text{règle } \textit{alt}_2) \\
(a_l; a_b); (a_l; a_b)* & \xrightarrow{a_l} & a_b; (a_l; a_b)* & (\text{règle } \textit{seq}_2) \\
a_b; (a_l; a_b)* & \xrightarrow{a_b} & (a_l; a_b)* & (\text{règle } \textit{seq}_1)
\end{array}
$$

La version graphique est présentée figure 3.2. Sur celle-ci et dans la suite, les transitions étiquetées par τ sont composées de tirets, grisées et n'ont pas de label.

La figure 3.3 correspond au NFA obtenu par l'outillage avec injection d'une faute $((a_l; a_b)* \|(f_1 \Box f_2))$. On peut distinguer six étages de cinq états correspondant respectivement à, de haut en bas : après injection de la faute f_1 (10-14, 1X), avant f_1 (2X), après $f_1 \Box f_2$ (3X), avant $f_1 \Box f_2$ (4X), avant f_2 (5X), après f_2 (6X). Pour chaque étage, les états sont ordonnés vis à vis de la répétition non-bornée $(a_l, a_b)*$ et correspondent respectivement aux sous-termes $(a_l, a_b)*$ (X0), $(a_l; a_b); (a_l, a_b)*$ (X1), $a_b; (a_l, a_b)*$ (X2), $(a_l, a_b)*$ (X3) et $\bot$ (X4).

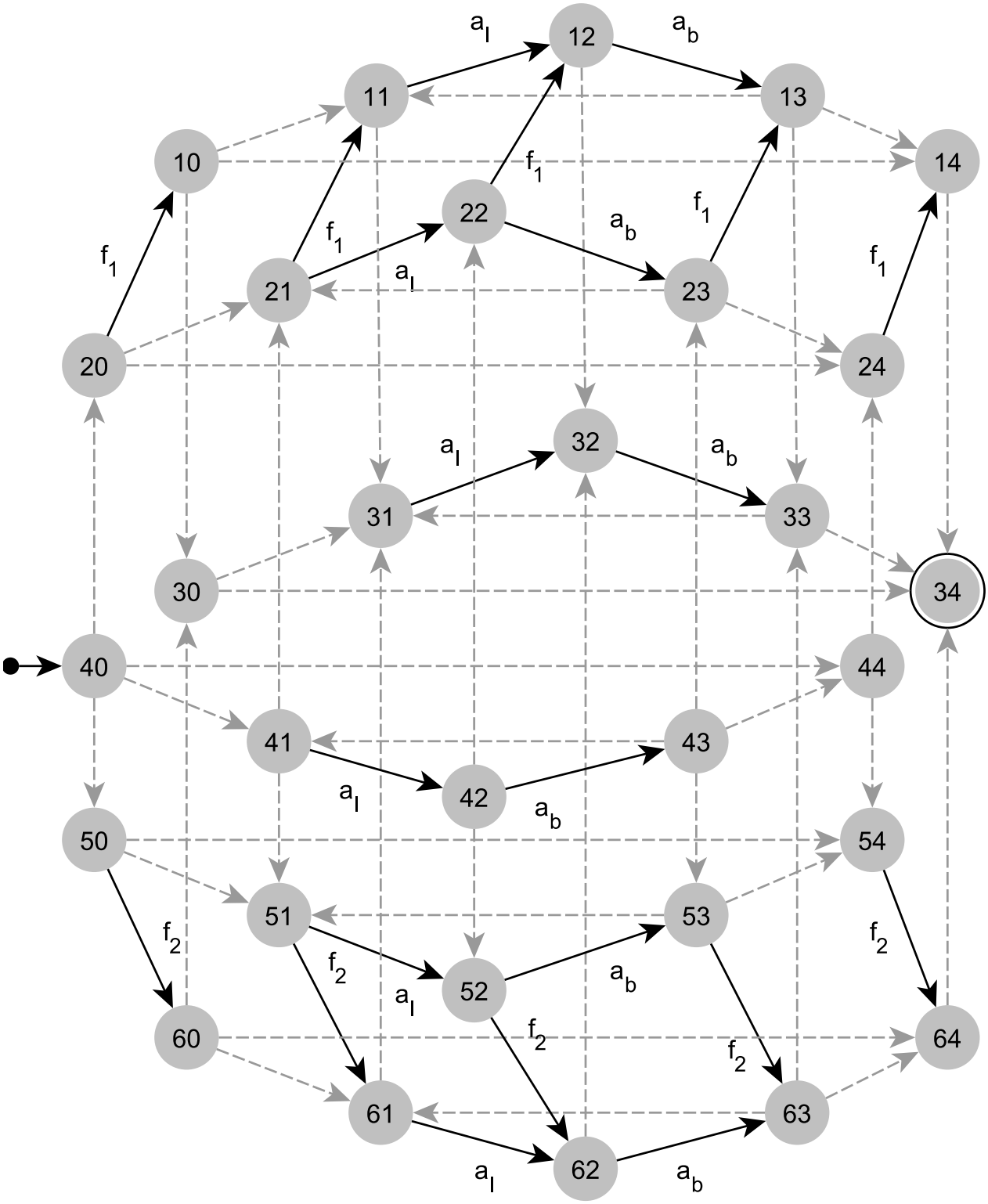

FIGURE 3.3 – $(a_l; a_b) * \|(f_1 \square f_2)$ transformé en NFA.

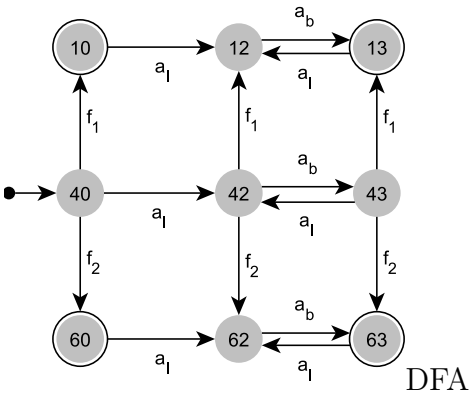

FIGURE 3.4 – $(a_l; a_b) * \|(f_1 \square f_2)$ transformé en DFA.

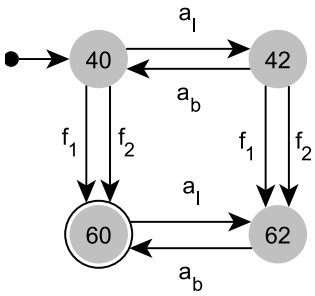

FIGURE 3.5 – $(a_l; a_b) * \|(f_1 \square f_2)$ transformé en DFA minimisé.

Exemple : transformation vers DFA La figure 3.4 présente le résultat de la transformation du NFA figure 3.3 vers un équivalent déterministe. Les labels de ce DFA correspondent à ceux du NFA avant transformation. On peut noter la disparition des transitions étiquetées par τ et l'équivalence de chemin entre les états conservés sur les deux automates. Ainsi qu'il existe au moins un chemin exclusivement étiqueté par τ vers un état final dans le NFA (34) pour tous et seulement les états finaux dans le DFA (10, 13, 60 et 63. Ex : $10 \xrightarrow{\tau} 14 \xrightarrow{\tau} 34$).

Exemple : minimisation du DFA La figure 3.5 présente le résultat de la minimisation du DFA figure 3.4. On peut constater que cette nouvelle et dernière étape de la compilation XGDL obtenu après génération d'un NFA, transformation vers DFA et minimisation correspond bien à la spécification en entrée : $(a_l; a_b) * \|(f_1 \square f_2)$.

3.3.2 Dépliage borné

Dans la suite de ce document, certains algorithmes présentés se basent sur l'hypothèse que le guide de vérification est **acyclique**. Le formalisme XGDL introduit section 3.2 autorise la définition de spécifications cycliques via les opérateurs de réplication non-bornées ($*$ et $+$).

L'approche présentée dans cette section opère sur la forme compilée d'un guide de vérifi-

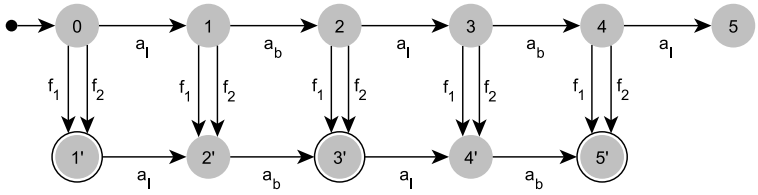

FIGURE 3.6 – $(a_l; a_b) * \|(f_1 \square f_2)$ compilé et déplié pour une borne à 5.

cation XGDL et la transforme dans une forme acyclique selon une borne spécifiée en entrée.

Algorithmie Le dépliage est effectué en deux phases : le dépliage proprement dit et la minimisation du résultat. Cette dernière est identique à celle présentée section 3.3.1, paragraphe correspondant. La première construit un DFA acyclique à partir de celui obtenu par compilation et d'une borne n donnée. Le résultat reconnaît les termes de longueur inférieure ou égale à n.

Exemple Dans le cas du LGS et de la spécification XGDL $(a_l; a_b) * \|(f_1 \square f_2)$, le DFA cyclique obtenu par compilation est présenté section 3.3.1, figure 3.5. L'automate acyclique résultat d'un dépliage pour une borne à 5 est présenté figure 3.6. Les nœuds sont étiquetés selon la longueur des termes reconnus. X' dénote un terme contenant une injection de faute.

On peut noter la présence du nœud 5 lequel ne ferait pas de sens dans le cas des expressions régulières (pas d'état d'acceptation atteignable). Cependant, comme déjà mentionné sections 3.2.2 et 3.3.1, la sémantique de XGDL est fermée par préfixe. Ici, le terme $a_l; a_b; a_l; a_b; a_l$ est un bien un terme de longueur 5 accepté du fait qu'il s'agit d'un préfixe de, entre autres, $a_l; a_b; a_l; a_b; a_l; f_1; a_b$.

3.4 Algorithme de vérification guidé par le contexte

L'algorithme Past-Free[ze] présenté ici est une spécialisation de l'algorithme d'exploration basée sur le fait que les sommets d'un graphe dirigé acyclique (DAG) peuvent être ordonnés en fonction de leur précédence. Il permet de libérer la mémoire de partitions sur l'espace d'état (indexées par les sommets du DAG) pendant l'exploration.

Cette section est basée sur les sections 2 et 3 de l'article "*Past-Free[ze] reachability analysis : reaching further with dag-directed exhaustive state-space analysis*", lequel a été publié dans *Software Testing, Verification and Reliability* en 2016 [34]. Il s'agit d'une traduction dont certaines parties ont été clarifiées dont, en particulier, la formalisation de l'algorithme et les métriques proposées.

L'intuition et les concepts nécessaires à sa transmission sont présentés section 3.4.1, la formalisation de l'algorithme section 3.4.2.

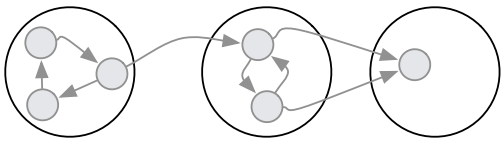

FIGURE 3.7 – Partitions induites par l'acyclicité du DAG

3.4.1 Concepts et intuition

DAG L'algorithme suppose donc un guide de vérification dont la forme compilée est un graphe dirigé acyclique (DAG), c'est à dire décrit sans les opérateurs de réplications non-bornées $*$ et $+$. Si cette contrainte d'acyclicité n'est pas respectée il est possible, en amont, d'utiliser l'algorithme de dépliage borné présenté section 3.3.2 pour y remédier.

Précédence La précédence est une relation d'ordre partiel sur les sommets d'un graphe. u précède v si il existe une transition du sommet u vers le sommet v ($u \to v$). Par transitivité, u précède v si il existe un chemin de u vers v. Ou, en d'autres mots, u précède v si et seulement si u est un parent de v.

Partitions La composition d'un système arbitraire avec un DAG induit des partitions, indexées par les sommets de ce dernier, respectant la structure du DAG. Acyclicité comprise. La figure 3.7 présente un exemple d'un tel partitionnement.

En d'autres mots, la relation de précédence sur les sommets du DAG est conservée par le graphe d'état obtenu par composition avec le système à vérifier. Cela implique, entre autres, que si u précède v alors il n'existe pas de chemin d'un état de la composition indexé par v vers un état indexé par u. Et donc, si l'ensemble des états restant à explorer sont indexés par v, les états connus et conservés en mémoire indexés par u ne seront plus atteints et peuvent être *oubliés*.

Tri topologique Pour permettre d'exploiter cette observation durant l'exploration, l'algorithme Past-Free[ze] :

— ordonne les sommets du DAG en fonction de la relation de précédence ;

— décompose l'espace d'état en partitions indexées par les sommets du DAG.

Un ordonnancement sur les sommets respectant la relation de précédence est communément appelé *tri topologique* [15]. Figure 3.8, l'ordonnancement présenté item *(b)* est un tri topologique des sommets du DAG en exemple item *(a)*. Un tel ordonnancement peut être construit dans un temps linéaire en fonction de la taille du DAG.

Relation d'ordre La relation d'ordre totale dérivée du tri topologique S_a retenu est notée $<$ par la suite. Par définition $u < v$ si et seulement si le sommet u est ordonné avant le sommet v dans S_a. On peut noter que :

— u précède $v \Rightarrow u < v$;

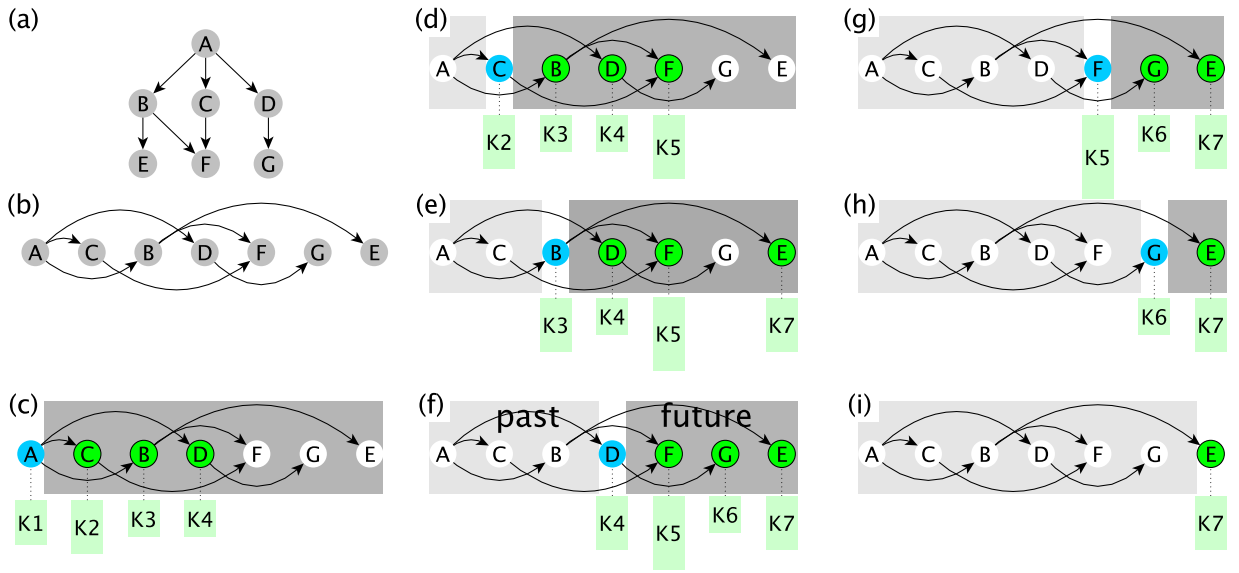

FIGURE 3.8 – Illustration de l'algorithme d'exploration Past-Free[ze].

— $u < v \Rightarrow v$ ne précède pas u

— $u < v$ n'implique pas nécessairement que u précède v

.

Past-Free[ze] L'algorithme d'exploration Past-Free[ze] se base sur ce tri topologique pour *oublier le passé*. C'est à dire libérer la mémoire des partitions déjà explorées et indexées par les sommets du DAG qui précédent ceux restant. Il est important de noter que un état atteignable à partir d'un état appartenant a une partition indexée par un sommet du DAG u appartient soit :

— à la même partition indexée par u ;

— à une autre partition indexée par un sommet v tel que u précède v et donc $u < v$.

Il est donc possible d'explorer *entièrement* une partition, c'est à dire ne considérer que des états appartenant à cette partition pour l'exploration, avant de passer à la suivante.

Illustration La figure 3.8 présente une illustration du déroulement de l'algorithme. Sur celle ci :

— item *(a)* : le DAG a composer avec le système ;

— item *(b)* : tri topologique des sommets du DAG, les liens entre les sommets illustrant la préservation de la relation de précédence ;

— items *(c)* à *(i)* : les étapes successives de l'exploration, les nœuds représentant les partitions d'états de la composition indexées par les sommets du DAG.

Et pour chaque étape de l'exploration (items *(c)* à *(i)*) :

— le nœud bleu est la partition actuellement explorée ;

— les nœuds blancs dénotent celles *oubliées* ou non encore découvertes ;

— les nœuds verts correspondent à celles *futures* dont certains états ont été atteints ;

— Les états atteints et conservés en mémoire sont représentés par les rectangles K1-K7.

Il est à noter que le tri topologique retenu parmi ceux possibles impacte grandement les performances de l'exploration (voir section 3.4.3).

3.4.2 Formalisation

Notations Dans la suite, $\langle s, u\rangle$ dénote un état de la composition d'un système avec un DAG, s correspondant à l'état du système et u au sommet du DAG. S_a correspond au tri topologique retenu, et $<$ à la relation d'ordre totale dérivée de S_a. S_a est aussi utilisé pour dénoter l'ensemble des sommets du DAG, en particulier $|S_a|$ correspond au nombre de sommets.

Retour vers le futur L'algorithme Past-Free[ze] est basée sur l'observation que si l'exploration de la composition d'un système avec un DAG traite les états atteints dans un ordre respectant un tri topologique sur les sommets de ce DAG, alors les états atteints *dans le futur* ne sont jamais indexés par un sommet du DAG ordonné avant ceux explorés *dans le passé*. Cela est capturé par le théorème 1.

Théorème 1. *Soit S_a un tri topologique sur les sommets d'un DAG donné et $<$ la relation d'ordre totale dérivée. Si l'algorithme d'exploration traite les états de la composition $\langle -, -\rangle \in \boldsymbol{\mathcal{S}}$ d'un système arbitraire et de ce DAG dans un ordre respectant ce tri ; alors, lorsque tous les états $\langle -, u\rangle, \forall u \in S_a$ ont été explorés, l'exploration ne peut atteindre un état $\langle -, v\rangle$ avec $v < u$.*

Démonstration. **Par contradiction, on suppose qu'il existe** u et v deux sommets du DAG tel que $v < u$ et qu'un état $\langle -, v\rangle$ est atteint après que l'ensemble des $\langle -, u\rangle$ aient été explorés. Alors :

– **soit il existe un chemin dans le DAG** de u à v, c'est à dire que u précède v et donc que $u < v$ **[contradiction]** ;

– **soit il existe un parent** p de v tel que l'ensemble des $\langle -, p\rangle$ est entièrement exploré après l'ensemble des $\langle -, u\rangle$, c'est à dire que p est ordonné après u dans S_a, équivalent par définition à $u < p$; comme p est un parent de v on a aussi $p < v$ et, par transitivité, $u < v$ **[contradiction]**. □

Algorithm 2 Past-Free[ze] context-aware reachability analysis algorithm

```
function PAST-FREE[ZE]REACHABILITY(Sys, DAG)
    let S_a ← all vertices of DAG sorted in topological order
    // initialize c, the current vertice, to DAG's root
    let c ← first(S_a)
    // initialize K_u (∀u ∈ S_a), clusters of reachable, discovered states
    for all u ∈ S_a do
        let K_u ← ∅
    end for
    // initialize Q, the set of discovered states yet to be explored
    let Q ← ∅
    for all s_0 ∈ initial(Sys) do
        Q ← Q ∪ {⟨s_0, c⟩}
        K_0 ← K_c ∪ {⟨s_0, c⟩}
    end for
    // explore reachable states of Sys‖DAG
    while Q ≠ ∅ do
        // get a minimal state ⟨s, u⟩ ∈ Q and remove it from Q
        let ⟨s, u⟩ ← minimal(Q, S_a)
        Q ← Q \ {⟨s, u⟩}
        // check if the algorithm moved to a different cluster
        if c ≠ u then
            // save the previous cluster to disk and free it from memory
            IO.saveAll(K_c)
            K_c ← ∅
            // update c accordingly
            c ← u
        end if
        // explore outgoing transitions in Sys‖DAG from ⟨s, u⟩
        let R ← all ⟨t, v⟩ such that ⟨s, u⟩ → ⟨t, v⟩ ∈ Sys‖DAG
        for all ⟨t, v⟩ ∈ R such that ⟨t, v⟩ ∉ K_v do
            Q ← Q ∪ {⟨t, v⟩}
            K_v ← K_v ∪ {⟨t, v⟩}
        end for
    end while
    // end the exploration
    IO.saveAll(K_c)
    return IO.getReferenceToResults
end function
```

Algorithmie L'algorithme, présenté par le pseudo-code Algorithm 2, prend un système Sys et un DAG (le guide de vérification) en entrées pour en explorer exhaustivement leur composition. Pour permettre l'exploitation du théorème 1 et donc l'oubli à la volée de partitions d'états déjà explorés, l'exploration respecte l'ordre défini par le tri topologique S_a sur les sommet du DAG.

Phase d'initialisation :

— $\mathbf{S_a}$: l'ensemble ordonné (par tri topologique) des sommets du DAG est calculé. Dans la suite, $u < v$ est équivalent à $indexOf(u, S_a) < indexOf(v, S_a)$.

— $\mathbf{c}$: le sommet *courant* est initialisé à la racine du DAG, c'est à dire le premier sommet de S_a par construction ($c \leftarrow first(S_a)$). Jusqu'à la fin de cette phase d'initialisation, cette variable est utilisée pour dénoter le sommet initial du DAG.

— $\mathbf{K_u}$: l'ensemble des états atteints par l'exploration est découpé en $|S_a|$ partitions indexées par les sommets du DAG ($\forall u \in S_a$) tel que $\langle -, u\rangle \in K_v \Rightarrow u = v$. Ces ensembles commencent vides ($K_u \leftarrow \emptyset$).

— $\mathbf{Q}$: l'ensemble des états de la composition découverts mais non explorés est initialisé par l'ensemble des états initiaux. C'est à dire l'ensemble des $\langle s_0, c\rangle$ où s_0 est un état initial de Sys ($s_0 \in initial(Sys)$). Ces états sont aussi rajoutés à K_c, la partition indexée par le sommet initial du DAG.

Phase d'exploration (itérations tant que $Q \neq \emptyset$) :

— **Nouvel état courant** $\langle \mathbf{s}, \mathbf{u}\rangle$: chaque itération correspond à l'exploration d'un état $\langle s, u\rangle \in Q$ *minimal* par rapport au tri topologique S_a, c'est à dire tel que $\forall \langle -, v\rangle \in Q, u < v \vee u = v$. Un état parmi ceux possibles est décidé par l'appel de fonction $minimal(Q, S_a)$. Ainsi, $\forall v \in S_a$ tel que $u < v$, l'ensemble des $\langle -, u\rangle$ seront explorés avant les $\langle -, v\rangle$; respectant ainsi le prérequis du théorème 1. Ce nouvel état courant est retiré de Q.

— **Finalisation d'une partition si** $\mathbf{c \neq u}$: tous les $\langle -, c\rangle$ atteignables ont été explorés, la partition K_c est *finalisée*, ne peut plus être atteinte (produit du théorème 1) et donc peut être stockée sur disque et *oubliée* ($K_c \leftarrow \emptyset$). c doit être mis à jour pour les prochaines itérations ($c \leftarrow i$). On peut noter que $c < i$ (avant mise à jour), et donc c ne peut que *avancer* dans S_a.

— **Pas d'exploration** $\langle \mathbf{t}, \mathbf{v}\rangle$: les *nouveaux* ($\langle t, v\rangle \notin K_v$) états atteignables à partir de $\langle s, u\rangle$ *directement* ($\exists \langle s, u\rangle \rightarrow \langle t, v\rangle \in Sys \| DAG$) sont rajoutés à Q et aux partitions K_v correspondantes. Dénotés atteints par l'ajout dans K_v, ces états ne peuvent donc être découverts (ajout dans Q) de nouveau. L'ensemble des états atteignables étant fini, la terminaison est donc assurée.

Illustration Les items de *(c)* à *(i)* sur la figure 3.8 illustrent le déroulement de l'algorithme sur un exemple présenté items *(a)* (DAG) et *(b)* (tri topologique).

A l'initialisation : $S_a \leftarrow \{A, C, B, D, F, G, E\}$, $c \leftarrow A$, les états initiaux de la composition $\langle s_0, A\rangle$ sont ajoutés à Q et K_A.

Item *(c)*, dans les premières itérations (jusqu'à $c \neq A$), seuls les états $\langle -, A\rangle$ sont explorés (partition dénotée en bleu), c'est à dire retirés de Q. Les états atteignables à partir de ces états sont ajoutés à Q (découvert) et aux K_u correspondants (atteint). Du sommet A, seuls les sommets A, B, C et D sont atteignables (partitions dénotées en vert).

Le passage de l'item *(c)* à l'item *(d)* correspond à $c \leftarrow C$ (nouveau sommet courant, vert $\rightarrow$ bleu). C'est à dire que tous les $\langle -, A\rangle$ ont été explorés, la partition correspondante est donc finalisée et peut être oubliée (bleu $\rightarrow$ blanc). Le sommet F est maintenant atteignable (blanc $\rightarrow$ vert).

Et ainsi de suite jusqu'à la finalisation de la dernière partition

Génération du potentiel contre-exemple Libérer la mémoire d'états qui ne peuvent plus être atteints permet d'alléger la charge mémoire durant l'exploration et donc de repousser le problème de l'explosion combinatoire. Cependant, si la vérification associée échoue (violation d'une propriété) il n'est plus possible de construire un contre-exemple (un chemin illustrant la mise en défaut). Pour remédier à ce problème, une possibilité est de stocker *sur disque* les partitions *oubliées* durant l'exploration.

En pratique, les vérifications de large taille peuvent être exécutées sans y recourir (l'écriture sur disque ralentie notablement le processus) et relancées le cas échéant. Dans ce dernier cas, il est possible de ne considérer que les chemins du DAG représentant l'environnement menant à l'état redouté.

3.4.3 Métriques

Pour un DAG donné, cette section propose des métriques pour en caractériser les sommets en fonction du tri topologique retenu (métriques locales aux sommets) et pour permettre de comparer différents tris possibles (métriques sur le tri topologique).

Passé, présent, futur Pour un DAG et un tri topologique S_a donnés, il est possible d'inférer, lorsque la partition courante est indexée par $u \in S_a$ ($c = u$ dans l'algorithme 2), $P_u(S_a)$ les sommets appartenant au **p**assé (partitions oubliées), $A_u(S_a)$ les sommets présentement **a**tteignables (partitions potentiellement non-vides) et $F_u(S_a)$ les sommets appartenant au **f**utur (partitions non-atteintes) ; c'est à dire respectivement :

— $P_u(S_a) : \{v \in S_a | v < u\}$, l'ensemble des sommets appartenant au passé de u dont les partitions correspondantes ont sont explorées, finalisées et oubliées avant celle indexée par u ;

— $A_u(S_a) : \{u\} \cup \{v \in S_a | u < v \wedge \exists p \in S_a, (p = u \vee p \in P_u) \wedge p \rightarrow v \in DAG\}$, le sommet u et l'ensemble des sommets après u dans S_a ($u < v$) atteignables directement de u ou d'un sommet atteint avant u, c'est à dire l'ensemble des sommets indexant des partitions potentiellement en mémoire (découvertes mais non finalisées) ;

— $F_u(S_a) : \{v \in S_a | v \notin P_u \wedge v \notin A_u\}$; l'ensemble des sommets non-atteints avant u, différents de u et non-atteignables directement de u ou d'un sommet atteint avant u, c'est à dire l'ensemble des sommet indexant des partitions non découvertes.

Dans la suite, le paramètre S_a est omis lorsqu'il n'y a pas d'ambiguïté possible.

Par construction, pour un DAG et un tri topologique sur les sommets de ce DAG S_a donnés, $\forall u \in S_a$:

— P_u, A_u et F_u disjoints ;

— $S_a = P_u \cup A_u \cup F_u$;

— $\Rightarrow |S_a| = |P_u| + |A_u| + |F_u|$.

Illustration Sur les différents items de la figure 3.8 :

— u correspond au sommet bleu ;

— P_u correspond aux sommets à gauche de u (blancs) ;

— A_u correspond aux sommets bleus et verts ;

— F_u correspond aux sommets blancs à droite de u.

Métriques locales aux sommets Les tailles de ces ensembles de sommets sur nous donne diverses informations sur l'exploration lorsque la partition courante est indexée par $u \in S_a$ ($c = u$ dans l'algorithme 2) :

— $|\mathbf{P_u}|$, le nombre de partitions oubliées (découvertes et finalisées) ;

— $|\mathbf{A_u}|$, le nombre de partitions potentiellement en mémoire (découvertes mais non finalisées) ;

— $|\mathbf{F_u}|$, le nombre de partitions non atteintes (non découvertes).

On peut noter que, pour un index i dans S_a et u le sommet à cet index, $|P_u|$ étant égal au nombre de sommet *à gauche* de u par construction, on a $|P_u| = i$ et $|A_u| + |F_u| = |S_a| - i$ (en remplaçant $|P_u|$ par i dans $|S_a| = |P_u| + |A_u| + |F_u|$).

Et donc, pour un DAG et un index i donnés, pour tous les tris topologiques S_a possibles sur ce DAG, avec u le sommet à l'index i dans S_a, $|P_u|$ (*à gauche de u exclus*) et $|A_u| + |F_u|$ (*à droite de u inclus*) sont des invariants.

Métriques sur le tri topologique Le problème adressé par l'approche proposée est celui de l'explosion combinatoire inhérent au modèle-checking. C'est à dire que l'on cherche à réduire la charge mémoire maximum de l'exploration.

Dans la suite, on suppose que tous les sommets du DAG sont atteignables par composition avec le système et que les tailles des partitions obtenues lors de l'exploration sont similaires.

Alors, pour un S_a donné, le **maximum des** $|\mathbf{A_u}(\mathbf{S_a})|$ correspond au nombre maximum de partitions conservées en mémoire lors de l'exploration guidée par Past-Free[ze], algorithme 2. Pour un DAG donné, cette métrique sur ses tris topologiques peut être considérée comme une indication de leurs *qualités*.

Pour la lisibilité des résultats et pour permettre la comparaison des résultats obtenus sur des DAGs différents, on divise cette valeur par le nombre de sommets du DAG. Le résultat

représente ainsi le **ratio maximum de partitions en mémoire sur le nombre total de partitions**, noté $\mathbf{C_{max}}$ par la suite :

$$C_{max}(S_a) = \frac{Max_{u \in S_a}|A_u(S_a)|}{|S_a|}$$

Toujours sous l'hypothèse que l'ensemble des partitions sont atteignables par composition avec le système et que leurs tailles (nombre d'états) sont similaires, C_{max} est une **sur-approximation de la charge mémoire maximale**.

Dans le cas *au pire*, si tous les sommets du DAG sont atteignables de la racine (ou dans le cas d'une exploration non guidée *classique*), $C_{max} = 1$.

Dans le meilleur cas, si le DAG n'est composé que d'un chemin entièrement linéaire, il n'existe qu'un tri topologique possible et il n'est nécessaire que de conserver deux partitions en mémoire à tout point de l'exploration ($Max_{u \in S_a}|A_u| = 2$). On a alors $C_{max} = \frac{2}{|S_a|}$.

Intuitivement, étant donnés S_a et S'_a deux tris topologiques sur le même DAG, S_a peut être considéré *meilleur* que S'_a si $C_{max}(S_a) < C_{max}(S'_a)$. C'est à dire moins de partitions à conserver en mémoire au maximum pour l'exploration du même espace d'états.

Illustration Sur la figure 3.8, le tri topologique est présenté item *(b)* et correspond à $\mathbf{S_a} = \{\mathbf{A}, \mathbf{C}, \mathbf{B}, \mathbf{D}, \mathbf{F}, \mathbf{G}, \mathbf{E}\}$. C_{max} est atteint aux étapes *(c)*, *(d)*, *(e)* et *(f)* avec 4 sommets dont les partitions sont découvertes mais non finalisées (sommets bleus et verts), c'est à dire $\mathbf{C_{max}} = \mathbf{4/7}$.

Si on considère $\mathbf{S'_a} = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}\}$, un autre tri topologique possible, lors de l'exploration des partition indexées par B et C il est nécessaire de conserver 5 partitions en mémoire ($A_B(S'_a) = \{B, E, F, C, D\}$), c'est à dire $\mathbf{C_{max}} = \mathbf{5/7}$.

Il est donc *préférable* d'ordonner l'exploration des partitions en se basant sur S_a au lieu de S'_a.

3.5 Décomposition récursive de l'espace d'états

Diviser pour régner La technique, nommée Split [17, 19, 16], présentée schématiquement figure 3.9, divise systématiquement et récursivement le graphe dirigé acyclique (DAG) correspondant à la forme compilée d'un guide de vérification acyclique lorsqu'une exploration échoue due à l'explosion combinatoire.

Les nouveaux DAGs générés après division sont successivement composés avec le système et les propriétés à vérifier. La vérification initiale est ainsi décomposée en plusieurs sous-problèmes. Mener la vérification sur ce nouvel ensemble est équivalent à la mener sur le problème initial[19].

Illustration La figure 3.9 présente un déroulement possible de l'algorithme avec un DAG similaire à celui présenté figure 3.8, item *a*. En partie haute de la figure, l'exploration de la composition initiale du système à vérifier (SUS) et de son guide de vérification compilé *explose*. Trois nouveaux DAGs sont alors générés par division de celui initial, lesquels sont

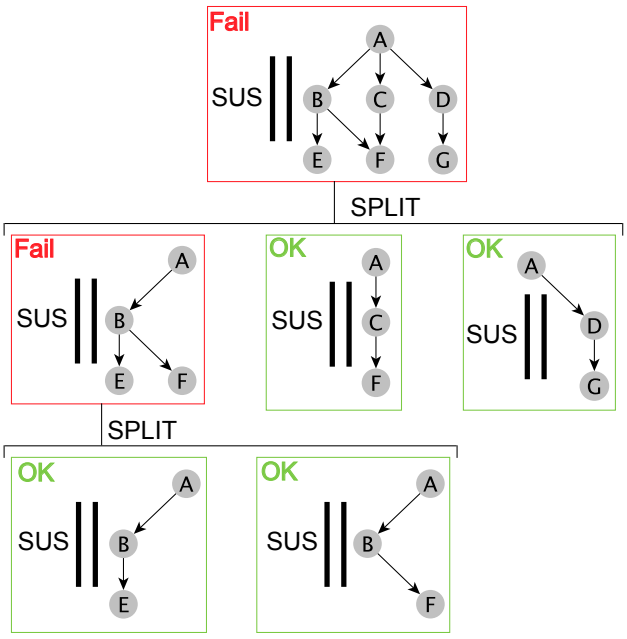

FIGURE 3.9 – Division récursive (Split) d'une vérification

successivement composé avec le système. Parmi ceux-ci, la première composition échoue et le DAG correspondant est de nouveau divisé.

On peut noter que à chaque étage, ou étape, les nouveaux DAGs sont *plus linéaires* que le parent. De ce fait, ils se prêtent d'autant mieux à une utilisation conjointe de l'algorithme Past-Free[ze] introduit section 3.4.

Paramétrage L'algorithme peut être paramétré de diverses manières. Les expérimentations menées [17, 19, 16] opèrent sur la forme compilée du guide de vérification et permet de spécifier une profondeur initiale et l'incrément à chaque itération. Dans l'exemple présenté figure 3.9 ces deux valeurs sont à 1.

D'autres approches sont possibles comme la prise en compte spécialisée de certains opérateurs XGDL en amont de la compilation. Pour l'exemple, dans le cas de l'opérateur de permutation et d'une injection de fautes $\{n, n\}of[f_1, ..., f_m]$, le guide initial peut être divisé pour chaque sous-partie de taille $m-1$ de l'ensemble $[f_1, ..., f_m]$; puis par récursion, dans les cas explosifs, pour chaque sous-partie de taille $m-2$ et ainsi de suite jusqu'à n. On peut noter que, dans ce cas, le guide peut avoir des composantes cycliques (hors celle contenant la permutation).

Compromis espace-temps Bien qu'ayant de bonnes propriétés pour contourner l'explosion combinatoire, par définition le Split induit la nécessité de conduire plusieurs vérifications. De plus, dans bien des cas, les ensembles d'états explorés sont partiellement redondants. Cela induit un processus de vérification relativement coûteux en terme de temps.

Cependant, en pratique ce compromis entre la mémoire et le temps nécessaire à une vérification est acceptable [19, 7, 33, 32] lorsque qu'il permet la terminaison du processus.

De plus, les unités de vérifications générées étant indépendantes de nature, le temps requis peut être considérablement réduits en distribuant ces tâches sur un réseau de calculateurs.

3.6 Liens avec le model-checking borné

Les algorithmes présentés section 3.4 et 3.5 font l'hypothèse qu'*il est possible d'exprimer l'environnement sous la forme d'un guide de vérification acyclique* (pas de réplication non-bornée xGDL $*$ ou $+$).

Cela implique que les chemins dans le graphe obtenu par compilation soient d'une longueur bornée. L'exploration d'un système à vérifier composé avec un tel guide ne garantie pas l'exhaustivité. Il est possible que certains états atteignables par des chemins plus longs que la borne retenue pour l'environnement échappent au processus de vérification.

Seuil d'exhaustivité Cette problématique est similaire à celle inhérente au BMC (Bounded Model-Checking [11], voir section 2.4). Cette technique consiste à borner la profondeur de l'exploration du système pour réduire l'espace d'états et en éviter l'explosion.

Comme évoqué en fin de section 2.4, garantir la validité du processus de vérification via BMC nécessite de démontrer que la borne retenue est supérieure ou égale au seuil d'exhaustivité [27].

Application à la CaV Dans le cas de la CaV et de xGDL, il s'agit de démontrer que la borne retenue pour la modélisation de l'environnement permet l'exploration du système jusqu'à une profondeur au moins égale au seuil d'exhaustivité.

Ce seuil portant sur le système fermé, si les interactions de l'environnement peuvent se succéder sans réaction du système ouvert (le *pire* des cas), la borne retenue pour le dépliage du guide xGDL ne peut être inférieure au seuil d'exhaustivité.

Si le seuil est connu, dans le cas d'une modélisation xGDL cyclique, il devient possible de déplier le DFA obtenu par compilation (voir section 3.3.2) jusqu'à la borne nécessaire et suffisante tout en conservant la garantie de l'exhaustivité du processus de vérification.

Des cas particuliers On peut noter que explorer le système jusqu'à une profondeur n revient à attribuer pour toutes les transitions la même étiquette a, et à explorer le dépliage borné de la composition avec le guide xGDL $a*$ (ou, de manière équivalente, la composition avec le guide xGDL $a\{n\}$).

En d'autres mots, le BMC peut être vu comme un cas particulier de la CaV.

Similairement, le model-checking borné partiellement revient à n'étiqueter que les transitions du système que l'on souhaite considérer pour l'étude.

Inversement, en proposant une définition de la profondeur basée sur le guide xGDL, la composition du système avec le dépliage borné de ce guide peut être vue comme un cas particulier du model-checking borné.

3.7 Conclusion

Section 2.5, nous avons identifié des limites à la mise-en-œuvre de la CaV avec CDL : manque d'une modélisation exhaustive des comportements de l'environnement en entrée, composition asynchrone à travers variables partagées uniquement, modélisation acyclique de l'environnement.

Ce chapitre a adressé ces points en généralisant les concepts mis en jeu par la CaV et en a proposé une mise-en-œuvre via un nouveau formalisme : xGDL.

Celui-ci a été présenté sections 3.2 et 3.3.1. Plutôt que de séparer explicitement le système ouvert de son environnement, l'approche avec xGDL se base sur la capacité de distinguer :

— les transitions du système ouvert ;

— les interactions avec l'environnement.

Ainsi, l'approche suppose en entrée le système fermé, c'est à dire une modélisation exhaustive des comportement à vérifier et de l'environnement. Celui-ci peut donc constituer un modèle de référence en amont de la division du problème en unités de preuves (desquelles il est une partie invariante).

De plus, le système exploré est obtenu via une composition synchrone basée sur la notion d'actions observables atomiques. A l'image de CDL ces actions peuvent être des communications asynchrones. xGDL permet cependant d'autres formes d'interactions avec pour seule limite l'expressivité du langage utilisé pour la modélisation du système.

Aussi, la sémantique de xGDL supporte la définition d'environnements cycliques. La section 3.3.2 a proposé un algorithme de dépliage borné pour permettre de revenir a une modélisation acyclique.

Les sections 3.8 et 3.5 ont présenté des algorithmes exploitant les spécificités de l'approche, notablement l'acyclicité de l'environnement.

La section 3.6 a montré des liens entre les problématiques de couverture du dépliage borné présenté ici et du model-checking borné. Il s'agit d'une piste intéressante pour y apporter des éléments de réponse.

Chapitre 4

Validation de l'approche

4.1 Introduction

Cette section illustre l'application de l'approche par CaV avec xGDL présentée chapitre 3 à différents cas d'études.

Comme précisé section 2.2, les techniques axées autour du model-checking (CaV inclue) nécessitent une expertise et un effort particulier. Ce coût se justifie dans le cadre des systèmes embarqués avec un besoin fort de validation (i.e. vies humaines en jeu).

De plus, pour permettre le passage à l'échelle de l'analyse, en pratique la validation via model-checking requière souvent la mise en place d'outils sur mesure, capables d'exploiter les caractéristiques propres au système étudié.

Les cas d'études retenus pour illustrer l'approche sont un train d'atterrissage (LGS, avionique, introduit section 2.6), un pacemaker (santé) et un régulateur de vitesse (CCS, automobile). Les trois sont de dimension industrielle. Le LGS a été proposé comme défi ouvert à l'occasion d'une conférence internationale (ABZ'14 [3, 8]) ; le Pacemaker est l'exemple fil rouge d'un ouvrage sur la vérification de modèles [1, 7].

Cependant, le LGS, le Pacemaker et le CCS présentent des caractéristiques les distinguant clairement les uns des autres. On peut noter que :

— pour le LGS, la cible de la vérification sont les composants logiciels responsables de détecter d'éventuelles fautes et de relayer l'information au pilote ;

— pour le Pacemaker, il s'agit d'un système générique dont le comportement diffère d'une mise en service à l'autre en fonction des paramètres d'initialisation retenus ;

— pour le CCS, la disponibilité du service n'est pas critique pour la sûreté (i.e. il est possible de désengager le système si celui-ci s'écarte de l'usage validé).

On verra que ces différentes spécificités peuvent être exploitées par xGDL pour aider le passage à l'échelle de l'analyse.

Les sections 4.2, 4.3 et 4.4 présentent les études et les résultats obtenus pour, respectivement, le LGS, le Pacemaker et le CCS.

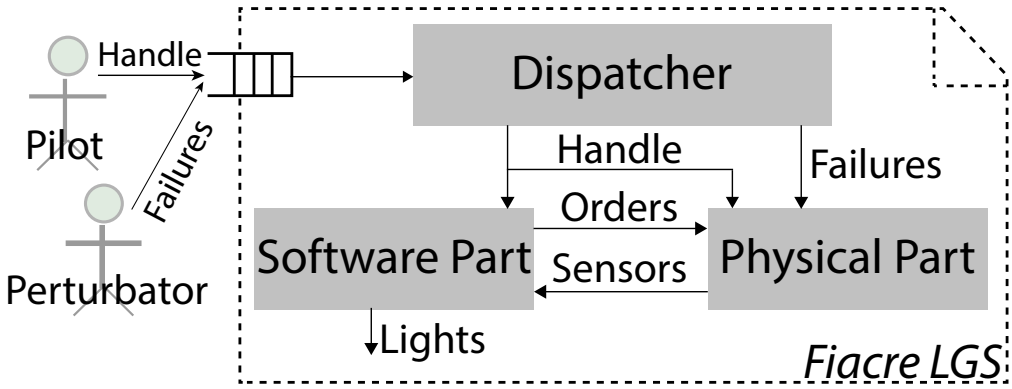

FIGURE 4.1 – Vue globale des composantes du LGS, de son environnement et des communications entre les différentes entités.

4.2 Cas d'étude : train d'atterrissage (LGS - avionique)

Les premiers résultats de la CaV appliquée sur ce cas d'étude (LGS), point d'entrée de la quatrième conférence internationale ABZ (ABZ'14) [8] présenté section 2.6, ont été publiés lors de cette même conférence dans l'article *Context aware verification of a landing gear system* [22] (Philippe Dhaussy et Ciprian Teodorov). Le modèle est exprimé en Fiacre, les guides de vérification (propriétés inclues) en CDL.

L'approche a été depuis été raffinée pour en améliorer les performances, en particulier avec l'introduction de l'algorithme Past-Free[ze] présentée section 3.4. Les nouveaux résultats sur le LGS ont été publiés dans l'article *Environnement-driven Reachability for Timed Systems* [32] (Ciprian Teodorv, Philippe Dhaussy et Luka Le Roux) en 2015 pour *Software Tools for Technology Transfer* (STT'15). Les résultats obtenus, meilleurs de plusieurs ordres de grandeur, s'expliquent aussi par un système revisité (moins de redondance d'états) et une gestion affinée du temps.

Cette section est basée sur les sections 3 et 4 de ce dernier article. Il s'agit d'une traduction dont certaines parties ont été revues voir enrichies pour plus de cohérence et de clarté. En particulier le langage utilisé pour la définition des guides de vérification est remplacé par xGDL (présenté section 3.2).

4.2.1 Modélisation du système

Architecture globale L'architecture retenue pour la modélisation Fiacre du LGS est présentée figure 4.1.

Le **système** est composé de deux parties : celle logicielle (*Software Part*) et celle physique (*Physical Part*). Les communications entre ces deux composants sont considérées synchrones (via les canaux *Sensors* et *Orders*).

L'**environnement** du LGS est composé de deux acteurs : le pilote et l'entité chargée de l'injection faute (nommée *Perturbator* sur les figures et *perturbateur* par la suite).

Le diagramme de séquence présenté figure 4.2 correspond à un scénario dans lequel le pi-

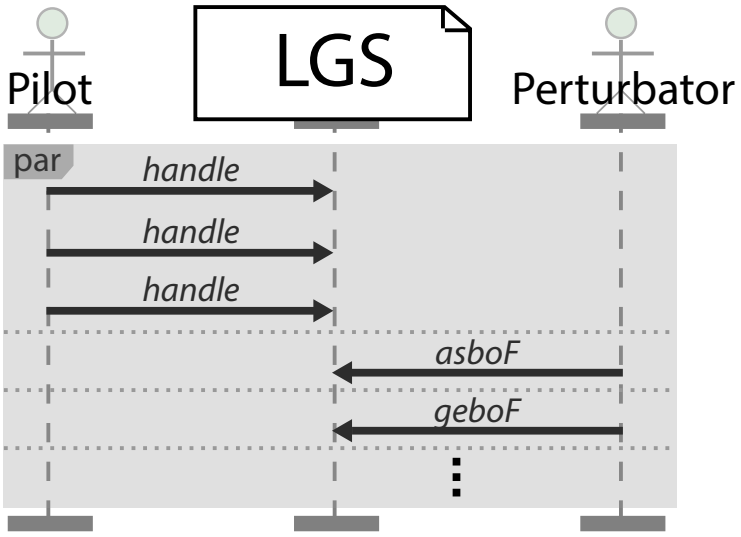

FIGURE 4.2 – Un scénario d'interactions entre le système et sont environnement (inclus dans un bloc parallèle et donc entrelacés)

lote actionne le train atterrissage trois fois et le *perturbateur* injecte deux fautes. Ces messages étant inclus dans un bloc parallèle, le scénario en représente l'ensemble des entrelacements possibles (similaire à la spécification XGDL (*handle*; *handle*; *handle*) | (*asboF*; *qeboF*)).

Cette définition diffère des exemples présentés chapitre 3, le message *handle* correspondant ici alternativement à l'abaissement (a_b précédemment) et à la levée (a_l) du train d'atterrissage.

De retour à la figure 4.1, l'environnement communique avec une entité chargée de relayer l'information (le *Dispatcher*). En aval de cette entité, les communications sont synchrones à travers les canaux *Handle* et*Failures*.

Enfin, les indicateurs lumineux permettant d'informer le pilote d'un éventuel dysfonctionnement sont modélisées par des variables globales maintenues par la partie logicielle (canal *Lights*).

Partie physique Les composantes mécaniques et hydrauliques du système sont modélisés par cinq processus Fiacre différents dont certains sont instanciés plusieurs fois pour un total de douze comportements parallèles : *a*) *Analog Switch*, le commutateur analogique ; *b*) *General_EV*, l'électrovanne générale ; *c*) *Generic_EV*, une électrovanne ; *d*) *Gear*, un train ; *e*) *Door*, une porte.

Chaque processus Fiacre peut être vu comme un automate. La table 4.1 (haut) donne le nombre d'états définis par les automates et le nombre d'instance requis par cette partie physique.

La figure 4.3 illustre l'automate (de 18 états) défini par le processus Fiacre correspondant

TABLE 4.1 – Processus Fiacre pour la partie physique (haut) et la partie logicielle (bas)

	Analog Switch	General_EV	Generic_EV	Gear	Door
# of states	18	34	24	23	20
# of instances	1	1	4	3	3

	Door synth.	Gear synth.	EV Manager	Status Manager
# of states	8	8	52	10
# of instances	3	3	1	1

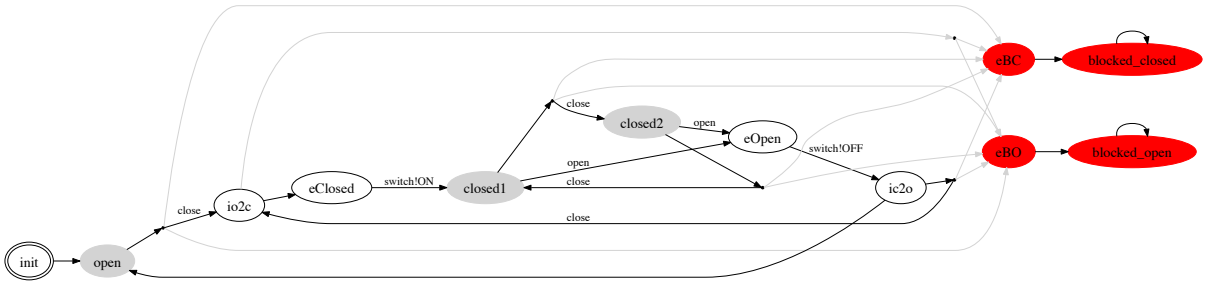

FIGURE 4.3 – Automate correspondant au processus Fiacre *Analog Switch*

au commutateur analogique. En fonctionnement nominal, l'automate passe de l'état **open** à l'état **closed** et inversement. Les états intermédiaires permettent de modéliser les délais imposés par la spécification. L'automate possède deux états finaux, **blockedOpen** et **blockedClosed**, modélisant des fautes pouvant être injectées à tous moment et donc toujours atteignables en fonctionnement nominal.

Partie logicielle De manière similaires, les composants logicielles sont modélisés par quatre processus Fiacre dont certains sont instanciés plusieurs fois : *a*) *Door sensor synthesis*, infère l'état actuel d'une porte en fonction des signaux émis par les senseurs ; *b*) *Gear sensor synthesis*, infère l'état actuel d'un train en fonction des signaux émis par les senseurs ; *c*) *EV Manager*, supervise les séquences d'extension et de rétraction du train d'atterrissage ; *d*) *Status Manager*, maintient à jour les variables globales modélisant les signaux lumineux.

La table 4.1 (bas) donne le nombre d'états définis par les automates et le nombre d'instance requis par cette partie logicielle.

Environnement Comme illustré figure 4.1, l'environnement du LGS est composé de deux acteurs : le pilote (manipulation du levier de commande) et le perturbateur (injections de fautes).

Ceux-ci sont définis en Fiacre par des processus communicant avec le *dispatcher* via une file de message (FIFO, asynchrone).

L'automate du **pilote** ne possède qu'un état et une seule transition. Celle-ci envoie le message *handle* (inverse la position du levier commandant le train atterrissage) au *dispatcher* et boucle dans l'état.

Similairement, l'automate du **perturbateur** ne possède qu'un seul état mais propose autant de transitions que de fautes possibles. Toutes envoient le message correspondant à la faute à injecter au dispatcher et bouclent dans l'état.

Le modèle Fiacre du LGS décrit ici s'étend sur 3,000 lignes de code et est disponible sur http://www.obpcdl.org, ainsi que l'outillage OBP.

Hypothèses et restrictions Ainsi modélisé, le système comporte quelques différences en regard des spécifications en entrée :

1. Une seule partie logicielle au lieu des deux requises.
2. Un seul câble en sortie des senseurs au lieu des trois requis.
3. Les fautes injectées sont permanentes.

Pour les deux premiers points, cette redondance est requise pour assurer le maintient du service si l'un des composants dupliqués est compromis. Il n'y a pas d'impact sur le comportement hors cas de défaillance générale (i.e. si les 3 câbles en sortie d'un senseur perdent le contact). Pour cette étude, ces composants sont considérés fiables.

Outre ces différences, l'ensemble des spécifications est prise en compte. En particulier, les contraintes temporelles portant sur les trains, les portes, les électrovannes et le commutateur analogique sont modélisées en temps continu. Aussi, la partie logicielle permet au pilote d'inverser la séquence actuelle à n'importe quel moment. Enfin, les différentes anomalies devant être remontées au pilote (porte droite non-fermée 7 secondes après requête par exemple) le sont à travers les signaux lumineux.

4.2.2 Guide de vérification xGDL

Fermeture du système Les transitions appartenant à la fermeture du système sont celles de l'environnement, c'est à dire celles des automates modélisant le pilote et le perturbateur.

Le système peut être exploré sans contraindre ces transitions. Le graphe d'états exploré correspondant alors à l'ensemble des comportements possibles sans borner le nombre de manipulation du train d'atterrissage (*handle*) ou le nombre de fautes injectées.

Cependant la taille de ce graphe est bien trop grande pour permettre la terminaison de l'algorithme classique de model-checking sans se heurter à l'explosion combinatoire.

Alphabet d'interactions xGDL Comme discuté dans la section 3.2.3, pour obtenir cet alphabet on défini une fonction d'étiquetage sur les transitions de la fermeture du système. Ici, il s'agit des transitions des automates représentant le pilote et le perturbateur.

Les étiquettes attribuées sont **handle** pour la seule transition du pilote et $\mathbf{f_i}$ avec $0 < i \leq n$ pour les n transitions du perturbateur (correspondantes aux n fautes définies par les spécifications).

f_{1_1}	f_{1_2}	f_{2_1}	f_{2_2}
$asbo$	$asbc$	gbo	gbc
analog switch		general EV	
Opened	Closed	Opened	Closed
exclusive		exclusive	

f_{3_1}	f_{3_2}	f_{4_1}	f_{4_2}	f_{5_1}	f_{5_2}	f_{6_1}	f_{6_2}
$debo$	$debc$	$drbo$	$drbc$	$gebo$	$gebc$	$grbo$	$grbc$
door electro-valves				gear electro-valves			
extension		retraction		extension		retraction	
Opened	Closed	Opened	Closed	Opened	Closed	Opened	Closed
exclusive		exclusive		exclusive		exclusive	

f_7	f_8	f_9	f_{10}	f_{11}	f_{12}
fd	ld	rd	fg	lg	rg
Front	Left	Right	Front	Left	Right
door			gear		

TABLE 4.2 – Survol des possibles injections de faute

Injection de faute La table 4.2 présente l'ensemble des 18 possibles injections de faute prévues par les spécifications. Certaines sont exclusives, pour l'exemple le commutateur analogique ne peut être à la fois bloqué en positions ouverte et fermée ($f_{1_1} \square f_{1_2}$).

De plus, les exigences portant sur un maximum de trois fautes injectées. En termes de permutations, les possibles séquences d'injection de fautes peuvent s'exprimer via le formalisme xGDL comme suis :

$$\{0,\ 3\}\ of\ [(f_{1_1}\ \square\ f_{1_2}),\ ...,\ (f_{6_1}\ \square\ f_{6_2}),\ f_7,\ ...,\ f_{12}]$$

Cela représente 1 séquence de taille 0, 18 séquences de taille 1, 294 séquences de taille 2 et 4320 séquences de taille 3 pour un total de 4633 séquences possibles.

L'expression xGDL correspondant à l'injection d'une faute parmi toutes celles possibles et prenant en compte les exclusivités est notée $\mathbf{F_{all}}$ par la suite. C'est à dire : $F_{all} = (f_{1_1}\ \square\ f_{1_2}),\ ...\ , (f_{6_1}\ \square\ f_{6_2}),\ f_7,\ ...\ ,\ f_{12}$.

Guide xGDL nominal Il est alors possible de définir un guide xGDL exhaustif comme suis :

$$handle* \parallel \{0,\ 3\}\ of\ [F_{all}]$$

C'est à dire une répétition non bornée de *handle* en parallèle avec l'ensemble des permutations de taille 3 ou inférieure parmi l'ensemble de fautes possibles, exclusions prises en compte.

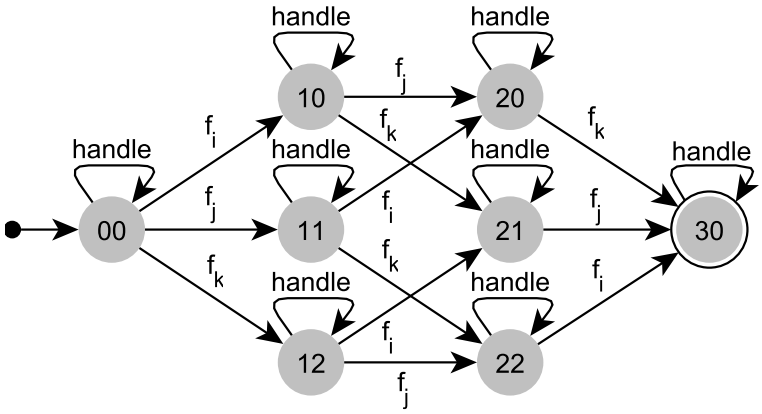

FIGURE 4.4 – DFA obtenu par compilation : $handle* \parallel \{3,\ 3\}\ of\ [f_i,\ f_j,\ f_k]$

Compilation La figure 4.4 présente le résultat de la compilation (telle que définie section 3.3.1) vers un DFA minimisé d'un guide limité à 3 fautes non exclusives (f_i, f_j et f_k). Qu'importe l'état courant, il est possible de tirer une transition *handle* ou l'une de celles correspondant à l'injection d'une faute non déjà introduite.

Note sur la borne inférieure La sémantique de XGDL étant fermée par préfixe, le DFA présenté figure 4.4 accepte aussi l'ensemble des chemins pour les combinaisons de 0, 1 et 2 de ces même fautes.

Cet automate est donc équivalent à celui qui serait obtenu à la compilation de $handle* \parallel \{\mathbf{0},\ 3\}\ of\ [f_i,\ f_j,\ f_k]$ pour lequel tous les états sont des états d'acception.

La borne inférieure de l'opérateur de permutation et de celui de répétition bornée n'a d'impact que lorsque composé séquentiellement. Par exemple, l'expression XGDL $a\{\mathbf{0}, 2\}; b$ reconnaît le terme b alors que ce n'est pas le cas pour $a\{\mathbf{1}, 2\}; b$.

Guide XGDL partiel Il est important de noter que $handle* \parallel \{0,\ 3\}\ of\ [F_{all}]$ est équivalent, en termes d'espace d'états exploré lors de la composition, à :

$$\{0,\ 3\}\ of\ [F_{all}]$$

C'est à dire en excluant *handle* de l'alphabet d'interactions XGDL ou, en d'autres mots, sans contraindre le pilote.

Cette seconde forme est particulièrement utile pour permettre l'application d'algorithmes nécessitant un guide acyclique sans borner le nombre de *handle*.

4.2.3 Stratégie de division

Comme mentionné section 3.5 il est possible de diviser ce guide en plusieurs guides plus *petits* tel que vérifier les propriétés sur l'ensemble de ces nouveaux guides soit équivalent à les vérifier sur le parent.

Taille	Index	Sous-partie F_{index}^{taille}
0	0	$\emptyset$
1	0	$[f_{1_1}]$
	...	
	17	$[f_{12}]$
2	0	$[f_{1_1},\ f_{2_1}]$
	1	$[f_{1_1},\ f_{2_2}]$
	...	
	15	$[f_{1_1},\ f_{12}]$
	16	$[f_{1_2},\ f_{2_1}]$
	17	$[f_{1_2},\ f_{2_2}]$
	...	
	146	$[f_{11},\ f_{12}]$
3	0	$[f_{1_1},\ f_{2_1}\ f_{3_1}]$
	1	$[f_{1_1},\ f_{2_1}\ f_{3_2}]$
	...	
	719	$[f_{10},\ f_{11}\ f_{12}]$

TABLE 4.3 – Sous-parties de fautes possibles

L'opérateur de permutation xGDL, si il n'est pas inclus dans une répétition, se prête particulièrement bien à cet exercice. On divise en fonction des sous-parties de l'ensemble en argument. Certaines options contenant du non-déterminisme (tel que $f_{1_1} \square f_{1_2}$), dénotant ici des fautes exclusives, sont ensuite mises à plat en autant de parties que nécessaires. La table 4.3 présente les sous-parties obtenues : 18 combinaisons pour 1 faute, 147 pour 2 fautes et 720 pour 3 fautes.

On peut noter que l'exclusivité des couples de fautes $(f_{i_1},\ f_{i_2})\ \forall i, 1 \leq i \leq 6$ est respectée.

Dans la suite, les guides correspondants sont référencés par la taille et l'index de la sous-partie de fautes considérées. Ainsi :

— $\mathbf{F_{id}^{len}}$ correspond à la sous-partie de taille *len* et d'index *id* ;

— $\mathbf{FailureGuide_{id}^{len}}$ correspond au guide xGDL $handle * \parallel \{3,\ 3\}\ of\ F_{id}^{len}$.

Pour l'exemple :

$$FailureGuide_{15}^{2} = handle * \parallel F_{15}^{2} = handle * \parallel \{2,\ 2\}\ of\ [f_{1_1},\ f_{12}]$$

Le théorème suivant est basé sur la définition du langage d'un LTS tel que introduit section 2.3.1. Ainsi $langage(G)$, avec G un guide xGDL, dénote le langage du LTS (le DFA sans ses états d'acceptation) obtenu par compilation de G tel que décrit section 3.3.

Théorème 2. ***LGS : Équivalence de langage*** *Le langage défini par le guide* xGDL $handle * \parallel \{0,\ 3\}\ of\ [F_{all}]$ *et l'union des langages définis par les guides* $FailureGuide_{id}^{3}$ *obtenus après division sont identiques :*

$$\mathbf{langage(handle} * \parallel \mathbf{\{0,\ 3\}\ of\ [F_{all}])} \quad = \quad \cup_{\mathbf{id=0}}^{\mathbf{719}}\ \mathbf{langage(FailureGuide_{id}^{3})}$$

Démonstration. **Par réécritures successives** du membre droit de l'égalité :

$0:\ \cup_{id=0}^{719}\ langage(FailureGuide^3_{id})$
$1:\ langage(\ FailureGuide^3_0 \square ... \square FailureGuide^3_{719}\ )$
$2:\ langage(\ (handle* \parallel \{3,\ 3\}\ of\ F^3_0) \square ... \square (handle* \parallel \{3,\ 3\}\ of\ F^3_{719})\ )$
$3:\ langage(\ handle* \parallel (\ (\{3,\ 3\}\ of\ F^3_0) \square ... \square (\{3,\ 3\}\ of\ F^3_{719})\ )\ )$
$4:\ langage(\ handle* \parallel \{3,\ 3\}\ of\ [F_{all}]\ )$

Les règles appliquées pour chaque pas de réécriture (jusqu'au terme 3) sont les suivantes :

$0 \to 1:\ \cup_{i=0}^{n}\ langage(G_i) \to langage(G_0 \square ... \square G_n)$
$1 \to 2:\ FailureGuide^{len}_{id} \to handle* \parallel \{3,\ 3\}\ of\ F^{len}_{id}$
$2 \to 3:\ (L \parallel R_0) \square ... \square (L \parallel R_n) \to L \parallel (R_0 \square ... \square R_n)$

Pour le pas de réécriture $3 \to 4$, les ensembles de fautes de F^3_0 à F^3_{719} étant définis comme l'ensemble exhaustif des sous-parties de taille 3 autorisés par F_{all} (exclusions comprises), pour toute séquence de taille 3 obtenue par permutations et (le cas échéant) mise à plat du non-déterminisme de F_{all} il existe une sous-partie F^3_{id} telle que cette séquence appartienne aux possibles permutations de taille 3 de F^3_{id} (et réciproquement). On a donc :

$$3 \to 4:\ (\{3,\ 3\}\ of\ F^3_0) \square ... \square (\{3,\ 3\}\ of\ F^3_{719}) \ \to\ \{3,\ 3\}\ of\ [F_{all}]$$

Il reste à prouver que :

$$langage(handle* \parallel \{\mathbf{0},\ 3\}\ of\ [F_{all}]) \ = \ langage(handle* \parallel \{\mathbf{3},\ 3\}\ of\ [F_{all}])$$

Trivial du fait que la sémantique de xGDL est fermée par préfixe : si un terme est accepté, ses préfixes le sont aussi. Ici, si un terme contenant 3 fautes est accepté, alors les préfixes contenant 0, 1 ou 2 fautes le sont aussi. Ce point est illustré plus haut en commentaire de la figure 4.4. □

Espace d'états du système Il est important de noter que l'équivalence de langage ci-dessus implique que l'espace d'états du système atteint par compositions avec $handle* \parallel \{0,\ 3\}\ of\ [F_{all}]$ et l'union des espaces d'états atteints par composition avec les différent $FailureGuide^3_{id}$ sont les même. Ce point est évoqué section 2.3.2 dans le paragraphe traitant de l'atteignabilité et l'espace d'état de la composition de deux LTS.

4.2.4 Dépliage borné

Dans le cas où l'exploration de la composition de l'un des guides obtenus après Split (tel que défini plus haut) avec le système se heurte au problème de l'explosion combinatoire, il peut être nécessaire de recourir à un dépliage borné du guide en question. En sus de restreindre le nombre de *handle*, il est alors possible de mener l'exploration avec l'algorithme Past-Free[ze] présenté section 3.4.

La figure 4.5 présente le dépliage borné à 5, tel que défini section 3.3.2, du contexte $handle* \parallel \{3,\ 3\}\ of\ [f_i,\ f_j,\ f_k]$ (dont la compilation est présentée figure 4.4).

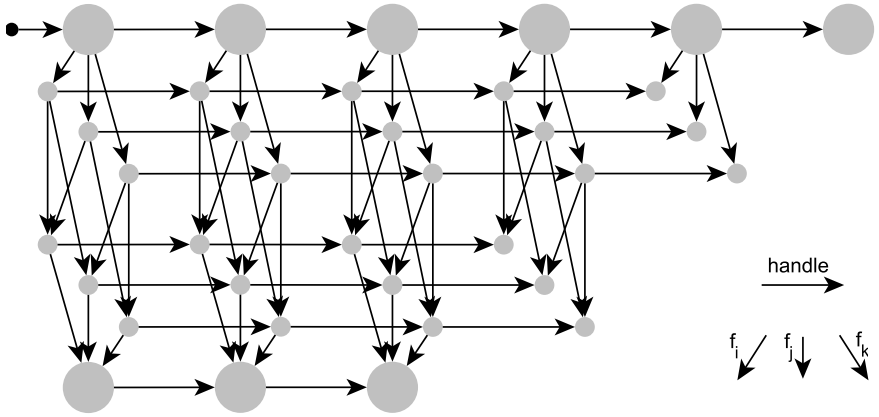

FIGURE 4.5 – Dépliage borné à 5 de $handle* \parallel \{3,\ 3\}\ of\ [f_i,\ f_j,\ f_k]$

Comme l'indique la légende, les transitions horizontales correspondent à l'étiquette *handle*, les transitions strictement verticales à f_j, les transitions en diagonale vers le bas et la gauche à f_i, vers le bas et la droite à f_k.

On peut noter que tous les chemins passent par au moins 2 *handle*. C'est à dire la profondeur en argument du dépliage (ici 5) minus le nombre de fautes injectées (ici 3).

Pour justifier de la couverture il faudra donc fournir une preuve, ou à minima des indicateurs de confiance, de la suffisance de ces $n-3$ tirages de *handle* (avec n la profondeur du dépliage) pour la vérification des propriétés.

4.2.5 Propriétés

Pour illustrer la formalisation des propriétés, on considère les exigences suivantes :

— **Exigence $\mathbf{R_1}$:** L'indicateur lumineux rouge doit toujours être éteint.

— **Exigence $\mathbf{R_2}$:** Après chaque interaction du pilote, l'indicateur vert doit être allumé sous 15 secondes.

Invariant R_1 est un invariant sur l'ensemble des états explorés et peut être exprimée sous la forme d'un prédicat : $redLight = false$.

Lors de l'exploration, ce prédicat doit être évalué sur tous les états atteint. Si l'évaluation renvoie faux, la propriété est considérée violée.

Automate observateur R_2 est une propriété portant sur les chemins, un prédicat ne suffit donc pas. Il s'agit d'une action-réponse temporisée.

Sans la partie temporelle il s'agirait d'une propriété de vivacité nécessitant une formalisation via logique temporelle telle que LTL et CTL. Avec, cependant, il s'agit d'une propriété de sûreté.

La figure 4.6 présente un automate observateur correspondant à la vérification de cette propriété.

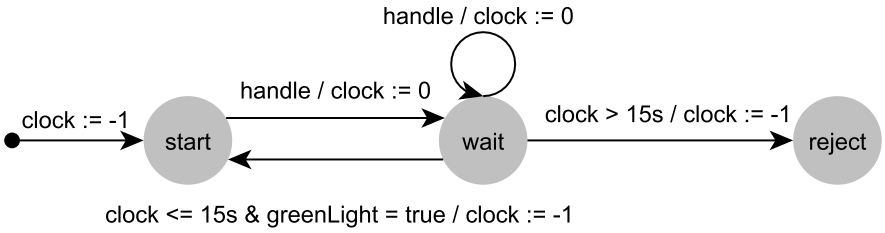

FIGURE 4.6 – Automate observateur pour la vérification de R_2

Les actions $clock := -1$ et $clock = 0$ désactive et lance l'horloge respectivement. Cette horloge respecte la sémantique du langage utilisé pour la modélisation du système. De ce fait, la garde $clock > 15s$ correspond à l'instruction Fiacre *wait*]15000, 15000] avec un pas de temps au millième de seconde.

L'évaluation de la garde $greenLight = true$ nécessite l'accès à la variable globale $greenLight$ du système (représentant l'état de l'indicateur lumineux vert). Le résultat de l'évaluation est obtenu en accord avec la sémantique du langage dans lequel est modélisé le système, Fiacre ici.

handle correspond à l'observation d'une interaction du pilote. Il s'agit de la même étiquette introduite section 4.2.2. En conséquence, les transitions correspondantes dans la composition du système avec le guide XGDL sont celles étiquetées par *handle*.

En synthèse, l'automate observateur présenté figure 4.6 suis l'exécution du système et rentre dans l'état *reject* si le temps entre une interaction du pilote et l'allumage de l'indicateur lumineux vert dépasse 15s.

Lors de l'exploration de la composition, si l'automate observateur rentre dans l'état *reject* la propriété est considérée violée.

4.2.6 Résultats d'exploration

Cette section présente les résultats obtenus par CaV [19] du système LGS présenté section 4.2.1 guidé par le guide XGDL présenté section 4.2.2 pour la vérification de propriétés telles que celles présentées section 4.2.5.

Ces résultats ont été obtenu par l'outillage OBP v1.4.8 sur deux configurations Linux 64-bit avec 64GB et 128GB de mémoire, référencée par la suite comme $L64$ et $L128$ respectivement.

En mode nominal (sans injection de faute) l'exploration termine sur $L64$ pour un total de 352 279 états et 1 447 197 transitions en près de 4 minutes.

Comme indiqué section 4.2.2, dans les cas où le pilote n'est pas contraint et n injections de fautes, les guides XGDL $handle * \parallel \{0,\ n\}\ of\ F^n_{id}$ et $\{0,\ n\}\ of\ F^n_{id}$ explorent le même espace d'états par composition avec le système.

Cette seconde forme est privilégiée pour sa nature acyclique. Cependant, si l'exploration échoue et qu'il semble nécessaire de contraindre le pilote, un dépliage borné de la première

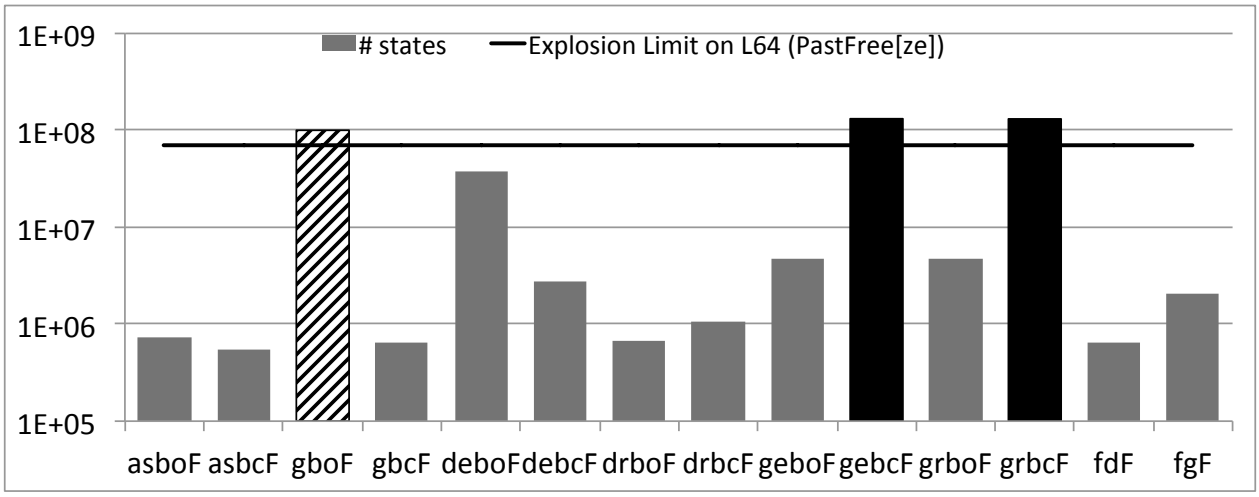

FIGURE 4.7 – Résultats d'exploration avec une injection de faute, pilote non-contraint.

forme peut être utilisé.

La suite présente les résultats obtenus pour une et une seule injection de faute. Le guide xGDL a été divisé en 18 partitions $FailureGuide^1_{id}$ (voir section 4.2.2, table 4.3), chacun représentant donc une répétition non-bornées de *handle* en parallèle avec une injection de faute spécifique.

La correspondance entre les indexes de fautes et les noms utilisés pour les guides est visible section 4.2.2, table 4.2.

Pilote non-contraint Les guides xGDL utilisé pour ces explorations ne contraignent pas le pilote ($\{0,\ 1\}\ of\ F^1_{id}$).

La figure 4.7 présente les résultats obtenus sur 14 de ces guides. Ceux absents (portes et trains de gauche et droite : ld, rd, lg, rg) sont identiques à d'autres déjà présents (porte et train de front : fd, fg). Les barres grises dénotent une exploration qui termine sur L64 et L128, la barre avec le motif en diagonal une exploration qui termine sur L128 mais pas sur L64 (gbo), les barres noires une exploration qui ne termine pas qu'importe la configuration utilisée ($gebc$ et $grbc$).

Ici, l'approche par CaV aide peu. Le pilote n'est pas contraint et les contextes sont limités à l'injection d'une et une seule faute, résultant dans des guides xGDL extrêmement petits (2 états 1 transition après compilation).

Nombre de handle borné à 3 Pour le passage à l'échelle des cas où l'on est confronté à l'explosion combinatoire, on se propose de borner, arbitrairement, les interactions du pilote à 3 interactions. Ne considérant toujours qu'une injection de faute, cela correspond à un dépliage borné à 4 du guide xGDL $handle * \parallel \{0,\ 1\}\ of\ F^1_{id}$ tel que présenté section 4.2.4.

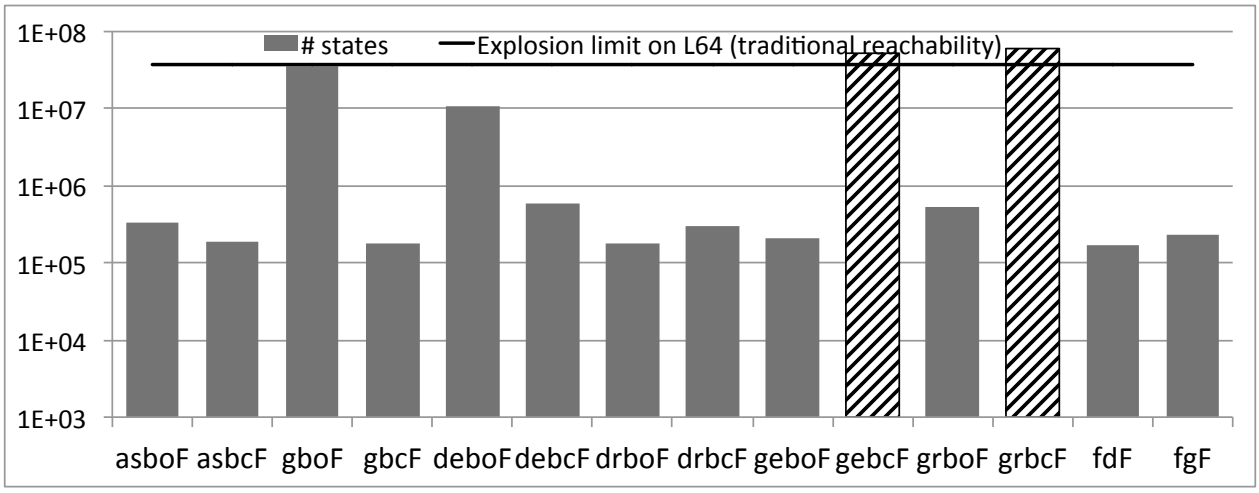

FIGURE 4.8 – Résultats d'exploration avec une injection de faute, pilote contraint à 3 interactions.

Les guides obtenus par compilation contiennent notablement plus d'états et transitions (9 et 11 respectivement) et se prêtent mieux à l'exploitation de l'algorithme Past-Free[ze] présenté section 3.4.

La figure 4.8 présente les résultats obtenus. Là encore, les barres grises correspondent à des explorations qui terminent qu'importe la configuration, les barres avec le motif en diagonal à des explorations qui terminent sur $L128$ mais pas sur $L64$.

Cette fois-ci, toutes les explorations terminent sur $L128$. Les deux explorations qui ne terminent pas sur $L64$ (*gebcF* et *grbcF*) correspondent à celles qui ne terminaient pas sans contraindre le pilote. Au moment de l'explosion sur $L64$, l'algorithme a exploré 79% et 58% de l'espace d'états respectivement pour ces deux guides.

Division en profondeur Dans les deux cas où l'exploration ne terminent pas sur $L64$, on divise ces guides à une profondeur de 2 (voir section 3.5). Les résultats obtenus sur $L64$ pour les quatre guides générés dans les deux cas sont présenté figure 4.9. Toutes les explorations terminent.

Premier bilan En synthèse des résultats présentés jusqu'ici, le model-checking classique ne permet pas l'exploration du système sans une division préliminaire en fonction de la faute injectée. Ce faisant, certaines fautes posent toujours problème (*gebc* et *grbc* sur $L128$, *gbo* sur $L64$).

Borner le nombre d'interactions du pilote à 3 permet, via Past-Free[ze], le passage à l'échelle de ces explorations sur $L128$. De plus, une division supplémentaire en fonction de la profondeur en permet le passage à l'échelle aussi sur $L64$.

Bien qu'illustratifs, ces résultats obtenus avec un nombre d'interactions du pilote borné à 3, posent le problème de l'exhaustivité de l'analyse.

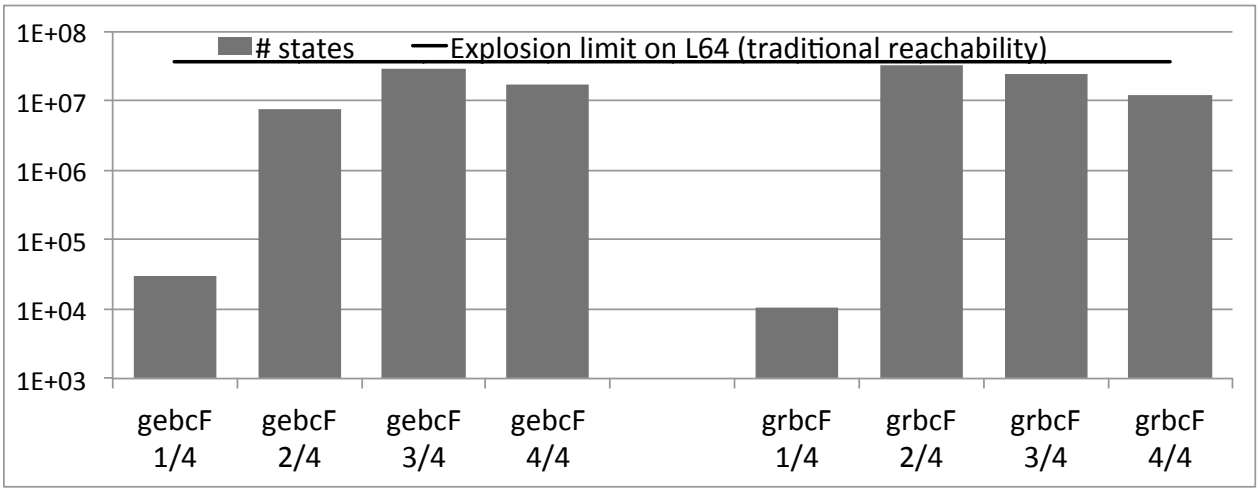

FIGURE 4.9 – Résultats d'exploration sur $L64$ après division en profondeur

Context step	H0	H1	H2	H3	H4	H5	H6	H7	H8	H9
Cluster size	1	448	12 904	44 331	72 650	85 042	87 780	87 746	87 780	87 746

TABLE 4.4 – Tailles des partitions explorées par Past-Free[ze] en mode nominal

Analyse exhaustive, mode nominal Pour garantir l'exhaustivité de la vérification, il convient de s'assurer que la borne retenue est suffisante.

Pour ce faire, on borne arbitrairement le nombre d'interactions à 1000 dans le mode nominal (sans injection de faute).

La table 4.4 présente la taille des partitions obtenues via Past-Free[ze]. On peut noter l'émergence d'un paterne : à partir de H_6 les tailles de partitions d'indice pair et celles d'indice impaire sont invariantes. Cette observation est respectée par l'ensemble des partitions explorées pour les 1000 interactions du pilote.

Il s'agit d'une forte indication d'un comportement cyclique modulo le guide xGDL composé avec le système. La preuve peut en être faite par induction basée sur la bi-simulation des partitions $H_6 + H_7$ et $H_8 + H_9$ (étape initiale d'induction).

Par conséquence, en mode nominal Il suffit de borner le nombre d'interactions du pilote à 7 pour garantir l'exhaustivité de l'exploration.

Le diamètre d'atteignabilité évoqué section 3.6 peut alors être obtenu en inférant du graphe d'états obtenu la profondeur maximale des chemins jusqu'à la partition H_7. Ici, en mode nominal, il est de 305.

La figure 4.10 présente le pourcentage de l'espace d'états libéré par Past-Free[ze] pour différents nombres d'interactions (*handle*).

Les résultats mitigés pour un petit nombre de *handle* (≤ 7) s'explique du fait que la dernière partition est notablement plus grande que les précédentes (voir table 4.4). Pour 8 ou plus cependant, Past-Free[ze] permet de libérer la mémoire de plus de 80% de l'espace d'états.

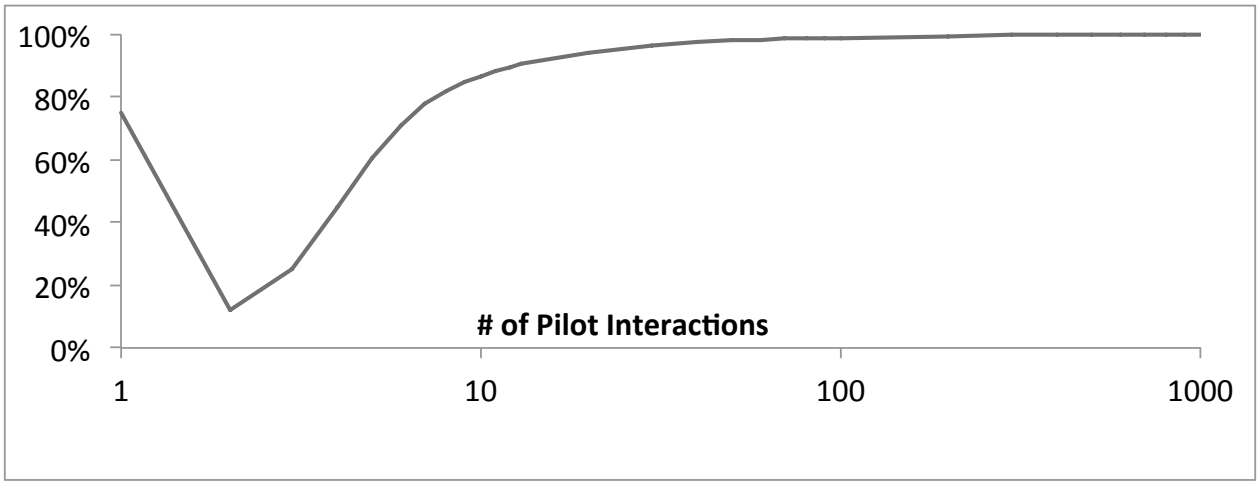

FIGURE 4.10 – Fraction de l'espace d'états libérée par Past-Free[ze]

Failure	asbo	asbc	gbo	gbc	debo	debc	drbo	drbc	gebo	gebc	grbo	grbc	fd	fg
Bound	16	16	18	17	20	20	18	20	20	X	18	X	20	20

TABLE 4.5 – Nombre d'interactions du pilote nécessaires pour les différentes fautes

Qu'importe la progression actuelle de l'analyse, seules les partitions indexées par l'état courant du DAG (obtenu par compilation du guide xGDL) et l'unique état suivant sont conservés. De ce fait, le passage à l'échelle de l'exploration guidée par Past-Free[ze] est dépendant de la taille de deux partitions successives et par une éventuelle explosion temporelle.

Pour l'illustration, l'exploration de 1000 *handle* occupe au maximum $10GB$, soit 15% de la mémoire disponible sur $L64$. Cette même exploration ne termine pas sans Past-Free[ze].

Analyse exhaustive, une injection de faute Le nombre d'interactions du pilote nécessaires pour garantir l'exhaustivité (7 en mode nominal) varie en fonction des fautes injectées et donc du guide xGDL. On applique la même méthodologie que précédemment pour les guides correspondants à l'injection d'une faute avec un nombre d'interactions (*handle*) fixé arbitrairement à 30 (dépliage borné à 31).

Mis à part les explorations de la composition avec les guides correspondants aux fautes *gebc* et *grbc*, les explorations ont permis de déduire le nombre minimum d'interactions du pilote requis pour l'exhaustivité. La table 4.5 récapitule ces résultats.

Pour les fautes *gebc* et *grbc*, cependant, l'exploration ne termine pas dû à la taille des partitions générées.

Spécialisation du guide xGDL pour gebc et grbc. On s'intéresse ici à $f_i \in \{gebc,\ grbc\}$, correspondants aux fautes pour lesquelles l'exploration de la composition précédente n'a pas aboutie.

Pour une seule injection de faute donnée, en utilisant l'atomicité de cette injection, on a :

$$handle* \parallel \{1,\ 1\}\ of\ [f_i] \quad = \quad handle* \parallel f_i \quad = \quad handle*; f_i; handle*$$

Basé sur les résultats obtenus en mode nominal plus tôt dans cette section, on peut construire un nouveau guide xGDL équivalent à ce dernier. Pour la vérification, il est suffisant d'explorer la composition avec l'injection de faute intervenant au plus tard après 7 interactions du pilotes.

En d'autres mots, pour la vérification, il est suffisant de composer le système avec le guide xGDL suivant :

$$handle\{0,\ 7\}; f_i; handle*$$

Ces guides, produisant par compilation des DFA aux structures très linéaires, se prêtent d'autant mieux à l'application du Past-Free[ze]. De plus ils peuvent être divisés en fonction du nombre d'interactions du pilotes en amont de l'injection de faute, résultant en 8 guides plus *petits* et linéaires :

$$\forall\ len,\ 0 \leq len \leq 7, \quad G_i^{len} :\ handle\{len,\ len\}; f_i; handle*$$

L'exploration de la composition avec ces guides pour $0 \leq len \leq \mathbf{3}$ et un dépliage borné toujours arbitrairement fixé à 31 (pour 30 interactions) a terminé avec succès et à permis d'observer, à nouveau, l'apparition d'un pattern cyclique.

Pour les guides avec $\mathbf{4} \leq len \leq 7$, la taille des partitions générée est prohibitive pour en permettre le passage à l'échelle.

4.2.7 Conclusion

Pour le cas d'étude du LGS [8], la vérification du logiciel embarqué est axée sur la détection de ces fautes afin d'assurer que le pilote soit tenu à jour. Pour ce faire il est nécessaire d'inclure l'injection de ces fautes dans la modélisation du système.

Cette section a montré que l'approche par CaV avec xGDL permet de produire des unités de preuve variées mais référençant toutes la même modélisation du système (invariant de ces unités de preuve). Il a été ainsi possible d'itérer à partir d'une première formalisation générale vers un ensemble de guides xGDL isolant les difficultés alors identifiées et ce sans instrumenter le système. Sur les 18 fautes possibles, la vérification a été possible sur 16 d'entre elles.

Les deux fautes restantes, posant problème pour le passage à l'échelle, ont pu faire l'objet d'une nouvelle itération en permettant une vérification partielle. La vérification a été possible pour les cas où ces fautes sont injectées après au plus trois interactions du pilote. L'explosion de l'espace d'état reste un problèmes avec un nombre supérieur d'interactions en amont de l'injection.

Contrairement à la mise-en-œuvre de la CaV avec CDL, l'approche proposée ici (avec xGDL) suppose un système fermé en entrée. Il s'agit d'un point de convergence avec bien d'autres techniques basées sur le model-checking [29, 12], et d'une piste pour permettre le passage à l'échelle des cas restants.

4.3 Pacemaker (santé)

Cette section est basée sur le chapitre dédié à la vérification par model-checking publié dans *Model-Based Analysis* [7]. Le cas d'étude, fil rouge de l'ouvrage, est celui d'un Pacemaker [1]. Il s'agit d'une synthèse du travail présenté alors, formulée via xGDL.

Système Le Pacemaker est un système capable d'observer et de stimuler le cœur d'un patient. Il est hautement configurable, le même appareil peut assumer différents comportements adaptés à diverses pathologies. Ses paramètres peuvent aussi être ajustés par le médecin lors de l'implantation et des rendez-vous de suivi.

L'architecture du système, présentée figure 4.11, comprend :

— un boîtier de commande (DCM) utilisé par le médecin pour paramétrer le système en fonction de la pathologie observée chez le patient ;

— un contrôleur responsable de relayer ces paramètres au *Cardilogist*, du lancement et de la mise en pause du système ;

— le *Cardiologist*, l'entité centrale responsable de la synthèse de l'observation et de la prise de décision pour la stimulation ;

— deux électrodes (*LeadV* et *LeadA*, pour le ventricule et l'auricule respectivement) capables d'observer et de stimuler le cœur ;

— le cœur du patient.

Les communications entre le DCM et le contrôleur, ainsi que entre le contrôleur et le *Cardiologist*, sont asynchrones et relayées par des files d'attentes (FIFO). L'observation et la stimulation du cœur est modélisée par des variables partagées (registres).

La figure 4.12 présente l'automate du *Cardiologist*. L'état *Pacing*, responsable de l'observation et la stimulation du cœur en fonction du mode et des paramètres sélectionnés, est le principal sujet de l'étude, le garant des différentes propriétés à vérifier.

Les différents modes offerts sont, en respect de la spécification fournie [1], XOO, XXI, XXT et VDD avec $X = V \vee X = A$ (pour cibler le ventricule ou l'auricule respectivement), soit 7 modes différents.

Pour l'exemple, les modes VOO et AOO (XOO sur la figure) correspondent à une stimulation à intervalles fixes de la chambre correspondante.

Context-aware verification Pour la vérification on se focalise sur la phase ambulatoire du cycle de vie, c'est à dire sur le comportement du pacemaker en dehors du cabinet médical.

Lors de l'implantation, du paramétrage, de la consultation et de l'extraction, le patient est pris en charge par le corps médical, lequel est alors responsable d'assurer le maintient de l'activité cardiaque.

Celant étant dit, les fonctionnalités nécessaires pour permettre au médecin de se connecter, de mettre en pause, de paramétrer et d'activer le pacemaker sont incluses dans la modélisation.

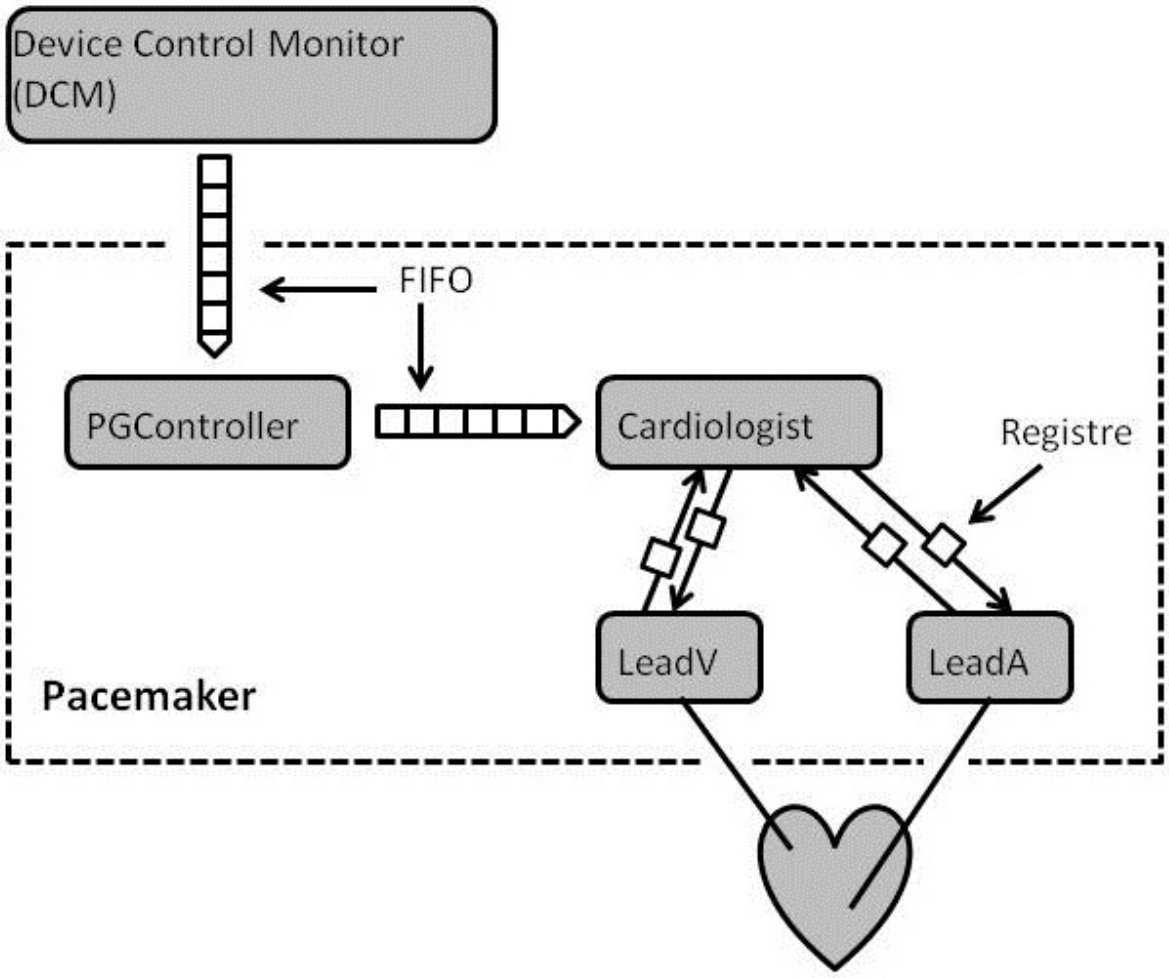

FIGURE 4.11 – Pacemaker : architecture du système

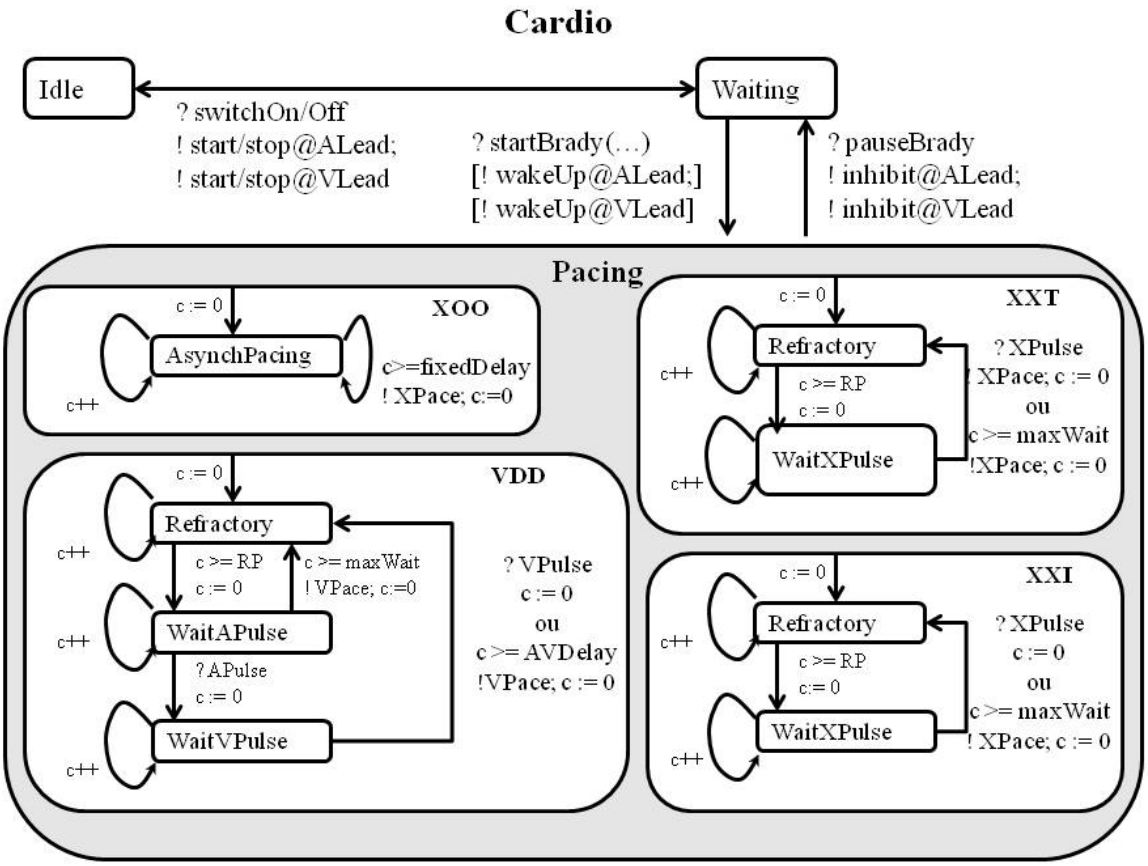

FIGURE 4.12 – Pacemaker : automate du *Cardiologist*

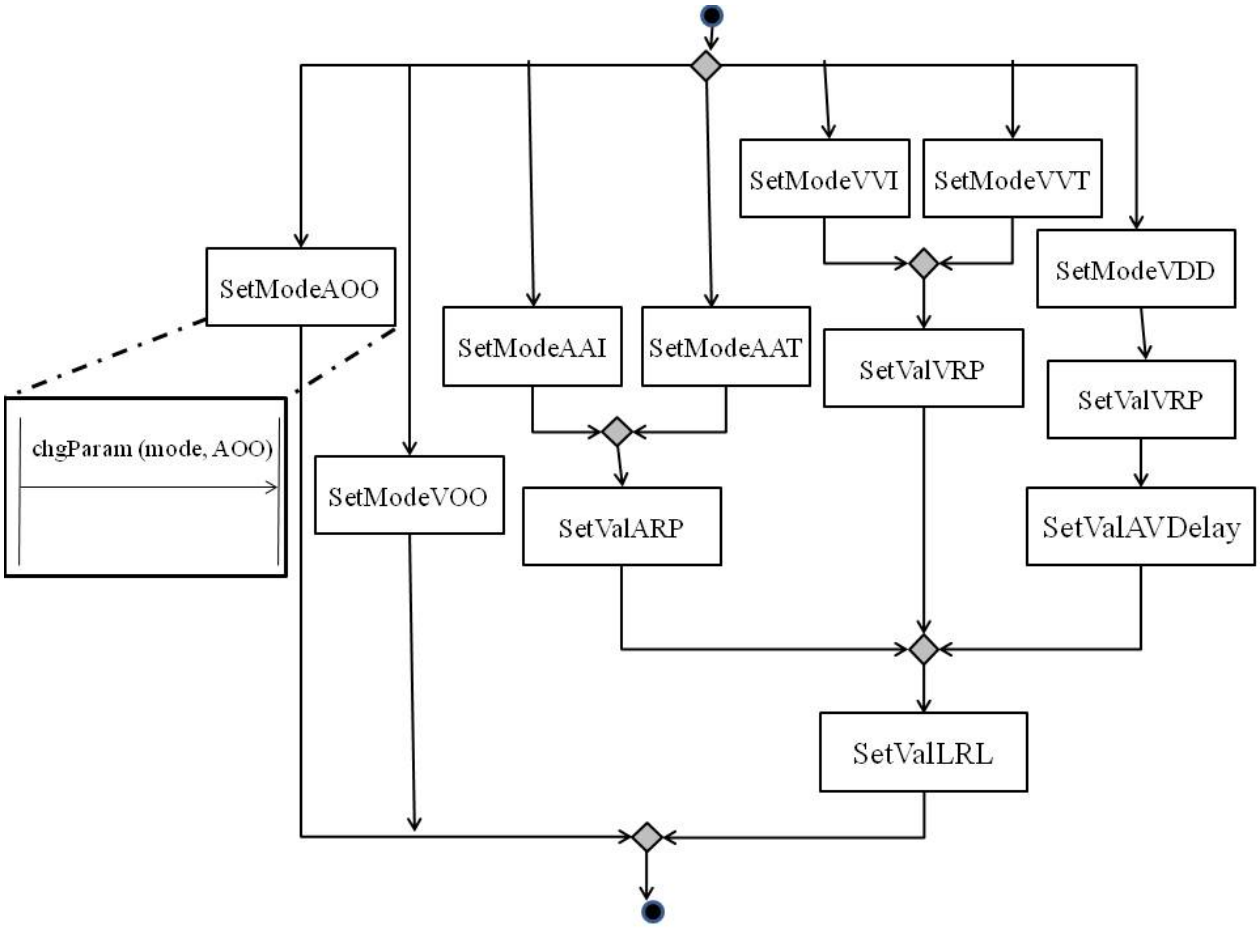

FIGURE 4.13 – Pacemaker : paramétrage

Dans ce cadre, l'utilisation de la CaV nous permet de restreindre l'exploration à la phase ambulatoire. Le guide XGDL est une séquence d'initialisation composée du paramétrage et de l'activation du système.

La figure 4.13 présente la partie de paramétrage de cette initialisation, c'est à dire la sélection du mode et de la valuation des paramètres pertinents pour le mode choisi. Pour l'exemple, comme indiqué sur la figure, l'étiquette $SetModeAOO$ correspond au tirage d'une transition représentant un envoie asynchrone du DCM au $PGController$ du message $chgParam(mode,\ AOO)$. Les nœuds tel que $SetValARP$ correspondent à des alternatives portant l'ensemble des valeurs possibles du paramètre. Les pas d'incréments sont précisés par la spécification [1]. Pour l'exemple le paramètre ARP, pertinent uniquement pour les modes AAI et AAT, doit être compris entre 150 et $500ms$ avec un pas de $10ms$, soit 36 valeurs possibles et autant d'étiquettes XGDL.

La taille de l'ensemble des paramétrages possibles, et donc le nombre de chemins dans cette séquence d'initialisation, est de 62 498. En sus des 165 étiquettes requises pour les

Nombre de combinaisons considérées	Nombre d'états explorés	Nombre de transitions explorées	Temps (sec)
7	40 455	48 781	2
42	299 740	364 391	10
110	744 224	909 908	25
226	1 783 438	2 238 305	64
352	Explosion	–	–

TABLE 4.6 – Vérification brute du Pacemaker.

paramètres, l'expression du guide XGDL en requière 2 autres pour activer et mettre en pause la stimulation. Cette dernière n'apparaît pas dans le guide mais son inclusion dans l'alphabet bloque la transition correspondante lors de l'exploration, permettant ainsi de se limiter à la phase ambulatoire.

Alternativement à cette approche, il aurait été équivalent de retirer le DCM du système, de générer autant de configurations initiales que de paramétrages possibles et de commencer l'exploration en phase ambulatoire. Cependant la spécification du guide XGDL nous permet d'éviter une instrumentation du système et constitue une spécification lisible et maintenable de l'ensemble des paramétrages considérés (tel qu'illustré figure 4.13).

Dans les deux cas, l'exploration peut être divisée en fonction de sous-parties de l'ensemble des configurations initiales. Pour l'approche par CaV, il s'agit d'une division en profondeur du guide XGDL tel que présenté section 3.5, un algorithme supporté par l'outillage.

Résultats La propriété que l'on cherche à vérifier porte sur la durée maximale d'un cycle cardiaque, laquelle doit être inférieure à l'un des paramètre (LRL). Il s'agit de l'exigence 2 listé par la publication [7] en source de cette synthèse, représentée par un automate observateur. Celui-ci est composé avec le système et le guide XGDL lors de l'exploration.

Pour l'illustration, en sus du guide XGDL représentant les 62 498 paramétrages possibles, on introduit plusieurs guides plus petits (mais non-exhaustifs). Pour chacun, le nombre de combinaisons considéré est précisé. Les résultats ont été obtenu par OBP [21] sur un OS Linux avec 64G de mémoire vive.

Les tables 4.6 et 4.7 présentent la taille des espaces d'états explorés sans et avec division du guide XGDL respectivement. Dans le deuxième cas, le processus de division (le Split présenté section 3.5) est automatique. Si l'exploration de la composition d'un guide XGDL avec le système et l'automate de la propriété échoue par explosion combinatoire, ce guide est divisé en profondeur et les différentes explorations correspondantes sont menées. Sur cette deuxième table, la deuxième colonne indique le nombre de guides générés récursivement lors de ce processus.

Sans division, la vérification termine avec succès pour un guide comprenant 226 combinaisons différentes, explorant ainsi prêt de 2 millions d'états en une minute, mais échoue pour 352 combinaisons.

Nombre de combinaisons considérées	Nombre de guides générés	Nombre (cumulés) d'états explorés	Nombre (cumulés) de transitions explorées	Temps d'exploration (sec)
352	7	2 680 017	3 362 387	92
578	12	4 519 385	5 729 766	238
884	13	7 033 893	8 987 094	241
1 282	14	10 179 272	13 069 412	437
3 746	72	30 869 430	40 072 535	1 686
8 194	228	67 893 326	88 639 720	3 963
15 202	344	126 267 775	16 5504 813	6 355
25 346	484	21 2075 376	27 8725 276	9 679
39 202	648	329 522 688	434 009 683	14 226
55 554	636	476 083 965	628 001 017	19 779
62 498	940	535 871 149	706 896 539	22 238

TABLE 4.7 – Vérification du Pacemaker avec division automatisée du guide XGDL.

Avec division l'exploration termine dans tous les cas. Pour le guide complet comprenant les 62 498 combinaison possible, l'algorithme de division génère 940 guides différents, explore plus de 555 millions d'états (plus de 300 fois l'espace obtenu pour 226 combinaisons) en 370 minutes (6 : 10 heures).

Conclusion de l'étude de cas Le pacemaker illustre le cas typique d'un système embarqué dont le comportement peut varier grandement d'une mise en service à une autre. Pour ces systèmes, l'approche par CaV et XGDL offrent un cadre adapté à la spécification des paramétrages possibles. De plus l'algorithme de division, présenté section 3.5, permet d'exploiter cette spécification pour faciliter le passage à l'échelle de la vérification par model-checking.

4.4 Régulateur de vitesse (automobile)

Cette section est basée sur l'article *Context-aware Verification of a Cruise-Control System* [33] publié en 2014 lors de la quatrième conférence internationale sur l'ingénierie des modèles et des données (MEDI'14). L'approche est ici reformulée et étendue à travers XGDL.

Le système embarqué que l'on cherche à valider ici est un régulateur de vitesse (CCS) dont la fonction principale est de contrôler automatiquement la vitesse d'un véhicule motorisé.

Un scénario nominal d'utilisation est présenté figure 4.14. Après la mise en marche du système (événement *powerOn*), le conducteur doit fournir une vitesse de consigne dite vitesse de croisière (événement *set*). Le système peut alors être engagé (événement *resume*), c'est à dire qu'il prend en charge le maintien de cette consigne de vitesse. A tout moment, le

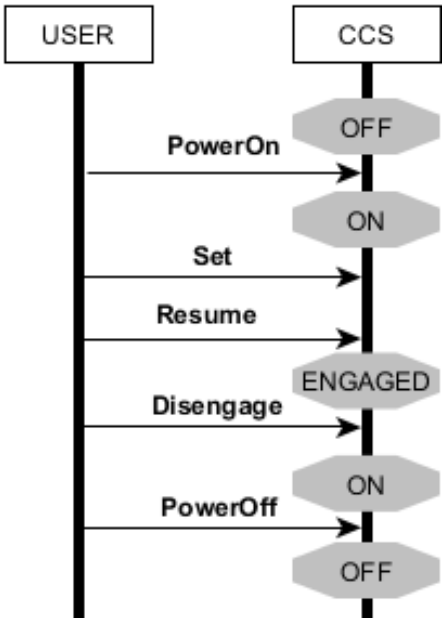

FIGURE 4.14 – Cruise Control : scénario nominal

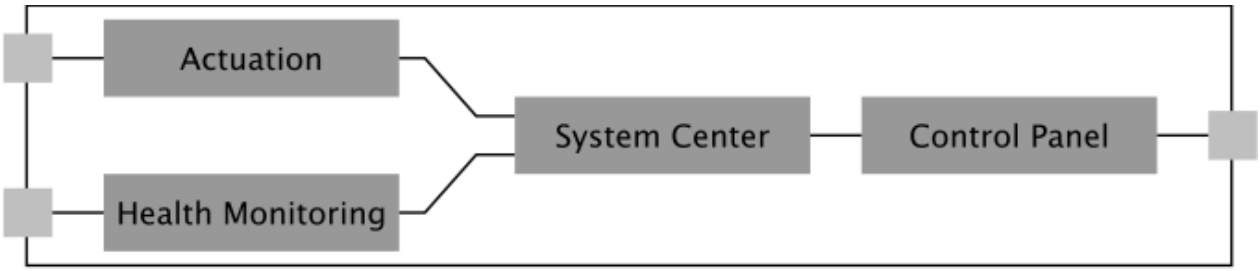

FIGURE 4.15 – Cruise Control : architecture du système ouvert

conducteur peut diminuer ou augmenter la vitesse de croisière. La ligne de vie du système (CCS) renseigne ses changements d'état en réaction a ces messages de l'utilisateur (OFF, ON, $ENGAGED$).

Le système doit garantir la sécurité pour une conduite sûre. En particulier il doit se désengager automatiquement lors des conditions suivantes : si une pression est appliquée sur la pédale de frein, si le conducteur débraye ou si la vitesse courante du véhicule (v) dépasse les bornes autorisées ($40km/h < v < 180km/h$). Le système doit aussi se mettre en pause lors d'une pression sur l'accélérateur et ce jusqu'à ce qu'il soit relâché.

Système La figure 4.15 présente l'architecture du système ouvert retenue pour la modélisation. Elle est composée de quatre entités :

— le *control panel*, responsable de filtrer les entrées de l'utilisateur ;

— l'*actuation*, responsable d'envoyer les consignes de régulation au moteur ;

— l'*health monitoring*, responsable de la détection des cas nécessitant un désengagement du système ;

— le *system center*, responsable de synthétiser les sorties du *control panel* et de l'*health monitoring* en instructions pour l'*actuation*.

En périmètre, la fermeture du système comprend l'utilisateur et le moteur du véhicule. Le premier peut interagir via les boutons de commande et les pédales d'accélération, de freinage et l'embrayage. Le moteur renseigne la vitesse actuelle et supporte la prise en compte de primitives de régulation (accélération, décélération).

Context-aware verification L'exploration du système fermé est explosive. La combinatoire induite par l'ensemble des possibles vitesses de croisière, de la vitesse actuelle et des communications asynchrones est prohibitive.

Pour permettre le passage à l'échelle, on se propose d'utiliser xGDL pour mieux contrôler l'environnent. Les étiquettes, et donc l'alphabet d'actions atomiques, couvrent les envois de l'utilisateur au système (boutons et pédales).

On capture l'ensemble des envois possibles dans des expressions xGDL comme suis :

$$
\begin{array}{lcl}
AnyButton & = & powerOn \,\square\, powerOff \,\square \\
 & & set \,\square\, resume \,\square\, disengage \,\square \\
 & & inc \,\square\, dec \\
AnyPedal & = & (accelPressed;\ accelReleased) \,\square \\
 & & (brakePressed;\ brakeReleased) \,\square \\
 & & (clutchPressed;\ clutchReleased) \\
AnyInteraction & = & AnyButton \,\square\, AnyPedal
\end{array}
$$

On peut noter que les pressions et relâchements des pédales sont d'abord composés en séquence pour en assurer l'alternance.

Il est alors possible de construire un guide xGDL exhaustif en encapsulant cette dernière alternative dans une répétition non bornée :

$$CCSGuide \;=\; AnyInteraction*$$

L'exploration de la composition de ce guide avec le système, équivalente avec celle du système sans xGDL, se heurte au problème de l'explosion combinatoire.

Pour bénéficier des différents algorithmes proposés dans le cadre de la CaV, on applique un dépliage borné sur ce guide xGDL. Cependant, les expérimentations menées ne nous ont pas permis d'atteindre une borne suffisante pour justifier de la couverture. En particulier, une borne fixée à 5 ne passe pas à l'échelle et le scénario nominal présenté figure 4.14 n'est donc pas couvert.

De plus, bien des chemins représentés par ce guide *polluent* l'effort de vérification (par exemple, ceux ne débutant pas par *powerOn*).

Pour adresser ces problèmes, on se propose de construire un autre guide xGDL, équivalent au premier, mais dont la forme se prête mieux aux traitements en aval.

Pour ce faire, on capture le scénario nominal présenté figure 4.14 dans une séquence xGDL :

$$NominalCase \;=\; powerOn;\; set;\; resume;\; disengage;\; powerOff$$

On défini alors un nouveau guide en composant ce scénario en parallèle avec la répétition non-bornée de l'alternative de l'ensemble des possibles :

$$CCSGuide' \;=\; NominalCase \parallel AnyInteraction*$$

Cette nouvelle forme du guide xGDL est équivalente, vis à vis des chemins représentés, à la précédente :

Théorème 3. ***CCS : Équivalence de langages.***

Avec langage(exp) le langage accepté par l'expression xGDL *exp :*

$$langage(AnyInteraction*) \;=\; langage(NominalCase \parallel AnyInteraction*)$$

Démonstration. **Par double inclusion :**

On veut montrer que :

1 : $langage(AnyInteraction*) \supset langage(NominalCase \parallel AnyInteraction*)$
2 : $langage(AnyInteraction*) \subset langage(NominalCase \parallel AnyInteraction*)$

But 1 : Trivial du fait que le vocabulaire de $NominalCase$ est inclus dans celui de $AnyInteraction$ et que $AnyInteraction*$ capture tous les termes exprimés avec ce vocabulaire.

But 2 : On veut montrer que :

$$\forall t,\;\; t \in langage(AnyInteraction*) \Rightarrow t \in langage(NominalCase \parallel AnyInteraction*)$$

Pour tous terme t reconnu par $AnyInteraction*$, le terme $t; NominalCase$ est reconnu par $NominalCase \parallel AnyInteraction*$. La sémantique de xGDL étant fermée par préfixe (si un terme est accepté alors ses préfixes le sont aussi), et t étant un préfixe de $t; NominalCase$, t est reconnu par $NominalCase \parallel AnyInteraction*$. □

De manière similaire au dépliage borné, il est alors possible de remplacer la répétition non-bornée par une répétition bornée :

$$CCSGuide^n \;=\; NominalCase \parallel AnyInteraction\{0,\; n\}$$

Pour $n = 0$ le scénario nominal est couvert. Pour chaque incrément une nouvelle interaction de l'utilisateur, parmi toutes celles possibles, est *injectée* dans ce scénario.

Cette nouvelle forme (elle aussi acyclique) présente plusieurs avantages. En particulier, qu'importe la borne n retenue :

— le scénario nominal est couvert ;

— tous les chemins comportent les interactions nécessaires à l'engagement du système ;

— il est possible de mettre en place une stratégie de division axée sur l'entrelacement induit par le parallélisme (i.e. en fonction d'où, dans le scénario nominal, sont *injectées* les interactions supplémentaires).

En conséquence, cette seconde forme offre une meilleure maîtrise de la couverture. Intuitivement, la borne nécessaire (n) pour garantir l'exhaustivité est plus petite que celle nécessaire dans le cas du dépliage borné de $AnyInteraction*$.

Similairement, si nécessaire pour la vérification, il est possible d'étendre l'approche à plusieurs scénarios nominaux simplement en remplaçant, dans le guide xGDL, $NominalCase$ par une alternative sur les scénarios retenus :

$$(NominalCase_0 \,\square\, ... \,\square\, NominalCase_i) \parallel AnyInteraction*$$

Il s'agit, la encore, d'une forme équivalente (vis à vis des langages acceptés) à $AnyInteraction*$. Elle offre un axe supplémentaire de division en fonction de l'alternative sur les scénarios nominaux.

Note sur la sûreté Une spécificité importante du système présenté ici est que la disponibilité du service n'est pas critique pour la sûreté. En d'autres mots, il est possible de désengager le régulateur à tous moment. Il s'agit d'une différence notable avec le Pacemaker traité section 4.3.

Mener la vérification via xGDL permet de fournir une formalisation claire des limites de la validation (comme le nombres d'interactions). Cet aspect peut alors être pris en compte par le modèle pour désengager le système lorsque l'exécution sort du cadre ainsi formalisé.

Il s'agit d'un autre axe exploitable pour contourner l'explosion de l'espace d'états. Cela n'est pas possible si le maintient du service est critique, comme dans le cas du Pacemaker.

Résultats En pratique, pour permettre le passage à l'échelle il a été nécessaire d'inclure l'horloge du système dans la fermeture du système (l'environnement) et d'étiqueter sa transition (*tick*, dénotant l'avancement du temps) pour permettre d'en borner le nombre d'occurrence [33]. Ci-dessous, la forme exhaustive du guide pour un scénario nominal $NominalCase$ donné :

$$CCSGuide \;=\; NominalCase \parallel AnyInteraction* \parallel tick*$$

La figure 4.16 présente une comparaison des résultats obtenus avec le model-checking *classique*, le Past-Free[ze] sans et avec la division automatique du guide xGDL.

L'approche par CaV permet, ici, d'explorer un espace d'états 4.5 fois plus grand que par model-checking *classique*.

FIGURE 4.16 – Cruise Control : synthèse des résultats d'exploration obtenus

Conclusion sur l'étude de cas Le régulateur de vitesse (CCS), traité cette section, illustre plusieurs possibilités offertes par l'approche par CaV pour faire reculer l'explosion de l'espace d'états.

En particulier, la composition du système avec un guide XGDL permet de raisonner et d'itérer sur ce dernier sans recourir à l'instrumentation du système (non-intrusif).

L'impact des choix effectués lors de la modélisation du guide exhaustif est illustré ($AnyInteraction*$ versus $NominalCase \parallel AnyInteraction*$). En faire varier la forme en amont permet d'obtenir un compromis satisfaisant entre la couverture et le passage à l'échelle.

La disponibilité du service n'est, ici, pas critique pour la sûreté. Il est donc possible de désengager le système à tout moment. Le cadre d'utilisation dans lequel le système a été validé peut être transmis en aval (pour le déploiement). Il est alors possible, par exemple, de générer des moniteurs pour désengager le système si celui-ci sort du cadre ainsi identifié.

4.5 Conclusion

Dans ce chapitre, l'approche proposée a été appliquée à trois cas d'étude présentant chacun des spécificités récurrentes dans le domaine de l'informatique embarqué.

LGS Le train d'atterrissage, traité section 4.2, illustre le cas où le système supporte l'injection de fautes et où la vérification porte sur leurs détections. Dans ce cadre, on a montré que l'approche par CaV avec XGDL permet la production d'unités de preuves spécialisées sans modifier le système en entrée. Il a été alors possible de raisonner indépendamment sur celles-ci et ainsi d'aboutir à la vérification de 16 parmi les 18 fautes possibles.

Pacemaker Le Pacemaker, traité section 4.3, illustre le cas d'un système générique. La diversité des paramètres lors de la mise en service couvrent un grand nombre de comportements différents. On a montré que l'approche par CaV avec XGDL offre un cadre adapté à la capture et la vérification de ces paramétrages.

CCS L'approche qui a été proposée pour le régulateur de vitesse, présentée section 4.4, exploite le fait que, dans ce cas, la disponibilité du service n'est pas critique pour la sûreté. Il est donc possible de désengager le système lorsqu'il sort du cadre dans lequel il a été validé. On a montré que XGDL permet, de manière non-intrusive au système, de faire évoluer la modélisation vers un compromis satisfaisant entre la couverture et le passage à l'échelle.

En synthèse des paragraphes précédents, ce chapitre a illustré l'approche par CaV avec xGDL sur trois cas d'étude différents. Pour chacun il a été possible d'exploiter leurs spécificités à travers les guides xGDL pour en permettre le passage à l'échelle. Tous les traitements réalisés sont non-intrusifs au système en entrée et se prêtent donc à une application conjointe avec d'autres techniques basées sur le model-checking [29, 12].

Chapitre 5

Conclusion et Perspectives

5.1 Synthèse des travaux

Les travaux présentés dans ce manuscrit s'inscrivent dans le cadre de la validation de système embarqués. Il s'agit d'une approche par model-checking spécialisée pour les systèmes dont les comportements à vérifier peuvent être clairement distingués de ceux de l'environnement. On peut distinguer trois niveaux de contributions :

— la formalisation de l'approche à travers la définition de xGDL, un langage dédié ;

— une algorithmique spécialisée incluant le dépliage borné d'un guide xGDL, le Split et le Past-Free[ze] ;

— l'application à des cas d'étude de dimension industrielle.

5.1.1 Formalisation

Le cadre formel proposé dans ce manuscrit est dédié à l'application de la CaV [20]. Il s'agit d'une généralisation des travaux précédents basés sur les contextes de preuve capturés à travers CDL [18, 21]. L'enjeu est ici de gagner en flexibilité et de renforcer la compatibilité avec d'autres approches de la littérature basées sur le model-checking.

Fonction d'étiquetage L'approche repose sur une hypothèse importante : il doit être possible de distinguer le système de son environnement. Cette distinction est réalisée par une fonction d'étiquetage des transitions du système sur $A_f \cup \{\tau\}$ avec :

— A_f l'alphabet d'actions observables correspondant aux interactions entre le système et l'environnement ;

— τ l'étiquette dénotant l'absence d'action observable.

Une transition étiquetée par $a \in A_f$ correspond alors à une interaction avec l'environnement, une transition étiquetée par τ correspond à une transition propre au système.

Cette capture des interactions est indépendante de leurs natures, levant ainsi la contrainte de modélisation via communications asynchrones imposée par CDL. De plus, elle suppose

une modélisation fermée en entrée, un point de convergence avec d'autres approches basées sur le model-checking.

Formalisme dédié Identifier les interactions avec l'environnement ne suffit pas pour permettre l'application de la CaV. Pour la vérification d'un ensemble donné d'exigences, il est nécessaire d'être capable d'exprimer des scénarios pertinents. Ceux-ci guideront l'exploration lors de la vérification.

Pour ce faire, ce manuscrit introduit un nouveau formalisme : xGDL. Ses actions atomiques sont les actions observables correspondantes aux interactions entre le système et son environnement (A_f). Sa sémantique couvre l'ensemble des opérateurs offerts par CDL (séquence, alternative, parallélisme). En sus, elle supporte la répétition non-bornée (levant ainsi la contrainte d'acyclicité) et les permutations.

La composition du système à vérifier et d'un guide xGDL est réalisée par composition synchrone des systèmes de transitions étiquetées (LTS) correspondants. Celui du système est le point d'entrée au model-checking *classique* augmenté de la fonction d'étiquetage sur $A_f \cup \{\tau\}$. Celui du guide xGDL est obtenu par compilation vers un automate fini déterministe (DFA).

5.1.2 Algorithmique

Dans ce manuscrit, plusieurs algorithmes spécialisés pour le traitement d'unités de preuve incluant un guide xGDL sont définis.

Dépliage borné Il s'agit de déplier un guide xGDL vers une forme acyclique (DAG) dont le langage est conservé pour tous les termes de taille égale ou inférieure à la borne spécifiée.

Les termes de taille supérieure ne sont donc plus inclus après dépliage. Il s'agit donc d'une sous-approximation du problème en entrée. Dans ce cadre, il est nécessaire de justifier de la couverture pour garantir la complétude de l'analyse. Cette justification peut être fournie au cas par cas.

Il s'agit d'une problématique similaire à celle du model-checking borné [11]. Le résultat de la composition du système avec le guide obtenu pour une borne n est équivalent au model-checking borné partiellement à n interactions de l'environnement. Il est donc possible d'exploiter les résultats dans ce domaine (ex : diamètre d'atteignabilité [27]) pour justifier de la couverture.

Le Split et le Past-Free[ze] décrits par la suite supposent un guide xGDL acyclique (DAG). Si cette hypothèse n'est pas respectée, et sous réserve de pouvoir justifier de la couverture, il est possible d'appliquer le dépliage borné en amont.

Split Cet algorithme exploite la structure du DAG composé avec le système pour la génération automatique de sous-problèmes. Valider l'ensemble des unités de preuve ainsi obtenues est équivalent à mener la vérification sur le problème initial.

Outre le fait que les sous-problèmes générés sont *plus petits*, leurs vérifications peuvent être spécialisées et menées indépendamment. Ce dernier point est développé dans les perspectives, section 5.2.1.

Past-Free[ze] Il s'agit d'une spécialisation de l'exploration de la composition d'un système avec un guide xGDL acyclique (DAG). L'espace des états atteignables est divisé en partitions indexées sur les sommets de ce DAG. Ces sommets sont ordonnés via un tri topologique. L'algorithme d'exploration *progresse* de partition en partition selon l'ordre ainsi défini. L'acyclicité et le tri topologique permettent d'*oublier* une partition lors du passage à la suivante.

Le Past-Free[ze] permet donc, au fur et à mesure, de libérer de la mémoire ces partitions appartenant au *passé* de l'exploration. La charge mémoire requise pour permettre à l'exploration de terminer en est d'autant réduite, adressant ainsi le problème de l'explosion de l'espace d'états intrinsèque au model-checking.

Il peut être possible de lever la contrainte d'acyclicité comme évoqué dans les perspectives, section 5.2.2

5.1.3 Applications et résultats

Il existe une grande diversité de systèmes embarqués. Cependant, on peut identifier des familles dont les caractéristiques peuvent être exploitées pour le passage à l'échelle de la vérification. Le cadre proposé dans ce manuscrit permet de capturer et de raisonner sur certaines d'entre elles.

Choix des cas d'étude Pour valider expérimentalement ce point, l'approche a été illustrée sur plusieurs cas d'étude voulus représentatifs de différentes familles :

— Pour le train d'atterrissage (LGS), la cible de la vérification est le composant responsable de la détection de faute. Il s'agit d'assurer que le pilote est tenu à jour d'éventuels dysfonctionnements.

— Le *Pacemaker* est représentatif des systèmes dont le comportement peut varier fortement d'une mise en service à une autre.

— Le régulateur de vitesse automobile (CCS) est un système d'assistance. Ses capacités mettent la sûreté en jeu. Cependant il est possible de désengager le système à tout moment. La disponibilité du service n'est donc pas critique.

Résultats Dans les trois cas, les résultats obtenus après exploration sont encourageants. En particulier, du point de vue de la modélisation pour la vérification :

— LGS : L'opérateur xGDL de permutations utilisé pour l'injection de faute peut être exploité par le Split pour une décomposition automatique du problème initial en plusieurs centaines d'unités de preuve. Parmi celles-ci, il est alors possible d'identifier clairement celles posant problème pour le passage à l'échelle et de spécialiser l'approche pour leurs traitements.

— *Pacemaker* : La phase d'initialisation à la mise en service est capturée par xGDL. Là encore il est alors possible d'appliquer le Split pour la génération automatique d'unités de preuve.

— *CCS* : La complexité du système est prohibitive pour le passage à l'échelle de la vérification. xGDL permet de raisonner sur les cas d'utilisation de manière itérative, et ce sans recourir à l'instrumentation du système (non-intrusif). On obtient ainsi un compromis satisfaisant entre la couverture et le passage à l'échelle. La disponibilité du service n'étant pas critique pour la sûreté, les hypothèses posées (et capturées par xGDL) peuvent alors être prises en compte lors du déploiement.

5.2 Perspectives

L'approche proposée dans ce manuscrit met l'accent sur la capture et le traitement des comportements de l'environnement pour le model-checking. Cette section liste deux perspectives de recherche s'inscrivant dans cet axe.

5.2.1 Split : formalisation et capture des stratégies

L'algorithme Split [17, 19, 16] présenté dans ce manuscrit permet la division automatique du guide xGDL. Il est paramétré par la profondeur dans le graphe dirigé acyclique (DAG) obtenu par dépliage borné. Son application permet de diviser le problème initial en plusieurs unités de preuve *plus petites* et adresse ainsi le problème de l'explosion de l'espace d'états.

Cependant, la combinatoire en entrée peut rester prohibitive pour le passage à l'échelle après division. De plus, il est difficile de raisonner sur ces unités de preuves générées automatiquement.

Dans ce manuscrit, des stratégies alternatives à celle basée sur la profondeur dans le DAG sont illustrées. Pour l'exemple, dans le cas du LGS, le guide xGDL d'origine est divisé selon la sous-partie des fautes considérées pour l'injection. Les unités de preuves ainsi générées portent alors un nombre restreint de fautes possibles. La vérification termine avec succès pour une majorité de ces sous-problèmes. Ceux ne passant pas à l'échelle peuvent être caractérisés par les fautes retenues et faire l'objet d'une étude approfondie exploitant cette connaissance.

Cette mise-en-œuvre du Split peut être automatique et paramétrée en fonction de la taille des sous-parties de fautes après division. Similairement à l'approche basée sur la profondeur, lorsque l'espace d'états reste prohibitif pour le passage à l'échelle, il est possible de diviser récursivement les unités de preuve *explosives*.

Comment généraliser l'algorithme et formaliser ces approches alternatives ? En particulier, comment capturer la cible (ex : l'injection de faute) et l'axe retenu (ex : taille des sous-parties) pour la division ? L'enjeu est de proposer une formalisation générale et flexible pour permettre la génération d'unité de preuves portant suffisamment d'information (sémantique) pour faire l'objet d'un traitement dédié.

5.2.2 Past-Free[ze] : extension aux guides cycliques

Pour pouvoir appliquer Past-Free[ze] avec un guide xGDL cyclique, il est pour le moment nécessaire de recourir à un dépliage borné. Une autre piste envisagée est de spécialiser l'algorithme pour la prise en compte native des cycles.

Intuition Il est possible d'identifier un ensemble de transitions à retirer dans le guide de vérification xGDL compilé pour revenir à une forme acyclique. Ces transitions sont alors ignorées pour la définition de la relation de précédence qui guide l'algorithme. Lors de la vérification, ces transitions sont prises en compte. Cependant, les partitions d'états indexés par les sommets cibles de ces transitions doivent être conservés en mémoire et ne peuvent donc pas être *oubliées*.

Bien que cette approche garantisse la terminaison, certains états de la composition peuvent être explorés et oubliés plusieurs fois. C'est à dire que, pour ces états, il existe des parents ne pouvant être oubliés, distincts mais indexés par le même sommet du guide, et atteints à une profondeur différente.

L'intuition présentée ici n'a pas été formellement validée, ni mise-en-œuvre. On peut aussi noter que plusieurs choix peuvent avoir un impact sur le passage à l'échelle : le tri topologique et l'ensemble de transitions à retirer pour la précédence. Est-il possible de définir des heuristiques pertinentes pour le passage à l'échelle de la vérification ?

Bibliographie

[1] Pacemaker system specification. Technical report, Boston Scientific, 2007.

[2] Karam Abd Elkader, Orna Grumberg, Corina S. Păsăreanu, and Sharon Shoham. Automated circular assume-guarantee reasoning with n-way decomposition and alphabet refinement. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 329–351, Cham, 2016. Springer International Publishing.

[3] Yamine Aït Ameur and Klaus-Dieter Schewe, editors. *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*. Springer, 2014.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[5] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification : Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[6] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[7] Frédéric Boniol, Philippe Dhaussy, Luka Le Roux, and Jean-Charles Roger. *Model-Based Analysis*, pages 157–184. Wiley, May 2013.

[8] Frédéric Boniol and Virginie Wiels. The landing gear system case study. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014 : The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 1–18. Springer International Publishing, 2014.

[9] Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors. *Context-Aware Verification of a Landing Gear System*, volume 433 of *Communications in Computer and Information Science*. Springer International Publishing, 2014.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 1020 states and beyond. *Information and Computation*, 98(2) :142 – 170, 1992.

[11] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1) :7–34, 2001.

[12] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5) :752–794, September 2003.

[13] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking : Algorithmic verification and debugging. *Commun. ACM*, 52(11) :74–84, November 2009.

[14] EdmundM. Clarke and E.Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[16] P. Dhaussy, Frédéric Boniol, and Jean-Charles Roger. Context aware model-checking for embedded software. In *Embedded System / Book 1*. Intech publisher, 2012.

[17] Philippe Dhaussy, Frédéric Boniol, and Jean-Charles Roger. Reducing state explosion with context modeling for model-checking. In *13th IEEE International High Assurance Systems Engineering Symposium (Hase'11)*, Boca Raton, USA, 2011.

[18] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, Amine Raji, Yves Le Traon, and Benoit Baudry. Formalisation de contextes et d'exigences pour la validation formelle de logiciels embarqués. *Technique et Science Informatiques*, 6/2012 :797–826, 2012.

[19] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Le Roux. Improving model checking with context modelling. *Advances in Software Engineering*, ID 547157 :13 pages, 2012.

[20] Philippe Dhaussy, Pierre-Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, and Benoit Baudry. Evaluating context descriptions and property definition patterns for software formal validation. In Bran Selic Andy Schuerr, editor, *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models'09)*, volume 5795, pages 438–452. Springer-Verlag, LNCS, 2009.

[21] Philippe Dhaussy, Jean-Charles Roger, Luka Le Roux, LabSticc ENSTA Bretagne, Brest France, and Frédéric Boniol. Context aware model exploration with obp tool to improve model-checking. *Embedded Real-Time Software and Systems (ERTS'12)*, 2012.

[22] Philippe Dhaussy and Ciprian Teodorov. Context-aware verification of a landing gear system. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014 : The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 52–65. Springer International Publishing, 2014.

[23] Patrick Farail, Pierre Gaufillet, Florent Peres, Jean-Paul Bodeveix, Mamoun Filali, Bernard Berthomieu, Saad Rodrigo, Francois Vernadat, Hubert Garavel, and Frédéric Lang. FIACRE : an intermediate language for model verification in the TOPCASED environment. In *European Congress on Embedded Real-Time Software (ERTS)*, Toulouse, january 2008. SEE.

[24] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2Nd International Workshop on Computer Aided Verification*, CAV '90, pages 176–185, London, UK, UK, 1991. Springer-Verlag.

[25] G.J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5) :279–295, 1997.

[26] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1) :60–65, March 2001.

[27] Daniel Kroening and Ofer Strichman. *Verification, Model Checking, and Abstract Interpretation : 4th International Conference, VMCAI 2003 NewYork, NY, USA, January 9–11, 2003 Proceedings*, chapter Efficient Computation of Recurrence Diameters, pages 298–309. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[28] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4) :66–73, March 2015.

[29] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 377–390, London, UK, UK, 1994. Springer-Verlag.

[30] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[31] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, Berlin, Heidelberg, 2000. Springer-Verlag.

[32] Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux. Environment-driven reachability for timed systems. *International Journal on Software Tools for Technology Transfer*, pages 1–17, 2015.

[33] Ciprian Teodorov, Luka Le Roux, and Philippe Dhaussy. Context-aware verification of a cruise-control system. *4th International Conference on Model & Data Engineering (MEDI'2014)*, 2014.

[34] Ciprian Teodorov, Luka Le Roux, Zoé Drey, and Philippe Dhaussy. Past-free[ze] reachability analysis : reaching further with dag-directed exhaustive state-space analysis. *Software Testing, Verification and Reliability*, 26(7) :516–542, 2016. stvr.1611.

[35] Ken Thompson. Programming techniques : Regular expression search algorithm. *Commun. ACM*, 11(6) :419–422, June 1968.

[36] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4) :297–322, Dec 1992.

[37] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I : Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK, 1998. Springer-Verlag.

Cette page est laissée intentionnellement blanche.

Validation par parties et non-intrusive de systèmes embarqués

Résumé La complexité grandissante des systèmes embarqués critiques pose de nombreux défis pour leur certification. Les techniques basées sur les méthodes formelles apportent des éléments des réponses. En particulier, le model-checking permet un processus de validation semi-automatique des exigences. Cependant, cette approche est limitée par l'explosion de l'espace d'états, lequel constitue un problème pour le passage à l'échelle dans un cadre industriel.

Les approches symboliques, la réduction d'ordre partiel, le model-checking borné sont autant de techniques visant à réduire l'impact de ce problème. De plus, il peut rester nécessaire de décomposer la vérification en plusieurs sous-parties. Cet axe méthodologique pose, lui-même, plusieurs questions. Comment garantir la complétude de l'analyse après division ? Comment, dans un processus itératif, assurer le maintien de la cohérence entre ces différentes sous-parties et vis-à-vis de la spécification ?

Les travaux présentés ici proposent un cadre formel pour la division du problème initial en sous-parties. Cette décomposition exploite la capacité de distinction entre les comportements à vérifier et les interactions avec l'environnement. Il s'agit de contraindre le système à un cadre d'utilisation pertinent au sous-ensemble des exigences considérées. La capture de l'environnement est réalisée à travers un formalisme basé sur les expressions régulières étendues nommé xGDL. En sus du gain méthodologique, des algorithmes spécialisés permettent d'aider le passage à l'échelle de la vérification par model-checking.

La mise en pratique est illustrée sur trois cas d'étude : un train d'atterrissage (avionique), un pacemaker (santé) et un régulateur de vitesse (automobile).

Mots-clés model-checking, système embarqués, modélisation de l'environnement, validation par partie

---

Critical embedded system verification, a non-intrusive approach to divide the initial challenge into a sound set of smaller ones

Abstract Critical embedded systems have to be certified prior to their deployment to fulfill legal requirements, if only that. And formal methods are a mandatory step to this legal process. It is widely admitted that each case study requires a specialized approach to its formal verification. This thesis focuses on explicit model-checking. While it requires few human interactions and can be automated more easily so than, say, theorem proving, it also exponentially suffers from the growing complexity of critical embedded systems. This is a major drawback to its scalability in a practical use.

Symbolic approaches, partial order reduction, bounded model-checking all address this particular problem. The aim is to reduce the memory needs of the algorithm so it scales better. However, it is often still not enough and it might be necessary to split the initial task into several smaller ones. This process raises a few challenges, from the production to the soundness of this set of smaller task against the specifications, including maintainability and the ability to further divide them.

The main contribution of this work is to provide a formal framework to allow for the division of a verification task. It assumes the ability to identify the interactions between the system, target of the verification, and its environment. *Use cases* of the system are then defined as interaction scenarios. Those are expressed via xGDL, a proposed formalism based on regular expressions and extended. From this starting point, a set of sound smaller task can be produced and maintained. Plus, specialized model-checking algorithms further help with scalability.

The approach has been experimented and illustrated over three case studies : a landing gear system, a pacemaker and a cruise-control.

Keywords model-checking, critical embedded systems, context aware verification

---

Luka Le Roux UBO, ENSTA Bretagne