Accélération matérielle de la vérification de sûreté et vivacité sur des architectures reconfigurables

Émilien Fournier

DOCTORAT BRETAGNE LOIRE / MATHSTIC

ENSTA Bretagne

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
DE TECHNIQUES AVANCÉES BRETAGNE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : « *Informatique* »

Par
Emilien FOURNIER

« Accélération matérielle de la vérification de sûreté et vivacité sur des architectures reconfigurables »

Thèse présentée et soutenue à « Brest », le « 05 Juillet 2022 »
Unité de recherche : « Lab-STICC UMR CNRS 6285 »

Rapporteurs avant soutenance :

Frédéric BONIOL	Maître de Recherche HDR, ONERA, DTIS
Bertrand GRANADO	Professeur des Universités, Sorbone Université, LIP6

Composition du Jury :

Président :	Bertrand GRANADO	Professeur des Universités, Sorbone Université, LIP6
Rapporteurs :	Bertrand GRANADO	Professeur des Universités, Sorbone Université, LIP6
	Frédéric BONIOL	Maître de Recherche HDR, ONERA, DTIS
Examinateurs :	Virginie FRESSE	Maître de Conférences HDR, Télécom Saint Etienne, Lab. Hubert Curien
	Cécile BELLEUDY	Maître de Conférences HDR, Université de Nice, LEAT
Directeur de thèse :	Loïc LAGADEC	Professeur, HDR, ENSTA Bretagne, Lab-STICC
Encadrant de thèse :	Ciprian TEODOROV	Enseignant-Chercheur, ENSTA Bretagne, Lab-STICC

Remerciements

Je tiens à remercier tout d'abord mon encadrant, Ciprian Teodorov, à la fois pour la patience dont il a sû faire preuve, le soutien qu'il a pu m'offrir, mais également le savoir qu'il m'a transmis. J'ai pu ainsi, à ses côtés, m'épanouir techniquement et scientifiquement au cours de ces belles années.

J'adresse également toute ma reconnaissance à Loïc Lagagec, mon directeur de thèse, qui a toujours été à l'écoute pour m'orienter, et me conseiller, et définir les grands axes de mes travaux de recherche.

Mes remerciements vont également à Bertrand Granado et Frédéric Boniol, qui m'ont fait l'honneur de prendre le temps d'évaluer mes travaux en tant que rapporteurs. Je remercie Virginie Fresse et Cécile Belleudy d'avoir accepté de faire parti du jury. Je souhaite ensuite remercier Matthias Brun et Sebastien Pillement pour avoir participé à mon comité de suivi de thèse.

D'une manière plus générale, je veux remercier également mes collègues des équipes CS et SL, avec qui j'ai eu la chance de travailler, ainsi que plus généralement l'ensemble des personnels de l'ENSTA Bretagne qui m'ont accompagné dans mes activités d'enseignement et de recherche.

Malgré cet environnement favorable, l'aboutissement de ce travail de recherche n'aurait pas pû aboutir sans le soutien indéfectible de mes amis et de ma famille, qui m'ont guidé, soutenu, supporté - souffert ? - pendant ces années aussi passionnantes que difficiles. C'est donc à vous que j'adresserai le mot de la fin : Cette réussite est aussi la votre, Merci !

Table of Contents

Introduction

Contexte

Les systèmes informatiques sont aujourd'hui omniprésents dans notre société, répondant à une multitude d'applications, depuis l'information jusqu'au transport et à la médecine. Ces systèmes sont de plus en plus complexes, offrant de plus en plus de fonctionnalités. Leur caractère interconnecté a introduit une modification drastique du fonctionnement de notre société. Notre dépendance à ces systèmes augmentant, le besoin de performances et de fiabilité est devenu un enjeu majeur, ces systèmes étant passés de l'ère des prototypes, au temps de l'industrialisation massive. Dans le même temps, cette croissance forte de la demande induit une réduction des délais alloués au développement, augmentant encore les contraintes sur ce processus.

Outre cette complexité de développement, les conséquences d'une faute peuvent être parfois dramatiques, et plusieurs erreurs logicielles sont restées dans les mémoires[1]. Un exemple parlant est celui du bug présent dans l'unité de division des entiers à virgule flottante de l'Intel Pentium, au début des années 90, qui a induit une perte estimée à 475 millions de dollars pour l'entreprise. Un autre est celui de l'explosion de la fusée Ariane 5, quelques dizaines de secondes après son premier lancement.

Pour maitriser cette complexité, les phases de vérification et de validation, correspondant respectivement à l'assurance de fiabilité et de fonctionnalité d'un système, ont fait l'objet d'un effort de recherche important, tant en termes d'outillage que de méthodologie. Parmi ces travaux, les méthodes formelles occupent une place particulière. Ces méthodes permettent en effet de s'affranchir du risque d'incomplétude des tests, en fournissant une preuve, au sens mathématique du terme, que le système respecte ses spécifications. Pour ce faire, il est d'abord nécessaire de formaliser le système de manière précise et non ambiguë, en s'affranchissant des obscurités du langage naturel. Le système est alors modélisé de manière intensionnelle - sous la forme d'un graphe implicite : Le système et son environnement d'exécution sont représentés sous la forme d'une relation de transition implicite, dont les états représentent l'ensemble des données nécessaires à l'évolution du système.

Une fois le système formalisé, débute la phase de vérification. Les outils formels sont nombreux pour l'effectuer, mais le Model-Checking constitue ici un premier choix par son

caractère automatisé, et la variété de ses cadres d'application. Cette technique de preuve est très utilisée dans un contexte industriel, en particulier dans le cadre de la conception électronique, ou de la vérification de protocoles de communication. Clarke propose par exemple l'utilisation du Model-Checking pour la vérification d'unités arithmétiques et logiques, reproduisant le bug FDIV de l'Intel Pentium évoqué précédemment[2]. Newcombe décrit plus récemment son utilisation pour la vérification de systèmes distribués industriels[3]. Cette technique fournit une preuve par contre-exemple : L'espace des états atteignables du système est énuméré exhaustivement, en évaluant les propriétés de manière synchronisée. La terminaison de la preuve garantit ainsi l'absence des comportements redoutés.

Cette technique se heurte malheureusement à un problème de taille dans le cadre de la vérification de systèmes complexes : ajouter une variable booléenne double le nombre d'états potentiellement atteignables. Un effort de recherche très important a été déployé pour contrer cette caractéristique intrinsèque à la méthode, au travers de techniques d'abstraction, ou d'optimisation algorithmique[4][5]. Cependant, le caractère exponentiel de l'explosion combinatoire en fait un enjeu toujours critique, car dans le cas de problèmes toujours plus complexes, la taille de l'espace d'état ne permet pas de terminer la preuve.

Plusieurs approches sont mobilisables pour contrecarrer ce problème, que l'on peut catégoriser en fonction de leur caractère sémantique ou algorithmique. Sur le plan sémantique, l'objectif est de réduire la complexité du modèle vérifié abstrayant le système, ne formalisant dans le modèle vérifié que les comportements relatifs à la preuve en cours tout en restant complet. De manière orthogonale mais complémentaire, des techniques algorithmiques visent à améliorer la scalabilité et la performance des outils. De manière générale, le problème adressé par la communauté dans ce cadre est la vérification de modèles plus grands pour un même budget en temps et en mémoire.

Dans ce cadre, l'accélération matérielle constitue une opportunité prometteuse. Des travaux de recherche récents revendiquent en effet une amélioration importante tant des performances que de la scalabilité pour la vérification swarm pour la sûreté[6], technique partielle basée sur l'exécution massive d'analyses partielles randomisées, produisant un facteur d'accélération proche de trois ordres de grandeur par rapport à une implémentation logicielle performante[7]. Ces travaux sont cependant à un stade exploratoire, et reposent sur une évaluation sur un cas d'étude unique, évalué sur une architecture monolithique dédiée à l'exécution d'un algorithme précis.

Questions de recherche

Dans ce contexte, l'enjeu global adressé ici est l'exploration des opportunités de l'accélération matérielle du Model-Checking sur des architectures reconfigurables.

Cette problématique se traduit par plusieurs aspects. Un premier est illustré par l'état actuel de la littérature sur le sujet. Malgré des gains de performance supérieurs de plusieurs ordres de grandeurs aux meilleures implémentations parallèles logicielles[6], seuls deux travaux - espacés d'une décennie - proposent l'exploitation de FPGAs pour l'accélération du Model-Checking. Nous interprétons celà par plusieurs fondements. Le premier est de toute évidence la complexité de conception d'une telle solution, cette complexité se traduit également dans les résultats proposés, difficiles à comparer et à mettre en perspective, élément essentiel pour permettre une démarche de recherche incrémentale et structurée sur le sujet. En outre, le caractère exploratoire actuel des travaux sur le sujet rendent difficile la perspective de l'exploitation à court terme d'une telle technologie dans un outil de vérification.

Un autre axe de complexité est introduit par la multiplicité des algorithmes et des langages de spécification existants, nécessaires pour adapter les choix effectués dans le plan de vérification d'un système aux caractéristiques du problème étudié. Ces choix se traduisent en pratique en l'utilisation d'un langage de spécification adapté au niveau d'abstraction du modèle étudié, ainsi que par l'utilisation des algorithmes adaptés aux contraintes en termes de ressources mémoires et temporelles.

Ces deux axes se rejoignent par le besoin de réduction de la complexité d'un tel développement. Cette complexité provient d'un manque de cadre architectural et méthodologique pour la conception de tels accélérateurs, y répondre constitue une première question adressée par ce manuscrit.

Parmi les applications de l'accélération matérielle pour le Model-Checking, la vérification swarm se détache en termes de performance. Cette technique s'accommode en effet d'un faible volume mémoire et bénéficie d'un double niveau de parallélisme - de tâches, mais également à l'échelle locale d'une tâche de vérification - en faisant un candidat idéal pour l'accélération matérielle. Cependant, le caractère indépendant et randomisé des taches de vérification induit un recouvrement entre les sous-ensembles de l'espace d'état parcouru par chacune d'entre elles. Ce recouvrement induit une forte déperdition de l'effort de vérification, certaines sections de l'espace d'état étant parcourues plusieurs fois. L'enjeu scientifique considéré ici est l'augmentation de l'efficacité des algorithmes de vérification swarm, par la réduction de cette duplication de travail. Un second enjeu allant

de pair pour l'accroissement des performances de la vérification swarm est la conception d'un coeur de vérification correspondant au cadre abstrait proposé, mais optimisé pour la performance.

La spécification d'un système se construit autour de deux classes de propriétés, énoncées par Lamport en 77 : sûreté et vivacité[8]. Cette catégorisation correspond respectivement à la formalisation de comportements proscrits et prescrits du système, mais définit également l'expressivité du formalisme nécessaire pour la vérification. Ce formalisme va de pair avec les algorithmes pour les vérifier, pouvant se réduire à une analyse d'atteignabilité dans le premier cas, mais nécessitant une détection de cycles dans le deuxième.

A ce jour, les travaux portant sur l'accélération matérielle du Model-Checking se limitent au support de la vérification de propriétés de sûreté. Une autre question de recherche étudiée ici est comment concevoir une architecture de vérification matérielle pour la vérification d'automates de Büchi, formalisme permettant la vérification de propriétés de vicacité. Une question en découlant immédiatement est celle de la reproductibilité des performances obtenues pour la vérification de sûreté dans le cadre de la vérification de propriétés de vivacité.

Enfin, un enjeu plus ouvert est celui de la réutilisation des sémantiques existantes. L'introduction d'une nouvelle architecture de calcul pour la vérification pose en effet la question de l'utilisabilité des langages conçus pour les plateformes de calcul actuelles. Il s'agit donc ici de s'intéresser à la réutilisation de ces langages de spécification existants pour la vérification accélérée en matériel.

Contributions

La première contribution présentée dans ce manuscrit pour apporter des éléments de réponse aux questions de recherche énoncées ci-avant est le framework de vérification matériel Menhir. Un élément structurant de cette architecture de vérification matérielle est l'introduction d'une interface langage générique, permettant une division forte du problème de conception entre le coeur algorithmique et un *model-frontend*, encapsulant la sémantique du modèle vérifié et de ses propriétés associées. Son architecture générique et modulaire permet une forte variabilité tant dans la représentation de l'information stockée, que dans la discipline de parcours de l'espace d'état du modèle.

La généricité de cette architecture est illustrée au travers de l'implémentation de 6 algorithmes de vérification, formant un dégradé depuis la vérification exhaustive vers des analyses partielles de l'espace d'état. En outre, l'isolation forte entre la sémantique du

modèle et l'algorithme de vérification permet l'intégration - transparente pour le coeur algorithmique - de trois langages d'entrée couvrant la vérification de modèles exprimés dans un formalisme industriel comme UML, jusqu'à des modèles de très bas niveau, spécifiés directement en RTL.

Dans un second temps, cette généricité est exploitée pour adresser le problème d'efficacité des algorithmes de vérification swarm au travers d'une exploration de l'espace de conception algorithmique. Cette étude théorique se traduit par la proposition d'un nouvel algorithme randomisé de vérification swarm, minimisant à la fois le recouvrement des analyses et l'utilisation mémoire induite pour d'adapter aux contraintes d'implémentation des architectures reconfigurables.

Dans le but de maximiser la performance globale de la vérification swarm, pour laquelle l'efficacité va de pair avec la performance brute de la plateforme d'exécution. L'architecture de vérification distribuée Carnac adresse ces deux points, implémentant l'algorithme de vérification identifié au sein d'un coeur de vérification qui constitue une réinterprétation de l'approche générique Menhir, spécialisée pour la performance. La structure distribuée de Carnac contribue de plus à la scalabilité de l'approche, permettant l'intégration d'un grand nombre de coeurs de vérification sur des FPGAs multi-die.

Le troisième enjeu identifié est constitué par la vérification des propriétés de vivacité. Un élément de réponse est apporté ici par la proposition d'un algorithme swarm pour la vérification de propriétés de sûreté et de vivacité encodées sous la forme d'automates de Büchi, adapté aux contraintes imposées par une implémentation matérielle. Une preuve de faisabilité est ensuite constituée par l'implémentation du coeur de vérification Dolmen pour la détection de cycles d'acceptation dans les automates de Büchi, réorientation du coeur de vérification Carnac, illustrant ainsi sa généralité.

D'une manière transverse, plusieurs éléments de réponse sont apportés sur le plan de la réutilisation sémantique : Le support des langages UML et DVE par le moteur de vérification Menhir en est un, démontrant la faisabilité d'une approche hybride FPGA-CPU. Les performances potentielles de cette approche restent cependant un sujet ouvert. Une autre solution potentielle est explorée, sous la forme d'un outil de génération de circuit à partir du langage de spécification DVE. Cette approche atteint les attendus en termes de performance, mais l'utilisation importante des ressources doit faire l'objet d'améliorations.

Plan du manuscrit

Le chapitre 1 présente des éléments de l'état de l'art de la vérification par Model-Checking. Après avoir exposé les principes de cette technique de vérification, nous proposons une formulation abstraite des algorithmes de vérification de sûreté sous la forme d'une spécification formelle. Enfin, l'enjeu de performance des outils est discuté, présentant des travaux existants visant la parallélisation du Model-Checking sur les architectures de calcul modernes.

Cette spécification est ensuite reformulée dans le chapitre 2 pour construire une architecture générique abstraite d'un moteur de vérification matériel, et identifier les dépendances de données pour cet ensemble d'algorithmes. Cette abstraction est exploitée comme guide de conception pour l'architecture du moteur de vérification matérielle Menhir, dont la structure modulaire se veut agnostique tant aux algorithmes implémentés qu'aux sémantiques vérifiées. Cette généricité est illustrée par l'implémentation de 6 algorithmes de vérification pour la sûreté, et de trois sémantiques, proposant un dégradé d'abstraction depuis des langages de spécification industriels comme UML, jusqu'à des implémentations dédiées exprimées en RTL.

Le chapitre 3 adresse le problème de l'efficacité des algorithmes swarm, en exploitant la variabilité algorithmique exposée dans le chapitre précédent pour conduire une exploration de l'espace de conception algorithmique qui vise à minimiser le recouvrement entre les tâches de vérification, et maximisant ainsi la couverture à effort de calcul constant. Cette étude théorique mène à l'identification d'un nouvel algorithme de vérification swarm, intégré à l'architecture de vérification Carnac sous la forme d'un coeur dédié, réinterprétation de l'approche générique Menhir visant à maximiser la performance à l'exécution. L'intégration de ce coeur au sein d'une architecture scalable, conçue réduire les contraintes de localité des données propres aux FPGA multi-die. L'analyse théorique effectuée sur l'efficacité algorithmique est consolidée par une évaluation expérimentale sur un cas d'étude de référence pour la vérification swarm matérielle.

Pour finir, la vérification de propriétés de vivacité est adressée dans le chapitre 4. Dans ce cadre, nous proposons un algorithme swarm pour la vérification de propriétés de sûreté et de vivacité exprimées sous la forme d'automates de Büchi. Cet algorithme est implémenté sous la forme du coeur de vérification Dolmen, qui, en couplant deux coeurs de vérification proposés précédemment, effectue une détection de cycles d'acceptation. Outre l'illustration de la généralité de l'architecture du coeur Carnac proposée précédemment, cette implémentation permet également de confirmer l'analyse théorique effectuée pour

contruire l'algorithme, au travers d'une évaluation effectuée sur un ensemble de modèles représentatifs des cas d'études classiques adressables par Model-Checking.

Le dernier chapitre conclue ce manuscrit, et présente des perspectives pour des travaux futurs selon deux axes principaux : sémantique et algorithmique.

Liste des publications

[1] E. Fournier, C. Teodorov et L. Lagadec, « Carnac : Algorithm Variability for Fast Swarm Model-Checking on FPGA », in *2021 31th International Conference on Field Programmable Logic and Applications (FPL'21)*, août 2021, p. 185-189. doi : 10.1109/FPL53798.2021.00038.

[2] E. Fournier, C. Teodorov et L. Lagadec, « Menhir : Generic High-Speed FPGA Model-Checker », in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, p. 65-72. doi : 10.1109/DSD51259.2020.00022.

[3] E. Fournier, C. Teodorov et L. Lagadec, « Dolmen : FPGA Swarm for Safety and Liveness Verification », in *2022 Design, Automation and Test in Europe Conference Exhibition (DATE'22)*, 2022.

Chapitre 1

Etat de l'Art

1.1 Introduction

La phase de vérification est une composantes les plus importantes du processus de conception d'un système. De plus, la complexité croissante de ces derniers conduit à une augmentation drastique tant de la criticité, que de la probabilité d'occurrence d'erreurs. Pour détecter et corriger ces erreurs, différents types d'analyse peuvent être mises en oeuvre. Certaines, dites formelles, visent à fournir une preuve au sens mathématique du terme qu'un système - ou une abstraction de ce dernier - satisfait certaines propriétés, qui modélisent sous une forme précise et non ambiguë les spécifications à respecter par le système. Parmi les techniques de vérification formelles, ce travail aborde le Model-Checking, technique de preuve par contre-exemple automatique. Cette technique effectue une recherche de contre-exemple sur l'ensemble des états atteignables d'un système, permettant ainsi de garantir leur absence.

Ce chapitre présente dans un premier temps le formalisme utilisé pour la vérification par Model-Checking, puis propose une vue abstraite des algorithmes de vérification pour la sûreté, sous la forme d'une spécification. Enfin, l'enjeu de la performance des outils de vérification est discuté par une description des approches proposées pour l'implémenation d'outils de vérification parallèles.

1.2 Model-Checking

Après avoir défini la formalisation du problème de vérification utilisée dans le cadre d'une preuve par Model-Checking, cette section discute de l'expression des propriétés vérifiées, avant d'introduire des techniques permettant d'accroitre la scalabilité de la méthode.

Pour plus de détails, le lecteur peut se référer à l'ouvrage *Principles of Model-Checking*[1].

1.2.1 Principes du Model-Checking

Le Model-Checking est une technique de vérification qui a été proposé de manière simultanée et indépendante au début des années 80 par les travaux de Clarke et Emerson [9] et ceux de Sifakis et Queille [10].

Cette approche constitue une technique de preuve générique et automatisée, basée sur l'analyse systématique de la validité des propriétés vérifiées sur l'ensemble des états atteignables d'un système. Cet ensemble est désigné par le terme *espace d'état*. Son applicabilité est basée sur l'hypothèse que le système vérifié, qui se présente sous la forme d'une relation de transition, peut être énuméré exhaustivement.

Cette hypothèse est vérifiée naturellement dans le cadre de la vérification d'un système informatique ou numérique concret. Cependant, dans le cas d'un système complexe, élaborer la preuve directement est souvent une tâche trop complexe, voire impossible en pratique. Ainsi, la preuve est construite - en général - non pas sur le système lui-même, ie son implémentation, mais sur une abstraction de ce dernier. L'abstraction d'un système permet de réduire la taille de l'espace d'état à vérifier, en construisant un *modèle* abstrait, cachant certains détails d'implémentations. Cette abstraction n'est pas toujours réalisée, et la vérification est parfois effectuée directement sur le programme implémenté [11], cependant, dans le vocabulaire associé au Model-Checking, le terme *Modèle* est utilisé pour désigner l'objet de la vérification.

Le *Modèle* vérifié prends la forme d'un système de transitions. Les noeuds de ce graphe représentent les différents états du système, et les arcs - ou transitions - désignent les pas d'exécutions effectués pour faire évoluer le système d'un noeud à un autre. L'état d'un système doit être compris comme l'ensemble de ses variables, ainsi que l'endroit du programme où on est arrivé (*program counter*). Ce dernier contient l'ensemble de l'information suffisante pour restaurer l'exécution du programme dans les conditions exactes dans lesquelles ce dernier a été interrompu. Dans la suite, le terme configuration désigne l'état du système vérifié et de son environnement.

Aussi, le modèle vérifié peut être interprété comme un simple graphe, contenant des noeuds, et des transitions. Ce graphe possède un, ou plusieurs états initiaux, représentant les états d'initialisation possibles du système. Ce graphe n'est pas nécessairement déterministe. Cette idée peut paraître contre-intuitive, dans la mesure où le système vérifié peut être un programme réel, dont l'exécution est déterministe. Cependant, dans le cadre d'une tâche de vérification par Model-Checking, le système vérifié est dit *fermé*, et ne possède donc ni entrées ni sorties. Dans le cas de la vérification d'un programme possédant

des entrées-sorties, la vérification porte sur le système évoluant dans son environnement d'exécution, qui produit les entrées, et lit les sorties. Une conséquence directe, intrinsèque à la méthode, est que la vérification par Model-Checking ne vérifie le système que *relativement à son environnement*. Dit autrement, le programme n'est vérifié qu'au regard des sollicitations décrites dans cet environnement[12].

Les propriétés vérifiées sont exprimées dans des langages de spécification de haut niveau. De multiples langages peuvent être utilisés, suivant le besoin d'expressivité de la tâche de vérification en question. Les langages de spécification sont basés sur des formalismes mathématiques plus expressifs que les expressions booléennes, comme la logique temporelle linéaire *LTL*, ou la logique temporelle arborescente *CTL*. Dans un contexte industriel, les langages utilisés pour la vérification combinent parfois plusieurs logiques pour permettre la couverture d'un maximum de cas d'usage : on peut citer les langages PSL et SVA, très utilisés pour la vérification de circuits [13][14].

Le langage utilisé pour l'expression des propriétés n'est pas interprété directement. En amont d'une tâche de vérification, les propriétés sont reformulées sous la forme d'un automate fini, de manière homogène à la représentation du modèle vérifié. Cet automate représente la négation de la propriété à vérifier sur le modèle.

Aussi, les mots acceptés par l'automate représentant la propriété correspondent à des contre-exemples de la propriété vérifiée.

Le problème de Model-Checking est formalisé de la manière suivante : Soit un modèle M décrit sous la forme d'un automate fini non déterministe, et une formule à vérifier f exprimée dans une logique temporelle. La vérification s'effectue de la manière suivante : [15]

— Construction de l'automate fini non déterministe représentant la négation de la formule *f*. Un mot accepté par cet automate corresponds alors à un mot qui met en défaut la propriété. On note cet automate $A_{\neg f}$.

— Composition synchrone de l'automate représentant le modèle M et de la propriété $A_{\neg f}$. Un mot accepté par l'automate résultat est alors accepté simultanément par le modèle et la propriété.

— Vérification que l'automate résultat n'est pas vide.

Cette formulation du problème de Model-Checking se veut générique et didactique. Cependant, la construction de l'automate produit précédent la vérification nécessite un volume mémoire important pour stocker cet automate. Edmund Clarke, l'un des initiateurs du Model-Checking, fait remarquer à ce sujet que la composition asynchrone de n

processus, chacun possédant m états, peut avoir dans le pire des cas m^n états [5]. En pratique, l'approche moderne consiste plutôt à effectuer la composition à la volée, pendant l'exploration, permettant d'éviter cet écueil, ce qui permet de conclure d'une violation des spécifications avant la construction complète du produit. Néanmoins, si la propriété n'est pas violée, le parcours de l'espace d'état reste exhaustif.

1.2.2 Catégories de propriétés

Le Model-Checking est une technique de vérification basée sur la vérification de propriétés, représentant des exigences que le programe doit respecter. Cette approche présente l'intérêt d'être à la fois abstraite et modulaire. Son caractère abstrait se traduit par le fait que la spécification est traduite en une liste de propriétés indépendantes les unes des autres, sans avoir à se soucier de comment elles intéragissent entre elles, évitant ainsi de sur-spécifier le système. Son caractère modulaire se traduit par le fait qu'une liste de propriétés, qui traduit la spécification, peut aisément être modifiée par l'ajout, ou le retrait de l'une d'entre elles.

Cependant, ce caractère à la foit abstrait et modulaire de la spécification de systèmes par la définition de propriétés formelles présente un risque de sous-spécification. Une question classique se posant lors de la spécification d'un système est sa complétude : La liste de propriétés est-elle suffisante pour traduire la spécification ? Un exemple simple de sous-spécification, proposé par Manna en 1990, est celui de la spécification d'un algorithme d'exclusion mutuelle. Une spécification faussée consisterait à s'assurer que les deux processus n'accèdent pas à la section critique de manière simultanée, sans préciser la nécessité qu'un des processus accède au moins une fois à la section critique [16]. Une implémentation fausse, mais qui respecterait la spécification, serait alors d'implémenter un algorithme dans lequel aucun des deux processus n'accède jamais à la section critique.

Une solution partielle, proposée par Lamport en 1977[17], à ce problème de sous-spécification consiste à classifier les propriétés rôle dans l'élaboration de la spécification, selon deux catégories : Sûreté et Vivacité . Cette catégorisation a depuis été raffinée, par Manna et Pnueli, qui proposent une hiérarchie des propriétés de vivacité [16]. Cependant, la classification de Lamport sera utilisée dans la suite. Bien que moins fine, elle présente l'intérêt de correspondre aux besoins algorithmiques pour les vérifier.

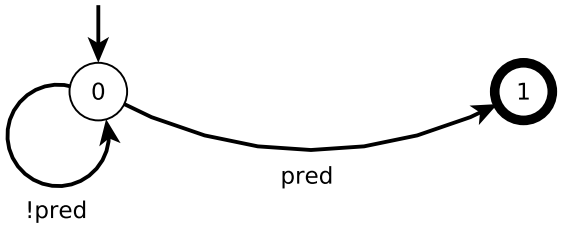


FIGURE 1.1 – NFA représentant une propriété portant sur un état - l'état 1 est un état d'acceptation

Propriétés de sûreté :

Une première classe de propriétés, dites de sureté, visent à exprimer des comportements non désirés du système. Elles sont formalisées sous la forme de prédicats, portant sur un chemin fini dans l'espace d'état du système. Elles sont modélisées sous formes d'automates finis non déterministes, dont certains états sont définis comme "acceptants", ie, caractérisent la fin d'un mot reconnu par l'automate. La section précédente - 1.2.1 - formalise le problème de Model-Checking comme une tâche vérifiant que le langage résultant de la composition synchrone du modèle et de la propriété vérifiée est vide. Aussi, pour vérifier des propriétés de sureté, il suffit de détecter l'absence d'états d'acceptation dans l'automate produit par la composition de la propriété avec le système. Les assertions portant sur un état, vérifiant qu'un état seul respecte un prédicat, sont une sous-classe triviale de propriétés de sûreté, dont le NFA est représenté dans la figure 1.1 ci-après.

Propriétés de vivacité

Pour construire une spécification complète d'un programme, après avoir modélisé les comportements non désirés, il est nécessaire de modéliser les comportements désirés du système : Un programme qui ne fais rien ne contient pas de fautes. Pour cela, on utilise un second type de propriétés, dites de vivacité. Ces propriétés peuvent se comprendre comme *"Le système fera toujours ... "*. Pour vérifier une telle propriété par contre exemple, il est nécessaire de prouver que *"le système ne fera jamais ... "*. Une telle assertion n'est pas vérifiable sur un chemin fini : Si un chemin fini est *"sur"*, une faute pourra se trouver *"plus tard"* dans l'exécution. Aussi, il est nécessaire de raisonner sur des chemins infinis, au sein d'un graphe fini, qui ne peuvent alors n'être que des cycles.

Plus précisément, ces propriétés sont modélisées via des automates dont la structure est similaire aux NFA, mais dont l'interprétation est très différente. Les automates de Büchi, utilisés pour modéliser les propriétés de vivacité, sont des automates non déterministes, comportant des états particuliers, désignés également comme états d'acceptation. Cependant, dans le cas des automates de Büchi, un mot n'est accepté que s'il contient un état d'acceptation infiniment souvent. Une des hypothèses du Model-Checking étant que l'espace d'état est fini, un tel état doit nécessairement faire partie d'un cycle.

Aussi, dans une formulation intuitive, la construction d'un contre-exemple pour des propriétés de vivacité consiste à détecter des cycles, contenant un état d'acceptation. On peut ainsi reformuler le problème de conception d'une architecture de vérification de propriétés de vivacité, en un problème de conception d'une architecture de détection de cycles dans un graphe.

1.2.3 Vérification partielle

Le Model-Checking est une technique de vérification automatisée basée sur une analyse systématique de l'espace-d'état correspondant à l'exécution du modèle sous-jacent. Cette technique est basée sur une énumération exhaustive de la relation de transition qui représente le modèle composé avec sa propriété. L'espace d'état est un graphe potentiellement cyclique. Aussi, son parcours nécessite d'accumuler les états parcourus au cours de l'exploration pour assurer la terminaison de l'algorithme.

Ce stockage de l'espace d'état est potentiellement très couteux en ressources mémoire. Pour des modèles complexes, il est fréquent que la preuve ne puisse pas terminer, par manque de mémoire disponible, le besoin mémoire étant exponentiel en le nombre de variables représentées dans le système (dans le pire des cas). Cette caractéristique intrinsèque à la méthode est souvent désignée par le terme d'explosion combinatoire, *state-space explosion* en anglais[5].

La recherche de solutions pour repousser ce problème d'explosion combinatoire a été un moteur majeur de la recherche sur le Model-Checking [5]. Des travaux riches ont été conduits dans le but de réduire le besoin mémoire d'une tâche de Model-Checking par des techniques d'abstraction automatique [18], symboliques [19], ou exploitant l'indépendance de certains évènements pour éviter l'exploration de certaines transitions [4].

Approches partielles : bitstate hashing, hash-compaction

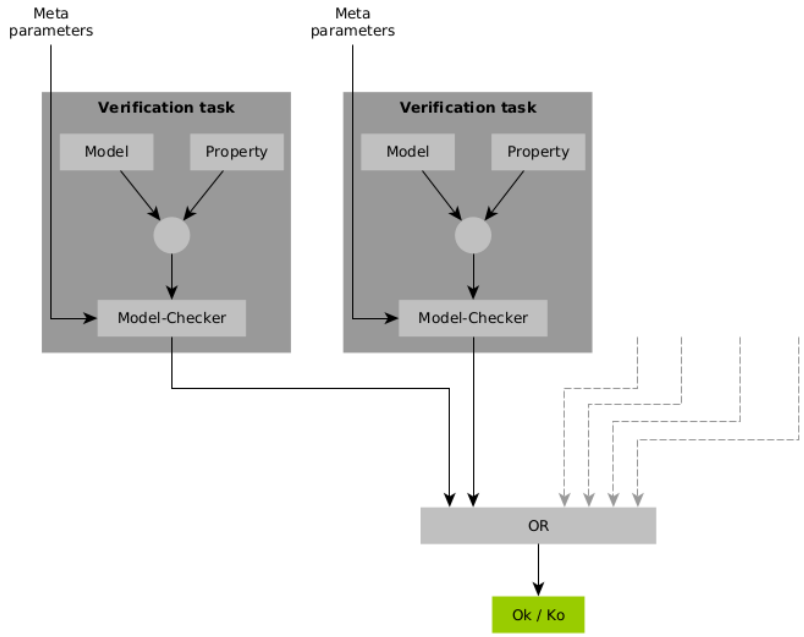
Les approches précédentes visent à augmenter la scalabilité du Model-Checking par des approches sans perte de données. Pour aller plus loin, et augmenter encore la scalabilité du Model-Checking, plusieurs techniques ont été proposées, reposant sur des techniques de compression avec perte pour stocker l'espace d'état de manière extrêmement compacte. Proposant un compromis entre couverture de l'espace d'état et scalabilité par rapport aux méthodes classiques, ces approches proposent l'utilisation d'une représentation probabiliste d'ensemble pour stocker l'espace d'état. Ce type de représentation peut introduire, contrairement à une représentation déterministe, des faux positifs lors de la requête d'appartenance à l'ensemble, qui implique alors que l'espace d'état n'est plus nécessairement parcouru exhaustivement. Aussi, l'absence d'erreur n'est plus garantie formellement, dans la mesure où il n'est plus garanti que l'espace d'état a été parcouru dans son intégralité. Cependant, la réduction très significative de l'espace mémoire requis en font une approche très utilisée pour la vérification de systèmes réels complexes. Ces approches incluent des techniques comme le *bitstate hashing*, qui exploite un *filtre de Bloom* pour stocker l'espace d'états, ou *hash compaction*, se basant sur une table de hachage pour stocker les hash des états parcourus, en lieu et place de leur forme complète [20][21][22].

Vérification swarm

Lors d'une tâche de vérification par bitstate-hashing, la couverture, que l'on peut définir comme la proportion de l'espace d'état couverte par une tâche de vérification, n'est pas totale. Seule une partie de l'espace d'état est vérifié par une tâche de vérification. Cette partition est définie par la discipline adoptée pour le parcours de l'espace d'état : BFS ou DFS, ainsi que par la - ou les - fonctions de hachage utilisées par le Bloom filter pour stocker l'espace d'état. Ce sont en effet les collisions occurant dans le Bloom filter qui sont à l'origine de *l'élimination* de certaines sous-parties de l'espace d'état lors du parcours.

Dans le but d'augmenter la partition de l'espace d'état vérifiée, Holzmann propose l'approche swarm [7][23]. Cette approche, esquissée dans la figure 1.2, consiste à combiner les résultats de plusieurs tâches de vérification, effectuées sur le même modèle, en faisant varier les méta-paramètres de ces tâches. Ces paramètres sont choisis pour leur influence sur la partition de l'espace d'état couverte par chaque tâche de vérification, la *seed* utilisée par la fonction de hachage du Bloom Filter en est un exemple.

FIGURE 1.2 – Vérification swarm : Vue conceptuelle



Cette approche, orthogonale et complémentaire aux techniques d'abstraction ou d'optimisation algorithmique, montre des résultats intéressants en termes de couverture, permettant la vérification d'espaces d'états de grande taille. Holzmann montre en 2013[24] des gains significatifs de couverture par l'utilisation de la vérification swarm intégrée dans le model-checker SPIN. Cette technique a par ailleurs été intégrée dans d'autres outils de vérification très utilisés, comme *Java pathfinder*[25].

Une caractéristique majeure de l'approche swarm est l'indépendance complète des tâches de vérification : Ces dernières ne sont définies que par leurs méta-paramètres, sans communication entre-elles. Ce point en fait une approche intrinsèquement concurrente, permettant une distribution des calculs sur des platformes de calcul parallèles multi-coeur ou multi-processeur.

L'approche swarm permet donc la vérification de modèles de taille très importante avec un volume mémoire faible - rapporté à la taille de l'espace d'état. Elle exploite pour cela deux leviers importants que sont la compacité mémoire offerte par l'utilisation d'un Bloom filter pour stocker l'espace d'état, associé à la diversification du parcours de l'espace d'état par le changement des méta-paramètres (algorithmes, *seeds*). La contrepartie de cette meilleure scalabilité est un cout en termes de calcul très important : dans un cas

typique, le nombre de tâches de vérification exécutées se chiffre en dizaines de milliers[6].

1.3 Présentation synthétique des algorithmes

Le Model-Checking fournit une technique de preuve automatique générique et puissante basée sur l'analyse de l'espace d'état sous-jacent à l'exécution d'un modèle. Son applicabilité est basée sur l'hypothèse que le modèle, vu comme un système de transition, induit un espace d'état fini qui peut être énuméré de manière exhaustive.

Cette section propose une formalisation abstraite et générique des algorithmes de vérification pour la sûreté principalement. Le cas des algorithmes de vivacité sera traité succinctement.

1.3.1 Formalisation abstraite des algorithmes d'atteignabilité

Visant à proposer une cadre de raisonnement générique, cette section propose une spécification abstraite des algorithmes de Model-Checking pour la sûreté.

Dans le cadre de ce chapitre, on s'intéresse à la résolution algorithmique du problème suivant : vérifier si un programme satisfait une propriété de sûreté. Il s'agit par exemple de vérifier si le programme ne se bloque pas, ou si un certain comportement indésirable n'arrive pas. Pour ce faire, il est d'abord nécessaire de préciser la définition d'un programme. On modélise ici un programme informatique comme un système de transitions. Ce système, prends ainsi la forme d'un graphe orienté, dont chaque noeud représente les valeurs de l'ensemble des variables du système. Chaque arc, ou transition, corresponds à la modification d'une ou plusieurs variables du système.

Si l'on considère un programme qui termine exprimé dans un langage d'implémentation classique, comme C, Python, .., l'exécution du programme est déterministe. Aussi, le graphe représentant le système état-transition du programme est purement linéaire. Cependant, le modèle considéré doit être fermé, ie, n'avoir ni entrées, ni sorties. Un tel programme est donc composé avec un environnement d'exécution, non déterministe, qui lit les sorties et génère les entrées.

On se base donc dans la suite sur un modèle exprimé sous la forme présentée dans la figure 1.3.

Spécification du modèle vérifié

La figure 1.3 présente l'implémentation d'un modèle simple exprimé en TLA+[26], langage proposé par L. Lamport pour décrire le comportement temporel de systèmes. Ce modèle simple est composé de trois fonctions principales, que l'on peut considérer comme l'interface publique d'un modèle.

FIGURE 1.3 – Modèle : Interface

MODULE *model3*

A simple model

$S \triangleq [a \mapsto \{\text{“b”}, \text{“c”}\}$
$\quad, b \mapsto \{\text{“d”}\}$
$\quad, c \mapsto \{\text{“f”}, \text{“d”}\}$
$\quad, d \mapsto \{\text{“e”}\}$
$\quad, e \mapsto \{\text{“c”}\}$
$\quad, f \mapsto \{\}]$
$ini \triangleq \{\text{“a”}\}$
$\_next(x) \triangleq S[x]$
$next(A) \triangleq \text{UNION } \{\_next(i) : i \in A\}$
$\_isSafe(x) \triangleq \text{IF } x = \text{“n”} \text{ THEN FALSE ELSE TRUE}$
$isSafe(N) \triangleq \forall\, i \in N : \_isSafe(i)$

La fonction *ini* renvoie l'ensemble, non vide, des états initiaux du système. On définit ensuite la relation de transition du système sous la forme de la fonction *_next(x)*. Cette fonction, pour un état source passé en argument, renvoie l'ensemble des successeurs de cet état. Ces deux fonctions sont suffisantes pour décrire l'évolution du modèle vérifié. Pour vérifier des propriétés, il convient d'enrichir ce modèle avec la fonction *_isSafe(x)* modélisant la propriété vérifiée, renvoyant, pour un état donné en entrée, son statut vis-à-vis de la propriété vérifiée (est-il *sûr*, ou viole t-il la propriété).

Ces deux dernières fonctions *_next* et *_isSafe* sont des fonctions internes à notre spécification. En elles sont utilisées en pratique via les fonctions *next* et *isSafe* qui ne sont que des sucres syntaxiques, permettant de travailler sur des ensembles. La fonction *next(A)* renvoie l'ensemble des états successeurs d'un ensemble d'état, et la fonction *isSafe(A)* envoie la conjonction de la fonction *_isSafe(x)* sur l'ensemble A, ie renvoie faux si au moins un état de l'ensemble A viole la propriété.

Ce triplet de fonctions encapsule le comportement du modèle. On présente dans la

suite une spécification des algorithmes de Model-Checking, permettant d'en dériver par la suite, une architecture générique présentée au chapitre 2.

Spécification des algorithmes

La spécification 1.4 propose une spécification abstraite d'un algorithme d'atteignabilité, exprimée dans le langage *TLA+*. Ce langage de haut niveau est utilisé principalement pour la modélisation de programmes informatiques et de systèmes concurrents et/ou distribués. Ce formalisme permet principalement la description d'ensembles et leurs relations.

Après avoir présenté la formalisation d'un modèle simple, que l'on peut résumer par son API publique - par convention de nommage - exprimée sous la forme de deux fonctions. Ces fonctions modélisent la relation de transition du modèle : *next(A)* qui renvoie l'ensemble des états successeurs à partir d'un ensemble d'états sources, et *ini()* qui renvoie l'ensemble des états initiaux du modèle considéré. A ces deux fonctions, on associe une fonction *isSafe(N)*, qui encapsule le prédicat à vérifier sur les états composites.

Dans cette formalisation, le terme *état* fait référence à l'état composite, concaténation de l'état du modèle vérifié et de sa propriété, supposant ainsi que la composition de ces deux automates a été effectuée a priori.

La figure 1.4 propose une abstraction des algorithmes d'atteignabilité proposés dans la littérature sous la forme d'une spécification générique.

Cette spécification se base sur la définition de deux ensembles :

— Un ensemble contenant les états déjà visités par l'algorithme - *Known* - noté K

— Un ensemble contenant les états dont les successeurs n'ont pas encore été visités - *Frontier* - noté F.

Associés à ces deux ensembles, deux variables de contrôle - booléennes - sont définies : I et S. Elles seront décrites lors de la description de la spécification ci-après.

La spécification des algorithmes d'atteignabilité est définie par la clause $Spec \triangleq Init \wedge \Box[Next]_{\langle K,F,I,S\rangle}$.

Considérons tout d'abord l'opérateur $\Box$. Cet opérateur de la logique temporelle affirme que la formule donnée en argument est toujours vraie. Aussi, la formule $\Box F$ affirme que la formule F est toujours vraie, ie que F est vraie à tous les pas d'exécution. Dans la clause *Spec*, l'opérateur $\Box$ porte sur la clause $Next = Step \vee TheEnd$, qui respectivement correspondent à l'exécution d'un *"pas"* de l'algorithme, ou à l'atteinte de la condition de terminaison. Cette formule peut se traduire en langage naturel comme *"Après avoir*

FIGURE 1.4 –

MODULE *r_frontier_nd_v2_safety*

EXTENDS *TLC*

$M \triangleq$ INSTANCE *model*

VARIABLES K, F, I, S

$Init \triangleq K = \{\}$
$\quad \wedge \quad F = \{\}$
$\quad \wedge \quad I = \text{TRUE}$
$\quad \wedge \quad S = \text{TRUE}$

$TheEnd \triangleq \quad (\neg I \wedge F = \{\} \wedge \text{UNCHANGED } \langle K, F, I, S \rangle) \vee \neg S$

$Step \triangleq \exists A \in \text{SUBSET } F :$
$(I \vee A \neq \{\})$
$\wedge \text{ LET}$
$N \triangleq \text{IF } I \text{ THEN } M!ini \text{ ELSE } M!next(A)$
IN
$K' = K \cup N$
$\wedge F' = (F \setminus A) \cup (N \setminus K)$
$\wedge I' = \text{FALSE}$
$\wedge S' = S \wedge M!isSafe(N)$

$Next \triangleq Step \vee TheEnd$

$Spec \triangleq Init \wedge \Box[Next]_{\langle K, F, I, S \rangle}$

exécuté la clause Init, exécute Step tant que TheEnd n'est pas atteint.". Il est à noter que la clause TheEnd est terminale dans la mesure où elle ne comprends pas d'assignation.

La formule *Init* définit l'initialisation de l'algorithme. Si l'on fait un parallèle avec l'implémentation d'un algorithme de vérification, cette étape corresponds au tirage des états initiaux du modèle, avant de parcourir transitivement l'espace d'état. Cette étape assigne l'ensemble vide aux ensembles K et F, et définit une variable booléenne I, initialisée à la valeur *true*. Cette variable de contrôle permettra de différentier le premier pas d'exécution, lors duquel les états initiaux du modèle sont tirés, de la suite de l'exécution, lors de laquelle les états successeurs sont demandés au modèle. Elle ne sera vraie que lors de ce premier pas d'exécution.

Définissons ensuite la terminaison de l'algorithme : L'algorithme termine lorsque plus aucun état n'est en attente d'être visité. Ces états sont représentés par l'ensemble F, d'où la condition : $F = \emptyset$. Il est cependant nécessaire de distinguer la phase d'initialisation,

en effet, lors de la première "itération" de l'algorithme, l'ensemble F est vide, avant d'y assigner les états initiaux $M!ini$. Cette distinction nécessite l'introduction de la variable booléenne I, dont la valeur *true* désigne "Les états initiaux n'ont pas encore étés tirés". Cette variable est initialisée à *true*, et est assignée à *false* dès la première itération de l'algorithme, matérialisée par la clause *Step* : $I' = FALSE$. Elle reste donc vraie uniquement lors de cette première phase. La clause *TheEnd* est donc enrichie de la condition $\neg$ pour prendre en compte ce cas particulier de l'initialisation. Pour finir, il est nécessaire d'imposer que rien ne change : Si F peut gagner des états après avoir été vide, alors la terminaison n'est pas encore atteinte.

Considérons maintenant un *"pas"* d'exécution de l'algorithme, défini par la clause *Step*. On commence par considérer un sous-ensemble A de F, qui correspondent aux états traités au cours du pas d'exécution courant de l'algorithme. Cet ensemble A est donc retiré de l'ensemble F, représentant l'ensemble d'états à traiter ultérieurement : $F' = F \backslash A$. Une requête est alors faite au modèle M pour obtenir les successeurs de A. On définit alors un ensemble intermédiaire N - *Neighborhood* (voisinage) - contenant ces états successeurs considérés à cette itération de l'algorithme.

Deux cas sont alors à distinguer :

— Si N contient au moins un état noté *unsafe* par le modèle, l'algorithme termine. Ce comportement est obtenu via le booléen S, qui est ajouté aux conditions de terminaison définies dans la formule *TheEnd*. Ce dernier prend la valeur *true* si un état de N est reconnu comme non sûr.

— Dans le cas contraire, les ensembles F et K sont rafraîchis des états traités au cours de cette itération : L'ensemble K, représentant les états déjà parcourus, ie les états dont on a déjà vérifié la sûreté, est étendu des états de N venant d'être vérifiés. L'ensemble F, représentant toujours les états à parcourir est lui aussi mis à jour, y ajoutant les états $N \backslash K$, qui correspondent aux états nouvellement identifiés, dont les successeurs n'ont pas encore été traités.

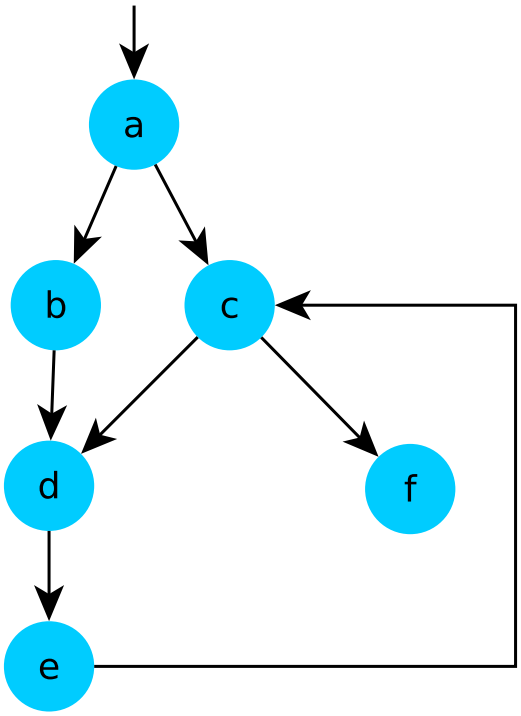
Cette formalisation spécifie la dynamique de parcours d'un algorithme d'atteignabilité arbitraire. Dans ce cadre, une exécution - définie comme un chemin dans l'espace d'état depuis un état initial jusqu'à un état terminal - correspond à une discipline de parcours appliquée à un modèle. Cette abstraction des algorithmes d'atteignabilité définit une famille d'algorithmes pour la vérification de propriétés de sûreté.

1.3.2 Algorithmes exhaustifs

La vue abstraite des algorithmes d'atteignabilité nous permet de poser un cadre de réflexion commun, décrivant un ensemble très large de parcours applicables au modèle considéré. Dans ce cadre, cette spécification peut être raffinée pour construire la spécification d'un algorithme particulier.

On considère comme support de raisonnement un modèle simple, composé de 6 états, comportant un cycle, et un *deadlock*. La figure 1.5 présente l'espace d'état du modèle considéré, dont l'implémentation dans le formalisme précédemment présenté est décrite dans la figure 1.3. C'est par ailleurs sur ce modèle que les traces d'exploration présentées ci-après seront construites, par le biais du model-checker OBP[27].

FIGURE 1.5 – Représentation de l'espace d'état du graphe considéré



La figure 1.6 montre les premiers pas d'éxecution de l'exploration. On note que l'espace d'état est un graphe non-déterministe : depuis le troisième état, trois pas d'exécution différents peuvent être effectués. Pour rendre cette exploration déterministe, et ainsi obtenir un parcours particulier du graphe, la spécification doit être précisée : depuis n'importe

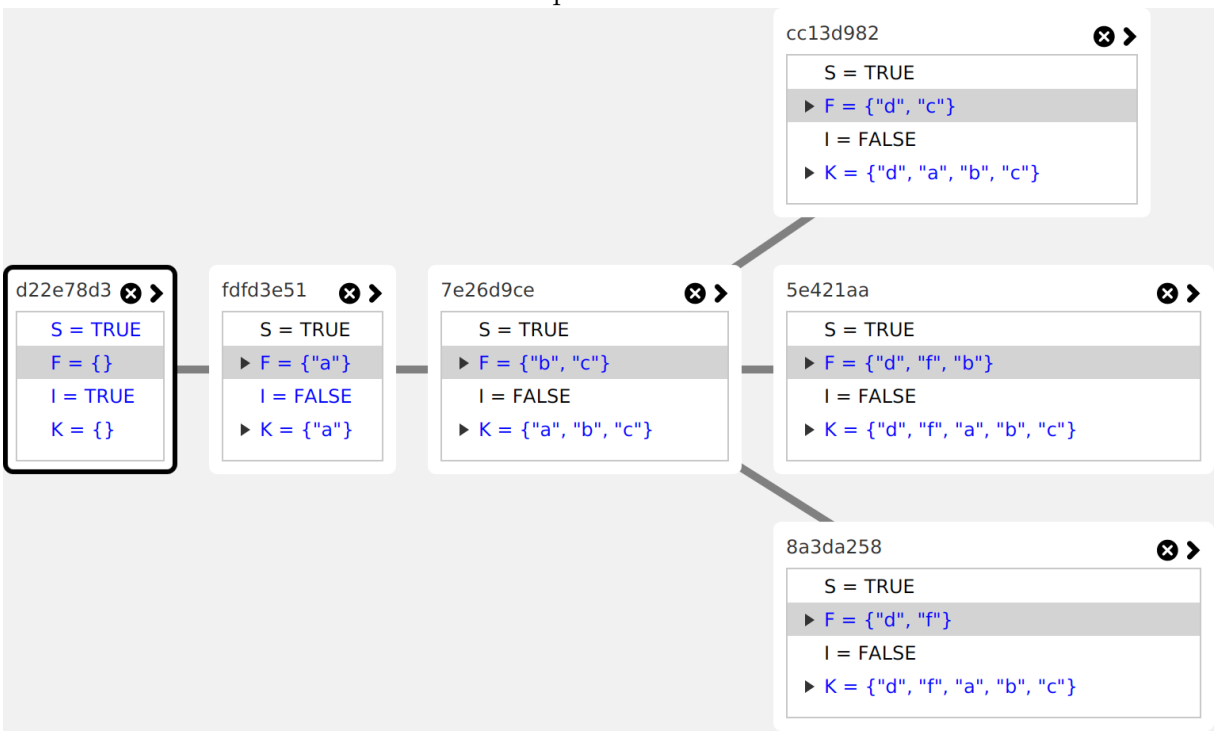
quel état, une seule action doit pouvoir être effectuée. On se propose ici d'illustrer cette procédure de raffinement en présentant quelques algorithmes typiquement utilisés pour la vérification de propriétés de sûreté.

Un premier point important à préciser dans la spécification est l'ordre de production des successeurs par la sémantique. En effet, parler de discipline de parcours d'un graphe, que ce soit BFS, DFS, ou autre, n'a de sens que pour un ordre donné de production des successeurs pour chaque état : Pour un même état, le modèle doit renvoyer l'ensemble de ses successeurs toujours dans le même ordre.

BFS

Le premier raffinement présenté est celui permettant de construire un parcours en largeur, son implémentation étant la plus proche de l'expression de la spécification. Cet algorithme est principalement utilisé pour sa capacité à proposer des contre-exemples courts [28].

FIGURE 1.6 – Premiers pas d'exécution de la simulation



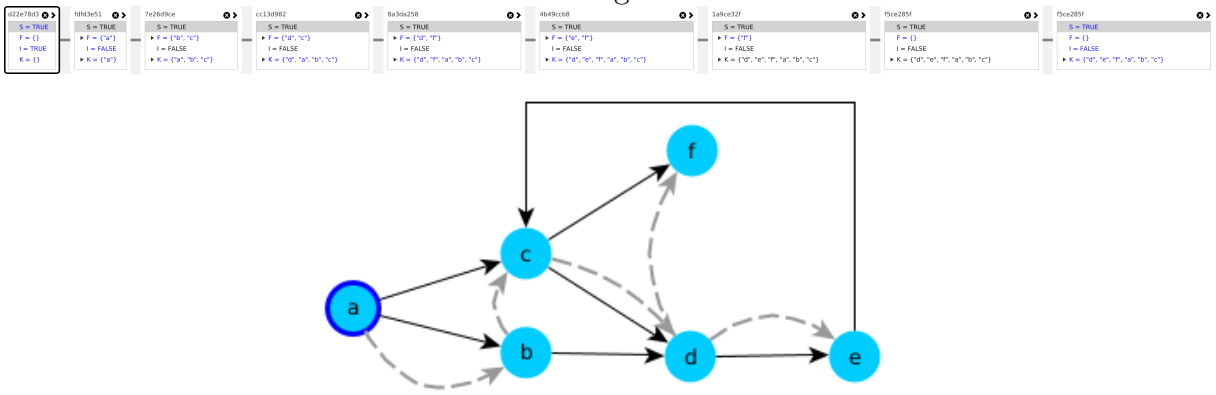
Dans son implémentation classique, un parcours en largeur se base sur une file d'attente - First In First Out - pour stocker les états en attente d'être visités. Ces états

correspondent exactement aux états de la frontière de l'exploration : tous les états contenus dans la FIFO d'un BFS sont des états dont les prédécesseurs ont été parcourus, mais dont les successeurs ne l'ont pas encore été. Ces états correspondent exactement aux états de l'ensemble *Frontière* - F - défini comme variable de la spécification.

Ils sont cependant décrits comme un ensemble non ordonné : il est nécessaire de préciser ce comportement, en y ajoutant une relation d'ordre, définie comme l'ordre d'insertion dans l'ensemble. Notons que l'appel à la relation de transition $M!next(\_)$ renvoie un ensemble. Ceci étant, la définition du raffinement est ici naturelle : l'ensemble de successeurs renvoyé par la fonction $M!next(\_)$ est supposé ordonné, les successeurs sont donc ajoutés dans cet ordre à la *frontière*.

Une fois cette précision effectuée, la spécification d'un parcours en largeur se résume simplement à contraindre la sélection de l'ensemble A d'états à traiter dans l'itération courante - défini initialement comme un sous-ensemble de F - à un singleton contenant l'état le plus "vieux" dans la frontière (la tête de la FIFO).

FIGURE 1.7 – Trace d'exécution d'un algorithme BFS du model-checker OBP2



Ce parcours est illustré dans la figure 1.7, sur laquelle les liens noirs représentent les transitions entre les états, et les flèches pointillées grisées décrivent le parcours effectué par l'algorithme.

A l'état initial, les ensembles F et N sont vides, et la variable I est vraie, désignant que les états initiaux n'ont pas encore été parcourus. Les états initiaux, en l'occurrence ici, "a", sont ajoutés dans au *Known*, et à la *Frontière*.

L'état le plus vieux de la *Frontière*, "a", est sélectionné et retiré de F, qui est étendue de ses successeurs "b" et "c".

L'algorithme parcours les états successeurs jusqu'à ce que l'élément "e" soit traité,

dont le successeur "c" a déjà été parcouru, et n'est donc pas ajouté dans la frontière.

A l'itération suivante, l'ensemble des états à traiter, F, est vide, et l'algorithme termine.

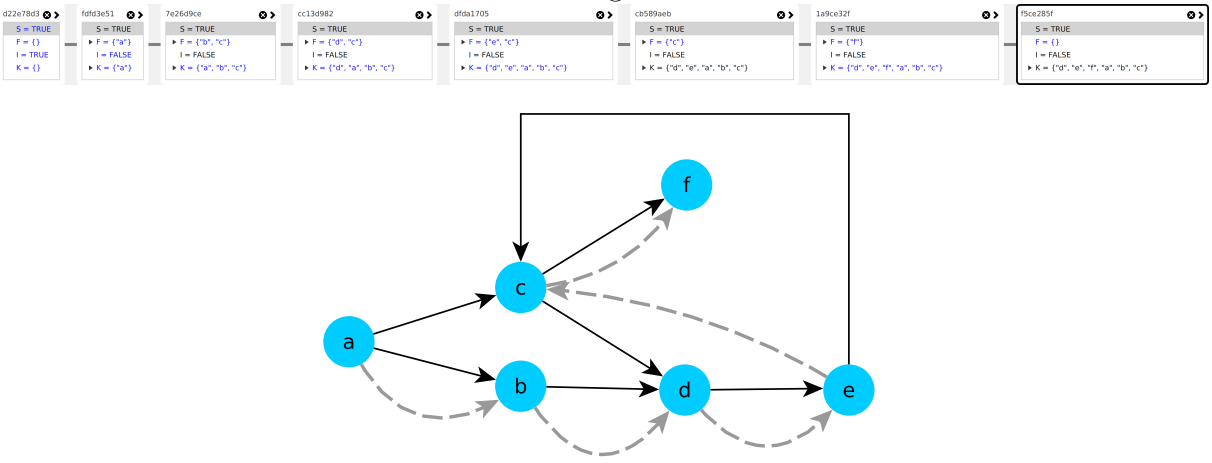
DFS

Le parcours en largeur possède intrinsèquement la propriété - majeure pour le débug - de fournir des contre-exemples de longueur minimale. Cependant, le parcours en profondeur est souvent préféré pour la vérification par Model-Checking pour des considérations d'efficacité mémoire. De plus, un parcours en profondeur permet de reconstruire le contre-exemple en se basant sur les seuls états contenus dans la pile d'états à parcourir.

Considérons tout d'abord le cas d'un parcours en infixe. Ce dernier se construit de manière analogue au parcours précédemment présenté, en imposant une relation d'ordre sur la Frontière. La seule différence entre ces deux parcours est la sélection de l'ensemble d'états à traiter dans l'itération courante : A. Dans le cas d'un DFS infixe, l'ensemble A sera toujours un singleton contenant l'état le plus récemment ajouté dans la Frontière.

Le parcours infixe, présenté à titre pédagogique, n'est cependant que rarement utilisé dans le cadre de la vérification de propriétés de sûreté. En effet, contrairement à un parcours postfixe, le parcours infixe ne permet pas la reconstruction du contre-exemple sans utiliser une structure de données additionnelle (arbre de parents).

FIGURE 1.8 – Trace d'exécution d'un algorithme DFS du model-checker OBP2



Dans le cas du parcours postfixe, l'implémentation diffère légèrement de la spécification : La pile utilisée dans le cadre d'un DFS ne corresponds pas directement à la frontière,

mais l'inclus. En effet, dans le cas de cet algorithme, la queue contient non seulement la frontière, mais également l'ensemble de ses prédécesseurs, jusqu'aux états initiaux. La preuve de raffinement de la spécification est moins naturelle, mais peut s'effectuer via la construction d'un observateur[29] qui reconstruit l'ensemble *frontière* de la spécification à partir des états contenus dans la pile.

Il est important de noter que l'exécution de la spécification de l'algorithme évolue de manière partiellement désynchronisée de l'implémentation : L'implémentation a besoin de plusieurs pas d'exécution avant que la spécification ne fasse un pas, chaque successeur étant traité individuellement.

Model-Checking Borné

Dans le cadre de la vérification de modèles complexes, les algorithmes précédents peuvent être limités par leur besoin mémoire important : Le stockage de l'espace d'état nécessite un volume mémoire qui dépasse souvent les capacités mémoires de la plateforme d'exécution.

Pour contourner cet écueil, une stratégie peut être de ne rechercher que des contre-exemples de longueur *au plus n*. Cette approche, partielle, présente l'intérêt d'une forte compacité mémoire, mais ne permet que difficilement de s'assurer que l'intégralité de l'espace d'état a été parcouru sans analyse complexe du modèle.

Une solution naturelle pour implémenter une telle recherche est l'implémentation d'un parcours en largeur dont la profondeur est limitée. Dans ce cadre, la représentation explicite de l'ensemble des états parcourus, nécessaire dans l'implémentation des deux algorithmes précédents pour assurer la terminaison de l'algorithme [1], n'est plus nécessaire : la terminaison est assurée par la borne imposée sur la longueur du contre-exemple.

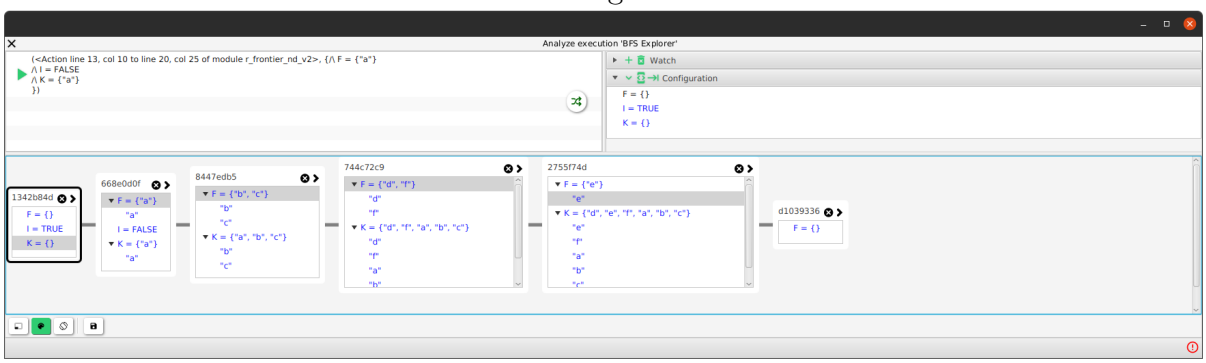
Aussi, le nombre d'états stockés pour assurer la terminaison peut être drastiquement réduit : Il n'est plus nécessaire que de stocker l'étage de profondeur en train d'être découvert, pour filtrer les états dupplicats parmi les successeurs de la frontière.

Dans ce cas, l'implémentation usuelle consiste à exploiter une file d'attente à deux étages, l'un stockant les états de l'étage courant - en train d'être dépilé - et l'autre stockant les états de l'étage suivant, successeurs de l'étage courant. Dans l'implémentation, l'ensemble stockant les états déjà parcourus ne contient que les états de l'étage suivant, et est vidé lorsque l'étage courant est entièrement visité.

1. L'espace d'état est un graphe potentiellement cyclique

Cet algorithme raffine également la spécification proposée, les ensembles K et F peuvent être reconstruits par deux observateurs, respectivement stockant tous les états parcourus à ce point de l'exploration, et concaténant les deux étages de la file d'attente utilisée dans l'implémentation.

FIGURE 1.9 – Trace d'exécution d'un algorithme DFS du model-checker OBP2



Algorithme symbolique

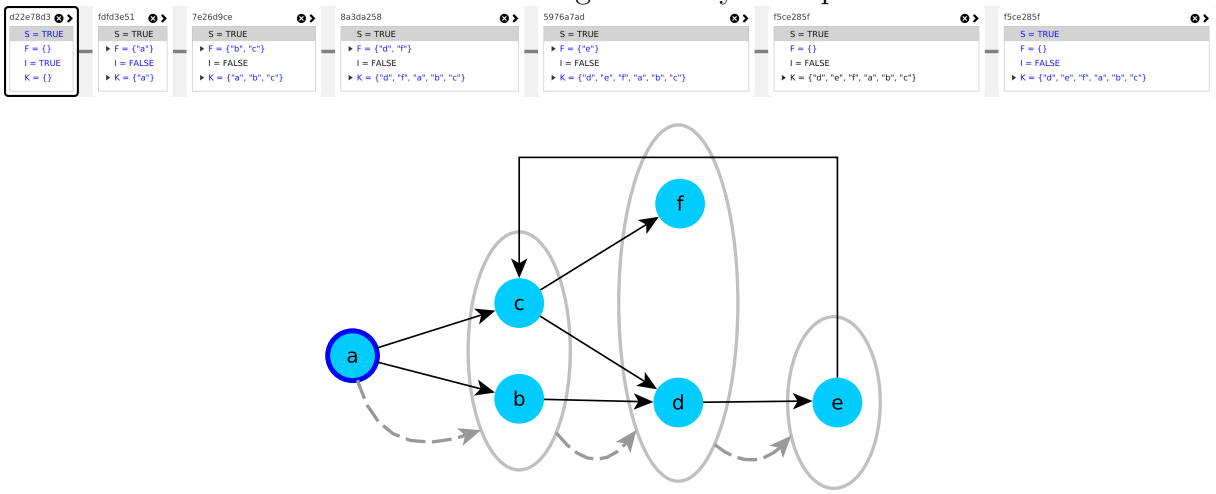
La section précédente a évoqué deux axes principaux pour repousser l'explosion combinatoire : l'abstraction du modèle d'une part, et l'optimisation algorithmique d'autre part. Cette disparité est valable dans le cadre du Model-Checking dit *explicite*, forme dans laquelle l'analyse est effectuée sur le modèle représenté explicitement.

A la fin des années 90[19], Mcmillan proposa un résultat prometteur : L'utilisation d'une représentation symbolique de la relation de transition - sous forme d'arbres binaires de décision(BDD) - permet une représentation bien plus compacte en mémoire que les techniques explicites, permettant ainsi de vérifier des modèles comportant un nombre extrêmement large d'états, gagnant plusieurs ordres de grandeur en scalabilité.

L'implémentation symbolique d'un Model-Checker est similaire à la version explicite. Une différence importante est cependant à noter : l'algorithme ne travaille pas sur un état explicite, mais sur un ensemble d'états représenté par un arbre de décision. L'implémentation de la fonction $M!next(\_)$ renvoie donc un ensemble, dont l'intersection et l'union avec l'ensemble K est calculée pour obtenir respectivement la nouvelle valeur des ensembles F et K, représentés dans le même formalisme. Ce type de parcours peut s'apparenter au parcours en largeur borné présenté précédemment, la différence majeure étant que les états sont traités de manière ensembliste, par des transformations algébriques sur

les formules booléennes les représentant, et non par énumération. La figure 1.10 illustre un de ces raffinements de la spécification.

FIGURE 1.10 – Trace d'exécution d'un algorithme symbolique du model-checker OBP2



Synthèse partielle

L'explosion combinatoire est une caractéristique intrinsèque du Model-Checking. L'effort de recherche important déployé pour trouver des contre-mesures a mené à la proposition de multiples algorithmes, présentant chacun un intérêt dans une situation particulière : minimisation ou capacité de reconstruction du contre-exemple, scalabilité. Cependant, ces algorithmes partagent une structure commune, illustrée ici au travers d'une spécification abstraite.

Cette structure commune des algorithmes, associée au besoin de multiples parcours pour mener à bien un effort de vérification, a influencé fortement l'architecture de Model-Checkers modernes, comme Divine3[11], LTSmin[30], ou OBP2[27].

Avec peu de modifications, cette spécification abstraite peut être généralisée pour illustrer aussi des algorithmes partiels.

1.3.3 Extension aux algorithmes de vérification partielle

La spécification précédemment présentée se veut être un outil de raisonnement, de manière à construire une architecture générique supportant des algorithmes multiples, ainsi qu'à, dans la suite, illustrer la généricité des architectures proposées. Cependant, cette

formulation de la spécification ne supporte qu'une sous-partie des algorithmes utilisables pour effectuer des tâches de vérification sur un modèle, constituée des algorithmes exhaustifs. On propose dans cette section une généralisation à des algorithmes partiels, offrant une compacité mémoire fortement améliorée par rapport à leurs versions exhaustives.

Les algorithmes de vérification explicites présentés précédemment reposent sur une représentation déterministe d'ensemble pour stocker l'espace d'état : on a donc une relation un-à-un entre les états effectivement stockés dans l'ensemble abstrait K, et son implémentation, généralement effectuée par une table de hachage dans le cas du Model-Checking explicite. Cette forme de représentation d'ensemble est intrinsèquement gourmande en mémoire : chaque élément ajouté est stocké explicitement, sans compression. Plusieurs techniques ont été proposées pour augmenter la compacité mémoire de ces ensembles (nombre d'états stockables par octet)[20][7][21]. Elles ont en commun de ne pas stocker les éléments directement, mais l'image de ces derniers par une fonction injective (fonction de hachage). Aussi, le test d'appartenance d'un état à l'ensemble d'états connus est dégradé : seule son image par la fonction f est prise en compte pour décider son appartenance. Cette particularité induit la possibilité d'occurrence de faux positifs : deux états dont l'image par f est identique ne pourront être différentiés lors du test d'appartenance.

Historiquement, l'exploitation de ce type de représentation compressée a été proposée en 1995 par Stern & Dill. La technique proposée, communément appelée *hash compaction*, consiste au stockage direct du hash des états dans une table de hachage, permettant ainsi de réduire l'occupation mémoire à quelques octets par état. Cette variation a été implémentée dans le Model-Checker $Mur\phi$[31], ainsi que dans la première implémentation d'un model-checker matériel : $PHAST$ [31][32].

La recherche de compacité mémoire a été ensuite poussée plus avant par Holzmann, proposant la technique du *bitstate hashing* [20]. Cette technique repose sur une représentation de l'ensemble *fontière* par un filtre de Bloom. Dans ce cas, l'ensemble est implémenté sous la forme d'une table de booléens, dont la valeur vraie corresponds à la présence de(s) l'état correspondant. Cette représentation présente l'intérêt d'une compacité mémoire extrême, pouvant être inférieure à 1bit/état dans le cas où plusieurs fonctions de hachage sont utilisées pour l'implémentation du filtre de Bloom [33].

D'un point de vue de la spécification ces algorithmes, la modification à effectuer par rapport à la spécification initiale est naturelle :

La figure 1.3.3 présente une généralisation de la spécification pour prendre en compte ces algorithmes partiels. L'ensemble K ne contient ici que l'image des éléments visités au

MODULE *r_frontier_nd_v2_safety*

$$
\begin{array}{l}
\mathit{Step} \triangleq \exists\, A \in \text{SUBSET } F : \\
\quad (I \vee A \neq \{\}) \\
\quad \wedge \text{LET} \\
\qquad N \triangleq \text{IF } I \text{ THEN } M!\mathit{ini} \text{ ELSE } M!\mathit{next}(A) \\
\quad \text{IN} \\
\qquad K' = K \cup \mathit{hashFct}(N) \\
\qquad \wedge F' = (F \setminus A) \cup \{x \in N : \mathit{hashFct}(x) \notin K\} \\
\qquad \wedge I' = \text{FALSE} \\
\qquad \wedge S' = S \wedge M!\mathit{isSafe}(N)
\end{array}
$$

FIGURE 1.11 – Extension de la spécification 1.4 aux parcours partiels

travers d'une fonction de hash *hashFct*. De manière analogue à la spécification initiale, l'ensemble *Fontière* est calculé en soustrayant les états traités pendant le cycle courant, et en y ajoutant les successeurs non présents dans l'ensemble d'états visités.

Cette spécification générale, permettant de formaliser les algorithmes de vérification partielle, est plus général que la version précédemment présentée : La preuve de raffinement est immédiate en représentant l'ensemble K en utilisant $\mathit{hashFct} = \mathit{Identite}$.

1.4 Enjeux de performance et solutions matérielles

Le milieu des années 2000 a sonné la fin le la Loi de Dennard, conséquence de la loi de Moore prédisant l'augmentation de la fréquence des circuits parallèlement à l'augmentation de leur niveau d'intégration. Aussi, l'augmentation de la puissance de calcul s'est traduite par l'introduction, puis la généralisation des processeurs multi-coeurs.

Dans le même temps, la complexité et la criticité des système n'a cessé de croitre, ainsi que le besoin de vérification. Ce constat a motivé un effort de recherche important visant à accroitre la performance en plus de la scalabilité des outils de Model-Checking.

1.4.1 Plateformes de calcul parallèles pour le Model-Checking

Les travaux portant sur la parallélisation des algorithmes de Model-Checking peuvent se classer en trois catégories, correspondant aux plateformes de calcul ciblées : multi-coeur/multi-processeur, distribuée, et plus récemment SIMD, avec l'exploration des pos-

sibilités offertes par le parallélisme massif offert par les GP-GPU.

	Distribué	Mémoire partagée	GPU	FGPA
NFA	[34] [35] [36]	[37] [38]	[39] [40] [41]	[6] [32]
ω-automates	[42] [43] [44] [45] [46]	[47] [48] [49] [50] [51]	[52] [53]	*N.A.*

FIGURE 1.12 – Parallélisation des algorithmes de Model-Checking

La figure 1.12 recense un certain nombre de travaux visant la parallélisation des algorithmes de Model-Checking. Ces travaux sont classés en fonction du modèle de programmation utilisé - mémoire distribuée, mémoire partagée, SIMD et FPGA - ainsi qu'en fonction du type de propriétés vérifiées. Comme précisé précédemment, cette distinction implique le type d'algorithme utilisé : un parcours d'atteignabilité est suffisant pour la vérification de propriétés de sûreté, mais un algorithme de détection de cycles est nécessaire pour la vérification de propriétés plus générales, représentées par des automates ω-réguliers comme les automates de Büchi, par exemple.

Les systèmes distribués sont une cible naturelle pour l'accélération du Model-Checking. L'intérêt de cette approche est liée principalement à l'augmentation linéaire du volume mémoire disponible avec le nombre de machines utilisées. Cependant, un point d'attention critique pour la mise au point d'algorithmes ciblant de telles plateformes réside en la minimisation des communications mises en jeu pendant l'exécution.

L'accélération du Model-Checking explicite sur de tels systèmes a été initiée par les travaux de Lerda [36], suivie d'un travail important mené par l'équipe Tchèque de J. Barnat et L. Brim, qui a mené à la proposition des algorithmes MAP [42], OWCTY [44], NEGC [46].

La généralisation des systèmes multi-coeurs et multi-processeurs a constitué une opportunité importante pour la vérification : partageant la mémoire principale entre les coeurs, ces systèmes permettent une communication beaucoup moins couteuse entre des processus concurrents. La structure hiérarchique de la mémoire peut de plus être avantageusement exploitée pour accélérer le stockage des états visités [38]. De nombreux travaux ont été menés sur ce sujet dans le contexte du développement du Model-Checker LTSmin, menés par J. Van De Pol [48] [49][50][51].

La scalabilité importante des algorithmes proposés ciblant des plateformes multi-coeurs suscite naturellement la question de leur scalabilité sur des plateformes massivement parallèles. Les GP-GPU - General Purpose Graphical Processing Units - sont de plus en plus utilisés pour le calcul parallèle. De plus, leur ratio consommation/perfor-

mance ainsi que leur coût relativement faible en font des outils de calcul plus efficaces que les processeurs pour nombre d'applications massivement parallèles. Quelques travaux portent sur l'exploitation de ces derniers proposent des résultats prometteurs, mais sans rupture majeure par rapport aux implémentations multi-coeurs [41] [53].

Ces trois modèles de calculs - distribué, mémoire-partagée, et gpu - ont chacun fait l'objet de travaux importants, visant à accélérer le Model-Checking. A contrario, seuls deux articles traitent de l'accélération matérielle du Model-Checking via l'utilisation de FPGA.

1.4.2 Accélération matérielle du Model-Checking

Les plateformes de calcul reconfigurables constituent un outil de plus en plus utilisé pour le calcul intensif. Outre le parallélisme intrinsèque et massif offert par de telles plateformes, la structure locale de la mémoire interne permet de bénéficier d'une latence en lecture et écriture constante et très faible.

Deux approches proposent l'exploitation de plateformes reconfigurables - FPGA - pour l'accélération matérielle du Model-Checking.

PHAST : un coeur de Model-Checking explicite reposant sur hash compaction

La première approche proposée pour l'accélération du Model-Checking explicite a été proposée en 2008, par M.E. Fuess et M. Lesser [54]. Dérivé du Model-Checker Murϕ, cette approche implémente un parcours en largeur, reposant sur la technique de *hash compaction* [21] présentée en section 1.2.3.

Cette technique de compression permet la réduction importante du volume mémoire utilisé pour stocker l'espace d'état, en ne stockant qu'une représentation réduite des états, sous la forme de son image par une fonction de hachage. Cette technique permet de réduire l'empreinte mémoire d'un état, dans cette implémentation, 40 bits. Cette compacité mémoire est cependant obtenue au prix de l'exhaustivité du parcours.

L'implémentation initiale repose sur l'utilisation de mémoire locale. Dans cette configuration, *PHAST* montre une accélération de *200x* par rapport à son pendant logiciel : Murϕ. Une seconde version de l'outil, décrite dans la thèse de master de M.E. Fuess, propose une implémentation reposant sur de la mémoire externe, dépréciant les résultats à une accélération de 30x, dû à l'augmentation de la latence mémoire. Ces résultats d'accélération sont obtenus en se basant sur un modèle simple, implémenté manuellement en

VHDL, dont l'espace d'état est constitué d'environ 10 000 états. Ce modèle est constitué de six compteurs, partiellement couplés. Pour ce cas d'étude, la propriété vérifiée est un prédicat portant directement sur l'état du modèle, évalué immédiatement après la génération des états successeurs. Ce choix de design permet de détecter les fautes au plus tôt.

Cet article aurait pu être l'origine d'une rupture majeure en termes de performances : Parmi l'ensemble des travaux portant sur l'accélération du Model-Checking présentés précédemment, aucun ne revendique de facteur d'accélération aussi élevé. Cependant, une faiblesse majeure de l'approche est mise en valeur par la différence de résultats entre l'implémentation originale, proposée en 2008, et sa mise à jour, proposée en 2012. Cette faiblesse, intrinsèque à la plateforme utilisée, réside dans le faible volume mémoire à disposition, limitant l'applicabilité à des modèles de taille raisonnable, pour lesquels les temps de vérification restent acceptables. Cependant, ce point met également en valeur l'intérêt important de la mémoire à faible latence dont disposent les FPGAs.

FPGASwarm : Accélération matérielle de la vérification Swarm pour la sûreté

Cette dernière observation est présentée comme motivation du développement de FPGASwarm par l'auteur, S. Cho [6]. Cet article propose une implémentation d'un accélérateur matériel de vérification swarm. Contrairement au Model-Checking explicite dans sa forme exhaustive, la vérification swarm s'accommode de volumes mémoires faibles au regard de la taille de l'espace d'état considéré. Il repose en effet sur l'exécution d'un nombre important de tâches de vérification sur un même modèle, reposant sur une exécution randomisée pour obtenir une couverture élevée. Cette technique est ainsi un candidat naturel pour l'accélération sur FPGA, bénéficiant d'un parallélisme massif et d'une mémoire à très faible latence.

L'implémentation exploite à la fois un parallélisme interne à chaque tâche de vérification par un pipelining profond, ainsi qu'un parallélisme de tâches, ces dernières étant exécutées sur 40 coeurs répliqués en parallèle. Plusieurs innovations sont proposées pour améliorer la diversification des tâches de vérification, comme l'utilisation d'un LFSR pour rendre pseudo-aléatoire l'ordre dans lequel les états successeurs sont générés, ainsi que l'utilisation d'une file bornée en profondeur pour le stockage des états en instance d'être parcourus (ensemble F de la spécification proposée en 1.4).

L'évaluation est effectuée sous la forme d'un test de couverture, sur un modèle dont la taille de 4.10^9 états est réaliste par rapport à des tâches de vérification sur des mo-

dèles industriels. Cette méthodologie d'évaluation pour la vérification swarm, consistant à recherche 100 entiers particuliers parmi un espace d'état constitué de l'ensemble des entiers de 32bits, a été initialement proposée par Holzmann [24] comme une mesure de la couverture obtenue par une tâche de vérification swarm.

Cette méthodologie d'évaluation sert de point de comparaison pour mesurer la vitesse d'exécution d'une tâche swarm, dans la mesure où aucune condition de terminaison autre qu'une violation de propriété n'existe en swarm - classiquement, un timeout est utilisé pour éviter cet écueil. L'implémentation exploite 32Kb de mémoire par coeur pour la version matérielle. Cette dernière est comparée au Model-Checker *SPIN* utilisant 256Mb de mémoire par coeur, sur 48 coeurs en parallèle. Malgré ce différentiel de mémoire important entre ces deux configurations, avantageant a priori la version logicielle, l'accélérateur FPGASwarm montre une accélération significative de 900x, et ce, en réduisant la consommation d'un facteur 10.

L'expérience menée par S. Cho et ses coauteurs montre l'intérêt de l'accélération matérielle pour la vérification swarm, compensant largement la faible mémoire à disposition par une latence très faible, ainsi que par le parallélisme massif offert par ces plateformes.

1.4.3 Synthèse

A la lumière des travaux existants sur le sujet, l'accélération matérielle du Model-Checking apparaît comme très prometteuse en termes de performances, montrant des gains significatifs par rapport aux implémentations SIMD, distribuées, ou multi-coeurs existantes.

Ce gain s'explique par deux aspects complémentaires : L'exécution bénéficie à la fois d'une mémoire locale, dont la latence est très faible et fixe. Outre le temps d'accès mémoire faible, évitant la situation classique de famine dans l'attente d'une valeur qui n'aurait pas été présente dans les différents niveaux de cache, une latence fixe permet des optimisations difficiles à mettre en oeuvre sur un processeur, comme l'entrelacement de ces accès. Ces caractéristiques sont particulièrement intéressantes pour l'implémentation de structures de données rapides reposant sur des fonctions de hachage, constituant un goulot d'étranglement majeur pour la performance des Model-Checkers. En outre, la maitrise architecturale fine permise par les plateformes reconfigurables permet des implémentations entièrement pipelinées, gagnant ainsi en efficacité par rapport à un traitement séquentiel effectué sur un processeur.

Dans le cas de la vérification swarm, ce micro-parallélisme au niveau des coeurs d'exé-

cution peut être augmenté, comme l'a proposé FPGASwarm[6], par un parallélisme de tâches intrinsèque à cette méthode en distribuant les calculs sur plusieurs coeurs d'exécution.

Plusieurs limitations sont cependant à noter. Tout d'abord, les deux approches proposées entremêlent sémantique du modèle vérifié et moteur d'exécution. Ce manque de modularité rends difficile l'extension à de nouveaux langages, et de nouveaux algorithmes. De plus, ces approches sont évaluées sur des modèles différents, nuisant fortement à la comparaison de l'une à l'autre.

En outre, dans le cas de FPGASwarm, l'algorithme proposé est nouveau par plusieurs aspects. En plus d'appliquer un ordre pseudo aléatoire sur l'évaluation des transitions, et donc la génération des successeurs, il se base sur une structure de données partielle pour le stockage des états en instance d'être parcourus - *Frontière* dans notre terminologie. Ces deux caractéristiques sont intéressantes, et bénéficient probablement à l'efficacité de la vérification, mais une analyse systématique sur l'influence de ces facteurs serait souhaitable.

Le dernier point à mentionner est leur limitation à la vérification de prédicats sur l'espace d'état, limitant fortement la l'expressivité des propriétés vérifiées.

1.5 Conclusion

Ce chapitre décrit dans un premier temps de cadre utilisé pour la vérification par Model-Checking. Dans cette formalisation, un programme est représenté sous la forme d'un graphe implicite définie par sa relation de transition. Pour vérifier des propriétés sur son comportement, les propriétés, formalisées sous forme d'automates finis non déterministes, ou d'automates de Büchi selon l'application, évoluent de manière synchrone avec le système, chacun avançant d'une transition à chaque pas d'exécution.

Dans cette représentation, il s'agit alors de détecter les états constituant la fin d'un mot accepté par l'automate composé - produit synchrone du modèle et de la propriété associée. Dans le cas de vérification de propriétés la sûreté, cette détection s'effectue par une analyse d'atteignabilité sur l'espace d'état de l'automate produit. Aussi, nous proposons une abstraction des algorithmes permettant d'effectuer une telle vérification, sous la forme d'une spécification générique.

Ces algorithmes ne peuvent s'exécuter sans plateforme adaptée. L'enjeu des performances de la vérification est abordée dans une troisième partie sous la forme d'une étude

de ces plateformes, et des opportunités associées.

Chapitre 2

Menhir : Un cadre modulaire pour la vérification formelle sur FPGA

L'accélération matérielle de la vérification formelle apparaît, à la lumière de l'analyse présentée dans la Section 1.4, comme une solution prometteuse pour permettre l'amélioration des performances des outils de vérification. D'autre part, il apparaît que les faibles ressources mémoires soient compensées par le gain de performances dans un contexte de vérification swarm.

Malgré des résultats en apparence prometteurs, seuls deux travaux - espacés d'une dizaine d'années - proposent l'exploitation des FPGAs pour accélérer la vérification par Model-Checking. Les raisons à cela peuvent-être de plusieurs ordres : 1. la complexité d'un tel développement, 2. le manque de capacité à comparer et à mettre en perspective les résultats proposés, élément nécessaire à une stratégie de développement incrémentale, ou encore 3. la difficulté d'exploiter un tel outil pour effectuer des tâches de vérification réalistes, dans le cadre du développement d'un produit. Ces éléments apparaissent être le symptôme d'un manque de structure, du manque d'une architecture de référence permettant de guider le développement d'un tel accélérateur.

En outre, les deux implémentations proposées, décrites dans la Section 1.4.2, n'implémentent qu'un seul algorithme, et n'évaluent leurs résultats que sur un seul modèle. Le Model-Checking souffre intrinsèquement de l'explosion combinatoire, aussi un grand nombre d'algorithmes ont été proposés pour adapter la stratégie de vérification aux caractéristiques de chaque problème. D'autre part, et pour la même raison, de nombreux langages de spécifications sont utilisés pour spécifier les problèmes à différents niveaux de raffinement. Il apparaît donc comme important et souhaitable qu'une architecture de référence soit générique tant d'un point de vue algorithmique que sémantique.

Pour y répondre, on propose dans ce chapitre Menhir, un framework de vérification matérielle, permettant dans une architecture unifiée d'implémenter de multiples algorithmes, ainsi que plusieurs langages d'entrée. Cette solution permet d'explorer un espace

de conception avec une variabilité dans la représentation de l'information stockée, dans la discipline de parcours de l'espace d'état du modèle, ainsi que dans l'occupation mémoire induite.

La crédibilité de l'approche est illustrée au travers de l'implémentation de 6 algorithmes, formant un dégradé depuis la vérification exhaustive vers l'exploration partielle. D'autre part, le découplage fort entre la sémantique du modèle et le coeur algorithmique permet la vérification depuis trois langages d'entrée, couvrant la vérification de modèles dans des formalismes de spécification de haut niveau comme UML, jusqu'à des modèles très bas niveaux, spécifiées directement en VHDL.

2.1 Architecture fonctionnelle

Visant à concevoir une architecture générique, le point de départ de ce chapitre est la spécification de haut niveau des algorithmes de Model-Checking proposée dans la Section 1.3. Cette spécification est d'abord rapportée d'un point de vue dataflow, afin d'identifier les composants fonctionnels la constituant, ainsi que leurs dépendances. L'identification de ces composants fonctionnels conduit ensuite à la proposition d'une architecture générique, supportant l'ensemble d'algorithmes explicites capturés par cette spécification.

2.1.1 Reformulation dataflow de la spécification

Avant de considérer la conception de l'architecture générique du coeur algorithmique, il est nécessaire de définir ses interfaces. En particulier, son interface avec la sémantique à vérifier.

Une idée clé, qui structure notre approche, est que le composant encapsulant la relation de transition du modèle vérifié, ainsi que celui représentant la propriété à vérifier sont intimement liés, étant les seuls composants dépendants de la sémantique du langage d'entrée. Par conséquent, il semble naturel de les regrouper en un composant unique, implémentation du problème de vérification.

Formulation du problème de vérification

Sans perdre de généralité, la Section 1.3.1 expose la représentation abstraite d'un modèle implicite construit sur un type de configuration arbitraire $\mathcal{C}$ sous la forme d'une

structure $\mathcal{M}(\mathcal{C})$, représentée dans le Listing 2.1, où **initial** est une fonction nullaire renvoyant l'ensemble des configurations initiales, **next** est une fonction renvoyant l'ensemble des configurations accessibles à partir d'une configuration donnée, et **is_safe** est un prédicat sur une configuration qui encode une assertion de sûreté. Cette abstraction permet aux algorithmes de vérification de traiter chaque configuration comme un mot binaire opaque, dont l'interprétation est déléguée au frontend du modèle. De plus, cette représentation abstrait la sémantique du problème de vérification, qui est encapsulée dans la fonction **next**. Ainsi, la composition synchrone entre le modèle et la spécification, étape préalable à une tâche de vérification est également cachée au backend algorithmique.

```
1    structure M (C : Type) :=
2        (initial : set C)
3        (next    : C → set C)
4        (is_safe : C → bool)
5
```

Listing 2.1 – Représentation implicite d'un modèle

Une telle interface cache les détails des langages de modélisation et de spécification ainsi que leur composition synchrone derrière une interface générique, agnostique au langage implémenté, suivant une approche similaire à celle présentée dans [27]. Composée de trois fonctions, cette interface peut être invoquée directement par l'algorithme de vérification.

Cependant, la latence élevée induite par l'invocation de "fonctions" matérielles nous pousse à affiner l'interface modèle-algorithme précédemment proposée 2.1. Chaque appel du prédicat **is_safe** nécessite l'envoi d'une configuration complète du noyau de vérification au frontend du modèle pour récupérer un bit d'information en échange. L'intégration de cet appel directement dans la structure du modèle réduit le coût de communication à un seul bit.

```
1     structure M_H (C : Type) :=
2         (initial : set (C × bool))
3         (next: C →   set (C × bool))
4
```

Listing 2.2 – Représentation implicite d'un modèle

Le raffinement proposé, présenté dans le Listing 2.2, inline les appels à la fonction **is_safe** dans les appels **initial** et **next**. Il en résulte de nouveaux types de retour pour ces derniers, un type produit ($\mathcal{C} \times bool$) composé de la configuration ($\mathcal{C}$ comme précédemment) et d'un bit is_safe (type *bool*).

Le Listing 2.3 montre la transformation mathématique, qui peut être appliquée à tout modèle suivant la structure du modèle précédent (Listing 2.1) pour obtenir le nouveau Listing 2.2. Le nouvel ensemble **initial** est obtenu en ajoutant le résultat de **m.is_safe(i)** à chaque configuration initiale i. Pour chaque source s, le nouvel ensemble de voisinage est le couple (n, **m.is_safe**(n)) pour tous les n dans le **m.next**(s).

```
def M2M_H (m : M C) : M_H C :=
<initial ← { (i, m.is_safe(i)) | ∀ i ∈ m.initial },
 next ← λ s, { (n, m.is_safe(n)) | ∀ n ∈ m.next(s) }>

```

Listing 2.3 – Convertion entre $\mathcal{M}$ et $\mathcal{M}_H$

Cette formulation de l'interface entre le backend algorithmique et la sémantique du problème vérifié permet d'isoler ce dernier, tout en réduisant le coût de la communication engendrée.

Spécification des algorithmes : Vue dataflow

La spécification proposée dans la Section 1.3 décrit le comportement d'un ensemble d'algorithmes de vérification. Cette spécification décrit l'évolution de deux ensembles : un ensemble d'états visités $\mathcal{K}$, et l'ensemble d'états *à la frontière des états connus* $\mathcal{F}$, tout deux initialement vides. Ce dernier ensemble $\mathcal{F}$ contient les états situés à la frontière du sous-espace d'état exploré, intuitivement, il contient donc les états explorés dont les successeurs n'ont pas encore été explorés. L'ensemble $\mathcal{K}$ tient à jour l'historique des états déjà parcourus, de manière à éviter que l'algorithme "régresse", ie qu'il ré-explore des états déjà explorés, ce qui mènerait à une boucle infinie dans le cas (nominal) où l'espace d'état est un graphe contenant des cycles.

FIGURE 2.1 – Etape *Step* de la spécification 1.4

$$
\begin{array}{l}
Step \triangleq \exists\, A \in \text{SUBSET } F : \\
\quad (I \vee A \neq \{\}) \\
\quad \wedge \text{LET} \\
\qquad N \triangleq \text{IF } I \text{ THEN } M!ini \text{ ELSE } M!next(A) \\
\quad \text{IN} \\
\qquad K' = K \cup N \\
\qquad \wedge\, F' = (F \setminus A) \cup (N \setminus K) \\
\qquad \wedge\, I' = \text{FALSE}
\end{array}
$$

Basé sur cette spécification, un algorithme de vérification pour un modèle m considéré peut être défini comme un prédicat **safety_checker**($m : \mathcal{M}$) **: bool**. La valuation de ce prédicat reflète l'état de la variable S de la spécification. Si cette variable prends la valeur *false*, alors le modèle m contient un contre-exemple pour la propriété considérée.

La spécification proposée dans la figure 1.4, se compose d'une *boucle* principale, exprimée en TLA+ par l'opérateur $\Box Step$. Pour des raisons de simplicité, on se concentre dans la suite sur cette l'étape *Step* de la spécification dans la figure 2.1. Les deux autres étapes, communes à tous les algorithmes spécifiés, que sont la définition de l'état initial *Init*, ainsi que la terminaison *TheEnd*, seront traitées ultérieurement.

D'un point de vue algorithmique, le prédicat $\Box Step$ de vérification corresponds au calcul d'un point fixe sur l'ensemble d'états connus en utilisant les données de l'ensemble frontière. Tant que l'ensemble frontière n'est pas vide, traduit dans la spécification par les deux premières lignes : Tant qu'il existe un sous-ensemble de F, noté A, non vide, on définit un ensemble de voisins (*neighbors* en anglais) N, auquel on assigne les successeurs de l'ensemble A considéré. Cet ensemble N, contenant l'ensemble des successeurs des états considérés pour cette *itération* de l'algorithme, est l'ensemble sur lequel porte la vérification de la propriété. Aussi, après avoir traité les états de N, les configurations dans $\mathcal{N}$ ayant été vérifiées, les ensembles *Known* et *Frontière* sont mis à jour : Les états de N sont ajoutées à l'ensemble d'états connus K. Parmi celles-ci, celles qui n'ont pas déjà été parcourues - $N \backslash K$ - sont ajoutées à l'ensemble frontière F.

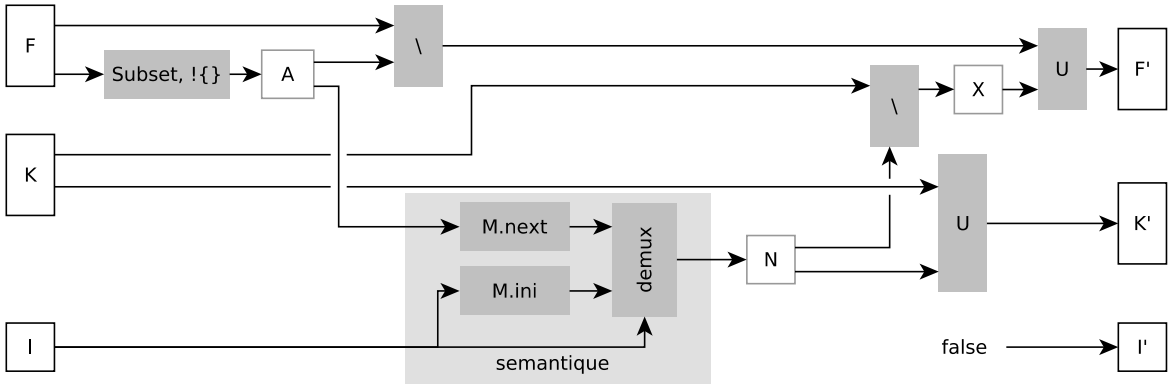
La figure 2.2 reformule l'étape *Step* présentée sous la forme d'un graphe de flot de données. Ce type de représentation représente sous la forme d'un graphe les opérations sur les données, en l'occurrence ici les ensembles F et K, et surtout leurs dépendances.

2.1.2 Architecture générique abstraite

La figure 2.2 décrit sous la forme d'un graphe le flot de données de l'algorithme. Cette formulation abstraite décrit comment les ensembles mis en jeux évoluent d'un pas d'exécution à un autre. Par analogie avec l'électronique, ce passage d'un pas d'exécution à un autre peut être modélisé par un registre, placé entre la valeur courante de l'ensemble, et sa valeur future (respectivement K et K' par exemple).

Une telle reformulation permet ainsi d'obtenir une architecture, mise à plat. Cependant, visant à concevoir un template architectural générique, il est souhaitable d'isoler les comportements propres à chacun des ensembles mis en jeux, de manière à isoler et spécifier des composants de base.

FIGURE 2.2 – Formulation dataflow d'un algorithme de Model-Checking



La figure 2.4 propose une architecture générique dérivée de la formulation précédente, identifiant les composants principaux de l'architecture abstraite. On identifie deux composants principaux. Le composant *Known*, encapsulant le comportement lié à l'ensemble d'états visité K, est composé d'un registre interne pour le stockage de ce dernier, et peut se comprendre comme un *filtre* rejetant certains états. La décomposition proposée pour le composant *Frontière* sépare deux comportements distincts : L'ajout des états entrants dans l'ensemble *Frontière* F - à gauche de l'architecture du composant -, et leur extraction - à droite.

	Type	Notation
Etat interne	ensemble	F
Entrées	ensemble	$X = N \backslash K$
Sorties	ensemble	A

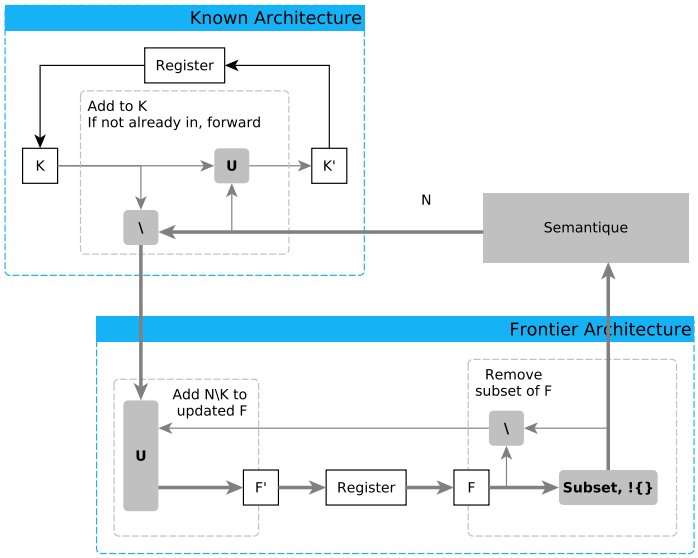
	Type	Notation
Etat interne	ensemble	K
Entrées	ensemble	N
Sorties	ensemble	$X = N \backslash K$

FIGURE 2.3 – Description des attributs des composants *Frontière* (à gauche) et *Known* (à droite)

On remarque sur cette représentation que du pipeline principale, représentée par des flèches épaisses sur la figure 2.4 décrit un cycle. Aussi, à chaque point du flot de données, aucune dépendance n'existe ni en avant, ni en arrière. Cette observation est intéressante du point de vue de la conception matérielle du système, car elle permet d'implémenter un pipeline linéaire, sans branchements qui forceraient la mise en place de synchronisations complexes.

Cette absence de dépendance a une autre conséquence : une fois l'ensemble des successeurs N transmis au composant *Known*, celui-ci n'est plus utilisé. Aussi, il n'est pas

FIGURE 2.4 – Formulation dataflow : Isolation des composants fonctionnels



nécessaire, ni de le stocker, ni que le composant *sémantique* maintienne sa valeur en sortie : ce dernier peut traiter un nouvel ensemble d'états dès l'instant où il a terminé de traiter le courant.

Cette remarque motive l'implémentation du coeur de vérification proposé dans le chapitre 3 sous la forme d'un pipeline auto-régulée : chaque composant transmet l'information à son prédécesseur qu'il est prêt à accepter de nouvelles données, dès l'instant où il l'est. Un tel mécanisme de communication est souvent désigné par le terme *backpressure* dans le vocabulaire métier de la conception matérielle. Ce type de mécanisme de régulation a des conséquences intéressantes pour la performance de la pipeline, en particulier, une réduction importante de la latence de communication entre deux éléments.

Avant de proposer une telle optimisation, on se concentre sur la conception d'une architecture de référence, permettant, dans un second temps, de proposer des améliorations dans le chapitre 3.

Pour cela, plusieurs points sont à considérer avant d'arriver à une implémentation réelle. Tout d'abord, l'architecture abstraite proposée dans la figure 2.4 pose un cadre, mais nécessite le traitement d'ensembles. La représentation d'un ensemble directement en matériel n'est - au mieux - pas naturelle, et - en pratique - très gourmande en ressources. D'autre part, et dans la même ligne, on observe dans le chapitre précédent 1.3 que les algorithmes explicites reposent sur l'utilisation d'un singleton pour l'ensemble intermédiaire A. On adopte cette restriction ici, néanmoins, étant donné une représentation symbolique

avantageuse, on peut imaginer la levée de cette contrainte lors de travaux futurs.

La section suivante propose une architecture concrète qui repose sur cette formulation abstraite et sérialise les communications, traitant les états un par un pour réduire les coûts de communication. D'autre part, la communication entre le modèle et l'algorithme présenté dans le Listing 2.2 est formalisée en une interface matérielle concrète, permettant le découplage entre le modèle et la sémantique du modèle vérifié.

2.2 Menhir : Un Model-Checker matériel modulaire

En se basant sur la spécification proposée dans le chapitre 1, la section précédente a construit une architecture abstraite, agnostique à l'algorithme implémenté. Cette analyse a permis d'identifier les dépendances de données d'un ensemble d'algorithmes.

La Section 1.3 a montré la généricité de la spécification algorithmique proposée. Cette spécification, précédemment raffinée par l'identification des composants fonctionnels mis en jeu, est exploitée ici comme guide de conception pour proposer une architecture modulaire pour le coeur algorithmique. L'analyse de ces dépendances permet de guider la construction d'une architecture concrète générique, supportant 6 algorithmes différents.

En outre, le développement de la section précédente repose sur une interface sémantique qui découple la sémantique des modèles vérifiés des algorithmes de vérification. Ce découplage abstrait est exploité dans cette section pour la conception d'une interface matérielle permettant ce découplage au niveau matériel. Cette interface matérielle générique permet ainsi l'implémentation indépendante de la sémantique du modèle vérifié et du coeur algorithmique, en n'exposant au backend algorithmique les seules fonctions nécessaires à son exécution. Ainsi, cette approche est structurellement composable : l'introduction d'un nouvel interpréteur conçu pour un langage de spécification de modèles est transparente pour le backend algorithmique, et il en est de même pour l'introduction d'un algorithme du point de vue de la sémantique.

2.2.1 Overview

L'architecture du Model-Checker Menhir est représentée dans la figure 2.5. Sa structure est organisée en deux couches paramétriques. Tout d'abord, le *Model-Frontend* encapsule la sémantique du langage. Cette couche est composée du *Next State Generator*, qui encapsule la représentation de l'automate produit par la composition du modèle avec

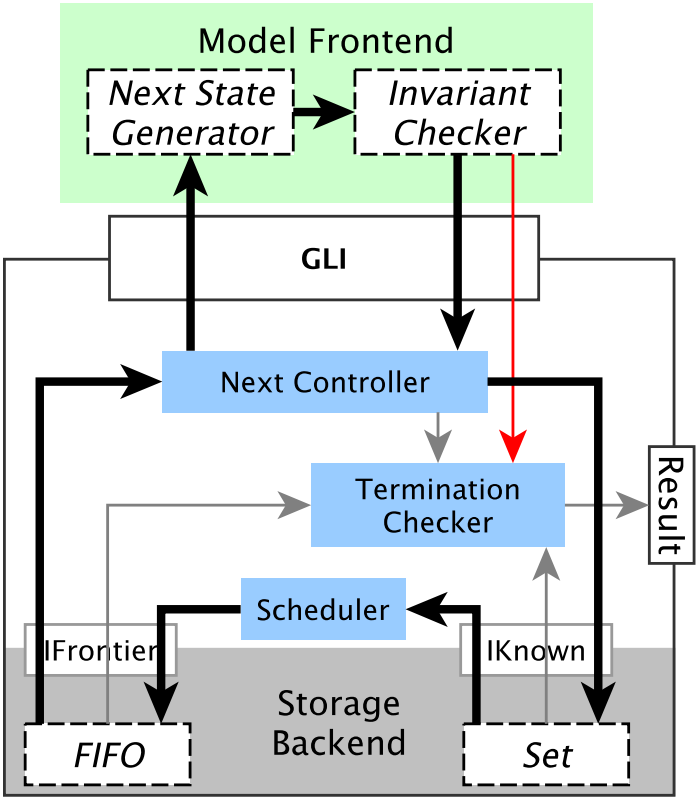


FIGURE 2.5 – Vue d'ensemble de l'architecture. Les cases en pointillés représentent des emplacements pour des modules spécifiques : les composants spécifiques au langage sur le *Model-Frontend*, et les implémentations des ensembles Known et Frontier dans le backend de stockage. Les rectangles bleus sont des composants internes du contrôleur Menhir. Les flèches montrent le flux des configurations (flèches épaisses) et des signaux de contrôle.

la propriété vérifiée. L'*Invariant Checker* encode de manière intentionnelle et générique l'ensemble des états d'acceptations du produit *modèle* $\otimes$ *propriété*. Ces composants sont cachés à l'algorithme de vérification au travers d'une interface langage générique, qui assure la médiation avec le noyau de vérification. En outre, pour la vérification de prédicat sur un état, l'*Invariant Checker* peut être utilisé directement pour évaluer ce prédicat, évitant d'effectuer la composition avec un NFA trivial.

Le noyau de vérification est également développé d'une manière modulaire, composé de deux parties : un ensemble de contrôleurs constituant le coeur algorithmique, associé à un backend, responsable du stockage, correspondant virtuellement aux composants décrits dans la section précédente. L'architecture globale est pilotée par un ensemble de contrôleurs, représentés par des boites bleues sur la figure 2.5 qui ordonnancent l'exécu-

tion.

Coeur algorithmique

Le coeur algorithmique de Menhir assure la coordination et l'ordonnancement des requêtes entre les composants. Il est composé de trois modules indépendants que sont le *Next Controller*, le *Scheduler*, et le *Termination Checker*.

— Le *Next Controller* se place, et assure l'isolation entre le backend algorithmique et le *Model-Frontend*. Il peut être vu comme un sérialiseur de l'ensemble des successeurs N : A partir d'un état source, il demande au modèle le prochain successeur, jusqu'à avoir reçu le dernier élément de N, avant de demander un nouvel état source au *Storage Backend*.

— Le *Scheduler* transmets les configurations nouvellement découvertes à l'ensemble des états déjà parcourus *Frontier*.

— Le *Termination checker* observe l'état de chacun des composants, et détermine la terminaison de l'algorithme.

Ces contrôleurs simples pilotent l'exécution de l'algorithme, et interagissent avec le *Storage backend* qui encapsule les implémentations concrètes des composants *Known* et *Frontier* décrits dans la section précédente.

Storage backend

Le *Storage Backend* héberge les représentations de l'ensemble connu et de l'ensemble frontière. Ces représentations sont exposées au coeur algorithmique par le biais de deux interfaces génériques : *IKnown* et *IFrontier*.

Ces deux interfaces simples correspondent fonctionnellement respectivement à l'interface d'un buffer, et celle d'un ensemble : L'interface *IKnown* expose une fonction add_if_absent, tentant l'ajout d'une nouvelle configuration dans l'ensemble K, et renvoyant une valeur booléene représentant le succès de l'opération. L'interface *IFrontier* est composée de deux fonctions *add*, et *get*, associées a des signaux de statut remontant l'information de remplissage de l'ensemble. On peut noter que l'ordre d'ajout-retrait n'est pas spécifié ici, mais laissé libre à l'implémentation concrète de F.

Ces opérations s'appliquent aux configurations de manière pipelinée. La Fig. 2.5 montre le chemin d'une configuration (flèches noires épaisses). Une nouvelle configuration n est générée par le *Next State Generator*, représentant la relation de transition du modèle.

Ce dernier produit soit une configuration **initiale** (au début) soit l'état **suivant** à partir d'une configuration source x. La nouvelle configuration passe par l'*Invariant Checker* qui affirme le prédicat **is_safe**. Si la configuration est sûre, elle est transmise à l'ensemble des états connus au travers de l'interface *IKnown*, isolant le coeur algorithmique de l'implémentation effective de l'ensemble des états parcourus, noté K dans la spécification proposée précédemment.

Si la configuration n'est pas déjà incluse dans l'ensemble K, elle est ajoutée et transmise à l'ordonnanceur, qui l'ajoute à l'ensemble *Frontier* $\mathcal{F}$.

En fonction de l'ordre imposé par l'implémentation de l'ensemble *Frontier*, une configuration frontière est sélectionnée comme source et envoyée au *Next State Generator* dès qu'elle a fini de produire les voisins de la source précédente.

Terminaison de l'algorithme

L'algorithme implémenté ne se termine que dans deux cas, conformément à la clause *TheEnd* de la spécification présentée dans la section précédente, et rappelée ici dans la figure 2.6.

FIGURE 2.6 – Etape *Step* de la spécification 1.4

$$TheEnd \triangleq \quad (\neg I \wedge F = \{\} \wedge \text{UNCHANGED } \langle K,\ F,\ I,\ S\rangle) \vee \neg S$$

L'algorithme termine tout d'abord lorsqu'un état est annoté comme *non sur* par l'*Invariant Checker*. Dans l'antithèse, la terminaison est acquise si l'ensemble d'états restant à parcourir (*Frontier*) est vide. Il est à noter que cette condition n'est pas suffisante pour détecter la terminaison de l'algorithme, un prédicat de terminaison plus exact serait *"Tous les états devant être parcourus l'ont été"*.

En effet, l'ensemble d'états à parcourir *Frontier* peut être temporairement vide, alors que des états sont en cours de traitement en amont de la pipeline. La terminaison est alors subordonnée à deux conditions : 1. L'ensemble *Frontier* est vide ; et 2. aucun état est en traitement en amont de l'ensemble *Frontier*. . Il faut en outre assurer que rien ne change : Des états peuvent être en cours de traitement.

Aussi, pour permettre une décision de terminaison juste, chacun des composants matériels doivent remonter au *Termination Checker* l'information que des états sont en cours

de traitement. Ces remontées d'information sont représentées par les flèches grises sur le schéma d'architecture 2.2.

2.2.2 Isoler le modèle du noyau de vérification

L'ensemble de composants décrits corresponds au coeur algorithmique de Menhir. Ce dernier doit communiquer avec le *Model-Frontend*, qui encapsule la relation de transition du modèle vérifié. Visant une généricité de l'architecture en termes d'implémentation concrète de la sémantique, il est nécessaire de proposer une interface claire isolant la sémantique du modèle vérifié.

La description fonctionnelle du modèle, présentée dans la Section 2.1 constitue une bonne base pour réaliser cette isolation. La spécification propose d'inliner la vérification d'invariant sur les états dans la fonction de transition du modèle qui renvoie alors la configuration, associée à un bit de statut, correspondant à la *sûreté* de l'état en question.

La traduction matérielle de cette interface est alors naturelle. La figure 2.7 expose les signaux matériels de l'interface *Generic Language Interface*, qui implémente le modèle dans le Listing 2.2. Conceptuellement, cette interface établit un canal de communication entre le frontend du modèle et le noyau de vérification, qui utilise des flux pour implémenter l'accès aux ensembles initial et suivant. Ce canal transfère les nouvelles configurations (*target*) ainsi que le booléen **is_safe** de manière simultanée.

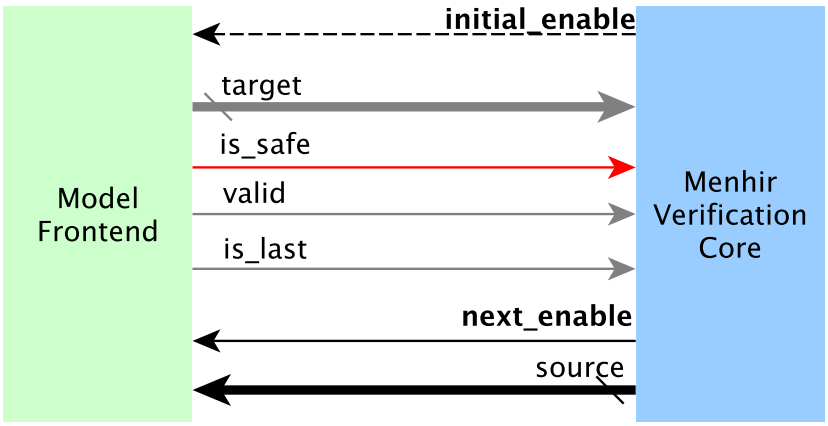


FIGURE 2.7 – Les signaux GLI entre le *Model-Frontend* et coeur algorithmique

Lorsque le signal **initial_enable/next_enable** est affirmé, le *Model-Frontend* produit une nouvelle configuration et affirme le signal **valid**. Pour la dernière configuration de

l'ensemble initial/suivant, le signal **is_last** est affirmé. L'état source s, nécessaire au calcul du prochain ensemble, est envoyé sur le canal **source** lorsque le signal **next_enable** est affirmé.

Les deux fonctions **initial** et *next* sont mutuellement exclusives : L'ensemble des configurations initiales est produit avant qu'un appel à *next* ne soit effectué. Cette exclusivité permet de mutualiser les signaux de communication entre le front-end du modèle et le noyau de vérification pour les deux transferts, réduisant ainsi les ressources matérielles utilisées.

En outre, cette structure de la GLI permet également de détecter la présence de deadlocks dans le modèle, correspondant à un état du modèle ne comportant pas de successeurs. Cette détection, optionnelle, peut être activée via une option du *Termination Checker*. Un deadlock peut arriver dans un modèle soit de manière triviale, si le modèle ne possède aucun état initial ($\mathbf{m.initial} = \emptyset$), soit si l'ensemble de successeurs d'une configuration donnée s est vide ($\mathbf{m.next}(s) = \emptyset$). Ce cas ne peut être représenté par le seul signal *valid* : un niveau bas de ce dernier représente le fait que le modèle est en train de calculer le successeur suivant, ou l'absence de toute transaction. Aussi, le cas du deadlock est représenté par *Model-Frontend* en affirmant le signal **is_last** sans affirmer le signal **valid**.

Pour permettre l'utilisation d'un langage de modélisation existant, le *Model-Frontend* présenté dans la Fig. 2.7 peut encapsuler un processeur sur lequel le runtime du langage peut être exécuté. Dans cette configuration SoC, le processeur est lié au coeur de vérification Menhir par un bus, qui expose directement les registres du *Model-Frontend* dans l'espace adressable du processeur.

Le logiciel exécuté sur le processeur est réduit à un pilote minimal mappé directement sur le GLI matériel. Ce pilote se présente sous la forme d'une couche d'adaptation, basée sur le Listing 2.2, pour intégrer des runtimes des langages de modélisation existants.

2.2.3 Variabilité algorithmique du coeur de vérification

La variabilité de Menhir s'articule autour de deux axes principaux. Outre la variabilité sémantique permise par la *Generic Language Interface* présentée dans le paragraphe précédent, l'architecture modulaire du coeur de vérification, construite à partir d'une spécification abstraite des algorithmes de Model-Checking, permet une forte variabilité algorithmique.

Historiquement, ce type d'architecture fortement modulaire a été exploité par les outils

de vérification formelle modernes, tels que Divine[11], OBP2[27], ou LTSMin [30], [55], [56], pour obtenir des résultats extraordinaires en termes d'extensibilité.

L'intérêt en termes de performances a déjà été démontré par des travaux précédents, l'objectif ici est de proposer une architecture matérielle modulaire présentant les mêmes caractéristiques en termes de modularité que son pendant logiciel, permettant ainsi à la fois une variabilité algorithmique forte, mais également une évolutivité simple, ouvrant la voie à une exploration architecturale incrémentale.

Dans cette section, nous nous concentrons sur quatre structures de données simples qui représentent l'épine dorsale de 6 algorithmes de Model-Checking explicite.

La figure 2.8 illustre par un diagramme de variabilité ces variantes d'implémentation présentes dans le noyau Menhir.

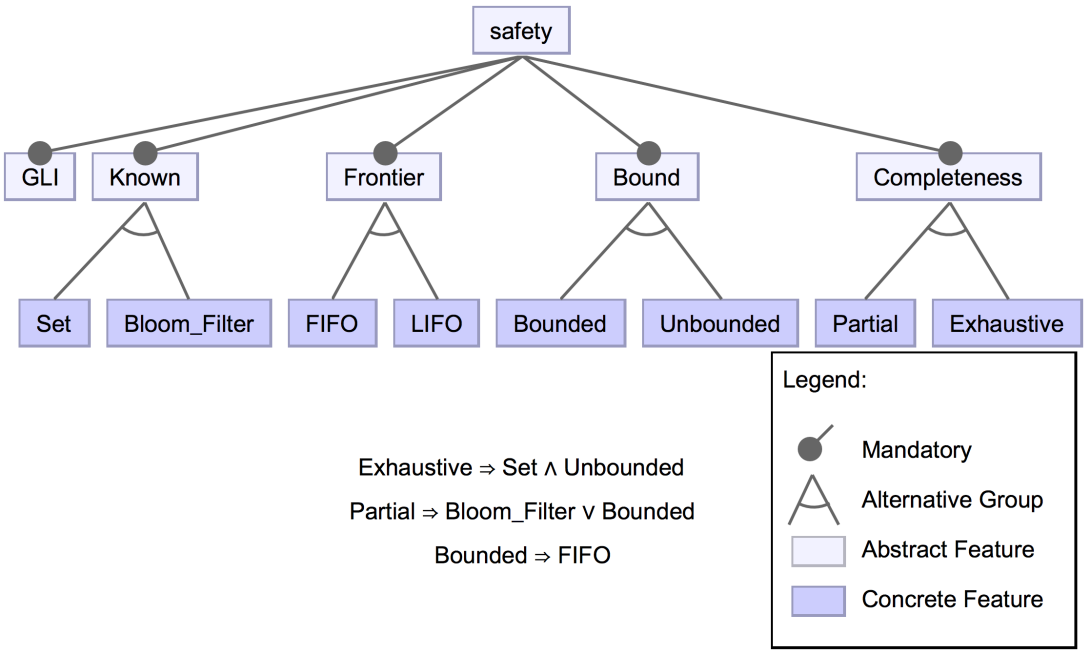


FIGURE 2.8 – Le modèle de variabilité du noyau Menhir

Menhir possède 3 axes de variabilité principaux (le GLI, l'ensemble connu et l'ensemble frontière) et deux caractéristiques dépendantes (Bound et Completeness).

Pour que l'exploration effectuée soit exhaustive, l'ensemble des états parcourus, représenté par le composant *Known*, doit mettre en oeuvre une représentation déterministe d'ensemble, ne produisant pas de faux-positifs. De plus, sans hypothèse sur le modèle vérifié, l'analyse ne doit pas être bornée en profondeur.

Dans le cas contraire, correspondant au cas où l'ensemble d'états parcourus est repré-

senté par une structure de données probabiliste, comme un filtre de Bloom par exemple, l'analyse obtenue ne sera que partielle.

Notez que, dans le cadre de cette étude, nous limitons l'analyse bornée à un algorithme de profondeur bornée basé sur FIFO (Bounded $\Longrightarrow$ FIFO), laissant d'autres approches, telles que [57], pour des travaux futurs.

Variantes de l'ensemble des états parcourus

L'ensemble des états parcouru K est représenté par le composant *Known* au travers de l'interface *IKnown*. Ce composant, dont la description fonctionnelle a été décrite dans la section précédente, expose une fonction **add_if_absent** [58]. Cette fonction ajoute une configuration dans l'ensemble K si elle n'y était pas déjà incluse, et renvoie soit la configuration ajoutée, soit null si la configuration était déjà dans l'ensemble.

Ce composant *Known*, est généralement implémenté en se basant sur une mémoire associative, comme une table de hachage, qui offre une complexité d'ajout dans l'ensemble en temps constant amorti.

Une telle implémentation, déterministe, permet, sous l'hypothèse que l'ensemble des états en instance d'être parcourus F soit complet, de garantir que l'espace d'état est parcouru exhaustivement. Cette exhaustivité du parcours de l'espace d'état sous-entends que la propriété vérifiée l'a été sur chacun des états de l'espace d'état, permettant ainsi d'obtenir une preuve d'absence de fautes dans le modèle.

Une telle preuve est désirable, mais se heurte rapidement au problème d'explosion combinatoire mentionné dans le premier chapitre, car l'espace d'état doit être stocké dans son intégralité dans l'ensemble K, impliquant un besoin mémoire important.

Dans un certain nombre de cas, particulièrement lors de la vérification de modèles industriels, réalistes, le volume mémoire requis pour effectuer une telle tache de vérification est trop important pour que la tâche de vérification termine avec succès.

Parmi les méthodes proposées pour permettre la vérification de modèles trop grands pour être parcourus exhaustivement, le bitstate-hashing est une technique basée sur l'utilisation d'une implémentation probabiliste d'ensemble pour stocker l'espace d'état. Cette technique, exploitant un filtre de Bloom pour l'implémentation de la fonctionnalité *add_if_absent*, permet le stockage d'un seul bit par état stocké. Cette compacité de la représentation de l'ensemble K viens avec un compromis important : la présence de faux positifs.

Ces faux positifs correspondent, dans ce cas, à des états non encore découverts annotés

comme déjà parcourus par le *Known*. D'un point de vue de l'exploration, ces faux positifs sont des états nouvellement découverts qui ne seront pas ajoutés dans la *Frontier*, et par conséquent, leurs successeurs ne seront pas découverts. L'implémentation de l'ensemble des états connus par un filtre de Bloom permet une scalabilité très importante pour la détection de bugs, fournissant une preuve de présence d'erreurs.

Variantes de l'ensemble frontière

L'ensemble d'états nouveaux à parcourir F est réalisé à travers l'interface *IFrontier*. Cette interface expose deux fonctions décrites dans le paragraphe 2.2.1, correspondant fonctionnellement à l'entrée du buffer, et la sortie du buffer. Le composant *Frontier*, qui implémente cette interface, a été spécifié dans la section précédente en laissant une liberté sur la discipline d'ordonnancement du buffer, précisant uniquement qu'un état doit être sélectionné, et retiré de l'ensemble. Aussi, l'ensemble frontière peut être implémenté, par exemple, sous forme de pile (LIFO), ou sous forme de file d'attente premier entré-premier sorti (FIFO). Selon la discipline de l'ensemble frontière, l'algorithme de vérification analyse l'espace d'état soit en profondeur, soit en largeur.

Model-Checking borné

Les quatre variantes du noyau de vérification sont augmentées de deux additionnelles en introduisant le paramètre *Bound*. Ce paramètre correspondà l'implémentation d'un algorithme de Model-Checking borné en profondeur.

Dans ce cas, le noyau de vérification exécute un algorithme BMC explicite, avec ou sans bitstate-hashing [20]. Dans cette configuration, le noyau ne stocke que deux couches de l'espace d'état en mémoire, la couche actuelle dans la frontière et la couche suivante dans l'ensemble connu. Ce comportement est obtenu en stockant l'ensemble frontière dans une FIFO à deux étages. Alors que le premier étage stocke les états restants de la couche courante, le second étage stocke les états de la frontière de la couche suivante. Une fois que la couche actuelle est entièrement explorée (c'est-à-dire que le premier étage est vide), l'ensemble *Known* est effacé, et les étages de la FIFO sont échangés.

La modularité de l'architecture proposée ici est bien illustrée par la possibilité d'implémentation d'un tel algorithme : les seuls changements à opérer pour l'implémentation d'un algorithme de Model-Checking borné en profondeur sont l'implémentation d'une structure de données à double étage, ainsi que l'ajout de la possibilité de vider l'ensemble *Known*.

La section suivante présente une évaluation de performances de l'implémentation proposée sur les 6 algorithmes implémentés, et, en complément, illustre l'intérêt d'une interface sémantique matérielle par l'implémentation de trois langages d'entrée différents, répondant à différents cas d'usage de la vérification formelle.

2.3 Résultats expérimentaux

Après avoir présenté Menhir, un coeur de vérification matériel conçu pour permettre à la fois une variabilité algorithmique importante, et un caractère agnostique au formalisme utilisé pour la spécification des modèles vérifié, on se propose dans cette section d'illustrer ces caractéristiques par une série d'évaluations.

Cette évaluation de l'architecture vise à confirmer les choix de conception selon les deux axes considérés : sémantique et algorithmique.

2.3.1 Mise en place de l'évaluation

Sur le plan algorithmique, les 6 algorithmes proposés dans la figure 2.8 sont évalués sur la base d'un modèle unique :

Modèle d'évaluation. Les résultats présentés sont basés sur un modèle paramétrique, décrit dans le Listing 2.4. La configuration du modèle est un tableau de n bits. La configuration initiale, représentée par la fonction constante **initial**, attribue 0 à tous les bits de configuration. La fonction **next** renvoie l'ensemble des configurations obtenues en retournant un bit dans la configuration source s.

Bien que simple, ce modèle présente un espace d'état exponentiel (2^n configurations), le pire cas pour une configuration à n bits. De plus, la structure de transition du modèle *nbits* expose un haut degré de non-déterminisme, pour chaque configuration s il y a n configurations cibles. En outre, pour les besoins de l'évaluation, nous supposons que tous les états sont sûrs, ce qui induit le pire temps d'exécution pour le Model-Checking (si aucune violation n'est trouvée, l'analyse continue pour toutes les configurations 2^n).

```
1  def nbits (w ∈ N+) : M_H C :=
2  ⟨initial ← { (n,T)|∀ i ∈ [0,w), n_i=0 },
3   next ← λs, { (n,T)| ∃ i ∈ [0,w), n_i=¬s_i }⟩
4
```

Listing 2.4 – Modèle paramétrique *nbits*

Pour illustrer la généricité sémantique de l'architecture Menhir, ce modèle *nbits* est implémenté dans trois formalismes de spécification formant un dégradé depuis un langage de modélisation industriel, jusqu'à une implémentation dédiée à une tâche de vérification.

Modèle en VHDL synthétisable

Une première approche pour représenter la relation de transition du modèle 2.4 est son implémentation directe dans un langage de description matérielle, en l'occurrence ici VHDL. Dans ce cas, le composant représentant le modèle implémente directement l'interface langage générique proposée précédemment, et encode les deux fonctions *next(_)* et *init()*. Cette implémentation est peu coûteuse en ressources, et permet une très faible latence de l'appel de ces deux fonctions. Cependant, du point de vue de la modélisation, cette implémentation est de très bas niveau, et rends difficile la vérification de modèles réalistes sans implémenter un générateur de code.

Modèle interprété : SoC

Formaliser un modèle à vérifier dans un langage de description matérielle peut s'avérer problématique pour plusieurs raisons. Tout d'abord, cette implémentation est coûteuse en temps, et difficilement réalisable par un non spécialiste. De plus, le fossé sémantique entre les langages de spécification, souvent non déterministes et composés d'opérateurs complexes, est très important. Visant à rendre réaliste la vérification accélérée en matériel, on se propose ici d'implémenter deux langages de spécification existants, respectivement utilisés dans un contexte académique et industriel. Pour cela, le modèle, exécuté en logiciel sur un processeur ARM, est connecté au coeur de vérification via un bus AXI4. L'interface langage générique est implémentée au travers d'un adaptateur, qui expose via un bus des registres de contrôle et de données au processeur. Côté logiciel, un driver simple fait l'interface entre le ces registres et la relation de transition.

Dans ce cas, cette adaptation a été implémentée pour deux langages :

Le langage UML [59], un langage de spécification de haut niveau couramment utilisé dans l'industrie pour la spécification de systèmes complexes. Parmi les nombreux efforts de recherche qui se concentrent sur la vérification des modèles UML, l'approche EMI [27] propose une approche intégrée, permettant d'effectuer la vérification directement sur le modèle exécuté, éliminant le besoin d'effectuer des transformations de modèles, qui introduisent un fossé sémantique potentiellement source d'erreurs. Cette approche

offre un runtime exécutable, qui peut être embarqué sur un processeur sans besoin d'un système d'exploitation, et est supporté nativement par le Model-Checker OBP2[27] qui servira de baseline pour notre évaluation. De plus, l'interpréteur offre une API d'exécution contrôlable qui a été utilisée pour l'implémentation de l'interface langage avec le coeur algorithmique.

Le langage DVE a été choisi comme langage typique du Model-Checking. Il est utilisé par deux outils logiciels de Model-Checking haute-performance : Divine [11] et LTSmin [30]. Ce langage permet la description de machines à états communicantes, classiquement utilisées pour la représentation de protocoles de communication, ou de problèmes d'ordonancements, idiomatiques pour la vérification par Model-Checking. Contrairement au langage de spécification précédent, celui-ci n'est pas interprété, mais peut être compilé vers du C via un générateur de code de l'outil Divine. Ce générateur produit un code C optimisé conforme à l'interface Divine CESMI [60], qui est fonctionnellement proche de l'interface langage générique présentée dans le Listing 2.7.

Plate-forme d'évaluation

Les évaluations ont été réalisées sur une plate-forme Zynq XC7Z020-CLG484. La configuration du SoC utilise un coeur ARM fonctionnant à 667 MHz, utilisée pour les langages UML et DVE. Les outils logiciels n'étant pas inter-compatibles, deux baselines ont été définies pour l'évaluation des performances :

— OBP2 EMI[61], utilisé comme référence pour UML, a été exécuté à l'aide d'OpenJDK v1.8.0, sur la plateforme Zynq utilisant Debian 8.

— Divine v3.3.3 [11], utilisé comme référence pour DVE, a fait l'objet d'une compilation croisée et a été exécuté sur la plate-forme Zynq à l'aide de gcc 5.4.

L'implémentation matérielle - synthèse logique et physique - a été effectuée via la suite Xilinx Vivado 2018.3, et est cadencée à une fréquence de 100MHz.

2.3.2 Présentation des résultats

Dans ce qui suit, les résultats des mesures de performances obtenues par notre moteur de vérification matériel sont comparés à leur référence logicielle. Ces deux outils définis comme référence sont limités à des tâches de vérification exhaustive, aussi, la comparaison des temps d'exécution n'a de sens que dans cette configuration.

Impact de l'implémentation du modèle

Une première question à se poser est celle de la compétitivité d'une architecture SoC, permettant la réutilisation des sémantiques existantes, au sein d'un moteur de vérification matériel. La figure 2.9 montre de facteur d'accélération pour un parcours en largeur par rapport aux références logicielles définies, en fonction de la taille du modèle considéré. Sept points de mesures ont été effectués, correspondant à une taille de configuration allant de 9 à 15 bits.

La ligne noire représente le rapport entre les temps d'exécution de OBP2 EMI et la configuration EMI-SoC utilisant le modèle UML. Dans ce cas, le gain de performance est compris entre 36X et 22X.

La ligne pointillée représente le rapport entre les temps de fonctionnement de Divine et de la configuration DVE-SoC à l'aide du modèle DVE. Dans ce cas, le gain de performance varie entre 23X et 5X.

La ligne grise montre le gain de performance obtenu par la configuration matérielle complète (modèle GLI basé sur VHDL) par rapport à Divine. Dans ce cas, le gain de performance est compris entre 296X et 50X.

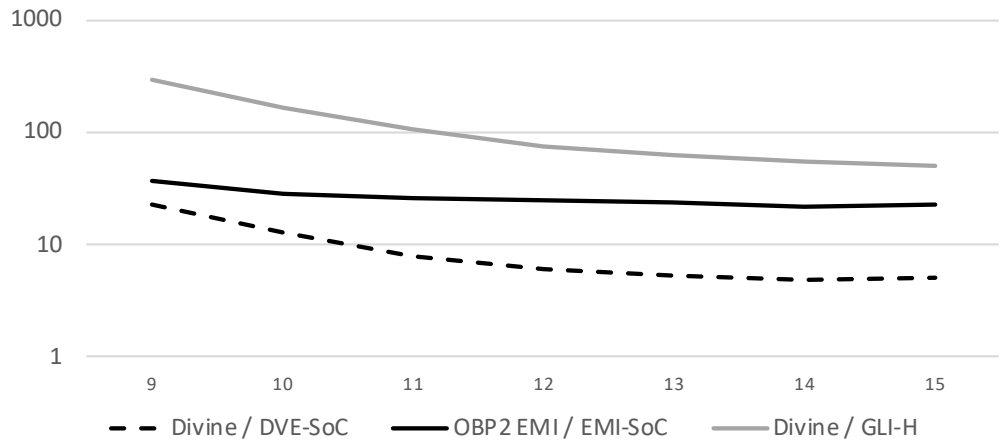


FIGURE 2.9 – Résultats SoC par rapport à Divine pour DVE et OBP pour UML

Les gains de performance élevés pour les petits modèles, dans la Fig. 2.9, illustrent le temps d'initialisation plus rapide moteur de vérification Menhir par rapport aux baselines logicielles. Cependant, lorsque le modèle devient plus grand, ce facteur pert de son influence et les gains de performances atteignent un plateau.

Ces résultats montrent une forte dépendance entre le temps d'exécution global de la

tâche de vérification et la latence du calcul de la relation de transition : L'implémentation directe du modèle en VHDL est la plus rapide, atteignant un débit proche d'une d'une configuration par cycle. La version DVE-SoC, pour laquelle le modèle est compilé en C, puis adapté directement au driver logiciel, est en effet beaucoup plus rapide que EMI-SoC, qui repose sur un interpréteur logiciel pour l'évaluation des transitions. D'après ces résultats, la configuration EMI-UML est en moyenne 14 fois plus lente que la vérification DVE SoC. Il est cependant à noter que les baselines logicielles OBP2-EMI et Divine présentent un facteur 52 en termes de temps d'exécution, en faveur du Model-Checker Divine.

Ces résultats permettent difficilement de conclure sur la compétitivité d'une archietcture SoC pour la vérification accélérée sur FPGA, dans la mesure où le processeur utilisé n'est pas représentatif des processeur hautes performances utilisés pour la vérification classique. Cependant, on montre ici un gain important par rapport au Model-Checker OBP2 dans la configuration EMI-SoC à processeur égal. Il serait ainsi intéressant de faire une étude des performances potentielles obtenues dans le cas de l'utilisation d'un processeur compétitif. Cependant, ces mesures illustrent la capacité du moteur de vérification à réutiliser des sémantiques existantes, grâce à l'introduction d'une interface langage générique.

Impact des algorithmes utilisés

La figure 2.10 illustre les résultats obtenus sur un modèle de 15 bits pour trois algorithmes non exhaustifs. La ligne rouge représente le temps de référence, choisi ici comme un parcours exhaustif en largeur.

Model-Checking borné : La ligne noire épaisse représentée dans la figure 2.10 représente le temps d'analyse en fonction de la limite en profondeur fixée pour l'analyse bornée. Il est à noter que le diamètre d'atteignabilité du modèle est de 16 par construction, l'analyse parcours donc exhaustivement le modèle à partir de ce point. Les résultats montrent que même si le Model-Checking borné réexplore des noeuds, pour une borne de profondeur faible, elle peut être plus rapide que l'analyse non bornée (le croisement s'opère pour une borne égale à 9 dans la figure). Lorsque l'analyse dépasse le diamètre d'atteignabilité, le BMC est 5.5 fois plus lent que la version exhaustive, dû au re-parcours de certains états.

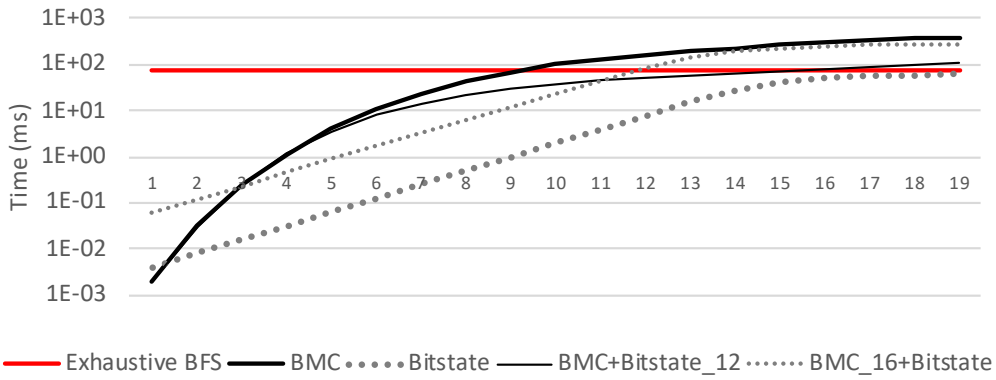


FIGURE 2.10 – Temps d'exécution de la vérification partielle.

Bitstate hashing : La ligne pointillée épaisse, dans la Fig. 2.10, indique le temps d'exécution de l'algorithme implémentant *bitstate-hashing* tout en faisant varier la largeur d'adressage du filtre bloom de 1 à 19. Bien que cette configuration divise les besoins en mémoire par un facteur proportionnel à la largeur de la configuration, elle perd la certitude d'une analyse exhaustive, à cause des faux-positifs potentiels. D'autre part, nous pouvons observer que cette analyse partielle est plus rapide que BMC, et atteint un temps d'exécution similaire à BFS lorsque le filtre bloom est suffisamment grand.

Hybride – BMC & Bitstate Hashing : La fine ligne noire, dans la Fig. 2.10, montre le temps d'exécution dans le cas d'un algorithme hybride qui combine Model-Checking borné - BMC - et bitstate-hashing. Dans ce cas, la taille du filtre de Bloom a été fixée à 2^{12} bits et la borne varie dans une plage de 1 à 19, comme dans le cas du BMC. Ici, nous pouvons observer que l'analyse correspond presque au temps d'exécution du BFS exhaustif lorsque la limite s'approche du diamètre d'atteignabilité. Cependant, la couverture de l'espace d'état est beaucoup plus faible en raison de la taille insuffisante du filtre bloom.

La ligne fine en pointillés, dans la Fig. 2.10, illustre le résultat dans une configuration hybride similaire qui fixe la limite de BMC au diamètre et fait varier la taille du filtre bloom. Il n'est pas surprenant que dans ce cas, le temps d'analyse suive la tendance de la configuration de bitstate-hashing (représentée par la ligne épaisse en pointillé). Ce qui est intéressant à noter cependant, c'est qu'en pratique, ces deux configurations hybrides permettent réguler la pression mémoire et le temps d'analyse pour une analyse non exhaustive (c'est-à-dire des tests), en influant sur les deux méta-paramètres de l'exploration :

borne et taille du filtre de Bloom.

Cet algorithme est unique, et n'a encore, à notre connaissance, jamais été proposé dans la littérature. Aussi, il constitue une illustration supplémentaire de l'intérêt d'une architecture générique et modulaire, permettant la construction naturelle de nouveaux algorithmes par composition.

Synthèse

Le processus de développement d'algorithmes dédié à une plateforme est nécessairement incrémental : Après une première implémentation basée sur l'intuition, seule la mesure et la comparaison permet de converger vers une implémentation optimale.

L'ensemble de mesures présenté en faisant varier les algorithmes utilisés montre l'intérêt majeur d'une architecture générique dans ce processus. En effet, l'implémentation d'un algorithme n'implique que de faibles modifications sur l'architecture elle-même, et aucune sur sa structure. Ainsi, l'introduction d'un composant pour implémenter un algorithme nouveau peut se faire en conservant le reste de l'architecture, permettant ainsi de mesurer précisément l'influence de chaque composant dans la performance globale.

2.4 Conclusion

Ce chapitre montre comment obtenir un coeur de Model-Checking générique en partant d'un spécification algorithmique de haut niveau. Le regard dataflow porté sur la spécification nous a permis d'identifier et d'isoler des composants fondamentaux d'une l'architecture abstraite, qui sert de guide dans la conception de l'architecture concrète du coeur de vérification.

En outre, ce coeur de vérification reprends des innovations architecturales des Model-Checkers logiciels modernes, comme LTSmin et Divine, en permettant notamment d'adresser plusieurs langages d'entrée, via l'introduction d'une interface langage générique.

Menhir peut être vu comme un *proof of concept* permet par sa modularité l'implémentation à la fois de plusieurs langages, mais aussi de plusieurs algorithmes, ouvrant ainsi la porte à un développement incrémental, partant d'un socle commun pour permettre l'évaluation et la comparaison des solutions.

Chapitre 3

Carnac : Variabilité algorithmique pour la vérification swarm efficace

L'accélérateur générique proposé dans le chapitre précédent propose l'implémentation, au sein d'une même architecture, d'un ensemble d'algorithmes formant un dégradé depuis des parcours exhaustifs, vers des parcours partiels, permettant d'adapter discipline de parcours à la tâche de vérification effectuée.

En complément des techniques d'abstraction et de réduction de symétries, les parcours partiels, vérifiant les propriétés sur une sous-approximation de l'espace d'état, permettent l'étude de modèles trop complexes pour être vérifiés par des techniques classiques - exhaustives. Dans ce cas, seul un sous-ensemble de l'espace d'état est couvert par la tâche de vérification, laissant ainsi la possibilité que des fautes restent non détectées.

Pour augmenter la proportion de l'espace d'état couvert par la vérification, une technique, dite *vérification swarm*, consiste à exécuter plusieurs tâches de vérification sur un même modèle en diversifiant les parcours. Une manière d'obtenir cette diversification est par exemple de faire varier les fonctions de hachage utilisées par chacune des tâches pour le stockage de l'espace d'état, typiquement effectué par un filtre de Bloom. De cette manière, les faux positifs, dus à l'existence de deux états possédant le même hash, se produisent pour des états différents, impliquant le parcours d'un sous-ensemble différent de l'espace d'état.

Cette technique se prête naturellement à l'accélération matérielle sur FPGA, bénéficiant à la fois du parallélisme interne de chaque coeur de vérification, mais également d'un parallélisme de tâches, qui sont réparties sur plusieurs coeurs. La section 1.4.2 a montré les résultats impressionnants qu'obtient l'accélérateur FPGASwarm dans un tel contexte.

Cependant, un calcul rapide montre la relative inefficacité de cet accélérateur : La

fréquence de découverte d'un nouvel état est de l'ordre d'un état tous les 28 cycles [1]. La cible d'un état nouveau découvert par cycle est certes irréaliste, mais il reste une marge de progression importante.

Visant à améliorer l'efficacité de ce type de vérification, on propose dans ce chapitre d'exploiter la variabilité algorithmique exposée dans le chapitre précédent sous la forme d'une exploration de l'espace de conception, visant à sélectionner un algorithme de vérification swarm qui maximise une métrique d'efficacité de la vérification. Pour un même effort de calcul, la couverture est ainsi maximisée.

Cet algorithme est ensuite intégré à l'architecture de vérification Carnac sous la forme d'un coeur dédié, spécialisation du coeur précédemment proposé visant à maximiser la performance. Ce coeur est intégré à une architecture permettant une distribution des calculs au sein d'une plateforme FPGA multi-die.

L'analyse effectuée sur l'efficacité algorithmique est consolidée par une évaluation expérimentale sur le même cas d'étude que l'architecture concurrente FPGASwarm. Le gain d'efficacité théorique est confirmé par l'évaluation matérielle, montrant une accélération de 3.96x pour un même nombre de coeurs, et une scalabilité plus importante sur des FPGA de grande taille, permettant de doubler le nombre de coeurs en conservant une fréquence 60% plus élevée, augmentant le facteur d'accélération à 7.58x. La fréquence de découverte des états est également significativement améliorée par les améliorations algorithmiques et architecturales, tombant à un état tous les 11 cycles à l'échelle du swarm entier.

1. Le modèle considéré constitue un test de couverture[7], cherchant 100 valeurs aléatoires parmi les valeurs d'un entier 32 bits. Si l'on fait l'approximation d'une couverture de 100%, les 2^{32} états sont parcourus en $f_{clk} * N_{cores} * T_{elasped}$ cycles, soit :

$$f_{discovery} = \frac{2^{32}}{250 * 10^6 * 40 * 12} \approx \frac{1}{28}$$

3.1 Exploration de la variabilité algorithmique pour la vérification swarm

La vérification swarm repose sur l'exécution diversifiée d'un grand nombre de tâches de vérification - *VTask* - sur un même modèle pour assurer statistiquement une couverture élevée de l'espace d'état.

Chacune de ces tâches de vérification sont caractérisées entièrement par les *seeds* des générateurs de nombres aléatoires utilisés par l'algorithme exécuté (fonctions de hachage, lfsr...). Aussi, elles sont indépendantes, et peuvent être exécutées en parallèle. Le caractère indépendant de ces tâches est une forte opportunité de parallélisation, mais implique également une sensibilité importante à l'algorithme utilisé, aucune communication entre les tâches de vérification ne permettant d'éviter la duplication de travail.

De fait, la performance d'une vérification swarm repose sur deux facteurs principaux, et orthogonaux : Le degré de parallélisme tout d'abord, mais également le niveau de diversification entre deux tâches, qui peut être défini comme l'inverse d'une mesure du recouvrement entre deux *VTask*.

Dans cette section, on se propose de se concentrer sur ce dernier point, en exploitant la variabilité algorithmique exposée par la spécification proposée précédemment pour augmenter la diversification des *VTask*.

Cette spécification expose plusieurs points de variabilité, qui sont exploités ici au travers d'une exploration de l'espace de conception algorithmique.

Cette procédure visant à optimiser l'efficacité globale d'une analyse swarm repose sur la minimisation d'une métrique qui réifie le recouvrement entre les espaces d'états révelés pendant l'exécution de l'essaim.

3.1.1 Exploiter la variabilité pour une plus grande diversification

Cette section présente les points de variabilité constituant la base de l'exploration de l'espace de conception algorithmique.

La spécification présentée dans la Section 2.1 identifie deux composants abstraits dont le raffinement mène à la définition de la discipline de parcours de l'espace d'état. On propose ici plusieurs options d'implémentation de ces composants qui permettront de construire les algorithmes qui seront ensuite étudiés au regard d'une mesure d'efficacité.

Le modèle vérifié ainsi que la sémantique des propriétés sont regroupés au sein du composant *Model-Frontend*. Conformément à l'approche Menhir proposée dans le chapitre précédent, ce composant est considéré comme externe au coeur de vérification. Une conséquence directe de ce choix de conception est que l'on cherche à éviter d'y ajouter des contraintes de conception d'ordre algorithmique, de manière à éviter d'y ajouter une complexité additionnelle.

L'article présentant FPGASwarm [6] soutient cependant l'intérêt du tirage aléatoire des transitions pour la diversification de l'exploration. Cette analyse ne serait pas équitable si ce point était négligé. Aussi, on propose l'introduction d'un composant supplémentaire pour permettre l'implémentation de cette randomisation sans apporter de modifications intrusives sur le *Model-Frontend*.

Diversification du *Frontier Stream*

Dans les algorithmes de Model-Checking, le *Frontier Stream* est un buffer stockant les états restant à parcourir. Dans le cas classique, sa profondeur n'est pas bornée, et respecte une discipline d'ordonnancement FIFO ou LIFO dans la plupart des cas. Dans [6], pour s'adapter au faible volume de mémoire présent sur les FPGA, les auteurs ont proposé d'utiliser une FIFO bornée en profondeur. Cette structure de données présente le même comportement qu'une FIFO standard, mais n'accepte des états que lorsqu'un slot est libre, sinon elle ignore l'état entrant, augmentant ainsi la diversité de l'exploration parmi les tâches VT. L'intuition derrière ce choix doit être approfondie, car elle est critique pour l'efficacité de la tâche de vérification swarm.

Dans le cas d'un buffer borné en profondeur, les politiques d'entrée et de sortie doivent être prises en compte, la première déterminant quels états sont ajoutés au buffer, et la seconde déterminant quel état stocké est renvoyé au *Model-Frontend*.

La politique d'entrée du *Frontier stream* détermine quels états sont stockés dans le tampon. Dans cette étude, deux cas sont considérés pour la **politique d'entrée** du *Frontier stream* : 1. L'état est ignoré lorsque le tampon est plein, ce qui entraîne le stockage des états les plus anciens. 2. L'état entrant écrase l'état déjà présent à l'emplacement du pointeur d'écriture, ce qui conduit au stockage des seuls N états les plus récents dans le tampon. Cette différenciation, pouvant paraître anecdotique, conduit à des algorithmes conceptuellement différents.

Conserver les états jeunes (en écrasant les anciens) conduit à une stratégie d'explora-

tion plutôt en profondeur, dans la mesure où, dans le cas où le *Frontier Stream* est plein, un état qui aurait dû être exploré lors d'un parcours classique DFS, ne le sera pas, et écrasé par un état plus "jeune" - à gauche sur la Fig. 3.1.

Conserver les états les plus anciens (en supprimant les jeunes) conduit à parcours qui privilégie l'exploration en largeur du modèle, sans être stricto-censu un BFS. L'ensemble d'algorithmes qui en découle est représenté sur le côté droit de la Fig. 3.1.

La politique de sortie mise en œuvre par le *Frontier Stream*, est un autre point de variabilité important.

L'extraction de l'état le plus jeune conduit à une traversée de type DFS (depth-first search). L'extraction de l'état le plus ancien conduit à un déroulement de l'espace d'état par une recherche en largeur (BFS). Cependant, il s'agit de deux choix extrêmement particuliers sur le continuum offert par la profondeur du *Frontier Stream*. Nous considérons également une stratégie de sortie aléatoire, qui choisit arbitrairement un membre du *Frontier Stream* comme successeur à traiter par le *Model-Frontend*. Ce choix garantit, au niveau de la tâche de vérification swarm, que deux *VTask* traitent des états différents y compris au début de l'exécution, lorsque le *Frontier Stream* n'est pas encore plein. Carnac implémente cette option à travers l'utilisation pour ce composant d'un buffer pour lequel l'indice du prochain élément sorti par le *Frontier Stream* est déterminé via un LFSR, initialisé avec l'ID du coeur de vérification *VCore* courant, de manière à ce que deux tâches de vérification ne traitent pas les états dans le même ordre.

Les 4 algorithmes utilisant cette stratégie de sortie aléatoire sont présentés en haut, et identifiés par la lettre **R** : (YR_FB et OR_FB) et en bas (MYR_FB et MOR_FB) de la Fig. 3.1.

Diversification du *Known Set*

Pour être complet, nous citons ici la diversification introduite par l'utilisation d'un filtre bloom pour stocker l'ensemble des états connus. Notre approche, similaire à celle de [6], [23], exploite directement cette structure de données probabiliste pour réduire les besoins en mémoire pour le stockage de l'espace d'état, et pour diversifier l'analyse d'accessibilité au niveau de l'essaim en se basant sur les collisions de hachage spécifiques au VCore, assurées par l'initialisation des fonctions de hachage de chaque noyau avec une *seed* différente.

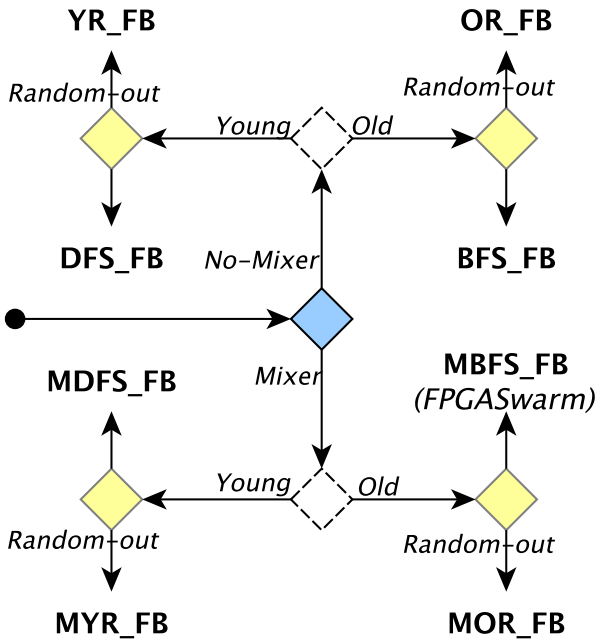


FIGURE 3.1 – Les algorithmes de VCore dérivés des points de variabilité

Diversification de la sortie du *Model-Frontend*

Pour augmenter le facteur de diversification, l'article FPGASwarm [6] introduit un nouvel axe de diversification : *exploration path randomization*. Cette approche rend aléatoire l'ordre des états produits par le *Next-State Generator*, ce qui réduit efficacement la duplication du travail au niveau de l'essaim, fait confirmé également par notre évaluation présentée dans la Section 3.2.

Cependant, cette approche, initialement proposée par Holzmann dans [7], nécessite des modifications intrusives du *Model-Frontend*, un composant hautement complexe encodant de la sémantique du modèle vérifié. En pratique, ces modifications sont difficiles à mettre en œuvre, nécessitant d'introduire des générateurs de nombres aléatoires directement dans l'interpréteur de la sémantique du langage de modélisation, et peuvent conduire à une dégradation sérieuse des performances, voire à des erreurs dans la sémantique. Pour pallier ce problème, notre VCore introduit un nouveau composant au sein de la pipeline, le *Mixer*, qui fonctionne comme un mélangeur dans la pipeline. L'implémentation du *Mixer* est basée sur l'utilisation d'un buffer dont la politique de sortie est aléatoire (utilisé également pour le *Frontier Stream*), qui change l'ordre des états sortant du *Invariant Checker* avant de les envoyer au *Known Set*. Une différence notable avec l'implémentation utilisée pour le

Frontier Stream est l'absence de perte d'états ici.

L'utilisation de la structure de pipeline autorégulée, basée sur le protocole AXI Stream, facilite l'intégration de ce bloc, et permet également la réalisation de différentes stratégies de *exploration path randomization*. Dans le cas le plus simple, ce module peut être réalisé par de simples fils, si le mélange de l'état suivant n'est pas nécessaire. En activant (Fig. 3.1 en bas) ou en désactivant (Fig. 3.1 en haut) le *Mixer*, différents algorithmes d'exploration apparaissent.

Synthèse

Pour illustrer les points de variabilité précédemment présentés, la Fig. 3.1 présente les 8 algorithmes induits ainsi qu'une convention de dénomination basée sur les choix dans l'arbre de décision. La partie supérieure de la Fig. 3.1 montre les algorithmes avec le *Mixer* désactivé, tandis que la partie inférieure montre ceux avec le *Mixer* activé dans le pipeline. Si le *Mixer* est activé, le nom de l'algorithme est préfixé par **M**. La conservation des états les plus anciens dans le *Frontier Stream* (partie droite de la Fig. 3.1) conduit à des algorithmes de type BFS (BFS_FB et MBFS_FB) ou à des algorithmes de sortie aléatoire : old-random-out (OR_FB) et mixed-old-random-out (MOR_FB).

D'autre part, le maintien des jeunes états dans le *Frontier Stream* (partie gauche de la figure 3.1) conduit soit à des algorithmes de type DFS (DFS_FB et MDFS_FB), soit à des algorithmes de sortie aléatoire : young-random-out (YR_FB) et mixed-young-random-out (MYR_FB). Le suffixe **_FB** indique que le *Frontier Stream* est borné. FPGASwarm [6] s'appuie sur un algorithme en largeur avec mixage des successeurs(MBFS_FB), qui laisse tomber les jeunes si la *Frontière* est pleine et extrait l'état le plus ancien stocké dans la *Frontière*.

3.1.2 Métrique d'évaluation

La section précédente identifie plusieurs candidats potentiels permettant la diversification de l'exploration. Cette diversification vise à réduire le recouvrement des tâches de vérification, et à tendre vers le cas idéal, où chacune des tâches de vérification exploreraient une sous-partie disjointe de l'espace d'état.

Pour mesurer ce recouvrement, on considère n tâches de vérification utilisant la même discipline de parcours. On note $\mathcal{S}(k)$ l'ensemble des états parcourus par la k-ième tâche.

L'espace d'état combiné couvert par les n tâches est constitué de $|\bigcup_{k=1}^{n} \mathcal{S}(k)|$ états.

En outre, un total de $\sum_{k=1}^{n} |\mathcal{S}(k)|$ états ont été parcourus par l'ensemble des tâches de vérification.

On définit la métrique d'efficacité comme le rapport de ces deux grandeurs, rapport entre le nombre d'états uniques et le nombre d'états total analysés :

$$\mathcal{E} = \frac{|\bigcup_{k=1}^{n} \mathcal{S}(k)|}{\sum_{k=1}^{n} |\mathcal{S}(k)|}$$

Sous cette définition, une valeur de 1 corresponds à la situation idéale où chacun des $\mathcal{S}(k)$ sont disjoints, et prends la valeur $\frac{1}{n} \xrightarrow[n\to\infty]{} 0$ lorsque toutes les tâches ont parcourus strictement le même espace d'état.

Cette grandeur permet de réifier le recouvrement entre les n tâches dans une grandeur numérique. Cependant, elle ne permet la comparaison d'algorithmes que pour un même modèle, sa valeur absolue étant dépendante de la structure de l'espace d'état du modèle.

Pour permettre la comparaison entre deux algorithmes - λ et λ_{ref} - de manière globale sur un ensemble de modèles, on considère cette métrique relativement à un algorithme de référence λ_{ref} :

$$\mathcal{E}_{rel} = \frac{\mathcal{E}(\lambda) - \mathcal{E}(\lambda_{ref})}{\mathcal{E}(\lambda_{ref})}$$

À l'aide de cette métrique $\mathcal{E}_{rel}$, les efficacités entre les modèles pour chaque algorithme sont centrées autour de 0, et normalisées par rapport à l'efficacité de l'algorithme de référence.

Une valeur négative -0.5 est interprétée comme le fait que l'algorithme correspondant a vu un état, en moyenne, moitié moins souvent que l'algorithme de référence.

3.2 Résultats expérimentaux

Dans cette section, nous présentons l'évaluation des 8 algorithmes présentés dans la Section 3.1.1. Les résultats conduisent à la sélection d'un candidat pour une implémentation matérielle dans le cœur FPGA de Carnac.

3.2.1 Configuration expérimentale

La conception matérielle est un processus lourd de développement, qui ne se prête pas bien à la conception incrémentale d'algorithmes, ni à l'exploration à grande échelle de l'espace de conception nécessaire pour choisir les meilleurs candidats.

Aussi, l'évaluation des algorithmes repose sur un modèle système qui implémente les algorithmes précédemment décrits. D'un point de vue architectural, ce modèle est structuré de la même manière que le coeur matériel présenté dans la Section 3.3, et permet une mise en œuvre simple et la collecte de données nécessaires pour discriminer entre les algorithmes proposés.

Le nouveau système est conçu autour d'un noyau central, interfacé avec une instance du modèle considéré, ainsi qu'avec les implémentations *Known Set*, *Frontier Stream* et *Mixer* spécialisées pour chacun des algorithmes. Chaque algorithme proposé a été implémenté en utilisant cette infrastructure modulaire, qui correspond au schéma architectural matériel.

Modèles étudiés

Pour sélectionner un algorithme présentant une bonne performance moyenne, sans hypothèse a priori sur la structure du modèle, nous avons effectué l'évaluation algorithmique sur le benchmark BEEM[62]. Ce benchmark contient un ensemble de modèles paramétriques spécifiquement conçus pour mettre à l'épreuve les outils de Model-Checking, constitué de problèmes classiques du Model-Checking comme les problèmes d'exclusion mutuelle, d'ordonnancement, ou des protocoles de communication. Notre évaluation a été effectuée sur les 66 plus grands modèles, ayant un espace d'état au moins 4 fois plus élevé que la capacité du *Known Set* : 2×10^6 états. Chaque modèle a été compilé en C++ à l'aide du compilateur DVE de l'outil Divine3 [11] et interfacé avec le modèle système du coeur de vérification.

Protocole expérimental

La vérification swarm ne fournit une condition de terminaison que dans le cas où une propriété est violée. Cependant, dans le cas où le modèle ne contient pas d'états qui violent les propriétés spécifiées, aucune condition de terminaison n'est fournie. Cette perte de la condition [2] de terminaison n'est pas nécessairement un problème dans le cas d'un

2. Une terminaison arbitraire est décidée en pratique, via un *timeout* par exemple, mais ceci ne permet pas de mesurer des performances.

scénario du monde réel, car la vérification swarm est appliquée dans le cas où les techniques de vérification classiques ne peuvent être utilisées, et permet dans ce cas d'accroitre la couverture, et ainsi d'augmenter la confiance dans un modèle à travers une meilleure couverture qu'un algorithme partiel utilisé seul, comme *bitstate-hashing*[20].

Cependant, pour mesurer les performances, ce deuxième cas est problématique, car aucune mesure de durée n'est possible sans l'information de terminaison. Ensuite, cela oblige à exclure un grand nombre de modèles, qui sont connus pour être sans défaillance, et aussi à exclure des modèles pour lesquels le statut des propriétés n'a pas été résolu, en raison de leur taille trop importante par rapport aux ressources mémoire actuelles.

Dans la section précédente, nous avons présenté une métrique d'évaluation, comme un proxy pour mesurer le chevauchement entre les tâches de vérification, car la duplication du travail est la cause première de l'inefficacité de l'essaim. Notre méthodologie d'évaluation est basée sur l'exécution de 500 tâches de vérification, en enregistrant les espaces d'état atteints. Pour mesurer l'efficacité, une hashmap en mémoire est construite, pour associer les états vus avec le nombre correspondant d'occurrences parmi les exécutions. À partir de cette table de hachage, l'efficacité de chaque algorithme a été calculée pour chaque modèle BEEM, comme la proportion d'états uniques vus par l'essaim simulé, par rapport au nombre total d'états vus pendant les 500 cycles. Le nombre d'exécutions a été limité à 500 en raison du coût du post-traitement. Le post-traitement est très gourmand en mémoire et a été effectué sur un supercalculateur SMP avec 6 To de mémoire partagée.

FIGURE 3.2 – L'efficacité algorithmique relative ($\mathcal{E}_{rel}$), par rapport à λ_{ref} = BFS_FB. La légende indique les algorithmes par ordre d'efficacité décroissante.

3.2.2 Résultats de l'évaluation

La figure 3.2 présente la métrique d'efficacité des 8 algorithmes considérés, par rapport à l'algorithme BFS_FB, choisi comme référence en raison de sa similitude avec l'algorithme BFS standard utilisé par les outils de vérification swarm logiciels[55]. L'axe horizontal représente les modèles considérés, triés par rapport à MYR_FB, l'algorithme le plus performant.

L'algorithme DFS_FB, similaire à l'algorithme DFS traditionnel, présente une efficacité similaire à celle de l'algorithme de référence. La version randomisée de cet algorithme, appelée MDFS_FB, présente des résultats légèrement meilleurs en moyenne, mais manque de régularité entre les modèles considérés, ce qui se traduit par des résultats extrêmements mauvais sur certains modèles, et bons sur d'autres.

Ces résultats peuvent mener à la conclusion que les algorithmes orientés vers la profondeur ne conviennent pas bien à la vérification swarm. Cependant, l'algorithme YR_FB, qui sélectionne à chaque itération un état choisi au hasard parmi les n états les plus récemment visités, montre une augmentation considérable des performances par rapport à l'algorithme de référence. Une justification intuitive de ce comportement est que si nous considérons deux cœurs à un moment donné avec le même contenu de *Frontier Stream*, les deux états sélectionnés pour être explorés par l'algorithme ont une grande chance d'être différents, car chaque cœur sélectionne un état aléatoire parmi ceux présents dans le *Frontier stream*.

Cette intuition est confirmée par les résultats relativement médiocres de l'algorithme OR_FB, pour lequel les états successeurs sont choisis aléatoirement parmi les états les plus anciens de la frontière. Ceux-ci ont de grandes chances d'être similaires, voire identiques au début de l'exploration, car les nouveaux états ne sont ajoutés à la frontière que lorsqu'un emplacement libre est disponible. De plus, au début de l'exploration, la probabilité de collision induite par le filtre bloom est faible. La conjonction de ces deux facteurs conduit à une forte probabilité de frontières similaires pour deux tâches OR_FB différentes, ce qui entraîne une importante duplication du travail.

En diminuant la duplication du travail, le MBFS_FB utilisé par FPGASwarm, montre de meilleurs résultats en moyenne, car le réordonnancement des successeurs induit différentes collisions dans l'*Known Set*. De plus, comme différents enfants d'un même état source sont considérés parmi les VCores, le *Frontier stream* est diversifié.

La combinaison des deux conduit à une forte probabilité que deux coeurs différents énumérant le même état conduisent à des états successeurs différents ajoutés dans la

frontière, ainsi qu'à des collisions différentes produites dans l'ensemble connu.

En profitant à la fois de la réorganisation sémantique et de la randomisation de la frontière, l'algorithme MOR_FB présente de meilleurs résultats en moyenne, étant plus efficace que le MBFS_FB. Néanmoins, son efficacité est inférieure à celle de l'algorithme MYR_FB.

Globalement, l'algorithme MYR_FB présente la meilleure efficacité, avec un gain d'efficacité moyen de 249 % par rapport à *BFS_FB* et jusqu'à 144 % par rapport à MBFS_FB sur 72 % des 66 modèles. Cet algorithme permet une plus grande diversification, grâce à la frontière randomisée et continuellement renouvelée utilisée conjointement avec le *Mixer*.

Ces bonnes performances font de MYR_FB le candidat sélectionné pour le moteur de vérification par essaimage de Carnac.

3.2.3 Synthèse

L'évaluation présentée compare un ensemble d'algorithmes de vérification swarm bornés en mémoire, dans le but d'accroitre les performances de la vérification swarm. Dans un contexte swarm, la performance globale est fortement liée à la duplication de travail, chacune des tâches de vérification explorant un espace d'état recouvrant - en partie - l'espace d'état couvert par d'autres tâches. Cette caractéristique est intrinsèque à la méthode, les tâches n'étant pas coordonnées, mais nuit à la vitesse globale de la vérification - ou, vu autrement, nuit à la couverture atteinte dans un quota de temps fixé.

Visant à le réduire, ce recouvrement a été mesuré pour un ensemble de 8 algorithmes adaptés à la vérification matérielle. Ces mesures, effectuées sur un 66 modèles issus du benchmark BEEM, permettent l'identification de l'algorithme MYR_FB comme le meilleur au regard de la métrique d'efficacité proposée, montrant un gain d'efficacité moyen de 249% par rapport à l'algorithme de référence.

3.3 Carnac : Un moteur de vérification Swarm sur FPGA

Le chapitre précédent propose une architecture générique pour un coeur de vérification matériel. Le caractère modulaire de cette architecture a montré son intérêt en termes d'extensibilité, supportant à la fois plusieurs langages et plusieurs algorithmes de vérification.

Construisant sur la compréhension acquise par la conception de ce coeur, on propose ici de construire un coeur de vérification conservant l'architecture modulaire du coeur Menhir, mais optimisée pour la performance.

Ce coeur de vérification est ensuite intégré au sein d'une architecture distribuée, permettant de faciliter le placement sur des FPGAs de grande taille, tout en optimisant les ressources.

3.3.1 Coeur de vérification

Le coeur de vérification de Carnac s'appuie sur l'architecture modulaire de Menhir mais adopte une approche différente pour le contrôle.

Le coeur proposé dans le chapitre précédent s'appuie en effet sur un contrôle externe des composants de la pipeline principale : Le l'ensemble parcouru *Known*, la fontière *Frontier* et le Model-Frontend exposent en effet des interfaces simples, similaires à l'interface langage générique décrite dans la figure 2.7. Ces composants sont, dans cette configuration, pilotés par ces contrôleurs *globaux*, que sont le *scheduler* et le *Next Controller*. Ce type d'interface, basée sur un fonctionnement réactif, ne permet pas un contrôle précis de la pipeline : Pour chaque donnée à traiter, le composant considéré reçoit un signal *enable*, accompagné de la donnée à traiter, la traite, puis renvoie un signal de retour, accompagné de la donnée traitée. Cependant, le contrôleur ne reçoit pas l'information suffisante permettant de décider si une nouvelle donnée peut être traitée, ou non.

Pour l'implémentation du coeur Carnac, on se propose de se baser sur une autre approche, distribuant le contrôle à chacun des *blocs unitaires* de la pipeline. Cette approche permet le contrôle du rythme d'opération à chacun des blocs constituant la pipeline, permettant une plus grande flexibilité, et d'éviter qu'un contrôleur *global* ne pilote un ensemble d'étages, comme c'était le cas dans l'architecture précédente, ce qui ajoute une complexité importante à la conception de ce contrôleur.

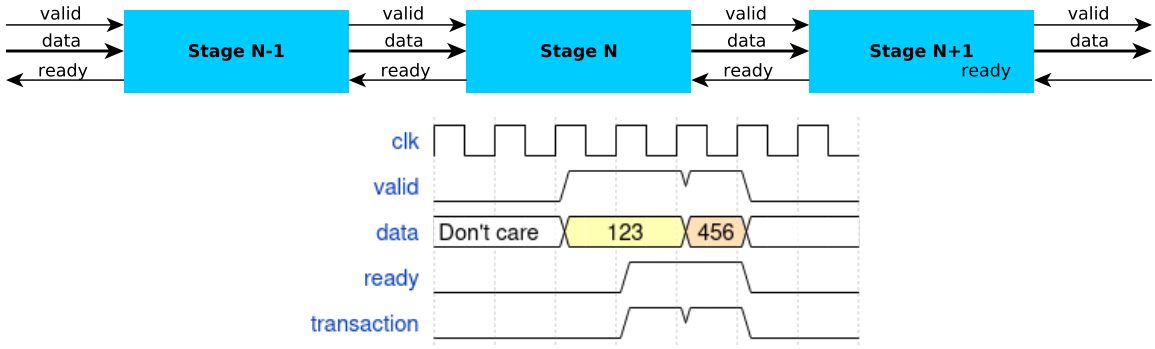
En outre, le coeur de vérification Menhir ne supporte que des configurations dont la largeur est égale à la taille du canal de communication. Cette limitation est problématique dans le cas de la vérification de modèles réalistes, dont les configurations peuvent atteindre plusieurs centaines de bits, entrainant de fortes contraintes sur la synthèse physique, et une dégradation importante de la fréquence d'horloge.

Pour répondre à ces limitations, le coeur de vérification Carnac repose sur une pipeline *ready-valid*, similaire au protocole de communication AMBA AXI Stream. Une transaction est ici basée sur une *poignée de main* (handshake en anglais) entre l'émetteur et le

receveur de la donnée : Le producteur indique par le signal *valid* que la donnée est prête à être lue, et le consommateur transmet de manière désynchronisée, via le signal *ready*, qu'il est prêt à recevoir une nouvelle donnée.

Dans ce cas, la transaction s'effectue lorsque ces deux signaux sont affirmés, et le consommateur, à l'étage suivant de la pipeline, a la possibilité de *ralentir* le flux de données s'il n'est pas prêt à en recevoir de nouvelles. La figure 3.3.1 esquisse ce fonctionnement : au cycle 3, la donnée est prête - signal *valid* affirmé - mais le consommateur n'est pas en état de pouvoir la consommer - signal ready à 0 -, la donnée est donc maintenue au cycle suivant, en attente d'être consommée.

Ce protocole unique, utilisé pour chacun des composants de la pipeline principale du moteur de vérification, est naturellement composable, une interface spécifique n'étant pas définie pour chaque bloc. De plus, le support d'états dont la largeur est supérieure à la taille du bus est naturelle, en ajoutant un signal *last* accompagnant la donnée, affirmé pour le dernier élément de chaque état.



Ce protocole réduit drastiquement le besoin de contrôle global, la pipeline étant auto-régulée : les données avancent dans la pipeline tant que l'étage suivant est prêt à les traiter, permettant également de réduire drastiquement les cycles à vide, où la pipeline est en attende de données.

L'architecture du coeur de vérification, décrite dans la figure 3.4, repose sur ce principe de contrôle local, réduisant donc le besoin de contrôleurs centraux, comme dans l'architecture Menhir présentée dans la Section 2.2.

De manière similaire à l'architecture précédemment présentée, le coeur de vérification s'articule autour d'une interface langage générique, permettant de conserver les propriétés de modularité sémantique de l'approche précédemment présentée.

Une différence importante réside cependant dans l'interprétation de cette interface,

par rapport à la version non pipelinée présentée dans la figure 2.7 du chapitre précédent : Contrairement à l'interface précédente, l'entrée et la sortie sont désynchronisées. Pour des raisons de performance, et éviter les cycles à vide, ici, un nouvel état source est transmis dès que possible, avançant dans la pipeline du *Model-Frontend* autant que possible. Cette différence n'est pas problématique pour l'algorithme implémenté ici, mais est néfaste dans le cas de l'implémentation d'un algorithme où l'information de *"qui est le précécesseur"* est nécessaire, comme dans le cas d'un algorithme de model-checking borné.

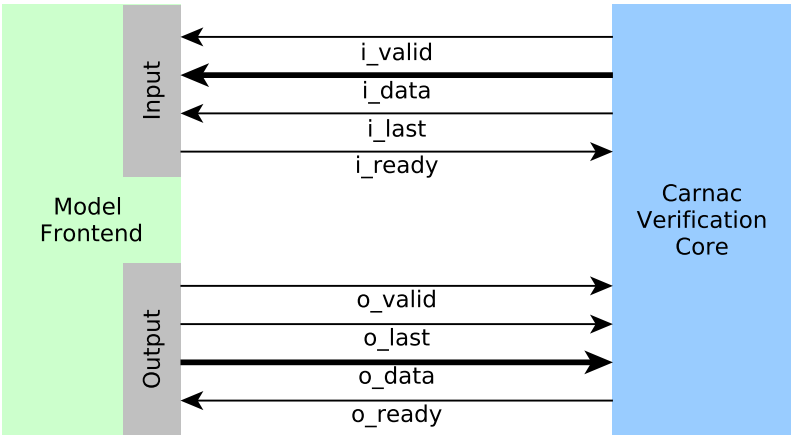


FIGURE 3.3 – Interface langage de Carnac

Une fois un état produit par le *Model-Frontend*, celui-ci est transmis au *Known Set*, puis au composant *Frontier* de manière similaire à l'architecture précédente. La différence majeure étant le paradigme choisi pour le contrôle, qui est cette fois de la responsabilité de chacun des éléments unitaires de la pipeline, au sein de chaque composant.

Ce choix d'un contrôle local induit une pipeline auto-régulée, qui corresponds de manière directe à l'architecture abstraite présentée dans la Section 2.2. Elle possède les mêmes propriétés de modularité, et ce choix d'une interface unifiée similaire au protocole AXI Stream permet une extensibilité naturelle de l'architecture. Un exemple direct de cette extensibilité est le composant *Mixer*, responsable de la randomisation de l'ordre dans lequel les états sont produits par le *Model-Frontend*. L'entrée et la sortie respectant rigoureusement la même interface, ce dernier peut être ajouté/supprimé de manière transparente, à la manière d'un pattern *wrapper* logiciel.

Le *terminaison checker*, responsable de la détection de la terminaison reçoit des informations de statut de chacun des composants, la pipeline devant être entièrement vide

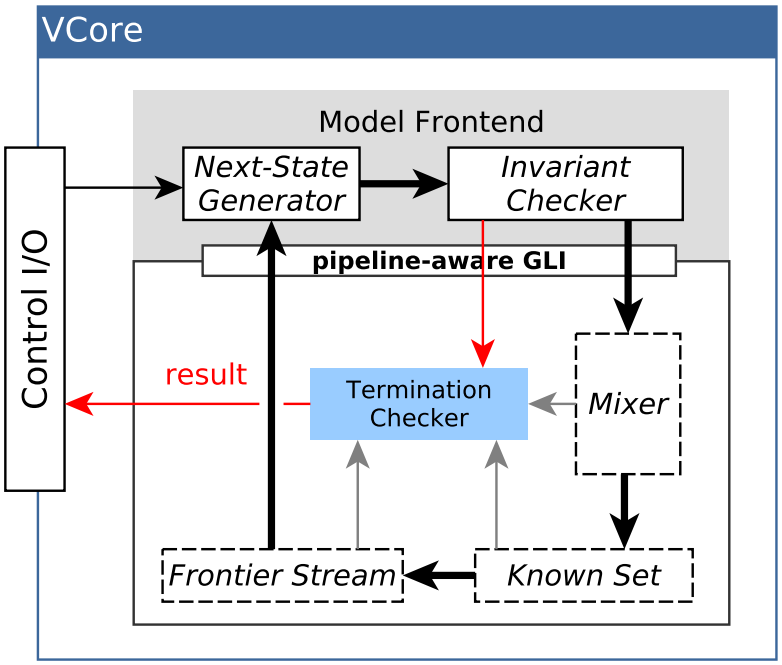


FIGURE 3.4 – Carnac : Architecture d'un coeur

avant de pouvoir affirmer la terminaison, de manière similaire à Menhir.

3.3.2 Carnac : Architecture de contrôle - niveau Swarm

Ce coeur de vérification est interfacé avec l'extérieur par un module de contrôle, à gauche sur la figure 3.4. Ce dernier est chargé d'effectuer la sérialisation et la déserialisation des ordres reçus de l'extérieur.

Ce pilotage est effectué via une architecture de contrôle distribuée, visant à réduire les contraintes liées à la localisation des coeurs sur la puce, ainsi qu'au besoin de routage.

Cette architecture de contrôle est représentée par la figure 3.5.

Le moteur de vérification est architecturé autour d'un contrôleur central, désigné par *Swarm Controler*, qui pilote l'exécution de l'ensemble des coeurs de vérification.

Ce contrôleur reçoit des informations de configuration du logiciel de contrôle implémenté sur un PC hôte, et pilote les tâches de vérification. En particulier, il est chargé d'arrêter l'ensemble des coeurs lorsque un état violant la propriété a été détecté par l'une des tâches.

Cette architecture est conçue pour être scalable sur des FPGA de taille importante,

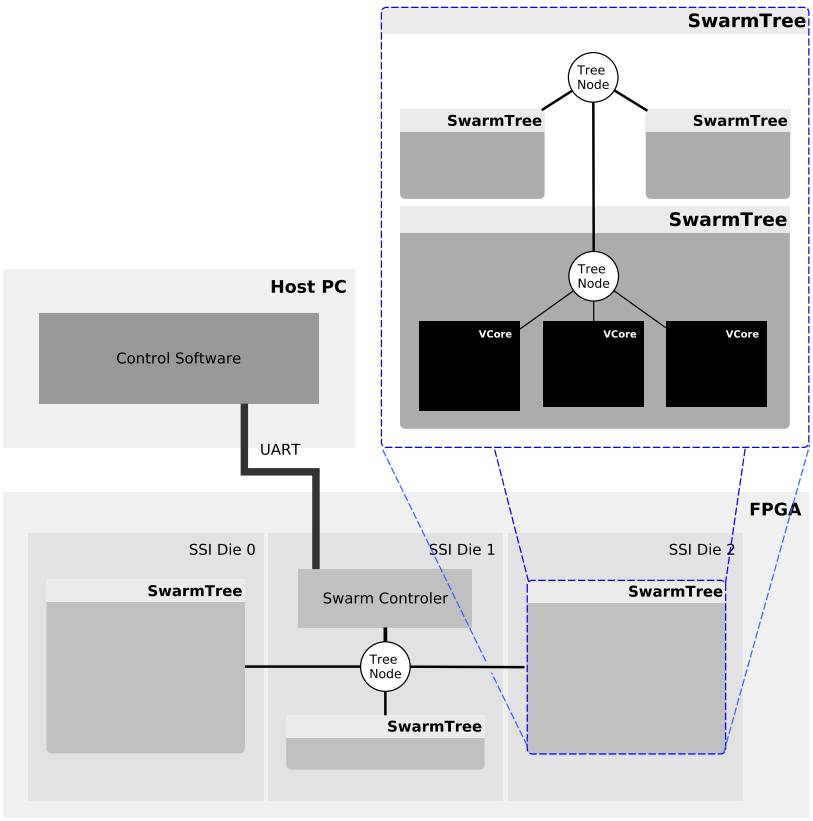


FIGURE 3.5 – Carnac : Architecture d'un coeur

et opérer à haute fréquence. Sur ce type de plateforme, plusieurs cycles d'horloge peuvent être nécessaire pour parcourir la puce (de l'ordre de la dizaine, sur le composant utilisé pour l'évaluation présentée dans la Section 3.4). Ce temps de communication introduit un problème de localité des données. D'autre part, les FPGA modernes haut de gamme reposent souvent sur l'aggrégation de plusieurs puces, empilées et connectées pour former une plus grande matrice. Ces composants permettent une taille très importante à un coût réduit par rapport à la production d'un chip unique, mais introduisent des manques d'uniformité dans la matrice, impliquant un retard plus important pour les signaux inter-die. Ces contraintes introduisent un problème de localité des données, et l'architecture de contrôle doit être prévue pour y faire face, et en particulier, pour réduire les communications inter-die.

On propose pour celà une architecture de communication sous la forme d'un arbre n-aire, esquissée dans la figure 3.5. Dans cette configuration, le contrôleur global interagit avec le seul noeud racine de l'arbre de communication, qui lui-même communique

avec n sous-arbres, dont les coeurs de vérification constituent les feuilles. Le nombre de sous-arbres n corresponds au nombre de sous-matrices du FPGA utilisé, pour réduire l'utilisation des ressources. La communication entre chaque noeud et ses successeurs est registrée dans les deux sens - montant et descendant - et effectuée sur un canal de largeur faible (8 bits), permettant ainsi de réduire drastiquement les contraintes de placement lors de la synthèse physique, à la fois inter-die et intra-die. En outre, cette architecture réduit grandement la congestion autour du contrôleur central par rapport à une communication directe entre ce controleur et les coeurs de vérification.

La scalabilité de cette architecture distribuée, visant à permettre la synthèse à haute fréquence sur des FPGA multi-die, est illustrée dans la section suivante par l'évaluation matérielle de l'algorithme sélectionné dans la Section 3.2.

3.4 Evaluation de l'implémentation matérielle

L'analyse théorique présentée dans la Section 3.1.1 a permis, au travers d'une exploration de l'espace de conception algorithmique, d'identifier un algorithme permettant d'accroitre l'efficacité de la vérification swarm. Cette analyse est basée sur l'intuition que la performance d'une tâche de vérification swarm pour un modèle quelconque repose sur deux facteurs : La performance brute du coeur de vérification - mesurée en nombre d'états traités par cycles - ainsi que le recouvrement entre les tâches de vérification. Ce chapitre addresse ces deux points en proposant d'une part un algorithme réduisant le recouvrement, et une architecture de vérification optimisée pour la performance.

Cette Section décrit les résultats expérimentaux obtenus par l'accélérateur Carnac.

3.4.1 Configuration expérimentale

On mesure la performance de l'accélérateur Carnac sous la forme du temps pour explorer un espace d'état défini. Cette évaluation repose sur la méthodologie décrite par les auteurs de l'article FPGASwarm [6], initialement proposée par Holzmann [7].

Cette analyse repose sur l'utilisation d'un modèle synthétique, qui peut s'interpréter comme un test de couverture visant à explorer un espace d'état de taille 2^{32}. Ce modèle est constitué d'un ensemble de 8 processus, chacun responsable d'un groupe de 4 bits au sein d'un entier 32 bits. A chaque pas de temps, un thread permutte un bit parmi son groupe. Parmi l'espace d'états, 100 états particuliers, choisis aléatoirement, sont désignés

comme des états fautifs. La vérification consiste à trouver l'ensemble de ces 100 états. Ce modèle permet une estimation de la couverture de l'espace d'état obtenue, les états fautifs étant distribués aléatoirement. Ce modèle synthétique est implémenté manuellement au sein de l'architecture Carnac, présentée dans la Section 3.3.

Ce modèle synthétique permet une comparaison directe des performances de FPGASwarm avec celles de l'accélérateur Carnac.

La scalabilité de l'architecture proposée est illustrée par l'implémentation du moteur de vérificaiton sur un FPGA moderne de grande taille : Xilinx Ultrascale+ XCVU37P, à une fréquence élevée de 400MHz. En complément d'une fréquence d'horloge 60% plus élevée, le moteur de vérification Carnac montre une scalabilité plus importante, permettant la synthèse de 80 coeurs de vérification, soit le double de son concurrent FPGASwarm. La synthèse physique a été réalisée à l'aide de Vivado 2020.1, ciblant une carte Xilinx VCU128.

3.4.2 Résultats

Le tableau 3.1 présente les résultats des performances de l'accélérateur proposé, comparées à l'accélérateur FPGASwarm proposé par [6], dans deux configurations, Carnac40 et Carnac80, intégrant respectivement 40 et 80 coeurs.

À des fins de comparaison, la configuration Carnac40 utilise un nombre de coeurs de vérification et une configuration mémoire identiques (*Frontier stream* de 1024 deep, *Known set* avec 2^{19} slots), sur un problème de vérification identique à FPGASwarm [6].

De plus, pour s'assurer de la validité des valeurs produites, chacune des mesures a été répétée 200 fois en changeant à chaque run les états *fautifs* recherchés dans l'espace d'état, choisis aléatoirement. Les résultats présentés dans la table 3.1 montrent une bonne répétabilité avec un coefficient de variation de seulement 22%. De plus, le coefficient de variation est similaire pour les configurations à 40 et 80 coeurs, ce qui est conforme à l'attendu.

La configuration Carnac40, correspondant exactement à la configuration utilisée par l'accélérateur FPGASwarm, termine la vérification en *3.02* secondes en moyenne, ce qui corresponds à un facteur d'accélération de *3.96x* par rapport à l'état de l'art. Ce facteur d'accélération s'explique à la fois par l'algorithme MYR_FB utilisé par le coeur, ainsi qu'une vitesse d'horloge plus élevée, et une architecture fortement optimisée, permettant d'explorer un état par cycle d'horloge.

En plus d'un facteur d'accélération important pour un même nombre de coeurs de

	FPGA Swarm	**Carnac40**	**Carnac80**
Temps moyen (sec)	12	3.02	1.58
Ecart type	-	0,683	0,349
Coefficient de variation	-	22.56%	22.08%
Fréquence de découverte	1/27.9	1/11.25	1/11.7

TABLE 3.1 – Résultats des performances de Carnac

vérification, l'architecture Carnac est plus scalable, permettant la synthèse du moteur de vérification dans une configuration 80 coeurs : Carnac80. Cette configuration résouds le problème considéré en *1.58* secondes, ce qui constitue une accélération conséquente d'un facteur 7.58x. Conformément à l'attendu pour ce problème *embarassingly parallel*, doubler le nombre de coeurs induit une accélération quasi-linéaire, de 1.91x.

Pour compléter cette étude, le tableau 3.4.2 présente l'utilisation en ressources du moteur de vérification Carnac dans les configurations 40 et 80 coeurs, ainsi que l'utilisation ressources correspondante pour l'implémentation FPGASwarm [6].

On constate que le doublement du nombre de coeurs double l'utilisation des ressources mémoires dédiées (BRAM + URAM). Cependant, en raison de contrôleur global partagé, ainsi que de la structure arborescente de l'arbre de communication, le nombre de LUTs n'augmente que de 84%.

	Cores	**LUT**	**FFs**	**Mémoire (kbit)**
Carnac80	80	360 147	780 707	53 280
Carnac40	40	177 974	381 211	26 640
FPGASwarm	40	112 272	271 723	36 972

TABLE 3.2 – Utilisation des ressources du moteur de vérification Carnac pour 40 et 80 coeurs

En outre, le surcoût en ressources de l'architecture Carnac40 est relativement faible par rapport à son concurrent (1,58 fois plus de LUT, 1,40 fois plus de FF), mais l'utilisation des ressources mémoires est optimisée, n'utilisant que 72.05% des ressources utilisées par FPGASwarm. Ce surcoût en ressources logiques est dû à l'optimisation fine de la pipeline, et l'implémentation d'un contrôle local de la pipeline, mais il est considéré comme non problématique ici, dans la mesure où le facteur limitant pour la vérification n'est pas constitué par les ressources logiques, mais la mémoire disponible sur la puce.

Ces résultats sont conformes aux attentes, et confirment les résultats théoriques en présentant une accélération importante par rapport à l'état de l'art. En outre, le calcul

de la fréquence de découverte d'un nouvel état, à l'échelle globale de la tâche de vérification swarm, montrent une amélioration significative de l'efficacité de l'algorithme et de l'implémentation, découvrant un nouvel état tous les 11.2 cycles d'horloges dans la configuration Carnac40, soit une amélioration de 59%. Il est noter que cet valeur est une moyenne à l'échelle de la vérification swarm complète, et ne doit pas être confondue avec le débit de traitement à l'échelle d'un coeur et d'une tâche de vérification, qui est d'un état par cycle.

3.5 Conclusion

L'amélioration des techniques de vérification swarm est un enjeu important pour accroitre la scalabilité du Model-Checking. Cette technique transpose en effet l'explosion combinatoire en une explosion en temps de calcul, sous la forme d'un ensemble de tâches de vérification indépendantes, visant à obtenir une couverture élevée de l'espace d'état vérifié avec une empreinte mémoire faible.

L'accélération matérielle de la vérification swarm est une opportunité majeure pour permettre la vérification de gros modèles dans des temps raisonnables, présentant des facteurs d'accélération de plusieurs ordres de grandeurs par rapport aux outils logiciels et GPU.

Un point d'attention est cependant l'efficacité de l'algorithme utilisé : les tâches étant non coordonnées, il est critique d'utiliser un algorithme qui minimise le recouvrement entre ces dernières.

Construisant sur la spécification générique d'algorithmes proposée dans la Section 2.1, ce chapitre propose l'interprétation de cette spécification sous la forme de points de variabilité. Ces points de variabilités sont exploités au travers d'une exploration de l'espace de conception algorithmique, visant à identifier un algorithme minimisant le recouvrement entre les tâches.

Cette étude théorique effectuée sur 8 algorithmes et 68 modèles du benchmark BEEM, parmi lesquels 7 sont nouveaux, a permis l'identification de l'algorithme MYR_FB, qui montre un gain d'efficacité moyenne de 249% par rapport à l'algorithme de référence considéré.

Pour confirmer ces bons résultats théoriques, cet algorithme a été implémenté au sein d'un moteur de vérification matériel, conçu pour maximiser la performance, tout en conservant les propriétés de modularité de l'approche Menhir présentée dans le chapitre

précédent. En outre, ce moteur de vérification a été conçu avec un soin particulier pour la scalabilité sur des plateformes reconfigurables multi-die, pour lesquelles la taille de la matrice, ainsi que son hétérogénéité met à l'épreuve les outils de synthèse physique.

L'évaluation expérimentale expose des performances intéressantes, montrant un facteur d'accélération de *7.58x* par rapport à l'état de l'art, traduction numérique des améliorations architecturales tant en termes de scalabilité, permettant l'intégration de deux fois plus de coeurs, qu'en termes de microarchitecture, permettant une fréquence d'horloge *60%* plus élevée.

Chapitre 4

Dolmen : Un coeur swarm pour la vérification de sûreté et de vivacité

Le chapitre précédent présente Carnac, une architecture de vérification matérielle distribuée, permettant l'accélération de la vérification swarm. Cette architecture montre une accélération significative par rapport à l'état de l'art, tout en conservant une approche modulaire, élément différenciant de l'architecture Menhir - présentée dans le chapitre 2 - permettant le support de multiples algorithmes.

Cependant, cette approche, ainsi que l'ensemble des implémentations proposées dans l'état de l'art à ce jour, se focalisent sur la vérification de propriétés de sûreté, une classe de propriétés permettant l'encodage de comportement non désirés dans le système étudié.

Pour construire la spécification formelle d'un système, la Section 1.2.2 a montré le besoin d'une seconde classe de propriétés, dites de vivacité, formalisant les comportements désirés du système. Cette seconde classe, utilisée de manière complémentaire aux propriétés de sûreté, nécessite le raisonnement dans un formalisme différent, apte à reconnaitre des traces infinies de l'exécution du système. Les automates de Büchi, sous-classe d'automates ω-réguliers sont communément utilisés pour formaliser d'une manière uniforme ces deux types de propriétés.

Le problème de Model-Checking, formulé dans la Section 1.2.1, se présente alors de la même manière que pour la vérification de propriétés de sûreté. Une différence majeure réside dans l'algorithme utilisé. Une trace constituant un mot accepté par un automate de Büchi se présentant intuitivement comme un *lasso*, composé d'un préfixe, et d'un cycle contenant un état d'acceptation. Du point de vue algorithmique, il s'agit donc de détecter de tels mots.

Ce chapitre montre comment construire un accélérateur matériel implémentant un algorithme de détection de cycles pour la vérification swarm de propriétés de vivacité

encodées sous la forme d'automates de Büchi.

L'approche présentée repose sur la réorientation du coeur de vérification Carnac pour permettre l'implémentation d'un algorithme de détection de cycles, dans une forme adaptée à la vérification swarm. Cette approche illustre par ailleurs la flexibilité de l'architecture Carnac présentée dans le chapitre précédent.

Le moteur de vérification Dolmen est confronté à l'outil logiciel Divine sur un ensemble de modèles extraits du benchmark BEEM, et montre une accélération importante de 4 ordres de grandeur par rapport à ce Model-Checker logiciel largement utilisé.

4.1 Algorithme

Cette section discute de l'algorithmie utilisée pour la détection de cycles dans le contexte présent de vérification swarm accélérée en matériel. Les adaptations apportées à l'algorithme de référence utilisé sont discutées, ainsi que leurs conséquences potentielles sur les performances de la vérification.

4.1.1 Algorithme de Courcoubetis

Du point de vue de l'algorithme, la vérification de propriétés de vivacité par Model-Checking se réduit à un problème de détection de cycles dans l'espace d'état : un mot accepté par un automate de Buchï est, par définition, un mot contenant un cycle, lequel contient un état d'acceptation. Aussi, une condition nécessaire et suffisante pour qu'un automate de Büchi soit non vide est qu'il contienne au moins un état d'acceptation atteignable depuis l'état initial, et atteignable depuis lui-même.

Parmi les algorithmes de détection de cycles proposés dans la littérature, on se base sur l'algorithme de Courcoubetis, présenté en 92[15], entrelaçant un parcours postfixe (post-order traversal), et un parcours préfixe (pre-order traversal) pour réduire le nombre d'états parcourus.

Le Listing 4.1.1 propose une implémentation récursive de cet algorithme référence. Cet algorithme détecte les cycles d'acceptation dans les automates de Buchï en entrelaçant une phase de détection de préfixe - dfs_1 -, avec une phase de confirmation du cycle - dfs_2.

Un premier parcours - dfs_1 - est initié à partir de l'état initial. L'ensemble d'états parcourus - k_1 - est initialisé à l'ensemble vide, puisque aucun état n'a été parcouru à ce stade. L'état traité est ajouté dans l'ensemble k_1, puis ses successeurs qui n'ont pas

encore été visités sont visités récursivement. Si un état d'acceptation est atteint, un second parcours est initié par l'appel à $dfs_2(s, s, k_2)$.

Quelques précisions sont à apporter sur cet appel. La fonction dfs_2 est déclarée avec trois paramètres : L'état courant traité s, un état cible *target*, ainsi que l'ensemble d'états parcourus par ce deuxième parcours k_2. Cette analyse d'atteignabilité imbriquée vise à détecter des cycles, ie des chemins d'un état vers lui-même. Aussi, pour chacun de ces états, ce parcours démarre - ligne 10 de dfs_1 - de l'état d'acceptation s, avec pour cible ce même état d'acceptation. De manière similaire au premier dfs, chacun des états successeurs qui n'a pas encore été visité - $\notin k_2$ - fais l'objet d'un appel récursif.

Si l'un des successeurs de l'état courant corresponds à la cible : un cycle a été détecté et les deux analyses imbriquées terminent. Précisons ici que le statement *report violation* est à interpréter comme un point de terminaison de l'algorithme. La logique complète liée à la terminaison lorsqu'un cycle d'acceptation est detecté n'a pas été décrite dans ce pseudo-code pour conserver une formulation simple. D'autre part, les états ne sont pas ajoutés directement dans les ensembles d'états parcourus k_i. Pour ce faire, on définit une fonction h, bijection de l'espace d'états vers un ensemble d'indices $1..m$ et la fonction d'appartenance se comprends comme *L'indice de l'état est contenu dans l'ensemble* k_i. Ce choix de conception proposé par Courcoubetis a l'avantage d'une meilleure compacité mémoire car les états peuvent ne pas être stockés *en clair* dans l'ensemble.

```
1  courcoubetisAlgo()
2      dfs1(initial, {}, {})
```

```
1  dfs1(s, k1, k2)
2      k1 = k1 ∪ { h(s) }
3
4      for t ∈ next(s) do
5          if h(t) ∉ k1 then
6              dfs1(t, k1, k2)
7          end if
8      end for
9      if s ∈ accepting then
10         dfs2(s, s, k2)
11     end if
```

```
1  dfs2(s, target, k2)
2      k2 = k2 ∪ { h(s) }
3
4      if target ∈ next(s) then
5          report violation
6      end if
7      for t ∈ next(s) do
8          if ht(t) ∉ k2 then
9              dfs2 (t, k2)
10         end if
11     end for
```

FIGURE 4.1 – Implémentation récursive de l'algorithme de Courcoubetis (version B)[15]

Après la description du principe, plusieurs observations sont à effectuer sur cet algorithme. Tout d'abord, notons que ces deux parcours sont notablement différents. Le premier *dfs* effectue la vérification d'acceptation des états en ordre postfixe (post-order en anglais). Avec cet ordre d'évaluation, chacun des états n'est évalué qu'après que l'en-

semble de ses successeurs a été évalué. Aussi, si un état d'acceptation A_1 est détecté par le dfs_1, il est certain qu'un état d'acceptation A_2 appartenant aux successeurs de A_1 n'est pas contenu dans un cycle, car la présence de cycles pour A_2 a déjà été vérifiée. Une conséquence directe est qu'il n'existe aucune transition depuis un des successeurs de A_2 vers un des prédecesseurs de A_2. En particulier, A_1 étant un prédecesseurs de A_2, il n'existe aucune transition depuis un des successeurs de A_2 vers un prédécesseurs de A_1. Aussi ce choix d'ordre de parcours implique qu'il n'est pas nécessaire, lors de la détection de cycle de A_1, de vérifier les successeurs de A_2.

Cette propriété intéressante se traduit dans l'implémentation de l'algorithme par une forme de mémoisation sur l'ensemble k_2 : Ce dernier, contenant les états déjà parcourus par le dfs_2, donc exemptés d'appels récursifs, est partagé entre les différentes phases de détection de cycles. Ce partage de l'ensemble k_2 explique la conservation de la référence vers l'ensemble k_2 dans toute la pile d'appel du dfs principal dfs_1, et qui diminue le nombre d'états visités par ce second parcours dfs_2.

4.1.2 Adaptation pour la vérification swarm

La vérification swarm repose sur l'exécution diversifiée d'un grand nombre de tâches de vérification sur un même modèle pour améliorer la couverture de l'espace d'état. Cette diversification de l'exécution des différentes tâches de vérification est obtenue en utilisant un algorithme probabiliste pour lequel chaque tâche de vérification exploite un générateur de nombres aléatoires différent.

Cette section présente trois adaptations apportées à l'algorithme présenté précédemment pour l'adapter à la vérification swarm, ainsi que répondre aux contraintes liées à l'implémentation sur des architectures reconfigurables.

Stack bornée

L'algorithme présenté précédemment imbrique deux parcours DFS, ce type de parcours repose sur une pile pour stocker l'ensemble d'états à parcourir. Cette dernière peut être implicite, comme c'est le cas dans l'implémentation récursive présentée dans la section précédente, ou explicite, comme ce serait le cas dans une implémentation itérative de l'algorithme en question [15].

Dans le cas d'une implémentation matérielle, la récursivité n'a pas de sens. Une construction récursive ne peux intervenir que statiquement, à la compilation, pour agencer

des composants d'une manière arborescente, par exemple.

Les deux explorations de graphe imbriquées reposent, dans leurs implémentations matérielles, sur l'utilisation de piles explicites, contenant les états en instance d'être visités. Ces piles sont généralement, en logiciel, de profondeur infinie (virtuellement infinies, en pratique, la taille augmente dynamiquement). Cependant, avant d'avoir exploré le modèle, ce nombre d'états est inconnu : le graphe étant défini de manière intensionnelle, par le biais d'une relation de transition, la seule manière de compter le nombre d'états est de le parcourir ; parcours dont la complexité est équivalente à celle d'une tâche de Model-Checking. Par conséquent, aucune hypothèse ne peut être faite sur la taille minimale de ces deux ensembles ordonnés.

Ce problème est généralement contourné en logiciel par le biais de l'allocation dynamique : l'espace mémoire alloué croit au cours de l'exploration. Une telle construction n'est pas possible en matériel, l'allocation mémoire étant fixée statiquement à la conception du moteur de vérification, ou, a minima, à la synthèse logique de ce dernier.

Une autre option aurait pu être d'assigner une profondeur fixe, et d'interrompre l'exploration - en relevant une erreur - si la pile atteint sa taille maximale. Cependant, étant donné l'espace mémoire limité à disposition sur les plateformes reconfigurables actuelles, cette option n'est pas souhaitable.

En se basant sur les bons résultats, proposés dans le chapitre précédent, des algorithmes de vérification swarm MY_FB, MYR_FB et DFS_FB, tous basés sur une structure de données ne conservant que les n états les plus récemment ajoutés, on propose ici d'utiliser ce type de structure de donnée pour la représentation des deux piles d'états utilisées par le parcours principal et imbriqué.

Pour rappel, cette structure de donnée se comporte comme un buffer LIFO classique tant que ce dernier n'est pas plein. Si aucun slot n'est disponible, l'état entrant est stocké en écrasant l'état le plus vieux du buffer. L'utilisation d'une telle structure de donnée pour stocker les états en instance de parcours induit un algorithme proche d'un DFS, qui priorise la profondeur de l'exploration.

Relaxation de la bijection

L'algorithme de référence définit l'appartenance d'un état à l'ensemble des états parcourus au travers d'une bijection h de l'espace d'états vers l'ensemble des entiers. Un état s est alors considéré comme déjà visité si et seulement si $h(s)$ est présent dans l'ensemble k_i. Une implémentation naturelle découlant de ce choix de conception est alors d'utiliser

une table de booléens : Un élément s est présent dans l'ensemble si et seulement si le booléen d'indice $h(s)$ est présent dans cette table.

Pour l'implémentation matérielle, on propose de relâcher la contrainte de bijectivité imposée à la fonction h, en se limitant à une fonction surjective : une fonction de hachage. Sans cette contrainte, la représentation de l'ensemble k_i devient probabiliste : deux états s_1 et s_2 peuvent avoir la même image par la fonction h. Aussi, en conservant la même routine d'appartenance à l'espace d'états, considérons le cas où s_1 et s_2 ont la même image par la fonction h. On insère l'état s_1 dans l'ensemble, le booléen à l'indice $h(s_1)$ prends alors la valeur *True*. Lors d'une requête d'appartenance de l'état s_2 à l'ensemble k_i, la routine d'appartenance lis alors la valeur du boléen stocké à l'adresse $h(s_2)$, qui est *True* car $h(s_1) == h(s_2)$. La requête d'appartenance à l'ensemble renvoie alors des faux-positifs pour tout état s_i pour lequel un état possédant le même hash a déjà été inséré dans l'ensemble. Cette implémentation correspond précisément à une structure de données connue et utilisée dans le chapitre précédent pour l'implémentation du *Known* dans un contexte swarm : le filtre de Bloom (Bloom filter)[7][20].

On note que lors de l'utilisation d'un filtre de Bloom, la probabilité de faux-positifs - ou collisions - croit avec le nombre d'états insérés dans l'ensemble représenté. Lors d'un parcours de graphe, ce Bloom filter est utilisé pour déterminer si un élément a déjà été visité. Aussi, un faux positif implique qu'un état n'ayant pas encore été visité sera considéré comme visité, et donc ne sera pas parcouru par l'algorithme. Par conséquent, pour chaque faux positif, le sous-graphe de ses successeurs sera ignoré par l'algorithme, car supposé déjà visité.

De manière analogue à ce qui a été présenté dans le chapitre précédent, la fonction de hachage utilisée est paramétrique, configurée par une *seed*. Chaque tâche de vérification lancée au cours d'une vérification swarm débute paramétrée par la *seed* unique utilisée pour la fonction de hash paramétrique du filtre de Bloom. Ainsi, si l'on considère deux tâches de vérification, les deux fonctions de hash utilisées seront différentes ; celà implique que les faux positifs induits par le remplissage du filtre de Bloom seront différents au cours de l'exploration. Ainsi, comme chaque faux-positif implique état "ignoré" par l'exploration, des faux positifs différents induisent que chaque tâche de vérification explore une partie différente de l'espace d'état, augmentant la portion de l'espace d'état couverte, et ainsi, la probabilité de trouver une faute.

Saturation et oubli

L'algorithme de référence présenté dans la section précédente repose sur un parcours postfixe pour la détection des états d'acceptation, entrelacé avec un parcours préfixe pour confirmer, ou infirmer la présence d'un cycle pour l'état d'acceptation courant. Ce parcours permet de réduire le nombre d'états visités car il assure que lorsqu'un état d'acceptation a été visité, les états d'acceptation parmi ses descendants ne sont pas contenus dans des cycles, il n'est donc pas nécessaire d'évaluer leurs successeurs.

Ce comportement est obtenu par mémoïsation : l'ensemble d'états visités utilisés par le parcours préfixe imbriqué est conservé d'état d'acceptation en état d'acceptation. De cette manière, un état qui a déjà été visité lors du traitement d'un état d'acceptation descendant ne sera pas visité à nouveau lors du traitement de l'état d'acceptation courant. Ce type de parcours est intéressant par la réduction du nombre d'états parcourus mais implique une saturation rapide dans le cas de l'utilisation d'un Bloom filter pour stocker l'espace d'états parcourus : Plus l'exploration avance, plus la probabilité de faux positifs du Bloom filter augmente. Cette caractéristique induit alors, pour les états d'acceptations traités plus tard dans l'exploration, une part importante des états successeurs éliminés de l'exploration. Aussi, plus l'exploration avance plus la profondeur d'exploration du parcours imbriqué sera faible, ce qui conduit à une chance réduite de trouver des contre-exemples.

Pour contrecarrer ce comportement, on propose de réinitialiser le Bloom-filter utilisé pour stocker les états parcourus du parcours imbriqué pour chaque nouvel état d'acceptation. Dans ce cas, à chaque nouvel état d'acceptation trouvé, le Bloom filter est vide au début de la phase de confirmation de cycle (parcours imbriqué), et le début de l'exploration, correspondant intuitivement à des cycles courts, s'effectuera avec peu de faux positifs, donc peu de chances d'ignorer un cycle suite à un faux positif.

Ce choix de design a plusieurs conséquences sur l'algorithme utilisé. En premier lieu, l'utilisation d'un parcours postfixe pour le parcours principal est dans ce cas, inutile. En effet, ce type de parcours n'est nécessaire que dans le cas de la conservation de l'ensemble d'états parcourus du parcours imbriqué (noté k_2 dans la section précédente). En renonçant à cette optimisation, on élimine la nécessité d'implémenter un parcours postfixe pour la boucle principale.

D'autre part, un autre intérêt d'imbriquer un parcours postfixe et infixe est la construction du contre-exemple : lorsqu'un cycle est détecté, le chemin depuis l'état initial, jusqu'à ce dernier, ainsi que les états constituant le cycle d'acceptation sont contenus respectivement dans les piles d'états en instance de visite des deux parcours principaux et imbriqués.

Ceci étant, dans le cas présent, pour répondre à la nécessité d'allocation statique de la mémoire ainsi qu'au faibles ressources mémoires à disposition, ces deux piles s_1 et s_2 sont bornées, de manière analogue à ce qui a été présenté dans le chapitre précédent : elles sont chacune représentées par un buffer de n états, contenant les n états les plus récemment ajoutés. Cette contrainte de design empêche la construction du contre-exemple, car seuls certains états du contre-exemple sont contenus dans les deux piles, les autres ont étés écrasés au cours de l'exploration.

Pour ces raison, nous faisons le choix de ne pas implémenter un parcours postfixe pour la détection des états d'acceptation, mais un parcours prefixe, qui permet de détecter les états d'acceptation au plus tôt au cours de l'exploration : l'évaluation du prédicat d'acceptation se fait à immédiatement après la découverte de l'état, et non après avoir traité ses successeurs. Les implications de ce choix en termes de temps d'exécutions sont discutées dans la section suivante.

4.1.3 Complexité algorithmique

La complexité algorithmique est une métrique communément utilisée pour formaliser la quantité de ressources nécessaire à l'exécution d'un algorithme.

Dans le cas de l'algorithme utilisé comme base dans cette étude, proposé par Courcoubetis en 92[15], une analyse naïve conviendrait d'une complexité de $\mathcal{O}(N^2)$, du fait des deux parcours récursifs imbriqués. Un oeil averti remarquera cependant que l'optimisation implémentée, entrelaçant un parcours infixe et un parcours postfixe, et mémoïsant l'ensemble d'états parcourus k_2 du parcours imbriqué rends cet algorithme linéaire en la taille de l'espace d'états : chaque état étant parcourus au plus deux fois.

De manière analogue, l'algorithme proposé, incluant les adaptations pour la vérification swarm, est constitué de deux parcours infixes imbriqués, renonçant à l'optimisation précédemment décrite. La complexité de ce parcours est donc, a priori, $\mathcal{O}(N^2)$, ou plus précisément, $\mathcal{O}(\mathcal{N} + \mathcal{A} * \mathcal{N}) = \mathcal{O}(\mathcal{A} * \mathcal{N})$ où N et A sont respectivement la taille de l'espace d'état, et le nombre d'états d'acceptations.

Cette analyse néglige l'influence du filtre de Bloom dans le temps de vérification. Le cadre d'utilisation sous-jacent aux algorithmes swarm est en effet que le volume mémoire disponible est très faible en comparaison avec la taille de l'espace d'état exploré. De plus, un filtre de Bloom ne peut représenter qu'un nombre d'états fini, majoré par la taille de son espace mémoire dédié.

Pour préciser cette analyse, on note n_1 et n_2 les tailles respectives des filtres de Bloom

utilisés par le parcours principal, détectant les états d'acceptations, et le parcours imbriqué, correspondant aux cycles d'acceptation.

L'hypothèse précisée ci-avant se traduit par $n_1 << N$ et $n_2 << N$. Le parcours principal peut ne parcourir qu'au plus n_1 états, qui, dans le pire des cas, sont tous états d'acceptation. Pour chaque état d'acceptation, le second parcours, débutant d'un état d'acceptation vers lui-même, ne peut parcourir qu'au plus n_2 états. Ainsi, au plus, l'algorithme parcourra $n_1 + A * n_2 < n_1 + n_1 * n_2 << N^2$ états.

L'analyse de complexité algorithmique apparaît donc comme un outil inadapté pour analyser cet algorithme, et l'estimation, fortement pessimiste, du nombre d'états parcourus montre que dans le cas de cet algorithme, le temps de parcours est majoritairement influencé par le nombre d'états d'acceptation, et la taille du filtre de Bloom du parcours secondaire, détectant les cycles.

L'intuition donnée par cette analyse est confirmée par l'évaluation matérielle du coeur de vérification, proposée dans la section 4.3. Cette évaluation considère deux configurations : la première pour $n_1 = n_2$, puis une seconde réduisant fortement la taille n_2 du filtre de Bloom secondaire, montrant alors un accroissement significatif des performances du moteur de vérification.

4.1.4 Synthèse

Les travaux présentés dans la section 1.4.2, ainsi que le chapitre précédent montrent l'intérêt majeur des plateformes FPGA pour l'accélération matérielle de la vérification swarm. Ces approches sont cependant limitées à la vérification de sûreté.

Pour dépasser cette limitation, cette section propose de construire un algorithme de vérification de propriétés de vivacité, exprimées sous la forme d'automates de Büchi, adapté à l'implémentation matérielle, mais également à une approche swarm diversifiée.

Pour effectuer la détection de cycles nécessaire à la vérification de telles propriétés, tout en fournissant une diversification des explorations dans un contexte swarm, cet algorithme construit sur les enseignements tirés du chapitre précédent, en utilisant à la fois un filtre de Bloom, mais également une pile ne conservant que les états récemment ajoutés.

4.2 Architecture

L'estimation du temps d'exécution d'une tâche de vérification décrite dans la section 4.1 donne une bonne intuition sur l'influence des méta-paramètres sur la performance globale de l'algorithme de vérification. Le chapitre 3 a cependant donné un aperçu de la complexité de la conception d'un algorithme de vérification swarm efficace.

Dans le cas de la vérification swarm d'automates de Büchi, l'algorithme est plus complexe, entrelaçant deux parcours du graphe pour permettre la détection des cycles d'acceptation. Pour gérer cette complexité, et aboutir, à terme, à la conception d'algorithmes de vérification swarm adaptés aux propriétés de vivacité, efficaces, et répondant aux contraintes du matériel, une approche est d'utiliser une méthodologie de conception incrémentale, construisant un coeur de vérification en parallèle de l'amélioration des algorithmes.

Pour permettre ce type de démarche de co-conception incrémentale des algorithmes et de l'architecture, il est souhaitable débuter à partir d'une architecture modulaire et évolutive, apte à recevoir de futures optimisations.

Les deux chapitres précédents de ce manuscrit ont proposé une approche descendante, depuis la spécification des algorithmes vers l'implémentation de l'architecture, pour concevoir un coeur de vérification pour la sûreté à la fois performant, et montrant de bonnes propriétés en termes de modularité, et division de la complexité, séparant par exemple l'algorithme de la sémantique vérifiée.

Cette section propose de réorienter le coeur de vérification de Carnac pour construire un coeur de vérification d'automates de Büchi. Outre l'illustration de l'évolutivité et de scalabilité de l'architecture précédemment proposée que cette conception constitue, ce coeur de vérification pour la vivacité bénéficie ainsi des bonnes caractéristiques de Carnac en termes de performance, et de modularité.

4.2.1 Architecture du coeur de vérification

L'algorithme présenté dans la section précédente détecte des cycles d'acceptation en entrelaçant une phase de détection de préfixe, avec une phase d'identification de cycle. Cet algorithme, dont le comportement de haut niveau est esquissé dans la figure 4.2, débute de l'état initial et effectue une analyse d'atteignabilité sur la composition du modèle et de sa propriété, tous deux représentés par des automates de Büchi Non Déterministes - NBA - à la recherche d'états d'acceptation. Le chemin depuis l'état initial vers un état

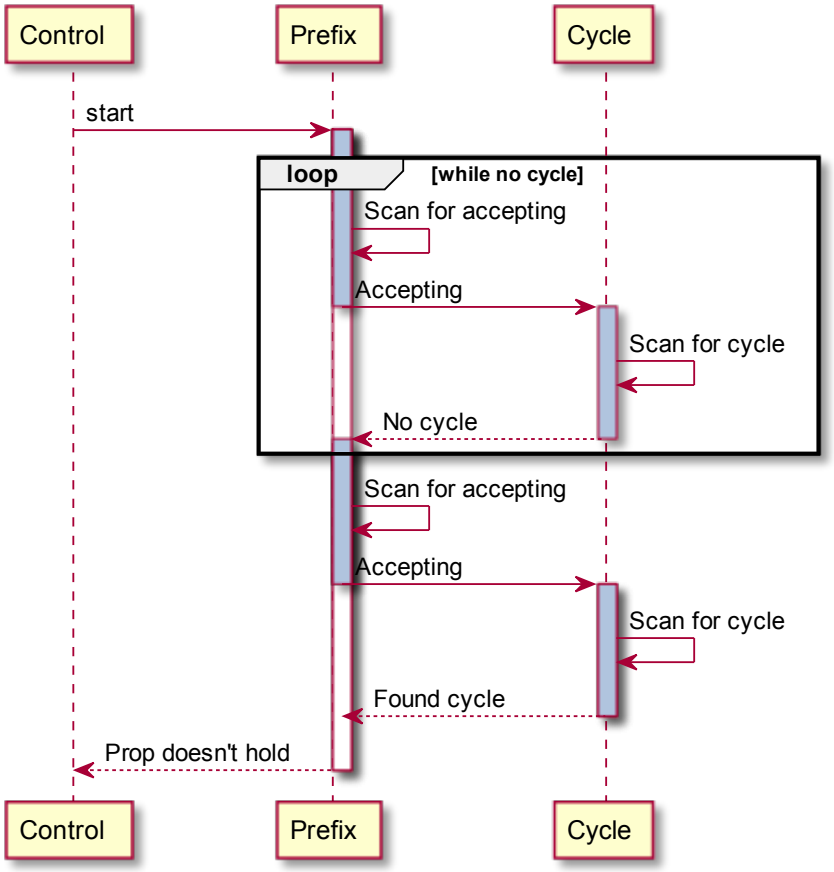


FIGURE 4.2 – Vue d'ensemble de haut niveau de l'algorithme utilisé

d'acceptation constitue le préfixe d'un potentiel contre-exemple.

Lorsqu'un état d'acceptation est atteint, cette tâche est mise en pause, et une tâche secondaire est lancée depuis l'état d'acceptation en question, à la recherche d'un cycle, qui corresponds à un chemin depuis l'état d'acceptation vers lui-même. Deux cas peuvent alors se produire : Si l'état d'acceptation n'est pas trouvé au cours de la tâche de détection de cycle, la tâche préfixe reprends, et l'opération précédente est répétée pour chaque état d'acceptation jusqu'à ce que la tâche préfixe termine naturellement. Alternativement, si un cycle est trouvé, les deux tâches se terminent, et l'état d'acceptation fautif est remonté au contrôleur.

L'architecture proposée pour la vérification des propriétés de vivacité correspond à cette structure présentée dans la figure 4.2.

Décrite dans la figure 4.3, cette architecture se compose de deux coeurs couplés. Le premier coeur, à droite de la figure, est responsable de l'identification des préfixes poten-

tiels, chemins depuis l'état initial vers l'état d'acceptation. Pour chaque état d'acceptation identifié, ce coeur est mis en pause, et l'état en question est transféré au second coeur, responsable de la confirmation - ou de l'infirmation - des cycles d'acceptation.

Chacun de ces coeurs est implémenté de manière similaire au coeur de sûreté présenté dans le chapitre précédent, composé d'un *Model-Frontend*, qui encapsule la sémantique du modèle et de la propriété vérifiée, et est responsable de l'implémentation de la composition de ces derniers.

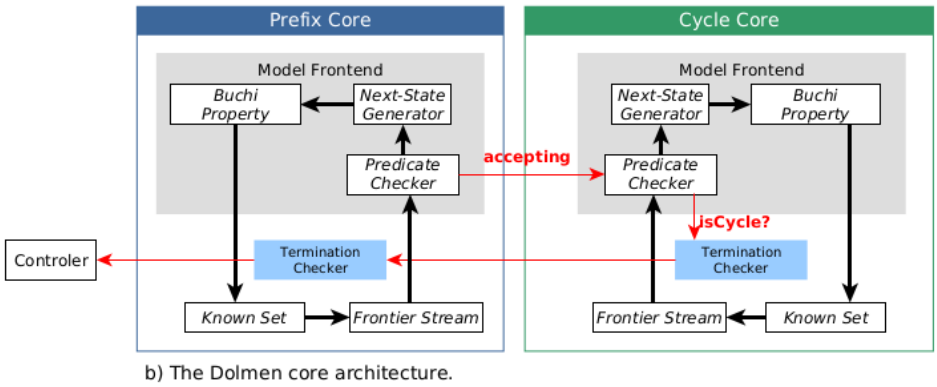


FIGURE 4.3 – Architecture d'un coeur de vérification pour la vivacité

Le Coeur de détection de préfixe est responsable de la détection des états d'acceptation, et corresponds à la fonction dfs_1 de l'algorithme présenté dans la section précédente. Ce dernier est adapté pour la vérification swarm par l'implémentation du *Known set* comme par un filtre de Bloom. Le composant Frontière, responsable du stockage des états en instance d'être parcourus, est implémenté comme une pile représentée par un buffer circulaire bornée en mémoire, pour lequel les états les plus vieux sont écrasés lorsqu'un état est ajouté alors que le buffer est plein. Ce choix de conception mène au même comportement algorithmique que celui de l'algorithme *DFS_FB* présenté dans le chapitre précédent.

Chaque nouvel état sortant de la FIFO est évalué par le *Predicate Checker* avant d'être transmis à la relation de transition qui représente le modèle, représentée par le composant *Next-State Generator*.

Le *Predicate Checker* est un composant générique qui, pour le coeur préfixe, évalue le prédicat d'acceptation sur chaque état qui lui parvient. Dans le cas où ce prédicat renvoie *vrai*, l'état est un état d'acceptation. Dans ce cas, cet état est transmis au coeur

de détection ce cycles pour vérifier si ce dernier est contenu dans un cycle. Dans ce cas, deux cas peuvent se produire : Si un cycle est détecté par le coeur secondaire, la tâche de vérification termine en renvoyant l'état en question au contrôleur central. Dans le cas contraire, l'état d'acceptation est transmis par le predicate-checker au *Next-state generator* pour générer ses successeurs et continuer l'exploration.

Le coeur de détection de cycles débute son exécution à la réception d'un état d'acceptation. Cet état est reçu et stocké par le *Predicate Checker* de ce coeur, avant de le transmettre au *Next-state-generator* qui va produire ses successeurs, et ainsi débuter l'exploration depuis cet état. Le *Predicate checker* a un rôle différent de celui qu'il porte dans le coeur *prefixe* : il ne détecte plus l'acceptation des états, mais, pour chaque état reçu, il vérifie l'égalité entre ce dernier et l'état reçu comme source de l'exploration, avant de le transmettre. La détection d'un tel état corresponds à la détection d'un cycle d'acceptation dans le graphe, et ainsi, constitue un contre-exemple pour la propriété vérifiée.

4.2.2 Composition modèle - propriété

L'introduction d'une interface langage générique, qui constitue un cadre important parmi les choix architecturaux proposés dans ce document, permet de raisonner de manière indépendante sur l'implémentation de la sémantique vérifiée, et celle du modèle.

Dans le cadre de la vérification de propriétés de vivacité, le modèle vérifié, ainsi que la propriété à vérifier sont représentés par des automates de Büchi non déterminites - NBA. Au cours d'une tâche de vérification, le modèle et la propriété évoluent de manière synchrone : Une transition du système composé est tirable seulement si le modèle et la propriété disposent d'une transition tirable depuis l'état considéré.

Une approche peut être d'effectuer cette composition sous la forme d'une transformation effectuée de manière statique, avant la vérification par model-checking. Cependant, cette approche est problématique dans la mesure où l'automate produit peut être très complexe : sa taille est exponentielle en la taille de l'automate de Büchi représentant le modèle.

Toute complexité ajoutée à ces automates se traduit directement en ressources utilisées pour la synthèse du modèle composé sur la plateforme cible. Aussi, pour réduire le besoin en ressources matérielles du *Model-Frontend*, on se propose d'implémenter la propriété vérifiée comme un moniteur. Dans cette configuration, les deux relations de transition sont placées en série, comme représenté dans la description d'architecture 4.3.

Dans cette configuration, chaque état produit par la relation de transition du modèle est transféré à la relation de transition de la propriété. Cette dernière observe l'état du système, et fait évoluer son état interne. Un état de la composition du modèle est de la propriété est alors une simple concaténation de l'état du système vérifié, avec l'état de la propriété. Cet état composite est alors transféré à l'algorithme de vérification.

4.2.3 Predicate checker

Comme précisé précédemment, la détection des états d'acceptation dans un automate de Büchi peut s'assimiler à une simple analyse d'atteignabilité. En effet, une fois le modèle et la propriété encapsulés sous la forme d'une relation de transition de la composition des deux NBAs les représentant, la détection d'un état d'acceptation s'effectue par l'évaluation d'un simple précidat portant sur l'état composite.

Ceci étant dit, au sein de la pipeline du coeur de détection de préfixe, présentée dans la figure 4.3, cette détection peut s'effectuer à plusieurs endroits, a priori de manière indifférente. Le seul endroit de la pipeline où il *n'est pas possible* de placer ce composant est entre le *next-state generator* et la *Buchi property*, représentant respectivement les relations de transition du modèle et de la propriété. Effectuer l'évaluation à cet emplacement n'a pas de sens dans la mesure où à ce point de la pipeline, les configurations composites représentent un état du modèle, concaténé avec l'état de la propriété de son prédécesseur, dans la mesure où la propriété n'a pas encore fait évoluer son état.

Parmi quatre composants constituant la pipeline, et après avoir éliminé un emplacement, il reste trois emplacements potentiels : Entre le *Model-Frontend* et l'ensemble d'états parcourus (*Known*), entre ce dernier et l'ensemble *Frontier*, ainsi qu'à la sortie de l'ensemble *Frontier*, avant le modèle.

Le premier emplacement évoqué, avant le composant *Known set*, n'est pas judicieux : ce composant est responsable du filtrage des dupplicats dans le flot d'états provenant du *model-Frontend*. Aussi, placer ce composant à cet emplacement induit potentiellement des détections multiples d'états d'acceptation, et, ainsi, une perte d'efficacité, dans la mesure où, dans ce cas, un même état serait traité plusieurs fois par le coeur secondaire.

Parmi les deux emplacements restants, avant et après le composant *Frontier*, le deuxième emplacement est préféré dans un contexte swarm. En effet, la diversification des explorations, due à l'action couplée des collisions du Bloom filter, avec le caractère borné de l'ensemble *Frontier*, mène, à un même instant du temps, à des états différents stockés dans la frontière entre deux tâches de vérification. Aussi, évaluer le prédicat d'acceptation

à cet emplacement contribue à ce que deux tâches de vérification n'explore pas les mêmes états d'acceptation, et ainsi, contribue à l'efficacité globale du swarm.

4.2.4 Génération de contre-exemples assistée par le logiciel

Un intérêt majeur du Model-Checking est sa capacité à fournir un contre-exemple lorsqu'une violation de propriété est détectée. L'architecture présentée dans ce chapitre permet de détecter ces derniers, cependant, la construction du contre-exemple s'avère impossible compte-tenu du choix de l'algorithme de parcours, mais également par le fait que la pile contenant les états à parcourir ne conserve pas tous les états. Aussi, seule une partie du contre-exemple est contenue dans cette dernière.

Cependant, une solution alternative est possible : lorsqu'un cycle d'acceptation est détecté le coeur en question transmets au contrôleur central non seulement l'information qu'un cycle a été détecté, mais également les *seed* des fonctions de hash utilisées par le coeur en question.

En se basant sur un modèle exact du comportement du coeur de vérification matériel, il est possible de re-calculer cette tâche précise, instrumentée pour construire un arbre de parents au cours de l'exploration, permettant ainsi la construction du contre-exemple de manière indirecte.

Une telle procédure est réaliste compte tenu du volume mémoire constant et réduit d'une tâche de vérification dans un contexte swarm.

4.2.5 Conclusion

A partir de l'algorithme de vérification de propriétés de vivacité proposé dans la section précédente, cette section construit un coeur de vérification matérielle adapté à la vérification de telles propriétés.

Ce coeur est construit comme la composition de deux instances du coeur Carnac présenté dans le chapitre précédent, bénéficiant ainsi tant des optimisations implémentées, que du caractère modulaire et évolutif de ce dernier.

En outre, l'architecture de contrôle distribuée proposée dans la section 3.3.2 est conservée pour l'agrégation des coeurs en le moteur de vérification swarm Dolmen, profitant ainsi de son caractère scalable et facilitant la synthèse physique.

Pour valider l'architecture proposée, la section suivante propose une évaluation des performances de l'accélérateur Dolmen, effectuée sur un ensemble de 10 couples modèles-

propriétés issus du benchmark BEEM [62].

4.3 Evaluation

Après avoir présenté le coeur de vérification matériel de Dolmen, construit comme l'aggrégation de deux instances du coeurs de vérification présenté dans le chapitre précédent, cette section propose une évaluation des performances de ce moteur de vérification.

L'approche utilisée pour mesurer ses performances repose sur l'utilisation de modèles, accompagnés de leurs propriétés de vivacité, extraits du benchmark BEEM. Pour construire le *Model-Frontend*, composant représentant la relation de transition du modèle vérifié dans le coeur de vérification proposé, l'approche repose sur un générateur de code dédié, compilant les modèles exprimés en DVE en leur implémentation matérielle.

La synthèse de la relation de transition matérielle du modèle permet ensuite de construire l'évaluation du moteur de vérification Dolmen sur des cas d'études représentatifs du cas d'usage du Model-Checking.

4.3.1 Méthodologie

Contrairement au Model-Checking classique, la mesure des performances dans le cadre d'une vérification swarm n'est pas évident. En effet, cette méthode ne dispose pas de condition de terminaison dans le cas où aucun contre-exemple n'est trouvé : Un critère de terminaison aurait pu être la couverture atteinte sur le modèle, mais cette dernière ne peux pas être mesurée. L'approche classique consiste à utiliser un *timeout*, ou définir un nombre maximal de tâches à lancer.

Dans le chapitre précédent, ainsi que dans certains travaux de la littérature, les évaluations de performance reposent sur l'utilisation d'un modèle synthétique, décrit dans [7]. Ce cas synthétique est un test de couverture, recherchant 100 états aléatoires parmi toutes les valeurs d'un entier de 32 bits. Ce cas de test est une bonne approximation de la performance dans le cas de la vérification de sûreté, mais n'est pas suffisant pour évaluer l'algorithme de détection du cycle d'acceptation, utilisé ici pour la vérification de propriétés de vivacité.

Aussi, pour évaluer la performance de l'architecture présentée dans ce chapitre, la métrique utilisée est le temps nécessaire pour trouver une violation dans un modèle pour lequel on sait que la propriété n'est pas satisfaite.

Pour des mesures de performance réalistes, nous nous appuyons sur le BEEM Benchmark[62], un ensemble complet de modèles regroupant des problèmes courants de Model-Checking tels que l'exclusion mutuelle, les protocoles de communication ou l'ordonnancement. Les modèles BEEM sont spécifiés à l'aide du langage DVE, un langage de spécification de bas niveau basé sur des machines à états finis communicantes. Des propriétés de sûreté et de vivacité sont incluses dans chaque modèle.

Ces modèles ne peuvent pas être exploités directement. L'approche proposée repose sur un générateur de code développé spécifiquement pour cette application. Cette génération s'effectue en plusieurs phases de simplification, éliminant la structure de machine à états pour obtenir un système de transition plat, sous la forme de couples prédicat-action, représentant respectivement le prédicat d'activation de la transition, et l'action à effectuer sur l'état courant. Ces transitions sont ensuite générées sous la forme d'un ensemble d'IPs, respectant l'interface AXI-Stream utilisée pour la pipeline. Ces IPs sont aggrégées par deux structures arborescentes, en amont, et en aval des transitions, qui effectuent respectivement la distribution de l'état source courant, puis la sérialisation des états cibles produits sur la pipeline.

L'implémentation de l'automate de Büchi représentant la propriété vérifiée est implémentée comme un moniteur (cf paragraphe 4.2.2), selon le même schéma de génération de code que l'architecture du modèle vérifié.

Compte tenu de la taille et de la complexité du circuit généré, les temps de simulation RTL sont prohibitifs. Ainsi, les valeurs de performance sont obtenues sur une carte Xilinx VCU128 qui comporte un FPGA Virtex Ultrascale+ XCVU37P, synthétisé à l'aide de Xilinx Vivado 2019.2. Les résultats de l'implémentation FPGA sont confrontés à des mesures logicielles, obtenues à l'aide du Model-Checker Divine, version 3.0.90, fonctionnant sur un processeur Intel(R) Xeon(R) E7-8890 v4 avec 128 Go de RAM.

L'évaluation de cette architecture, visant à confronter le moteur de vérification Dolmen à des cas d'étude réalistes, est effectuée sur 10 modèles extraits du benchmark BEEM, compilés depuis le langage DVE vers une architecture VHDL représentant leur relation de transition. La sélection de ces modèles a été faite suivant deux critères : Le respect du cas d'usage propose à la vérification swarm, appliquée pour des modèles dont la taille de l'espace d'état est très supérieure à la taille des filtres de Bloom utilisés par l'algorithme. Le cas contraire implique une capacité à effectuer une exploration exhaustive, l'utilisation d'une approche partielle n'a donc pas de sens dans ce cadre. Ainsi, les modèles sélectionnés ont une taille de 1 à 506 millions d'états. Le deuxième critère est la capacité à synthétiser

les modèles générés par l'outil de génération. Certains modèles contiennent en effet des constructions, comme les accès dans des tableaux imbriqués rendent difficile la synthèse logique sans un travail complexe de *retimming*, ce que l'outil ne supporte pas à ce jour. Cet ensemble de modèles constituent donc les 10 modèles du benchmark synthétisables par la procédure utilisée, et dont la taille de l'espace d'état est la plus grande.

Les performances mesurées sont mises en perspective avec une mesure effectuée sur un outil logiciel, sans limitation de mémoire. Cette valeur de référence est obtenue sur chaque modèle à l'aide du logiciel Divine3, outil de Model-Checking mature et utilisé industriellement.

En outre, pour confirmer l'analyse de temps d'exécution effectuée dans le paragraphe 4.1.3, le moteur de vérification est évalué tout d'abord en fixant une taille égale pour les filtres de Bloom du coeur primaire et secondaire, puis en réduisant fortement ce paramètre pour le coeur secondaire.

model	Divine			Dolmen(19,19)		Dolmen(19,12)				
	Time(s)	States	Accepting	Time(s)	Speedup	Time(s)	Speedup	Cores	LUT	FFs	Memory (kbit)
bakery6prop2	35.52	3.44E+06	1.97E+06	4.27	**8.32**	0.0406	**874.38**	32	753737	1377486	40320
bakery6prop3	25.34	2.56E+06	1.73E+06	2.54	**9.96**	0.0936	**270.78**	32	753907	1377481	40320
bakery7prop2	307.10	2.05E+07	9.07E+06	4.29	**71.63**	0.0407	**7544.51**	32	753847	1377483	40320
bakery7prop3	240.13	1.78E+07	9.39E+06	9.13	**26.31**	0.0941	**2552.02**	32	753850	1377487	40320
bakery8prop3	793.45	4.03E+07	2.05E+07	14.32	**55.41**	0.1361	**5827.93**	32	969126	1755526	40320
elevator4prop2	84.34	1.36E+06	4.74E+05	258.02	**0.33**	1.7922	**47.06**	32	1138247	1809188	40320
elevator4prop3	49.03	1.01E+06	1.18E+05	647.88	**0.08**	4.5003	**10.90**	32	1138149	1809189	40320
iprotocol5prop4	1171.75	7.73E+07	1.57E+07	320.82	**3.65**	3.2683	**358.51**	32	1102224	1759617	40320
szymanski3prop3	35.21	2.06E+06	1.06E+06	0.42	**83.28**	0.0043	**8172.40**	16	942043	1830547	20160
szymanski4prop3	103.37	4.61E+06	2.31E+06	0.42	**243.98**	0.0045	**23090.60**	16	942997	1830545	20160

TABLE 4.1 – Dolmen : Table des résultats

4.3.2 Évaluation des performances

La table 4.1 présente les résultats des mesures effectuées sur l'ensemble de modèles considéré. Ces mesures, effectuées sur 10 modèles, comparent les temps d'exécution du Model-Checker Divine3 avec ceux obtenus par le moteur de vérification matérielle Dolmen.

Cette évaluation est effectuée avec 32 coeurs de vérification pour une majorité des modèles, à l'exception des modèles *szymanski*, pour lesquels les ressources limitées de la plateforme utilisée ont contraint de réduire le nombre de coeurs de vérification à 16.

Les trois premières colonnes présentent le temps d'exécution de l'outil logiciel Divine3, ainsi que le nombre d'états parcourus, et le nombre d'états d'acceptation parcourus. On constate ici que pour tous les modèles, le nombre d'états d'acceptation du même ordre de grandeur que le nombre d'états total : un état sur trois est un état d'acceptation en moyenne sur les modèles considérés. Ce cas se rapproche du pire des cas en termes de temps d'exécution de l'algorithme utilisé, au regard de l'analyse effectuée dans la section 4.2.2. Cette remarque consolide cette analyse, et motive l'évaluation effectuée pour un second paramétrage du filtre de Bloom du coeur secondaire.

Les colonnes suivantes présentent les résultats de Dolmen dans deux configurations : Dolmen(19,19), et Dolmen(19, 12). Ces paramètres font référence à la largeur d'adressage utilisée pour les Bloom filter, respectivement pour les coeurs primaires et secondaire.

Dans la configuration $(19, 19)$, le moteur de vérification Dolmen montre une accélération moyenne de 50x par rapport au logiciel. Ce résultat est décevant, mais tempéré par les très bons résultats de la configuration $(19, 12)$. Dans cette configuration, la réduction de la taille du Bloom filter secondaire induit un gain de performance significatif de près de deux ordres de grandeur par rapport à la configuration précédente, et présente ainsi une accélération moyenne de 4 875x par rapport au logiciel. Ces résultats sont du même ordre de grandeur que l'accélération obtenue par l'accélérateur Dolmen, confirmant l'intérêt de l'accélération matérielle non seulement pour la vérification de propriétés de sûreté, mais également pour la vivacité.

Les résultats sont cependant très dispersés, allant de 1 à 4 ordres de grandeur pour la configuration $(19, 12)$. Cette dispersion peut provenir de plusieurs facteurs, et la largeur de l'état des modèles considéré peut en être un. La pipeline des coeurs de vérification a en effet une largeur de 32 bits, sur lesquelles les configurations sont sérialisées de manière contigüe. Cette sérialisation implique que plusieurs cycles sont nécessaires pour traiter un état : de 6 cycles pour *bakery6*, jusqu'à 11 cycles pour *elevator4*, réduisant ainsi le nombre d'états parcourus par unité de temps. On remarque également une corrélation

entre la taille de l'espace d'état, et le facteur d'accélération, particulièrement flagrante pour les modèles *elevator*, pour lesquels l'espace d'état est de taille relativement faible par rapport à la taille du Bloom filter. Cette situation constitue la limite du cas d'usage de la vérification swarm, au profit de l'algorithme exhaustif utilisé par Divine.

Les trois dernières colonnes présentent l'usage des ressources du FPGA utilisé pour chacun des modèles. Ces ressources sont, en grande partie, consommées par l'implémentation du modèle, représentant à 93% des ressources utilisées dans le cas du modèle *iprotocol5prop4*. De nombreuses optimisations pourraient être implémentées, comme le partage du *Model-Frontend* entre le coeur principal et secondaire, possible dans la mesure où ces derniers travaillent de manière exclusive : le coeur principal est en attente lorsqu'un état d'acceptation a été détecté, et que le coeur secondaire effectue une recherche de cycle. Ce point, laissé pour des travaux futurs, pourrait réduire drastiquement l'utilisation des ressources, permettant l'intégration d'un plus grand nombre de coeur, et ainsi contribuer un accroissement significatif des performances.

4.4 Conclusion

La formalisation complète d'un système ne peut s'envisager en ne spécifiant que des propriétés de sûreté. Les travaux existants portant sur l'accélération matérielle du Model-Checking ont montré gains significatifs en termes de performance, allant jusqu'à 3 ordres de grandeur par rapport au logiciel, mais n'ont concerné que cette première classe de propriétés.

Ce chapitre propose d'aller au dela de la vérification des propriétés de sûreté, en proposant Dolmen, accélérateur matériel pour la vérification swarm de propriétés de vivacité spécifiées sous la forme d'automates de Büchi.

La vérification de propriétés exprimées dans ce formalisme s'effectue par un algorithme de détections de cycles, pour lequel ce chapitre propose des adaptations par rapport à un algorithme de référence, pour s'adapter à la fois aux contraintes de l'implémentation matérielle, mais également aux spécificités de la vérification swarm.

L'implémentation de cet algorithme repose sur les développements architecturaux proposés dans le chapitre précédent, par un couplage de deux coeurs. Cette architecture bénéficie ainsi à la fois du caractère modulaire et des performances élevées de son prédécesseur, mais illustre également le caractère flexible et évolutif de ce dernier.

La crédibilité de cette architecture est illustrée par une évaluation effectuée en matériel

sur un large panel de 10 modèles issus du Benchmark BEEM, représentatifs de problèmes idiomatiques du Model-Checking, qui montre des gains de performance substantiels par rapport au logiciel, de 4 875x en moyenne.

Conclusion

Contexte

La conception de systèmes ne peux s'envisager sans phase de vérification. Cette phase permet de s'assurer, d'une part, que le système ne commette pas d'erreurs nuisibles, et d'autre part, que le système effectue correctement les missions pour lesquelles il a été conçu. En pratique, cela se matérialise par la vérification de l'absence, ou de la présence de certains comportements.

Dans le cadre de la conception de systèmes critiques, qu'ils le soient en termes de risques humains ou financiers, il est souvent nécessaire de formaliser ces comportements dans un langage plus précis, et moins sujet à l'interprétation que le langage naturel. Aussi, dans ce cadre, on modélise les comportements dans des langages formels, dont la sémantique est précise garantit une interprétation non ambiguë. Dans ce cadre, on modélise les systèmes de manière intensionnelle : Le système, associé à son environnement d'exécution, est représenté sous la forme d'une relation de transition, dont les états représentent l'ensemble de l'information nécessaire à l'évolution du système.

Parmi les outils existants pour vérifier le comportement de systèmes, le Model-Checking est une technique automatique très utilisée tant pour la vérification de spécifications, que pour la vérification d'implémentations, comme dans le cadre de la conception électronique, par exemple. Cette technique repose sur une preuve par contre exemple : on effectue une énumération des états atteignables d'un système, en évaluant les propriétés de manière synchronisée avec le système.

Cependant, malgré un effort de recherche très important, tant sur le développement de techniques d'abstraction, que sur l'optimisation des algorithmes utilisés, l'explosion combinatoire reste un problème majeur : ajouter une variable booléenne à un système double le nombre d'états potentiellement atteignables. Le nombre d'états potentiels d'un systèmes augmente donc de manière exponentielle par rapport à son nombre de variables. Cette situation est critique, car pour des problèmes complexes, la taille de l'espace-d'états ne permet pas de terminer la preuve.

Pour pousser plus loin la capacité de vérification de systèmes complexes, plusieurs techniques de Model-Checking partiel ont été introduites. Ces techniques proposent un

compromis : négliger la production d'une preuve d'absence d'erreurs, pour une capacité à vérifier des modèles plus grands. Ces techniques, comme le bitstate-hashing, proposent d'effectuer une sous-approximation de l'espace d'états par l'utilisation de structures de données probabilistes, plus compactes en mémoire.

Bien qu'exhibant de très bons résultats, la qualité de la preuve peut encore être améliorée par la vérification swarm. Cette technique consiste à lancer un grand nombre de tâches de vérification partielles indépendantes sur un même modèle. L'enjeu consiste à obtenir une portion de l'espace d'état couverte par vérification plus large, et donc augmenter la précision de la sous-approximation effectuée.

Cette amélioration se paie au prix fort en termes de temps. Chaque tâche de vérification est couteuse, or, le nombre de tâches de vérifications indépendantes effectuées est typiquement de l'ordre de plusieurs dizaines de milliers. Un enjeu clef est donc d'accélérer ce processus.

Une solution consiste à exploiter l'accélération matérielle. Dans ce cadre, l'utilisation d'architectures reconfigurables pour la vérification swarm a été rapportée dans la littérature, avec des gains significatifs de deux à trois ordres de grandeurs par rapport au logiciel. Cette apport peut donc constituer une rupture importante dans notre capacité de vérification. Cependant, les travaux sur le sujet sont encore à l'état d'études préliminaires, et souffrent d'un manque avéré de modularité pour permettre une démarche de développement incrémentale. Ces travaux permettent donc d'envisager l'accélération globale du Model-Checking, mais s'avèrent encore trop peu matures pour un usage réaliste.

Questions scientifiques

L'enjeu scientifique consiste ici à augmenter la complexité des modèles adressables par Model-Checking. On s'intéresse pour cela aux opportunités offertes par l'accélération du Model-Checking sur des plateformes reconfigurables.

Pour repousser l'explosion combinatoire, problème intrinsèque du Model-Checking, un grand nombre d'algorithmes de vérification ont été proposés pour répondre aux contraintes spécifiques de chaque problème. De manière orthogonale mais complémentaire, de nombreuses sémantiques sont exploités pour spécifier le problème de vérification au juste niveau de raffinement. Une première question adressée est alors celle de la conception d'un accélérateur adaptable non seulement à de multiples algorithmes de vérification, mais également à de multiples sémantiques, permettant de définir les caractéristiques de

la tâche de vérification au problème vérifié.

Dans un second temps, on constate dans la section 1.4.2 des résultats prometteurs de l'accélération matérielle pour la vérification swarm. On se pose ici la question de l'accroissement de ces performances et de la scalabilité, par des optimisations tant algorithmiques que micro-architecturales.

Pour finir, l'ensemble des travaux dans la littérature n'ont porté que sur l'accélération matérielle de la vérification de propriétés de sûreté. Cependant, la spécification des systèmes repose sur ces propriétés, mais également sur les propriétés de vivacité, qui spécifient les comportements désirés du système. Ainsi, la question qui se pose ici est celle de l'extension des accélérateurs existants pour la vérification de propriétés de vivacité.

Contributions

L'étude de ces questions scientifiques a donné lieu à plusieurs contributions, détaillées dans ce manuscrit.

Le besoin de généricité algorithmique et sémantique est adressé ici par la modularité. Cette modularité, telle que nous la présentons, concerne en premier lieu l'isolation de la sémantique du modèle vis-à-vis du moteur de vérification ; à ce titre, elle est gage d'une capacité de réutilisation du moteur de vérification pour différentes sémantiques d'entrée. Celle-ci concerne également le coeur algorithmique du moteur de vérification qui, par l'isolation de composants clefs des algorithmes implémentés au travers d'interfaces abstraites, permet l'implémentation de multiples algorithmes au sein d'un même moteur de vérification. Ces deux volets sont illustrés par l'implémentation au sein d'une même moteur de vérification, de 6 algorithmes et de trois langages de spécification, formant un dégradé respectivement depuis la vérification exhaustive vers la vérification partielle, et depuis des langages de spécification industriels de haut niveau comme UML, vers des implémentations dédiées en RTL.

Le caractère modulaire de cette architecture mène directement à l'identification de points de variabilité algorithmiques, qui sont exploités dans le chapitre 3 pour effectuer une exploration de l'espace de conception algorithmique, visant à accroitre l'efficacité des algorithmes de vérification swarm. Cette étude théorique, effectuée sur un ensemble de 8 algorithmes, mène à la sélection d'un algorithme, présentant un gain d'efficacité moyen de 249% par rapport à un algorithme de référence. Ces gains théoriques ne sont intéressants que si implémentés dans une architecture performante. Aussi, l'algorithme sélectionné est implémenté au sein d'un moteur de vérification conçu pour maximiser

les performances, tout en conservant les avantages de l'approche précédente en termes de généricité algorithmique et sémantique. Les gains combinés de cet algorithme et du moteur de vérification, conçu avec un soin particulier pour la scalabilité sur des FPGAs multi-die, atteignent un facteur *7.58*x par rapport à l'état de l'art.

Dans un troisième temps, le problème de la vérification swarm de propriétés de vivacité est considéré. L'étude d'un algorithme de référence pour la vérification des propriétés de vivacité mène ici à la construction d'un algorithme de vérification swarm pour la vérification de ces propriétés formalisées sous la forme d'automates de Büchi, adapté à la fois aux spécificités du swarm, mais également aux contraintes imposées par une implémentation matérielle. La vérification d'automates de Büchi est effectuée par le couplage de deux coeurs de vérification de sûreté pour effectuer la détection des cycles d'acceptation. Cette réorientation constitue une bonne illustration du caractère générique tant des coeurs de vérification que du moteur qui les agrège dans une architecture distribuée. Cette étude, première du genre à notre connaissance tant sur la vérification swarm des propriétés de vivacité que sur son accélération matérielle, montre des résultats extrêmement prometteurs, et en ligne avec les performances démontrées pour la sûreté, atteignant un facteur d'accélération moyen de 4 875x en moyenne sur 10 modèles du benchmark BEEM, représentatifs de cas d'applications réels de la technique.

Perspectives

Au cours de ces travaux de thèse, de nombreuses perspectives ont émergé. On présente ici des pistes pour des travaux futurs, sur un axe algorithmique tout d'abord, puis sur un axe sémantique.

Axe algorithmique Le chapitre 4 propose une nouvelle approche pour la vérification de propriétés de vivacité, conjuguant accélération matérielle et un nouvel algorithme de vérification swarm pour la vivacité. Cette étude montre des résultats en ligne avec les performances exceptionnelles présentées dans le cadre de la vérification de sûreté, en présentant des facteurs d'accélération de trois à quatre ordres de grandeur par rapport à un Model-Checker logiciel. Il s'agit, à notre connaissance, de la seule étude proposant l'exploitation de méthodes swarm pour la vérification de propriétés de vivacité, a fortiori en matériel. Ces travaux très prometteurs méritent d'être approfondis, et il est pour cela critique d'étude une méthodologie rigoureuse de caractérisation des algorithmes de vérification swarm pour la vivacité.

En premier lieu, l'algorithme présenté doit être caractérisé précisément dans un contexte swarm.

Cette démarche visera en premier lieu à chiffrer l'impact des choix de conception algorithmiques sur le recouvrement des tâches de vérification, et les conséquences sur le temps d'exécution global de la vérification swarm. Cette caractérisation devra de plus prendre en compte l'influence des méta-paramètres de l'algorithme, en particulier, le volume mémoire alloué à chacun des filtres de Bloom.

La mise en place de cette méthodologie d'évaluation permettra, dans un second temps, de mettre en place une démarche de conception incrémentale, visant à optimiser les algorithmes proposés pour maximiser la capacité d'exploration dans un volume mémoire et un temps contraint.

Axe sémantique Le deuxième enjeu concerne l'implémentation de la relation de transition représentant le modèle vérifié, associé à la propriété mise en jeu. Deux questions majeures se posent ici.

En premier lieu, serait opportun d'étudier la possibilité de réutiliser les implémentations existantes pour la représentation du modèle. Cette question va de pair avec la question de l'affranchissement du coût de synthèse des modèles vérifiés, coût qui n'a pas été discuté dans ce manuscrit, mais qui constitue un enjeu critique pour rendre réaliste l'utilisation de moteurs de vérification matériels dans un cadre industriel. Notons qu'il ne s'agit ici que du coût de synthèse logique et physique du modèle vérifié, dans la mesure où les coeurs algorithmiques et le moteur de vérification en lui-même peuvent être générés au préalable, par l'utilisation de techniques de synthèse partielle, ou de reconfiguration dynamique.

Deux axes se présentent ici. Il s'agit tout d'abord d'évaluer l'opportunité d'implémenter en logiciel les sémantiques, que ce soit sur des plateformes multi-coeurs, ou SIMD. Cette solution permettrai de s'affranchir intégralement du coût de synthèse, et de réutiliser directement les implémentations existantes de langages de spécifications. Cependant, la faisabilité de cette approche doit être évaluée, en particulier, il est nécessaier de préciser la capacité de telles plateformes hybrides à fournir un débit de configurations suffisant pour atteindre la cadence de traitement de l'accélérateur algorithmique matériel. Il est ici nécessaire de prendre en compte les canaux entre la plateforme d'interprétation de la sémantique, et l'accélérateur, en termes de bande passante, mais également l'impact de la latence engendrée sur l'algorithme.

Une autre approche consiste à réaliser l'implémentation de la sémantique sur le FPGA. Deux travaux existent à ce jour, dont l'une d'entre elle a fait l'objet d'une publication [63]. Cet article propose l'implémentation d'une pipeline programmable dédié à l'interprétation du langage DVE, permettant de s'affranchir totalement du coût de synthèse. Il est cependant notable qu'aucune évaluation des performances de cette implémentation n'a été proposée[63]. La génération de code, évoquée et utilisée pour l'évaluation effectuée dans le chapitre 4, constitue une deuxième opportunité potentielle pour permettre l'implémentation de sémantiques abstraites en matériel. Cette approche présente l'intérêt de présenter des performances intéressantes, mais nécessite d'être consolidée pour atteindre à la fois le débit optimal des coeurs de vérification - 1 état/clk - mais également de réduire fortement l'utilisation des ressources physiques.

Le problème qui se pose ici, dans son ensemble, est de fournir une solution générique permettant l'exécution de sémantiques arbitraires avec un débit suffisant pour alimenter le coeur algorithmique matériel. L'ordre de grandeur à atteindre est ici de 50 millions d'états par seconde [1], correspondant à une largeur de pipeline de 32 bits, opérant à 400MHz, pour un état de 256 bits [2].

1. Le calcul ici est le suivant : Un état de 256 bits est traité tous les 8 cycles d'horloge dans un pipeline de 32 bits de large, à une fréquence de 400MHz, soit une fréquence de traitement des états de 50MHz.
2. On se réfère ici à la largeur moyenne des configurations du benchmark BEEM, qui est de 239 bits.

BIBLIOGRAPHIE

[1] C. BAIER et J.-P. KATOEN, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008, ISBN : 026202649X.

[2] E. M. CLARKE, M. KHAIRA et X. ZHAO, « Word Level Model Checking—Avoiding the Pentium FDIV Error », in *Proceedings of the 33rd Annual Design Automation Conference*, sér. DAC '96, Las Vegas, Nevada, USA : Association for Computing Machinery, 1996, p. 645-648, ISBN : 0897917790. DOI : 10.1145/240518.240640. adresse : https://doi.org/10.1145/240518.240640.

[3] C. NEWCOMBE, T. RATH, F. ZHANG, B. MUNTEANU, M. BROOKER et M. DEARDEUFF, « How Amazon Web Services Uses Formal Methods », *Commun. ACM*, t. 58, *4*, p. 66-73, mar. 2015, ISSN : 0001-0782. DOI : 10.1145/2699417. adresse : https://doi.org/10.1145/2699417.

[4] E. M. CLARKE, O. GRUMBERG, M. MINEA et D. PELED, « State space reduction using partial order techniques », *International Journal on Software Tools for Technology Transfer*, t. 2, *3*, p. 279-287, 1999.

[5] E. M. CLARKE, E. A. EMERSON et J. SIFAKIS, « Model Checking : Algorithmic Verification and Debugging », *Commun. ACM*, t. 52, *11*, p. 74-84, nov. 2009, ISSN : 0001-0782. DOI : 10.1145/1592761.1592781. adresse : https://doi.org/10.1145/1592761.1592781.

[6] S. CHO, M. FERDMAN et P. MILDER, « FPGASwarm : High Throughput Model Checking on FPGAs », in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, août 2018, p. 435-4357.

[7] G. HOLZMANN, R. JOSHI et A. GROCE, « Swarm Verification Techniques. », *IEEE Trans. Software Eng.*, t. 37, p. 845-857, jan. 2011. DOI : 10.1109/ASE.2008.9.

[8] L. LAMPORT, « Proving the Correctness of Multiprocess Programs », *IEEE Transactions on Software Engineering*, t. SE-3, *2*, p. 125-143, 1977. DOI : 10.1109/TSE.1977.229904.

[9] E. M. Clarke et E. A. Emerson, « Design and synthesis of synchronization skeletons using branching time temporal logic », in *Logics of Programs*, D. Kozen, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 1982, p. 52-71, isbn : 978-3-540-39047-3.

[10] J. P. Queille et J. Sifakis, « Specification and verification of concurrent systems in CESAR », in *International Symposium on Programming*, M. Dezani-Ciancaglini et U. Montanari, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 1982, p. 337-351, isbn : 978-3-540-39184-5.

[11] V. Štill, P. Ročkai et J. Barnat, « DIVINE : Explicit-State LTL Model Checker », in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik et J.-F. Raskin, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 2016, p. 920-922, isbn : 978-3-662-49674-9.

[12] J.-R. Abrial, *Modeling in Event-B : system and software engineering*. Cambridge University Press, 2010.

[13] « IEEE Standard for Property Specification Language (PSL) », *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, p. 1-182, 2010. doi : 10.1109/IEEESTD.2010.5446004.

[14] « IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language », *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, p. 1-1315, 2018. doi : 10.1109/IEEESTD.2018.8299595.

[15] C. Courcoubetis, M. Vardi, P. Wolper et M. Yannakakis, « Memory-efficient algorithms for the verification of temporal properties », *Formal Methods in System Design*, t. 1, *2*, p. 275-288, oct. 1992, issn : 1572-8102. doi : 10.1007/BF00121128. adresse : https://doi.org/10.1007/BF00121128.

[16] Z. Manna et A. Pnueli, « A hierarchy of temporal properties (invited paper, 1989) », in *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, 1990, p. 377-410.

[17] L. Lamport, « Proving the Correctness of Multiprocess Programs », *IEEE Transactions on Software Engineering*, t. SE-3, *2*, p. 125-143, 1977. doi : 10.1109/TSE.1977.229904.

[18] E. Clarke, O. Grumberg, S. Jha, Y. Lu et H. Veith, « Counterexample-guided abstraction refinement », in *International Conference on Computer Aided Verification*, Springer, 2000, p. 154-169.

[19] K. L. McMillan, « Symbolic model checking », in *Symbolic Model Checking*, Springer, 1993, p. 25-60.

[20] G. J. Holzmann, « An Analysis of Bitstate Hashing », *Formal Methods in System Design*, t. 13, *3*, p. 289-307, 1998.

[21] U. Stern et D. L. Dill, « Improved probabilistic verification by hash compaction », in *Correct Hardware Design and Verification Methods*, P. E. Camurati et H. Eveking, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 1995, p. 206-224, isbn : 978-3-540-45516-5.

[22] J. Barnat, J. Havlíček et P. Ročkai, « Distributed LTL Model Checking with Hash Compaction », *Electronic Notes in Theoretical Computer Science*, t. 296, p. 79-93, août 2013. doi : 10.1016/j.entcs.2013.07.006.

[23] G. J. Holzmann, R. Joshi et A. Groce, « Swarm Verification », in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, sept. 2008, p. 1-6.

[24] G. J. Holzmann, R. Joshi et A. Groce, « Tackling Large Verification Problems with the Swarm Tool », in *Model Checking Software*, K. Havelund, R. Majumdar et J. Palsberg, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 134-143, isbn : 978-3-540-85114-1.

[25] M. B. Dwyer, S. Elbaum, S. Person et R. Purandare, « Parallel Randomized State-Space Search », in *Proceedings of the 29th International Conference on Software Engineering*, sér. ICSE '07, USA : IEEE Computer Society, 2007, p. 3-12, isbn : 0769528287. doi : 10.1109/ICSE.2007.62. adresse : https://doi.org/10.1109/ICSE.2007.62.

[26] L. Lamport, *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers*. USA : Addison-Wesley Longman Publishing Co., Inc., 2002, isbn : 032114306X.

[27] V. Besnard, C. Teodorov, F. Jouault, M. Brun et P. Dhaussy, « Unified verification and monitoring of executable UML specifications », *Software and Systems Modeling*, t. 20, *6*, p. 1825-1855, déc. 2021, issn : 1619-1374. doi : 10.1007/s10270-021-00923-9. adresse : https://doi.org/10.1007/s10270-021-00923-9.

[28] S. Edelkamp, S. Leue et A. Lluch-Lafuente, « Directed Explicit-State Model Checking in the Validation of Communication Protocols », *International Journal on Software Tools for Technology Transfer*, t. 5, p. 247-267, jan. 2004. doi : 10.1007/s10009-002-0104-3.

[29] L. Lamport et S. Merz, « Auxiliary variables in TLA+ », *arXiv preprint arXiv :1703.05121*, 2017.

[30] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom et T. van Dijk, « LTSmin : High-Performance Language-Independent Model Checking », in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier et C. Tinelli, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 2015, p. 692-707, isbn : 978-3-662-46681-0.

[31] D. L. Dill, « The Murϕ verification system », in *Computer Aided Verification*, R. Alur et T. A. Henzinger, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 1996, p. 390-393, isbn : 978-3-540-68599-9.

[32] M. E. Tie, « Accelerating explicit state model checking on an FPGA : PHAST », mém. de mast., Northeastern University, Boston, mai 2012.

[33] G. Holzmann, « An Analysis of Bitstate Hashing », *Formal Methods in System Design*, t. 13, mai 2003. doi : 10.1023/A:1008696026254.

[34] G. Behrmann, « Distributed reachability analysis in timed automata », *International Journal on Software Tools for Technology Transfer*, t. 7, *1*, p. 19-30, 2005.

[35] S. Aggarwal, R. Alonso et C. Courcoubetis, « Distributed reachability analysis for protocol verification environments », in. nov. 2006, t. 103, p. 40-56. doi : 10.1007/BFb0042303.

[36] F. Lerda et R. Sisto, « Distributed-memory model checking with SPIN », in *International SPIN Workshop on Model Checking of Software*, Springer, 1999, p. 22-39.

[37] A. Dalsgaard, A. Laarman, K. Larsen, M. Chr, Olesen et J. Pol, « Multi-Core Reachability for Timed Automata », sept. 2012, isbn : 978-3-642-33364-4. doi : 10.1007/978-3-642-33365-1_8.

[38] A. Laarman, J. Pol et M. Weber, « Boosting multi-core reachability performance with shared hash tables. », jan. 2010, p. 247-255.

[39] A. Wijs et D. Bosnacki, « Many-core on-the-fly model checking of safety properties using GPUs », *International Journal on Software Tools for Technology Transfer*, t. 18, avr. 2015. doi : 10.1007/s10009-015-0379-9.

[40] A. Wijs, « BFS-Based Model Checking of Linear-Time Properties with an Application on GPUs », t. 9780, juil. 2016, p. 472-493, isbn : 978-3-319-41539-0. doi : 10.1007/978-3-319-41540-6_26.

[41] R. DeFrancisco, S. Cho, M. Ferdman et S. A. Smolka, « Swarm model checking on the GPU », *International Journal on Software Tools for Technology Transfer*, t. 22, *5*, p. 583-599, 2020.

[42] L. Brim, I. Cerna, P. Moravec et J. Šimša, « Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking », jan. 2005, p. 352-366, isbn : 978-3-540-23738-9. doi : 10.1007/978-3-540-30494-4_25.

[43] J. Barnat, L. Brim et J. Chaloupka, « Parallel breadth-first search LTL model-checking », nov. 2003, p. 106-115, isbn : 0-7695-2035-9. doi : 10.1109/ASE.2003.1240299.

[44] I. Cerna et R. Pelánek, « Distributed Explicit Fair Cycle Detection (Set Based Approach) », jan. 2003, p. 49-73, isbn : 978-3-540-40117-9. doi : 10.1007/3-540-44829-2_4.

[45] L. Brim, I. Cerna, P. Moravec et J. Simsa, « How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors », *Electronic Notes in Theoretical Computer Science*, t. 135, p. 3-18, fév. 2006. doi : 10.1016/j.entcs.2005.10.015.

[46] L. Brim, I. Černá, P. Krčál et R. Pelánek, « Distributed LTL model checking based on negative cycle detection », in *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer, 2001, p. 96-107.

[47] C. P. Inggs et H. Barringer, « CTL* Model Checking on a Shared-Memory Architecture », *Electronic Notes in Theoretical Computer Science*, t. 128, *3*, p. 107-123, 2005, Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004), issn : 1571-0661. doi : https://doi.org/

10.1016/j.entcs.2004.10.022. adresse : https://www.sciencedirect.com/science/article/pii/S1571066105001714.

[48] A. Laarman, M. Olesen, A. Dalsgaard, K. Larsen et J. Pol, « Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction », t. 8044, juil. 2013, p. 968-983, isbn : 978-3-642-39798-1. doi : 10.1007/978-3-642-39799-8_69.

[49] S. Evangelista, A. Laarman, L. Petrucci et J. Pol, « Improved Multi-Core Nested Depth-First Search », t. 7561, oct. 2012, p. 269-283, isbn : 978-3-642-33385-9. doi : 10.1007/978-3-642-33386-6_22.

[50] A. Laarman, R. Langerak, J. Pol, M. Weber et A. Wijs, « Multi-core Nested Depth-First Search », t. 6996, oct. 2011, p. 321-335, isbn : 978-3-642-24371-4. doi : 10.1007/978-3-642-24372-1_23.

[51] S. Evangelista, L. Petrucci et S. Youcef, « Parallel Nested Depth-First Searches for LTL Model Checking », t. 6996, oct. 2011, p. 381-396, isbn : 978-3-642-24371-4. doi : 10.1007/978-3-642-24372-1_27.

[52] D. Bošnački, S. Edelkamp et D. Sulewski, « Efficient probabilistic model checking on general purpose graphics processors », in *International SPIN Workshop on Model Checking of Software*, Springer, 2009, p. 32-49.

[53] E. Bartocci, R. DeFrancisco et S. A. Smolka, « Towards a GPGPU-parallel SPIN model checker », in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, 2014, p. 87-96.

[54] M. E. Fuess, M. Leeser et T. Leonard, « An FPGA Implementation of Explicit-State Model Checking », in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, avr. 2008, p. 119-126.

[55] G. J. Holzmann, « The model checker SPIN », *IEEE Transactions on Software Engineering*, t. 23, *5*, p. 279-295, mai 1997, issn : 2326-3881.

[56] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill et L. J. Hwang, « Symbolic Model Checking : 1020 States and Beyond », *Inf. Comput.*, t. 98, *2*, p. 142-170, juin 1992, issn : 0890-5401.

[57] A. Udupa, A. Desai et S. Rajamani, « Depth Bounded Explicit-State Model Checking », in *Model Checking Software*, A. Groce et M. Musuvathi, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, p. 57-74, isbn : 978-3-642-22306-8.

[58] A. Laarman, « Scalable multi-core model checking », Undefined, IPA Dissertation Series No. 2014-06, thèse de doct., University of Twente, Netherlands, mai 2014, isbn : 978-90-365-3656-1.

[59] OMG, *Unified Modeling Language*, déc. 2017. adresse : https://www.omg.org/spec/UML/2.5.1/PDF.

[60] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill et J. Weiser, « DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs », in *Computer Aided Verification*, N. Sharygina et H. Veith, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, p. 863-868, isbn : 978-3-642-39799-8.

[61] V. Besnard, M. Brun, F. Jouault, C. Teodorov et P. Dhaussy, « Unified LTL Verification and Embedded Execution of UML Models », in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, sér. MODELS '18, Copenhagen, Denmark : Association for Computing Machinery, 2018, p. 112-122, isbn : 9781450349499.

[62] R. Pelánek, « BEEM : Benchmarks for Explicit Model Checkers », in *Model Checking Software*, D. Bošnački et S. Edelkamp, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, p. 263-267, isbn : 978-3-540-73370-6.

[63] M. Patel, S. Cho, M. Ferdman et P. Milder, « Runtime-Programmable Pipelines for Model Checkers on FPGAs », in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, p. 51-58. doi : 10.1109/FPL.2019.00018.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

ENSTA Bretagne

Titre : Accélération matérielle de la vérification de sûreté et vivacité sur des architectures reconfigurables

Mot clés : Model-Checking, Swarm verification, Reconfigurable Architectures

Résumé : Le Model-Checking est une technique automatisée, utilisée dans l'industrie pour la vérification, enjeu majeur pour la conception de systèmes fiables, cadre dans lequel performance et scalabilité sont critiques. La vérification swarm améliore la scalabilité par une approche partielle reposant sur l'exécution concurrente d'analyses randomisées. Les architectures reconfigurables promettent des gains de performance significatifs. Cependant, les travaux existant souffrent d'une conception monolithique qui freine l'exploration des opportunités des architectures reconfigurable. De plus, ces travaux sont limités a la verification de sûreté.

Pour adapter la stratégie de vérification au problème, cette thèse propose un framework de vérification matérielle, permettant de gagner, au travers d'une architecture modulaire, une généricité sémantique et algorithmique, illustrée par l'intégration de 3 langages de spécification et de 6 algorithmes. Ce cadre architectural permet l'étude de l'efficacité des algorithmes swarm pour obtenir un coeur de vérification de sûreté scalable. Les résultats, sur un FPGA haut de gamme, montrent des gains d'un ordre de grandeur par rapport à l'état de l'art. Enfin, on propose le premier accélérateur matériel permettant la vérification des exigences de sûreté et de vivacité. Les résultats démontrent un facteur d'accélération moyen de *4875*x par rapport au logiciel.

Title: Hardware Acceleration of Safety and Liveness Verification on Reconfigurable Architectures

Keywords: Model-Checking, Swarm verification, Reconfigurable Architectures

Abstract:

Model-Checking is an automated technique used in industry for verification, a major issue in the design of reliable systems, where performance and scalability are critical. Swarm verification improves scalability through a partial approach based on concurrent execution of randomized analyses. Reconfigurable architectures promise significant performance gains. However, existing work suffers from a monolithic design that hinders the exploration of reconfigurable architecture opportunities. Moreover, these studies are limited to safety verification.

To adapt the verification strategy to the problem, this thesis first proposes a hardware verification framework, allowing to gain, through a modular architecture, a semantic and algorithmic genericity, illustrated by the integration of 3 specification languages and 6 algorithms. This framework allows efficiency studies of swarm algorithms to obtain a scalable safety verification core. The results, on a high-end FPGA, show gains of an order of magnitude compared to the state-of-the-art. Finally, we propose the first hardware accelerator for safety and liveness verification. The results show an average speed-up of *4875*x compared to software.