Encodage Efficace des Systèmes Critiques pour la Vérificaton Formelle par Model Checking à base de Solveurs SAT

Guillaume Baud-Berthier


THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Guillaume Baud-Berthier**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Encodage efficace des systèmes critiques pour la vérification formelle par model checking à base de solveurs SAT

Date de soutenance : 20 septembre 2018

Devant la commission d'examen composée de :

Armin Biere	Professeur des universités, FMV JKU	Rapporteur
Sylvain Conchon	Professeur des universités, LRI	Rapporteur
Jean-Frédéric Etienne	Ingénieur de recherche, SafeRiver	Examinateur
Frédéric Herbreteau	Maître de conférences, LaBRI	Examinateur
Laurence Pierre	Professeur des universités, TIMA	Présidente du jury
Laurent Simon	Professeur des universités, LaBRI	Directeur

– 2018 –

Résumé Le développement de circuits électroniques et de systèmes logiciels critiques pour le ferroviaire ou l'avionique, par exemple, demande à être systématiquement associé à un processus de vérification formelle permettant de garantir l'exhaustivité des tests. L'approche formelle la plus répandue dans l'industrie est le Model Checking. Le succès de son adoption provient de deux caractéristiques : (i) son aspect automatique, (ii) sa capacité à produire un témoin (un scénario rejouable) lorsqu'un comportement indésirable est détecté, ce qui fournit une grande aide aux concepteurs pour corriger le problème. Néanmoins, la complexité grandissante des systèmes à vérifier est un réel défi pour le passage à l'échelle des techniques existantes. Pour y remédier, différents algorithmes de model checking (*e.g.*, parcours symbolique des états du système, interpolation), diverses méthodes complémentaires (*e.g.*, abstraction, génération automatique d'invariants), et de multiples procédures de décision (*e.g.*, diagramme de décision, solveur SMT) sont envisageables.

Dans cette thèse, nous nous intéressons plus particulièrement à l'induction temporelle. Il s'agit d'un algorithme de model checking très utilisé dans l'industrie pour vérifier les systèmes critiques. C'est également l'algorithme principal de l'outil développé au sein de l'entreprise `SafeRiver`, entreprise dans laquelle cette thèse a été effectuée. Plus précisément, l'induction temporelle combine deux techniques : (i) BMC (*Bounded Model Checking*), une méthode très efficace pour la détection de bugs dans un système (ii) k-induction, une méthode ajoutant un critère de terminaison à BMC lorsque le système n'admet pas de bug. Ces deux techniques génèrent des formules logiques propositionnelles pour lesquelles il faut en déterminer la satisfaisabilité. Pour se faire, nous utilisons un solveur SAT, c'est-à-dire une procédure de décision qui détermine si une telle formule admet une solution.

L'originalité des travaux proposés dans cette thèse repose en grande partie sur la volonté de renforcer la collaboration entre le solveur SAT et le model checker. Nos efforts visent à accroître l'interconnexion de ces deux modules en exploitant la structure haut niveau du problème. Nous avons alors défini des méthodes profitant de la structure symétrique des formules. Cette structure apparaît lors du dépliage successif de la relation de transition, et nous permet de dupliquer les clauses ou encore de déplier les transitions dans différentes directions (*i.e.*, avant ou arrière). Nous avons aussi pu instaurer une communication entre le solveur SAT et le model checker permettant de : (i) simplifier la représentation au niveau du model checker grâce à des informations déduites par le solveur, et (ii) aider le solveur lors de la résolution grâce aux simplifications effectuées sur la représentation haut niveau.

Une autre contribution importante de cette thèse est l'expérimentation des algorithmes proposées. Cela se concrétise par l'implémentation d'un model checker prenant en entrée des modèles AIG (*And-Inverter Graph*) dans lequel nous avons pu évaluer l'efficacité de nos différentes méthodes.

Mots-clés Vérification Formelle, Model checking, Induction temporelle, BMC, k-induction, Satisfaisabilité, Solveur SAT

Title Effective Encoding of Critical Systems for SAT-Based Model Checking

Abstract The design of electronic circuits and safety-critical software systems in railway or avionic domains for instance, is usually associated with a formal verification process. More precisely, test methods for which it is hard to show completeness are combined with approaches that are complete by definition. Model Checking is one of those approaches and is probably the most prevalent in industry. Reasons of its success are mainly due to two characteristics, namely: (i) its fully automatic aspect, and (ii) its ability to produce a short execution trace of undesired behaviors, which is very helpful for designers to fix the issues. However, the increasing complexity of systems to be verified is a real challenge for the scalability of existing techniques. To tackle this challenge, different model checking algorithms (*e.g.*, symbolic model checking, interpolation), various complementary methods (*e.g.*, abstraction, automatic generation of invariants) and multiple decision procedures (*e.g.*, decision diagram, SMT solver) can be considered.

In this thesis, we particularly focus on temporal induction. It is a model checking algorithm widely used in the industry to check safety-critical systems. This is also the core algorithm of the tool developed within `SafeRiver`, company in which this thesis was carried out. More precisely, temporal induction consists of a combination of BMC (*Bounded Model Checking*) and k-induction. BMC is a very efficient bug-finding method. While k-induction adds a termination criterion to BMC when the system does not admit bugs. These two techniques generate formulas for which it is necessary to determine their satisfiability. To this end, we use a SAT solver as a decision procedure to determine whether a propositional formula has a solution.

The main contribution of this thesis aims to strengthen the collaboration between the SAT solver and the model checker. The improvements proposed mainly focus on increasing the interconnections of these two modules by exploiting the high-level structure of the problem. We have therefore defined several methods taking advantage of the symmetrical structure of the formulas. This structure emerges during the successive unfolding of the transition relation, and allows us to duplicate clauses or even unroll the transitions in different directions (*i.e.*, forward or backward). We also established a communication between the solver and the model checker, which has for purpose to: (i) simplify the model checker representation using the information inferred by the solver, and (ii) assist the solver during resolution with simplifications performed on the high-level representation. Another important contribution of this thesis is the empirical evaluation of the proposed algorithms on well-established benchmarks. This is achieved concretely via the implementation of a model checker taking AIG (*And-Inverter Graph*) as input, from which we were able to evaluate the effectiveness of our algorithms.

Keywords Formal Verification, Model Checking, Temporal Induction, BMC, k-induction, Satisfiability, SAT Solver

Table des matières

Chapitre 1

Introduction

Les systèmes informatiques (matériels et logiciels) sont omniprésents dans nos vies quotidiennes. On peut ainsi les trouver au cœur des systèmes de conception et de pilotage dans l'automobile, le ferroviaire et l'aéronautique. Cette position centrale soulève cependant un problème : la défaillance d'un système peut avoir de lourdes conséquences, que ce soit en pertes humaines ou économiques. C'est pourquoi il est nécessaire de s'assurer que le comportement de ces systèmes dits *critiques* soit sans faille. Pour cela, les industries des secteurs concernés ont mis en place des techniques de **vérification formelle** dans leur processus de développement. En effet, contrairement aux procédures classiques de tests, consistant à vérifier le bon comportement de ces systèmes sur un nombre limité de scénarios, ces méthodes sont exhaustives et renforcent ainsi la confiance accordée aux systèmes.

Le processus de développement des systèmes critiques, se résume globalement en trois grandes étapes (i) modélisation du système, (ii) vérification du modèle, et (iii) génération de code à partir du modèle. Ce processus est illustré par la figure 1.1. L'étape de modélisation consiste à encoder les spécifications du système dans un langage formelle, c'est-à-dire avec une sémantique mathématique bien définie, *e.g.*, AltaRica (Arnold *et al.* [1999]) ou Lustre (Halbwachs *et al.* [1991]). Ensuite, les risques potentiels du système sont identifiés, et à partir de ces risques, des exigences sont définies. Ces exigences sont des propriétés que le modèle doit respecter pour s'assurer du bon comportement du système. Il y a deux types de propriétés : (i) les propriétés de sûreté, et (ii) les propriétés de vivacité. Les premières servent à décrire des comportements à éviter, en d'autres termes le système doit toujours être dans un état sûr. Les secondes décrivent des comportements souhaités, qui doivent se produire, autrement dit le système reste réactif, disponible, et n'est pas *bloqué*. Ces propriétés sont généralement définies en logique temporelle, comme LTL (*Linear Temporal Logic*) ou CTL (*Computational Tree Logic*), car ces logiques possèdent des opérateurs permettant de définir des comportements relatifs au temps, qui sont très pratiques pour exprimer ces exigences, par exemple : *toujours*, *dans le futur*,

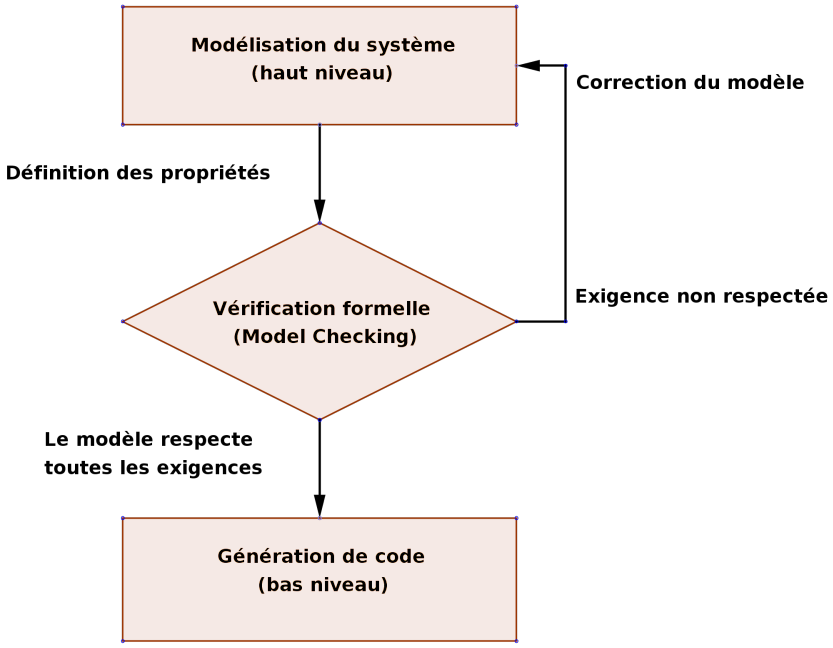


FIGURE 1.1 – Processus global simplifié du développement d'un système critique.

tant que, etc (voir Emerson [1990] pour une référence complète sur les logiques temporelles). Puis, à partir du modèle et des propriétés, la spécification, la conception et l'implémentation haut niveau du système peuvent être vérifiées à l'aide de méthodes formelles.

Il existe différentes méthodes de vérification, par exemple, l'analyse statique (interprétation abstraite), la vérification déductive (démonstration de théorème à l'aide d'un assistant de preuve ou de façon automatique), ou le *Model Checking*. Le Model Checking (MC, Clarke *et al.* [1986, 1999]) est une technique automatique de vérification formelle qui détermine si un modèle vérifie une propriété donnée, autrement dit la propriété est-elle vraie pour tous les chemins d'exécution du système. Le MC est certainement la méthode de vérification formelle la plus répandue dans le milieu industriel. Une des raisons de son adoption est sa capacité à fournir un témoin, lorsqu'un modèle ne vérifie pas une exigence, c'est-à-dire falsifie une propriété. Ce témoin est généralement appelé *contre-exemple* (CEX). Ce CEX correspond à un chemin d'exécution du système, il peut donc être simulé (rejoué), ce qui fournit une aide considérable aux concepteurs pour comprendre et corriger les erreurs si nécessaire.

Enfin, c'est seulement une fois que toutes les propriétés sont vérifiées pour un modèle que le code cible peut être généré avec un niveau de confiance élevé dans la sûreté du système. Le code cible est une représentation du système d'un

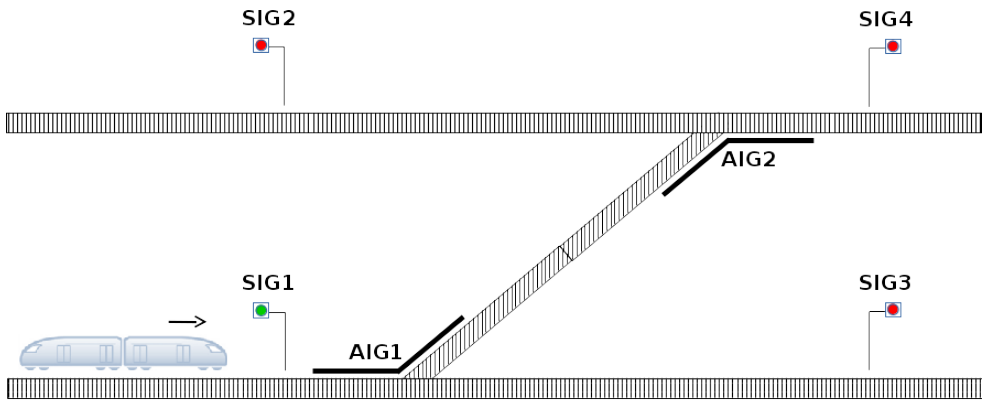


FIGURE 1.2 – Exemple d'un plan de voies.

niveau inférieur par rapport au modèle. En pratique, il ne s'agit généralement pas d'un code bas niveau (assembleur ou binaire), mais plutôt d'un code haut niveau comme le C par exemple. Ce passage du modèle au code introduit alors les mécanismes de gestion de la mémoire (*e.g.*, pointeurs, allocation mémoire) qui sont transparents lors de la modélisation.

Exemple 1.1 (Système et propriétés)**.** Dans le domaine ferroviaire, l'enclenchement est le système qui gère la signalisation et les aiguillages. Comme ce système peut générer des cas critiques, comme la collision entre deux trains ou un déraillement, il est généralement vérifié formellement. Les propriétés de sûreté décrivent alors le *bon* comportement que l'enclenchement doit vérifier afin d'éviter ces situations dangereuses, par exemple : lorsqu'un train passe sur une aiguille alors celle-ci est verrouillée, autrement dit elle ne peut pas changer de direction (*e.g.*, pas de commande de mouvement émis par le système). Cette propriété est générique, c'est-à-dire qu'elle s'applique à toute aiguille, quelque soit le plan de voies. Pour le plan de voies défini sur la figure 1.2, cette propriété générique serait alors instanciée pour chaque aiguille, *e.g.*, lorsqu'un train passe sur l'aiguille 1 (AIG1), alors AIG1 est verrouillée. D'autres exemples de **propriétés de sûreté** pourraient être :

— lorsque le signal 1 (SIG1) devient permissif (passe au vert) alors l'aiguille 1 (AIG1) est bien positionnée (vers SIG3 ou SIG4) et verrouillée ;

— lorsque SIG1 devient permissif, si AIG1 est en direction de SIG4, alors AIG2 est verrouillée en direction de SIG1 ;

— lorsque SIG1 devient permissif, si AIG1 est en direction de SIG3, alors aucun autre train n'est présent sur la voie SIG1-SIG3.

Un exemple de **propriétés de vivacité** serait : *les signaux ne restent pas toujours au rouge*, ou dit autrement : *il est toujours vrai que, dans le futur, au moins l'un des signaux passera au vert.*

1.1 Model Checking

L'objectif des méthodes de model checking est de permettre la vérification **automatique** de modèles **complexes**. La notion d'automatisme fait référence au besoin de minimiser le plus possible l'intervention d'*ingénieurs en méthodes formelles* dans le processus de vérification et donc de fournir une solution dite « push-button ». Autrement dit, en théorie, lorsque le système a été modélisé et les propriétés définies et formalisées, les algorithmes de MC qui sont généralement complets et garantis de toujours terminer, sont en mesure de déterminer la validité d'une propriété si les ressources en temps et mémoires sont illimitées. Dans la pratique, les ressources sont limitées, et donc l'intervention d'un utilisateur est très souvent requise pour les modèles complexes.

La notion de complexité d'un modèle fait quant à elle référence à la difficulté du problème. Intuitivement, un système est dit *complexe* si la vérification d'une propriété pour ce système demande d'importantes ressources (temps et mémoires). La *scalabilité* est, encore une fois intuitivement, la propension d'une technique à pouvoir vérifier des systèmes informatiques de plus en plus complexes. La complexité est souvent associée à la *taille* d'un modèle, c'est-à-dire que l'on caractérise généralement de complexe un système contenant beaucoup de variables d'état (mémoires). En effet, le nombre d'états d'un système augmente exponentiellement en fonction du nombre de variables d'état et de leur domaine. Ce phénomène, bien connu, est souvent formulé comme le problème de l'explosion combinatoire du nombre d'états d'un système.

Cependant, il est important de préciser que ce n'est pas le seul facteur qui doit être pris en considération. Par exemple, un modèle sans mémoire mais contenant un grand nombre d'entrées et/ou d'opérateurs *complexes* pour la procédure de décision utilisée (*e.g.*, multiplication pour un solveur SAT ; Biere [2016]) peut également poser des problèmes de scalabilité lors de la vérification.

Pour remédier de manière pragmatique à ce problème de scalabilité, les algorithmes de model checking ont beaucoup évolué et de nombreuses méthodes ont vu le jour. Dans la suite de cette partie (1.1), nous passons rapidement en revue les grandes lignes du MC afin de contextualiser cette thèse.

1.1.1 Model checking explicite et symbolique

Une des premières idées proposées consistait à analyser chaque état individuellement (*explicit state model checking*, Clarke *et al.* [1986]). Dans cette approche, tous les états du modèle sont vérifiés un à un, en commençant par les états initiaux, puis en parcourant chaque nœud du graphe des états atteignables. Cette méthode n'est donc généralement efficace que pour des modèles avec peu d'états atteignables.

Le premier grand succès du model checking se fit avec l'utilisation des ROBDDs (*Reduced Ordered Binary Decision Diagram*, Bryant [1986]; Knuth

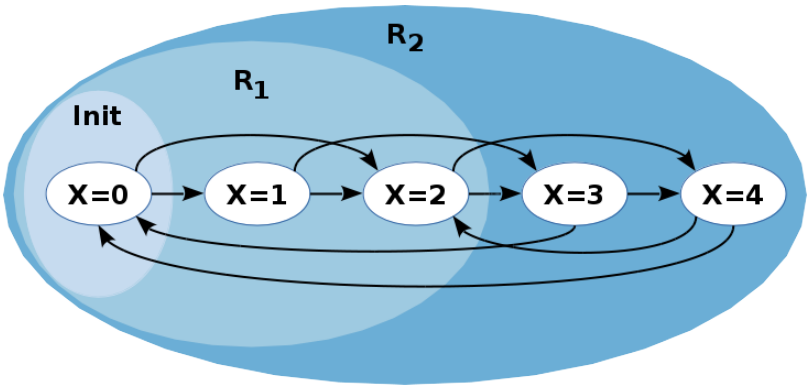


FIGURE 1.3 – Ensemble des états atteignables.

[2009]). Les ROBDDs sont des diagrammes de décisions qui encodent de façon canonique des formules propositionnelles, et pour lesquels certaines opérations usuelles (*e.g.*, conjonction logique) ont une complexité faible (*e.g.*, polynomiale en fonction du nombre de nœuds pour la conjonction). Ils sont utilisés pour encoder des fonctions caractéristiques, et permettent ainsi de représenter et de calculer efficacement des ensembles d'états. La méthode du Model Checking symbolique (*symbolic model checking*, Coudert *et al.* [1989]; Burch *et al.* [1990]) consiste à : calculer un ROBDD de l'ensemble des états atteignables du système, calculer un ROBDD correspondant à l'ensemble des états ne vérifiant pas la propriété, et vérifier si l'intersection de ces deux ROBDDs est vide. Cette méthode permet de vérifier des modèles industriels de circuits électroniques avec plusieurs centaines de variables d'état, ce qui représente potentiellement un nombre gigantesque d'états atteignables.

Exemple 1.2 (Compteur modulo 5)**.** Soit x une variable entière initialisée à 0, qui est incrémentée aléatoirement de 1 ou 2, modulo 5, à chaque pas de calcul. La figure 1.3 représente l'ensemble des états du système et les transitions possibles entre ces états. **Init** correspond à l'ensemble des états initiaux, ici le singleton $\{x = 0\}$. $\mathbf{R}_1$ représente l'union de l'état initial et des états atteignables en une transition depuis cet état initial, c'est-à-dire l'ensemble : $\{\{x = 0\}, \{x = 1\}, \{x = 2\}\}$. $\mathbf{R}_2$ représente l'union de $\mathbf{R}_1$ et les états atteignables en une transition depuis $\mathbf{R}_1$, soit l'ensemble : $\{\{x = 0\}, \{x = 1\}, \{x = 2\}, \{x = 3\}, \{x = 4\}\}$. À l'inverse du MC explicite qui va parcourir chaque nœud individuellement jusqu'à visiter l'ensemble des nœuds, le MC symbolique va construire itérativement ces ensembles d'états atteignables (ici **Init** puis $\mathbf{R}_1$ et enfin $\mathbf{R}_2$), jusqu'à atteindre un point fixe, c'est-à-dire lorsque $R_i = R_{i+1}$. Autrement dit, tous les états atteignables depuis $\mathbf{R}_i$ sont déjà contenus dans $\mathbf{R}_i$. Sur l'exemple courant, le point fixe a été atteint par $\mathbf{R}_2$: tous les états atteignables depuis $\mathbf{R}_2$ sont déjà contenus dans $\mathbf{R}_2$.

1.1.2 Model checking avec solveurs SAT

C'est à la fin des années 1990, avec l'introduction du *Bounded Model Checking* (BMC, Biere *et al.* [1999]), combiné à l'efficacité grandissante des solveurs SAT (GRASP, Silva et Sakallah [1997] et Chaff, Moskewicz *et al.* [2001]), que le model checking basé sur les solveurs SAT s'est développé et que des systèmes avec plusieurs milliers de variables ont pu être analysés. Un solveur SAT est simplement un algorithme répondant au problème de satisfaisabilité booléenne. BMC consiste à chercher s'il existe un chemin d'exécution de taille fixée qui, en partant d'un état initial peut mener à un état du système ne vérifiant pas l'exigence souhaitée. Cette méthode est en général très efficace pour trouver des défaillances. Cependant, il est soit difficile de trouver la taille minimale du chemin qui permet de conclure que la propriété est vérifiée pour l'ensemble du système, soit la taille nécessaire est trop grande pour être efficacement calculable. Alors, d'autres approches utilisant des solveurs ont été mises au point. Elles sont généralement qualifiées de « *méthodes basées sur l'induction* ». La grande majorité des méthodes qui sont usuellement employées dans l'industrie et les outils académiques, peuvent être rangées dans l'une des 3 catégories de stratégies suivantes : (i) la k-induction (Sheeran *et al.* [2000]), (ii) l'interpolation (ITP, McMillan [2003]), (iii) ou plus récemment IC3/PDR (*Incremental Construction of Inductive Clauses for Indubitable Correctness* Bradley [2011], *Property Directed Reachability* Een *et al.* [2011]). Elles sont toutes basées sur le principe de preuve par induction. Cependant, leurs fonctionnements diffèrent sur la *construction* de l'invariant inductif.

La k-induction est une extension de la preuve par induction, où l'invariant inductif est un chemin de longueur k vérifiant la propriété. ITP calcule une surapproximation des états atteignables à l'aide d'interpolants (Craig [1957]), qui sont eux-mêmes calculés à partir de preuves de non satisfaisabilité retournée par un solveur SAT, lors d'appels à BMC. Une idée proposée plus récemment par Vizel et Grumberg [2009] consiste à construire une séquence d'interpolants (bien que la séquence d'interpolants fut déjà introduite pour la vérification de logiciel dans des papiers précédents : Jhala et McMillan [2005]; McMillan [2006]). À noter que les techniques de k-induction, d'interpolation ou de séquence d'interpolants nécessitent le dépliage de la relation de transitions du système.

IC3, quant à lui, raffine une séquence d'ensembles d'états pas à pas et localement, c'est-à-dire avec des requêtes à un solveur ne contenant qu'une seule relation de transition. Plusieurs améliorations de l'algorithme original ont aussi été proposé : Hassan *et al.* [2013] (*Counterexample to generalization*), Ivrii et Gurfinkel [2015] (*Counterexample to pushing*), etc. Vizel et Gurfinkel [2014] ont également proposé une idée pour tirer parti des deux méthodes ITP (global) et IC3 (local). L'efficacité d'IC3 appliqué au model checking sur les circuits a aussi motivé son extension vers la vérification de logiciel. Par exemple,

Welp et Kuehlmann [2013] proposent une extension d'IC3 en remplaçant des cubes booléens par des cubes entiers (et des polyèdres), ainsi que l'utilisation d'un solveur SMT à la place du solveur SAT.

Vizel *et al.* [2015] ont récemment publié une étude claire et concise de toutes ces méthodes de model checking utilisant des solveurs SAT.

1.1.3 Approches complémentaires

Toutes ces stratégies de vérification ont pour objectif de permettre la preuve automatique de modèle de plus en plus complexe. Cependant, il existe également de nombreuses autres méthodes utilisées pour permettre de contrer ce problème de passage à l'échelle, et qui peuvent en général être appliquées indépendamment de l'algorithme de MC utilisé.

Abstraction

L'abstraction est une méthode très utilisée pour réduire la complexité d'un modèle. Le principe consiste simplement à ignorer des détails du modèle qui ne sont pas utiles pour vérifier une propriété donnée. Par exemple, une technique simple à mettre en place consiste à calculer le cône d'influence (COI, *Cone Of Influence*) de la propriété qui doit être vérifiée. Cela qui permet d'abstraire toutes les parties du modèle qui ne participe pas au calcul de la propriété. Le cône d'influence est généralement obtenu à partir d'un calcul de dépendance syntaxique des variables de la propriété.

Avant de voir un exemple de cône d'influence, il nous faut définir la notion de variable libre. **Une variable est libre** si, à chaque cycle, la variable peut prendre n'importe quelle valeur de son domaine.

Exemple 1.3 (Cône d'influence)**.** L'ensemble des variables définies ci-dessous sont des variables entières. Soit a, b, c, d des entrées (variables libres), et w, x, y, z des variables d'état initialisées à 0, et qui sont affectées à chaque pas de calcul comme suit :

$$\begin{aligned} w &= a + b \\ x &= b + c \\ y &= c + d \\ z &= w + x + y \end{aligned}$$

Supposons $x \leq y$ la propriété à vérifier. Lors de la vérification, il est possible, à l'aide d'une simple analyse de dépendance des variables nécessaires pour calculer x et y, de retirer du modèle les variables a, w et z (ainsi que leur définition).

Le COI est une technique syntaxique, ayant ainsi l'avantage d'être précis. En d'autres termes, si un CEX est trouvé dans le modèle abstrait alors il s'agit

également d'un CEX dans le modèle concret. Cependant, le COI ne réduit pas nécessairement le modèle comme on le voudrait, par exemple si toutes les variables sont syntaxiquement dépendantes de la propriété, comme illustré ci-dessous.

Exemple 1.4 (Limite du cône d'influence)**.** Soit a, b des entrées (variables libres), et x, y des variables d'état initialisées à 0.

$$x = a + b$$
$$y = x * y$$

Si la propriété à vérifier est $y = 0$, alors l'application du COI ne réduira pas le modèle. Or, comme 0 est l'élément absorbant de la multiplication et que y est initialisé à 0, quelque soit la valeur de x, y sera toujours égal à 0. La variable x pourrait donc être remplacée par une variable libre (tout en retirant sa définition), et la propriété serait toujours vérifiée dans le modèle modifié.

L'objectif des méthodes d'abstraction non syntaxique est donc de pallier cette limitation forte. Elles vont ainsi chercher les variables qui peuvent être remplacées par des variables libres tout en préservant la validité de la propriété dans le modèle abstrait.

CEGAR (*CounterExample-Guided Abstraction Refinement*, Clarke *et al.* [2000]) est l'une des techniques d'abstraction les plus connues. Résumé simplement, un ensemble de variables du modèle est abstrait (*i.e.*, elles sont remplacées par des variables libres) et un algorithme de vérification est exécuté sur ce modèle abstrait. Si la propriété est vérifiée dans le modèle abstrait, alors elle l'est également dans le modèle concret. Cependant, s'il existe un contre-exemple dans le modèle abstrait, celui-ci n'existe pas nécessairement dans le modèle concret. Les affectations des variables du contre-exemple sont alors rejouées sur le modèle concret. Si ce contre-exemple n'est pas valable pour le modèle concret, alors le modèle abstrait est raffiné, généralement en utilisant la preuve d'insatisfaisabilité retournée par le solveur SAT. Ce processus est réexécuté jusqu'à terminaison, c'est-à-dire jusqu'à ce que la propriété soit vérifiée dans le modèle abstrait ou qu'un contre-exemple valide soit trouvé pour le modèle concret. La principale difficulté réside donc dans le choix de l'abstraction initiale : si le modèle est *trop* abstrait, CEGAR devra raffiner le modèle de nombreuses fois.

PBA (*Proof-Based Abstraction*, McMillan et Amla [2003]) est une méthode qui utilise les preuves d'insatisfaisabilité retournées par le solveur SAT lors d'appels BMC pour construire le modèle abstrait. Ainsi, ce modèle abstrait possède la propriété de ne pas contenir de contre-exemple d'une longueur inférieure à la profondeur utilisé dans l'appel BMC.

Invariants

La génération d'invariants est une autre approche ayant pour but de permettre le passage à l'échelle des algorithmes de vérification. Un invariant est un prédicat logique qui caractérise une partie du modèle que l'on cherche à simplifier en raison de sa complexité. Ces invariants permettent de réduire l'espace de recherche. Ils sont généralement ajoutés directement au modèle. Ils peuvent également être utilisés comme méthode d'abstraction pour remplacer des parties complexes du modèle. Dans ce contexte, ils *imitent* alors le comportement des parties souhaitées du modèle, qui peuvent ainsi être supprimées. Un invariant peut être complet, c'est-à-dire qu'il représente exactement le même comportement que la partie du modèle qu'il remplace. Il peut aussi être partiel et sur-approximer (abstraire) le comportement original, afin de vérifier la propriété sans ajouter d'information inutile.

Exemple 1.5 (Invariants)**.**

$$\left.\begin{array}{l} x = y \\ \textbf{for } i = 0 \textbf{ to } y \ : \\ \quad x = x + 2 \end{array}\right\} \begin{array}{l} \Leftrightarrow y \leq x \leq 3y \\ \Leftrightarrow x = 3y \end{array}$$

Supposons que la partie à gauche de l'accolade ci-dessus soit contenue dans un modèle que l'on souhaite vérifier. Ce morceau de pseudo-code contient une boucle **for** dont le nombre d'itérations est dépendant de la variable y. Cela nécessite donc la génération d'un grand nombre de contraintes lors de l'encodage dans un langage de plus bas niveau. Or, fonctionnellement, cette partie pourrait être remplacée par les deux invariants donnés à droite de l'accolade, qui fournissent généralement un encodage beaucoup plus simple à résoudre pour les procédures de décisions. Le premier invariant : $y \leq x \leq 3y$ est une sur-approximation du comportement de la partie gauche. Il ne peut donc pas toujours être utilisé s'il est nécessaire d'avoir la valeur exacte de x par rapport à y pour vérifier la propriété.

Plusieurs approches pour trouver automatiquement des invariants ont été proposées, par exemple (i) l'interprétation abstraite introduite par Cousot et Cousot [1977], (ii) la découverte dynamique par Ernst *et al.* [2001], (iii) la recherche d'invariant basée sur template, *e.g.*, des invariants linéaires par Colón *et al.* [2003]; Gupta et Rybalchenko [2009], (iv) le raffinement itératif à l'aide d'IC3 Welp [2013] (voir également le chapitre 4). Néanmoins, cela reste un problème difficile, si bien que lorsque des invariants complexes sont nécessaires, cette étape est généralement confiée à un humain.

Lemmes

Les lemmes sont également des invariants mais nous avons choisi de les distinguer car il y a une différence primordiale sur l'objectif recherché et l'utilisa-

tion qui en est faite. La génération automatique de lemmes est donc également une méthode où des formules caractérisant des comportements du système sont recherchés. Cependant, ces formules n'ont pas pour objectif premier de réduire l'espace de recherche ou de remplacer une partie du système. Elles ont pour but de renforcer l'hypothèse d'induction afin de rendre la propriété à vérifier inductive. IC3, par exemple, génère de nombreux lemmes afin de vérifier la propriété mais ceux-ci ne sont pas forcément des invariants pour l'ensemble du modèle, ils peuvent être locaux (temporellement). Champion *et al.* [2013] proposent par exemple une génération de lemmes automatiques à partir des contre-exemples d'induction calculés en utilisant la stratégie de k-induction.

1.1.4 Procédure de décision

Comme l'utilisation des solveurs SAT/SMT est directement au cœur de la majorité des model checkers, améliorer ou spécialiser (pour le model checking) les algorithmes de ces procédures de décision implique directement de meilleurs performances pour les model checkers.

Encodage

Les circuits électroniques sont en général composés uniquement de variables booléennes et d'opérateurs logiques. Par conséquent, lorsque les modèles traités requièrent un espace mémoire *trop important* lors de la construction des ensembles d'états en utilisant les ROBDDs, le solveur SAT est la procédure de décision la mieux adaptée à la résolution des formules générées pour vérifier ces circuits. Néanmoins, la plupart des solveurs SAT n'acceptent en entrée que des formules en forme normale conjonctive (CNF, *Conjunctive Normal Form*). Il est donc nécessaire d'effectuer une étape de transformation du circuit en CNF. Il existe plusieurs approches pour transformer une formule propositionnelle quelconque en CNF. Les deux plus connus sont : (i) L'encodage direct, qui consiste à réécrire la formule en CNF à l'aide des lois de De Morgan, de distributivité, etc. Néanmoins, cette transformation peut, dans le pire des cas, engendrer une nouvelle formule de taille exponentielle par rapport à l'originale. (ii) La transformation de Tseitin [1968] crée une formule CNF dont la taille est linéaire en fonction de l'originale. Cela est rendu possible par l'ajout de nouvelles variables. La formule générée n'est plus équivalente mais équisatisfaisable avec la formule originale. Il existe de nombreuses autres techniques pour l'encodage, comme la méthode de Plaisted et Greenbaum [1986], ou encore des techniques plus spécifique aux circuits comme celles utilisées par Een *et al.* [2007] ou Manolios et Vroon [2007]. Plus de détails sont fournis dans la partie 3.2.2.

Même si ces techniques sont efficaces dans le cas propositionnel, les programmes utilisent généralement des entiers, qui sont représentés par des vec-

teurs de bits, et donc des opérations arithmétiques ou relationnelles sur les vecteurs de bits, *e.g.*, l'addition ou la multiplication. Il est donc nécessaire d'avoir une procédure de décision pour ce genre de problèmes, à savoir, une procédure de décision pour la théorie des vecteurs de bits. Une solution repose sur l'encodage de ces vecteurs de bits et opérations en formule propositionnelle, communément appelé « *bit-blasting* » ou « *flattening* ». La formule propositionnelle ainsi générée est ensuite vérifiée à l'aide d'un solveur SAT. Toutefois, comme évoqué précédemment les équivalents booléens de certaines opérations comme la multiplication peuvent poser problème lors de la résolution avec un solveur SAT (Biere [2016]). Aussi, même si les relations sémantiques entre les variables demeurent, il est plus difficile de retrouver la structure du problème haut niveau après cette transformation (*bit-blasting*). C'est pourquoi l'utilisation de solveurs SMT est, dans ce contexte, une alternative intéressante aux solveurs SAT.

Les solveurs SMT (*Satisfiability Modulo Theories*) sont des procédures de décision qui acceptent différentes théories en entrée (*e.g.*, arithmétiques entières et réelles, manipulation de tableaux), et sont capables de déterminer la satisfaisabilité de combinaisons de théories (Nelson et Oppen [1979]; Shostak [1984]; Tinelli [2002]; Conchon *et al.* [2008]). L'idée générale est d'utiliser des algorithmes de décision spécialisés pour chaque théorie et de les combiner. L'approche la plus utilisée consiste à transformer le problème en une formule propositionnelle en CNF en (i) purifiant la formule, *i.e.*, chaque atome n'appartient plus qu'à une seule théorie, (ii) remplaçant les atomes de chaque théorie par une nouvelle variable booléenne. Cette formule est donc une abstraction du problème initial, sa satisfaisabilité n'implique pas la satisfaisabilité du problème. Ainsi, lorsqu'un solveur SAT retourne une affectation possible des variables, les procédures de décision respectives à chaque théorie vérifient si cette affectation est cohérente. Si c'est le cas, alors le problème est satisfaisable. Sinon, un lemme, par exemple la négation de l'affectation courante, est ajouté à la formule abstraite et le processus est répété. Pour plus de détails sur les solveurs SMT, voir par exemple Kroening et Strichman [2008]; Barrett *et al.* [2009]; Monniaux [2016].

La théorie qui nous concerne plus particulièrement, c'est-à-dire celle que nous utilisons pour représenter les modèles dans notre contexte est la théorie des vecteurs de bits de taille fixe sans quantificateur QF_BV (*Quantifier-Free logic over fixed-size Bit-Vectors*) de la librairie SMT Barrett *et al.* [2010]. Pour la résolution des problèmes de décision de cette théorie, la technique qui est la plus couramment utilisée (*e.g.*, dans `Boolector` Niemetz *et al.* [2015], `CVC4` Barrett *et al.* [2011] ou `MathSAT` Cimatti *et al.* [2013]) et qui semble toujours être la plus efficace en pratique aujourd'hui, consiste à appliquer de nombreuses règles de réécriture pour simplifier le problème avant d'effectuer le *bit-blasting* de l'ensemble de la formule, puis d'utiliser un solveur SAT. À noter néanmoins que dans les dernières versions de `Boolector`, vainqueurs

notamment des dernières compétitions SMT dans la catégorie QF_BV, ils combinent l'approche décrite ci-dessus avec une approche de recherche locale appliquée au niveau de la théorie, *i.e.*, avant *bit-blasting* (Niemetz *et al.* [2016]).

Optimisations

Quel que soit l'origine du problème, circuit ou logiciel, effectuer une étape de prétraitement est devenu primordial lors de la vérification de modèles complexes. Cette étape vise à simplifier ou réécrire la formule originale en une formule équivalente (ou équisatisfaisable dans certain cas). L'objectif est de construire une nouvelle formule qui sera plus *simple*[1] et donc, dans la plupart des cas, plus *facile*[2] à résoudre par la procédure de décision. Cependant, les ressources nécessaires pour construire puis résoudre cette formule simplifiée ne doivent pas excéder les ressources nécessaires pour résoudre la formule originale. Comme dans la plupart des processus de prétraitement, un compromis entre la réduction du problème initial et le temps passé pour le simplifier doit être trouvé.

Dans un processus classique de vérification de logiciels basé sur le modèle, il y a en général 3 niveaux de représentation interne : le modèle haut niveau après simplification des structures complexes (boucles, fonctions, etc.), le modèle sous forme de circuit après *bit-blast*, et enfin la formule CNF. Les simplifications sont donc principalement effectuées à ces 3 niveaux distincts. Nous donnons plus de détails dans l'exemple de la partie 1.3. En général, la première étape de simplification est la normalisation, qui consiste à obtenir une représentation strictement identique mais avec moins d'opérateurs, afin de ne pas écrire des règles de simplification redondantes. Par exemple, chaque opérateur binaire « $-$ » est remplacé par un « $+$ », ou chaque « $\wedge$ » par un « $\vee$ ».

Exemple 1.6 (Normalisation)**.** Pour éliminer les opérateurs binaires « $-$ », une nouvelle variable et sa définition sont introduites :

$$\begin{array}{lcl} & & t := -y \\ z := x - y & \rightarrow & z := x + t \end{array}$$

Pour éliminer un « $\wedge$ » dans une définition, la partie droite de la définition est remplacé par sa négation, puis toutes les occurrences de la partie gauche sont remplacées par sa négation :

$$\begin{array}{lcl} c := a \wedge b & & c := \neg a \vee \neg b \\ d := c \vee e & \rightarrow & d := \neg c \vee e \end{array}$$

Ensuite, en fonction du niveau de la représentation, différentes simplifications sont effectuées. Par exemple, à haut niveau, les optimisations sont plus

1. Nombre de variables, nombre de définitions ou nombre de clauses réduits.
2. Que ce soit en temps d'exécution et/ou en ressource mémoire nécessaire.

orientées vers les simplifications arithmétiques, *e.g.*, détection d'équivalence comme : $a \times (b + c) \Leftrightarrow ab + ac$. Alors qu'au niveau du circuit, les optimisations sont plus focalisées sur les simplifications logiques, *e.g.*, contradiction : $a \wedge \neg a \Leftrightarrow \bot$. On notera que certaines simplifications, comme la propagation de constantes ou le calcul du cône d'influence, peuvent être effectuées à plusieurs niveaux. Davantage de détails sur l'optimisation sont fournis dans les parties 3.2.1 et 5.2.1.

Résolution

Après les étapes d'encodage et d'optimisation, il faut généralement déterminer la satisfaisabilité de la formule[3]. Les solveurs SAT sont des algorithmes/outils qui répondent au problème SAT et la plupart sont en mesure de fournir un modèle lorsque la formule est satisfaisable ou une preuve lorsque la formule est insatisfaisable. Leurs performances ont beaucoup évolué depuis les premières tentatives de Davis et Putnam [1960]; Davis *et al.* [1962], avec notamment l'apprentissage de clauses et le retour en arrière non-chronologique par Silva et Sakallah [1997]; Moskewicz *et al.* [2001], ainsi qu'un ensemble de techniques comme les heuristiques de sélection de variables, les politiques de redémarrage, la gestion paresseuse des clauses (*two-watched literals*), etc. Nous décrivons plus en détails l'ensemble de ces techniques dans la partie 2.1. De surcroît, les solveurs sont toujours en train d'évoluer et sont de plus en plus performant comme le montre la figure 1.4. Sur cette figure, chaque courbe représente la capacité d'un solveur SAT à résoudre un certain nombre de problèmes parmi les 300 problèmes de la compétition SAT 2011, dans un temps donné. Ces courbes portent le nom de « Cactus Plot » et sont communément utilisés lors des compétitions pour représenter la performances de ces solveurs. Les performances accrues de ces solveurs permettent bien évidemment de vérifier des modèles de plus en plus complexes. Cependant, on notera ici que ces solveurs ne sont généralement pas spécialisés pour un type de problèmes plutôt qu'un autre. Le même solveur sera utilisé, comme boîte noire, sur des problèmes de cryptographies comme sur des problèmes de model checking. Même si leur aspect boîte noire est aussi l'une de leurs forces, l'une des idées défendue dans ce manuscrit est justement de permettre aux solveurs SAT d'avoir une meilleur connaissance du problème, ou de pouvoir communiquer avec une représentation plus haut niveau d'un même problème.

3. Parfois, l'étape de simplification seule permet de conclure.

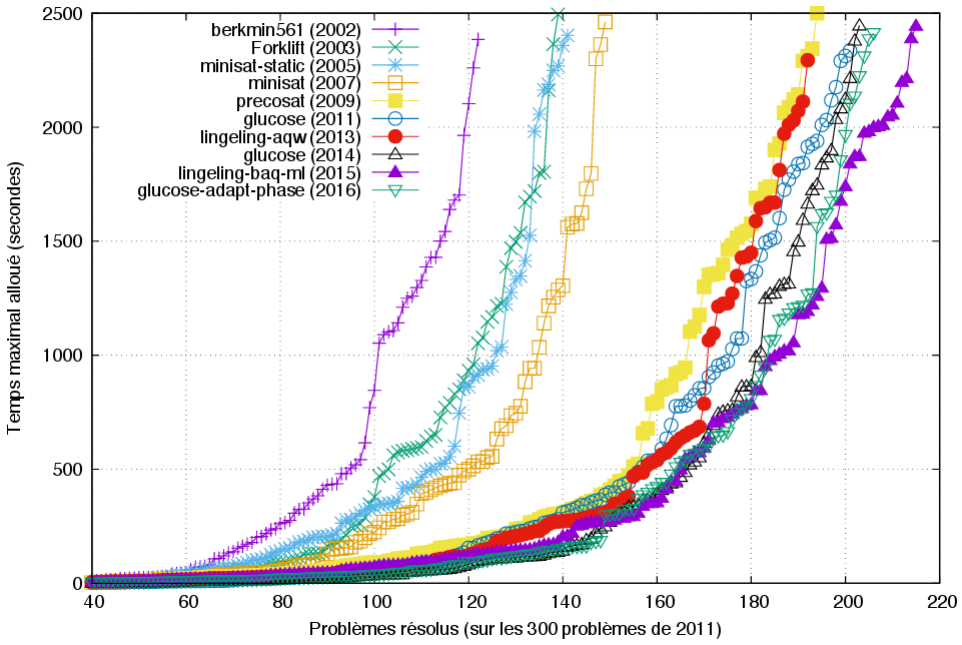


FIGURE 1.4 – Évolution du nombre de problèmes résolus pour différents solveurs sur l'ensemble des problèmes de la compétition SAT de 2011.

1.2 Contexte

1.2.1 Environnement

Cette thèse s'est déroulée dans l'entreprise `SafeRiver` qui est spécialisée dans les problèmes de sûreté de fonctionnement et de sécurité, et plus précisément dans la vérification formelle. Une partie importante de l'activité de l'entreprise est la vérification de systèmes ferroviaires (grande ligne et métro) à l'aide de MC. Dans ce but, l'entreprise développe un outil de vérification de modèles à l'aide d'algorithmes de Model Checking basé sur solveurs SAT/SMT, aussi appelé *Model Checker*. La figure 1.5 décrit succinctement les grandes étapes usuelles d'un model checker basé sur solveur SAT/SMT, tout en mettant en avant les différences en fonction du type de solveurs utilisé. Cet outil prend en entrée des modèles définis dans des langages *hauts niveaux* comme `SCADE` (Dormoy [2008]) ou `Simulink` (Dabney et Harman [2004]) par exemple. Ces langages encodent les entiers comme des vecteurs de bits et autorisent des constructions non triviales comme les boucles, les fonctions (récursives) ou encore les tableaux. Cependant, il y a tout de même certaines restrictions qui garantissent le *caractère fini* du modèle[4] : pas de notion de pointeur, les

4. Autrement dit, ces restrictions nous assurent que le modèle est exprimable en logique du premier ordre sans quantificateur.

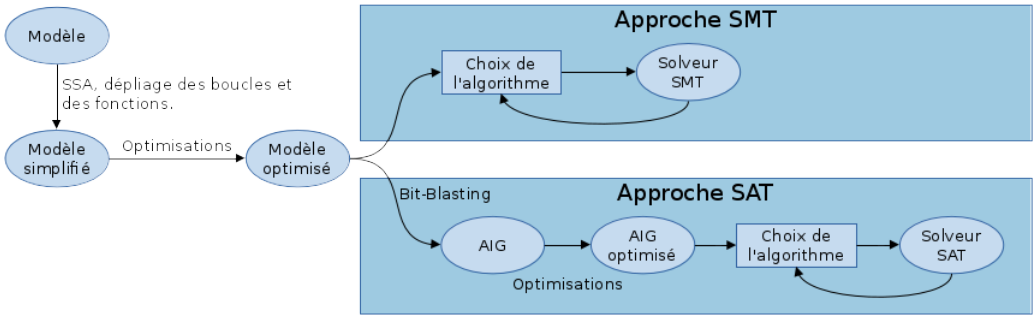


FIGURE 1.5 – Aperçu des étapes internes d'un Model Checker utilisant un solveur SAT ou SMT.

boucles ont un nombre d'itérations bornées, il est possible de vérifier statiquement la terminaison des fonctions récursives, etc.

La première étape consiste à « mettre à plat » l'ensemble de ces structures complexes : SSA (*Single Static Assignment*, Cytron *et al.* [1991]), dépliage des boucles, etc. Après cette étape, la représentation interne du modèle est similaire aux formules exprimables dans la théorie QF_BV. Une première série d'optimisations est alors effectuée sur cette représentation. Ces optimisations visent à simplifier le modèle, *i.e.*, réduire le nombre de variables, d'opérateurs, etc. À noter que certaines optimisations sont parfois effectuées avant même l'étape de mise à plat. Ensuite, en fonction du choix du solveur, *i.e.*, SAT ou SMT, les approches divergent : Pour le SMT, il suffit d'encoder le modèle en fonction de l'algorithme de MC utilisé, alors qu'avec le solveur SAT, il est nécessaire d'effectuer l'étape de *bit-blasting*, *i.e.*, transformer les variables et les opérateurs de vecteurs de bits en variables et opérateurs booléens. L'ensemble de ces étapes est illustré dans la partie 1.3 ci-dessous.

Les étapes de mise à plat et d'optimisation du modèle haut niveau ayant déjà été réalisées dans l'outil développé par l'entreprise, l'objectif global de cette thèse était alors de participer à l'amélioration de l'outil à partir du modèle optimisé. La partie 1.1 montre l'étendue des possibilités d'amélioration. Le premier choix que nous devions faire est de choisir entre l'approche SMT ou SAT. Cependant, comme évoqué précédemment l'approche qui est globalement la plus efficace pour résoudre QF_BV à l'aide d'un SMT consiste à effectuer le *bit-blast*, puis à utiliser un solveur SAT. Cela revient donc à utiliser un solveur SAT quelque soit l'approche à ceci près qu'en utilisant l'interface d'un SMT, le contrôle de l'optimisation de la représentation après *bit-blast* et du solveur SAT est perdu. Nous avons alors opté pour l'approche à l'aide d'un solveur SAT.

Les algorithmes de MC majoritairement utilisés dans l'entreprise ou plus généralement dans l'industrie ferroviaire sont BMC et la k-induction. La méthodologie habituellement utilisée consiste à d'abord exécuter BMC seul jusqu'à

un seuil fixé, afin de s'assurer qu'il n'existe pas de contre-exemples. Puis, l'algorithme d'induction temporelle, *i.e.*, la combinaison de BMC et de k-induction, est exécuté afin de vérifier la propriété. Comme la plupart des modèles à vérifier sont complexes, le temps de calcul nécessaire pour la vérification est d'ordinaire assez long (plusieurs heures). C'est pourquoi BMC est lancé seul d'abord, car cela permet en général de détecter rapidement certaines erreurs de conception qui peuvent être introduites malencontreusement lors du processus de vérification. Enfin, si l'induction temporelle ne permet pas de vérifier la propriété dans un temps raisonnable, des lemmes sont ajoutés manuellement par les utilisateurs afin d'accélérer le processus. Nous avons donc focalisé nos recherches d'amélioration sur ces deux algorithmes, *i.e.*, BMC et k-induction.

1.2.2 Contributions

Comme évoqué précédemment, l'objectif principal que nous avons poursuivi sur l'ensemble des travaux effectués au cours de cette thèse est de renforcer la collaboration entre le solveur SAT et le model checker. Nous avons tenté d'accroître l'interconnexion de ces deux modules en exploitant la structure haut niveau du problème. Nous avons alors défini des méthodes qui profitent de la structure symétrique des formules, provenant des dépliages successifs de la relation de transition. Nous avons également proposé de spécialiser notre solveur SAT en lui fournissant des informations additionnelles sur les variables. En outre, nous avons aussi instauré un dialogue entre bas niveau et haut niveau : des informations déduites par le solveur sont extraites et importées dans le model checker pour être exploitées lors de nouvelles simplifications haut niveau.

Plus précisément, les travaux effectués au cours de cette thèse sont les suivants :

— Dans un premier temps, nous avons proposé une nouvelle méthode d'élimination de variables à haut niveau permettant pour la suite de conserver une relation d'équivalence entre la représentation interne du model checker et celle du solveur.

— Cette relation d'équivalence nous permet de définir un dialogue simple entre le solveur et le model checker, car pour chaque variable présente dans l'une ou l'autre des représentations internes, il existe son équivalent dans l'autre. Ainsi, nous avons proposé d'exploiter les clauses apprises au niveau du model checker en important les déductions simples du solveur, à savoir, les variables qui sont des constantes et les équivalences entre deux variables.

— Ensuite, nous avons retravaillé la duplication de clauses pour l'induction temporelle à l'aide des littéraux d'activation, tout en limitant la duplication au niveau du model checker, ce qui nous fournit une approche ne nécessitant aucune modification de l'algorithme original et très simple à

mettre en œuvre. En outre, nous proposons des stratégies de sélection des clauses à dupliquer afin de ne pas surcharger le solveur avec un nombre trop important de clauses.

— Nous avons effectué une comparaison expérimentale pour mesurer l'impact du sens de dépliage de la relation de transition pour l'induction temporelle, vers l'avant ou vers l'arrière. Puis, nous avons proposé une méthode de dépliage hybride, qui cherche à profiter des avantages des deux sens de dépliage.

— Nous avons réalisé une étude pour observer s'il existe un lien entre la profondeur de dépliage des variables et la structure en communautés des formules CNF, mais également entre la profondeur et les clauses apprises (et notamment le score LBD (*Literal Block Distance*), un moyen d'estimer l'intérêt des clauses apprises dans les solveurs SAT), avec pour objectif de spécialiser notre solveur pour BMC.

— Pour finir, nous avons implémenté un model checker utilisant le solveur SAT `Glucose`, dans lequel nous avons pu expérimenter les propositions d'amélioration ci-dessus, afin de tester leur efficacité.

1.2.3 Cadre précis

Dans ce document, nous ne considérerons que le model checking de systèmes de transitions à état fini. Les variables des modèles ne seront que des variables booléennes. Les propriétés seront toujours des propriétés de sûreté simple, c'est-à-dire des propriétés sous forme d'assertions. Ces restrictions sont peu contraignantes dans le sens où la plupart des langages formels décrivant des systèmes logiciels ou matériels peuvent être vérifiés via ce formalisme, ainsi que tout type de propriétés. En effet, Biere *et al.* [2002] propose un moyen de transformer les propriétés de vivacité en propriété de sûreté et Kupferman et Vardi [2001] présente une réduction de toutes propriétés de sûreté en propriétés assertives.

Néanmoins, certaines configurations ne rentrent pas dans le cadre de cette thèse, par exemple (i) les modèles Lustre qui utilisent des entiers naturels (ii) le code C, contenant des pointeurs ou des boucles non bornées.

1.3 Exemple complet

De manière à bien clarifier l'ensemble des étapes évoquées précédemment, nous présentons maintenant un exemple complet du processus de vérification. Nous passons en revue les différentes étapes de normalisation et d'optimisation, la mise à plat, le *bit-blasting*, et les appels au solveurs pour BMC. Cela permettra de bien clarifier l'enchaînement de ces étapes, en partant du modèle

haut niveau jusqu'à l'appel au solveur SAT. L'ordre présenté ci-dessous ne reflète pas nécessairement ce qui est fait dans la pratique mais permet d'illustrer plus didactiquement chaque étape.

Modèle haut niveau

Voici un exemple d'un modèle très simple défini dans un langage haut niveau avec une sémantique usuelle. Soit x, y deux variables d'état (mémoires) de type entier qui sont initialisées à $x = 3$, $y = 2$. À noter qu'il s'agit d'entiers représentés par des vecteurs de bits. Leur taille importe peu pour l'instant, nous la définirons par la suite lorsque nous effectuerons le *bit-blasting*. À chaque pas de calcul et en fonction de leur valeur au cycle précédent, x et y prennent les valeurs suivantes :

if $x \neq 0$ **then**
    $x = x - 1$
else
    $x = 3$
if $y \neq 0$ **then**
    $y = y - 1$
else
    $y = 2$
Propriété : $x \neq y$

Dit simplement, il s'agit de décrémenter x et y, modulo 4 et 3 respectivement. En simulant ce modèle, voici les premières valeurs de x et y en fonction du cycle de calcul i :

$\boldsymbol{i}$	0	1	2	3	4	5	6	...
$\boldsymbol{x}$	3	2	1	0	3	2	1	...
$\boldsymbol{y}$	2	1	0	2	1	0	2	...

Modèle normalisé

L'étape suivante consiste à transformer le modèle d'entrée, qui est généralement écrit par un utilisateur [5]. Ce modèle d'entrée peut contenir de nombreux opérateurs, fonctions, structures complexes visant à faciliter la conception du point de vue utilisateur. Néanmoins, ces constructions complexifient la représentation interne du modèle en pratique. Cette étape consiste alors à : « mettre à plat » les différents modules/classes, « *inliner* » les fonctions, déplier les

5. Le modèle peut également être généré depuis une interface graphique par exemple.

boucles, et transformer l'ensemble du modèle en une liste d'affectations (de définitions) tel que chaque variable n'est affecté qu'une seule fois (SSA). Dans notre contexte, la représentation en SSA permet notamment de simuler les affectations séquentielles et/ou conditionnelles pour une même variable en logique propositionnelle. La représentation SSA du modèle donné en exemple est la suivante :

$$
\begin{aligned}
&x_1 = x_0 - 1 \\
&x_2 = 3 \\
&x_3 = x_0 \neq 0 \ ?\ x_1 \ : x_2 \\
&y_1 = y_0 - 1 \\
&y_2 = 2 \\
&y_3 = y_0 \neq 0 \ ?\ y_1 \ : y_2
\end{aligned}
$$

Normalisation et premier niveau d'optimisation

L'objectif de cette étape est de normaliser et de simplifier les définitions du modèle pour obtenir une représentation plus uniforme, permettant la factorisation des règles d'optimisation et de transformation. Par exemple, quelques règles de normalisation usuellement appliquées sont : éliminer certain opérateur, limiter le nombre d'opérandes par définition en fonction de l'opérateur, restreindre les opérandes à des constantes ou à des variables, etc. Ainsi, par exemple, la définition $x := a+b+c$ serait transformée en 2 définitions $t := a+b$ et $x := t + c$. Généralement, après cette étape de normalisation, une première passe d'optimisation est effectuée : propagation de constantes, simplification arithmétique, minimisation des ITEs (*If Then Else*), etc. Voici l'application de cette étape sur notre exemple :

$$
\begin{aligned}
&x_1 = x_0 + (-1) \\
&c_x = x_0 \neq 0 \\
&x_3 = c_x \ ?\ x_1 \ : 3 \\
&y_1 = y_0 + (-1) \\
&c_y = y_0 \neq 0 \\
&y_3 = c_y \ ?\ y_1 \ : 2
\end{aligned}
$$

Bit-blasting

Le but de cette étape est de transformer le modèle, variables et opérateurs, afin de convertir toutes les variables en variables booléennes pour préparer

l'encodage en logique propositionnelle. Dans les langages formels que nous traitons, les variables sont toujours définies avec une taille, soit sous forme d'intervalle soit en définissant la taille du vecteur de bits. Ainsi, il est toujours possible de déterminer le nombre de bits nécessaire pour représenter un entier. Il existe ensuite différents encodages. Par exemple, pour une variable x définie dans l'intervalle $[-2, 3]$, une technique simple consiste à générer une variable par valeur $x_{\text{-}2}$, $x_{\text{-}1}$, x_0, x_1, x_2, x_3 et tel que la variable x_i est vraie quand $x = i$. Il faut donc généralement ajouter un ensemble de contraintes spécifiant qu'au moins et qu'au plus une variable de cet ensemble soit vraie pour que la variable x ne puisse pas être égale à plusieurs valeurs en même temps. Néanmoins, cet encodage n'est pas toujours adapté pour les domaines de valeurs importants.

L'encodage le plus commun, et celui que nous utilisons également, est l'encodage binaire avec la méthode du complément à deux, à savoir, chaque variable est représenté par un vecteur de bits et tel qu'un nombre négatif est obtenu en prenant la négation de ce nombre à laquelle 1 est ajouté. Dans la suite, nous utiliserons la notation $x_{[n]}$ pour indiquer que la variable ou la valeur x est encodée sur n bits. Par exemple, une variable $x_{[3]}$ est composé de 3 bits : x_2, x_1 et x_0 avec x_2 le bit de poids fort (MSB, *Most Significant Bit*) et x_0 le bit de poids faible (LSB, *Least Significant Bit*). L'ensemble des valeurs possibles de $x_{[3]}$ (une variable signée) associés aux valeurs correspondantes de ses bits sont donnés dans le tableau suivant :

$x_{[3]}$	x_2	x_1	x_0
$3_{[3]}$	0	1	1
$2_{[3]}$	0	1	0
$1_{[3]}$	0	0	1
$0_{[3]}$	0	0	0
$-1_{[3]}$	1	1	1
$-2_{[3]}$	1	1	0
$-3_{[3]}$	1	0	1
$-4_{[3]}$	1	0	0

Une fois que tous les bits de toutes variables sont bien définis, il faut ensuite encoder chaque opérateur sur les vecteurs de bits en logique propositionnelle. Il s'agit simplement de traduire la sémantique de l'opérateur en utilisant les bits des variables et des opérateurs logiques. Il existe également différents encodages, mais comme c'est un processus assez fastidieux nous ne donnons qu'un exemple pour l'addition de deux variables $x_{[2]}$ et $y_{[2]}$ et où le résultat est affecté

à $z_{[2]}$:

$$z_{[2]} = x_{[2]} + y_{[2]} \rightarrow \begin{cases} x_{[2]} \rightarrow x_1\ x_0 \\ y_{[2]} \rightarrow y_1\ y_0 \\ z_{[2]} \rightarrow z_1\ z_0 \\ z_0 = x_0 \oplus y_0 \\ c_0 = x_0 \wedge y_0 \\ z_1 = x_1 \oplus y_1 \oplus c_0 \end{cases}$$

c_0 correspond à la retenue (*carry*). $\oplus$ correspond à l'opérateur « OU » exclusif (XOR). À noter que, généralement, avant d'effectuer une opération entre deux variables, celles-ci sont transformées (étendues ou tronquées) pour être du même type, *i.e.*, avoir le même nombre de bits, si ce n'est pas le cas. Notons également qu'en fonction de l'opération, le résultat peut dépasser (*overflow* ou *underflow*) la capacité de représentation de la variable dans laquelle le résultat est affecté. Pour notre exemple, nous considérons toutes les variables comme des entiers signés encodés sur 3 bits, voici alors le résultat après l'étape de *bit-blast* :

$$\begin{aligned}
x_{0[3]} &\rightarrow x_{02}\ x_{01}\ x_{00} \\
x_{1[3]} &\rightarrow x_{12}\ x_{11}\ x_{10} \\
x_{3[3]} &\rightarrow x_{32}\ x_{31}\ x_{30} \\
y_{0[3]} &\rightarrow y_{02}\ y_{01}\ y_{00} \\
y_{1[3]} &\rightarrow y_{12}\ y_{11}\ y_{10} \\
y_{3[3]} &\rightarrow y_{32}\ y_{31}\ y_{30} \\
(-1)_{[3]} &\rightarrow 1\ 1\ 1 \\
3_{[3]} &\rightarrow 0\ 1\ 1 \\
2_{[3]} &\rightarrow 0\ 1\ 0
\end{aligned}$$

$$x_1 = x_0 + (-1) \rightarrow \begin{cases} x_{10} = x_{00} \oplus 1 \\ x_{11} = x_{01} \oplus 1 \oplus (x_{00} \wedge 1) \\ x_{12} = x_{02} \oplus 1 \oplus ((x_{01} \wedge 1) \vee (x_{00} \wedge 1 \wedge (x_{01} \vee 1))) \end{cases}$$

$$c_x = x_0 \neq 0 \rightarrow c_x = x_{02} \vee x_{01} \vee x_{00}$$

$$x_3 = c_x\ ?\ x_1\ :3 \rightarrow \begin{cases} x_{30} = (\neg c_x \vee x_{10}) \wedge (c_x \vee 1) \\ x_{31} = (\neg c_x \vee x_{11}) \wedge (c_x \vee 1) \\ x_{32} = (\neg c_x \vee x_{12}) \wedge (c_x \vee 0) \end{cases}$$

$$
\begin{array}{lcl}
 & & y_{10} = y_{00} \oplus 1 \\
y_1 = y_0 + (-1) & \rightarrow & y_{11} = y_{01} \oplus 1 \oplus (y_{00} \wedge 1) \\
 & & y_{12} = y_{02} \oplus 1 \oplus ((y_{01} \wedge 1) \vee (y_{00} \wedge 1 \wedge (y_{01} \vee 1)))
\end{array}
$$

$$
c_y = y_0 \neq 0 \quad \rightarrow \quad c_y = y_{02} \vee y_{01} \vee y_{00}
$$

$$
\begin{array}{lcl}
 & & y_{30} = (\neg c_y \vee y_{10}) \wedge (c_y \vee 0) \\
y_3 = c_y \ ? \ y_1 \ : 2 & \rightarrow & y_{31} = (\neg c_y \vee y_{11}) \wedge (c_y \vee 1) \\
 & & y_{32} = (\neg c_y \vee y_{12}) \wedge (c_y \vee 0)
\end{array}
$$

Normalisation et second niveau d'optimisation

Généralement, après l'étape de *bit-blast*, le modèle est d'abord normalisé à nouveau, afin d'effectuer des optimisations. Sur notre exemple, afin de rester concis, nous n'appliquons pas la normalisation et effectuons seulement la propagation de constantes :

$$
\begin{array}{lcl}
x_{10} = x_{00} \oplus 1 & \rightarrow & x_{10} = \neg x_{00} \\
x_{11} = x_{01} \oplus 1 \oplus (x_{00} \wedge 1) & \rightarrow & x_{11} = \neg x_{01} \oplus x_{00} \\
x_{12} = x_{02} \oplus 1 \oplus ((x_{01} \wedge 1) \vee (x_{00} \wedge 1 \wedge (x_{01} \vee 1))) & \rightarrow & x_{12} = \neg x_{02} \oplus (x_{00} \vee x_{01})
\end{array}
$$

$$
c_x = x_{02} \vee x_{01} \vee x_{00} \quad \rightarrow \quad c_x = x_{02} \vee x_{01} \vee x_{00}
$$

$$
\begin{array}{lcl}
x_{30} = (\neg c_x \vee x_{10}) \wedge (c_x \vee 1) & \rightarrow & x_{30} = \neg c_x \vee x_{10} \\
x_{31} = (\neg c_x \vee x_{11}) \wedge (c_x \vee 1) & \rightarrow & x_{31} = \neg c_x \vee x_{11} \\
x_{32} = (\neg c_x \vee x_{12}) \wedge (c_x \vee 0) & \rightarrow & x_{32} = (\neg c_x \vee x_{12}) \wedge c_x
\end{array}
$$

$$
\begin{array}{lcl}
y_{10} = y_{00} \oplus 1 & \rightarrow & y_{10} = \neg y_{00} \\
y_{11} = y_{01} \oplus 1 \oplus (y_{00} \wedge 1) & \rightarrow & y_{11} = \neg y_{01} \oplus y_{00} \\
y_{12} = y_{02} \oplus 1 \oplus ((y_{01} \wedge 1) \vee (y_{00} \wedge 1 \wedge (y_{01} \vee 1))) & \rightarrow & y_{12} = \neg y_{02} \oplus (y_{00} \vee y_{01})
\end{array}
$$

$$
c_y = y_{02} \vee y_{01} \vee y_{00} \quad \rightarrow \quad c_y = y_{02} \vee y_{01} \vee y_{00}
$$

$$
\begin{array}{lcl}
y_{30} = (\neg c_y \vee y_{10}) \wedge (c_y \vee 0) & \rightarrow & y_{30} = (\neg c_y \vee y_{10}) \wedge c_y \\
y_{31} = (\neg c_y \vee y_{11}) \wedge (c_y \vee 1) & \rightarrow & y_{31} = \neg c_y \vee y_{11} \\
y_{32} = (\neg c_y \vee y_{12}) \wedge (c_y \vee 0) & \rightarrow & y_{32} = (\neg c_y \vee y_{12}) \wedge c_y
\end{array}
$$

Un exemple de règles additionnelles d'optimisation sur notre exemple pourrait être la propagation d'alias, *i.e.*, remplacer toutes occurrences de x_{10} par $\neg x_{00}$, ou des règles plus fonctionnelles comme :

$$x_{32} = (\neg c_x \vee x_{12}) \wedge c_x \quad \rightarrow \quad x_{32} = x_{12} \wedge c_x$$

Dans notre contexte, l'étape de normalisation vise à obtenir un modèle en format AIG (*And-Inverter Graph*, décrit dans la partie 2.2), c'est-à-dire que les définitions doivent être de la forme : un opérateur unique ($\wedge$) et deux opérandes au maximum, et cela grâce à l'ajout de variables intermédiaires. Voici un exemple pour la définition x_{11} :

$$\begin{aligned} x_{11} &= \neg x_{01} \oplus x_{00} \\ &\downarrow \\ x_{11} &= (\neg x_{01} \vee x_{00}) \wedge (x_{01} \vee \neg x_{00}) \\ &\downarrow \\ t_0 &= x_{01} \wedge \neg x_{00} \\ t_1 &= \neg x_{01} \wedge x_{00} \\ x_{11} &= \neg t_0 \wedge \neg t_1 \end{aligned}$$

Ce format permet d'appliquer plus facilement les règles de simplification, mais aussi de détecter plus d'équivalences structurelles. Cela facilite également la transformation du modèle en CNF.

CNF et troisième niveau d'optimisation

En reprenant la définition normalisée x_{11} et en utilisant la transformation de Tseitin [1968], la transformation en CNF est mécanique :

$$\begin{array}{lcl} & & \neg t_0 \vee x_{01} \\ t_0 = x_{01} \wedge \neg x_{00} & \rightarrow & \neg t_0 \vee \neg x_{00} \\ & & t_0 \vee \neg x_{01} \vee x_{00} \\ & & \neg t_1 \vee \neg x_{01} \\ t_1 = \neg x_{01} \wedge x_{00} & \rightarrow & \neg t_1 \vee x_{00} \\ & & t_1 \vee x_{01} \vee \neg x_{00} \\ & & \neg x_{11} \vee \neg t_0 \\ x_{11} = \neg t_0 \wedge \neg t_1 & \rightarrow & \neg x_{11} \vee \neg t_1 \\ & & x_{11} \vee t_0 \vee t_1 \end{array}$$

Enfin, une dernière étape de simplification peut être effectuée sur la formule CNF : élimination de variables, subsomption, etc. Plus de détails sont fournis dans la partie 3.2.3.

Vérification (Model Checking)

Cette étape a généralement lieu lorsque le modèle est dans un format où il est simple de le transformer en CNF et lorsque toutes les simplifications ont été effectuées. Ensuite, en fonction du choix de l'algorithme, le modèle est traduit en CNF, puis des appels aux solveurs sont effectués. Nous montrons ici un exemple de l'application de BMC sur l'exemple courant. Afin de simplifier la présentation, nous utiliserons les définitions de x et y avec le format haut niveau suivant :

$$x = x \neq 0 \ ? \ x - 1 \ : 3$$
$$y = y \neq 0 \ ? \ y - 1 \ : 2$$

BMC commence par vérifier si la négation de la propriété est atteignable aux états initiaux. La formule suivante est alors construite :

$$\text{BMC}_0 \ := (x_0 = 3 \wedge y_0 = 2) \wedge (x_0 = y_0)$$

Puis, sa satisfaisabilité est déterminée avec un solveur SAT. Comme cette formule est insatisfaisable, BMC vérifie si la négation de la propriété est atteignable à la profondeur suivante :

$$\begin{aligned}\text{BMC}_1 \ := & (x_0 = 3 \wedge y_0 = 2) \quad \wedge \\ & (x_1 = x_0 \neq 0 \ ? \ x_0 - 1 \ : 3 \ \wedge \ y_1 = y_0 \neq 0 \ ? \ y_0 - 1 \ : 2) \quad \wedge \\ & (x_1 = y_1)\end{aligned}$$

Comme BMC_1 est également insatisfaisable, la satisfaisabilité de BMC_2 est testée :

$$\begin{aligned}\text{BMC}_2 \ := & (x_0 = 3 \wedge y_0 = 2) \quad \wedge \\ & (x_1 = x_0 \neq 0 \ ? \ x_0 - 1 \ : 3 \ \wedge \ y_1 = y_0 \neq 0 \ ? \ y_0 - 1 \ : 2) \quad \wedge \\ & (x_2 = x_1 \neq 0 \ ? \ x_1 - 1 \ : 3 \ \wedge \ y_2 = y_1 \neq 0 \ ? \ y_1 - 1 \ : 2) \quad \wedge \\ & (x_2 = y_2)\end{aligned}$$

Il n'est pas possible d'atteindre un mauvais état en deux transitions, la formule est donc de nouveau insatisfaisable. Ce processus est alors répété, jusqu'à ce que la formule soit satisfaisable (si le modèle admet un contre-exemple). Ici, comme le modèle est très simple, il suffit de regarder les premières valeurs de x et y pendant l'exécution pour s'apercevoir que BMC_9 sera satisfaisable, et donc conclure que la propriété n'est pas vérifiée par le modèle, *i.e.*, il existe un CEX :

$\boldsymbol{i}$	0	1	2	3	4	5	6	7	8	**9**	...
$\boldsymbol{x}$	3	2	1	0	3	2	1	0	3	**2**	...
$\boldsymbol{y}$	2	1	0	2	1	0	2	1	0	**2**	...

1.4 Organisation

La suite de ce document est organisée comme suit. Le chapitre 2 introduit les notions qui seront utilisées dans le reste du document. Plus précisément, nous introduisons certains principes des solveurs SAT, des AIG et du model checking. Le chapitre 3 décrit les travaux que nous avons réalisés pour améliorer l'interconnexion entre le model checker et le solveur : élimination des variables au niveau du modèle, dialogue solveur/model checker, duplication des clauses, et dépliage de la relation de transition. Le chapitre 4 décrit une étude que nous avons effectuée avec Jesús Giráldez Crú, avec comme objectif de spécialiser notre solveur pour les formules générées par BMC. Le chapitre 5 donne plus de détails sur les techniques implémentées dans l'outil développé au cours de cette thèse. Enfin, nous proposons une conclusion en proposant quelques pistes ouvertes par nos travaux.

Chapitre 2

Définitions

2.1 Solveurs SAT

Le problème de satisfaisabilité (SAT) est le problème NP-complet canonique qui vise à déterminer s'il existe une affectation de valeurs aux variables d'une formule propositionnelle rendant cette formule vraie. Malgré sa complexité théorique intrinsèque, les progrès récents dans sa résolution pratique en ont fait une des méthodes les plus utilisées pour résoudre les problèmes difficiles, notamment dans le milieu de la vérification formelle.

Une formule propositionnelle est constituée d'un ensemble de variables booléennes, des constantes logiques : $\top$ (vrai, 1) et $\bot$ (faux, 0), et des opérateurs logiques : $\neg$ (non), $\vee$ (ou), $\wedge$ (et). Un *littéral* est une variable ou sa négation : x, $\neg x$. Une clause est une disjonction de littéraux, *e.g.*, $x \vee \neg y \vee z$. Une formule est dite sous *forme normal conjonctive* (CNF) si elle est une conjonction de clause, *e.g.*, $(x \vee \neg y \vee z) \wedge (\neg x) \wedge (y \vee z)$. Nous pouvons parfois, pour des raisons de simplicité, également considérer une clause comme un ensemble de littéraux, *e.g.*, $\{x, \neg y, z\}$, et une CNF comme un ensemble de clauses, par exemple : $\{\{x, \neg y, z\}, \{\neg x\}, \{y, z\}\}$. Une affectation des variables est une fonction associant à chaque variable une valeur booléenne. Une affectation satisfait une formule si la formule s'évalue à vrai en substituant chaque variable par sa valeur associée.

Dans cette partie, nous nous focalisons sur `Glucose` (Audemard et Simon [2013]), un solveur basé sur `Minisat` (Eén et Sörensson [2003a]), que nous avons utilisé au cours de nos travaux. Nous essayons d'être le plus concis et clair possible pour présenter les concepts et fonctionnalités des solveurs qui nous paraissent primordiales. Une présentation exhaustive des solveurs SAT dépasse cependant le cadre de ce chapitre. Nous invitons le lecteur intéressé à consulter par exemple Biere *et al.* [2009] pour une référence complète.

2.1.1 Principes généraux des solveurs SAT

Un solveur SAT est souvent utilisé comme une boîte noire très optimisée, résolvant le problème SAT. Cette utilisation en « boîte noire » est d'ailleurs l'une des raisons du succès de SAT. La plupart des solveurs prennent ainsi en entrée une formule sous forme CNF et sont capables de produire une affectation des variables si la formule est satisfaisable, sans aucune connaissance sur le sens des variables et des clauses encodées (certaines approches spécialisées ne rentrent pas dans ce cadre). Les solveurs peuvent également produire une preuve de non-satisfaisabilité (Wetzler *et al.* [2014]).

Une des premières approches pour résoudre le problème SAT, DP (Davis et Putnam [1960]), consiste à éliminer les variables une par une, à l'aide de la *règle de résolution*. L'étape d'élimination est répété jusqu'à ce que la clause vide soit trouvée (UNSAT), ou jusqu'à ce que la formule devienne vide (SAT). Toutefois, en raison du problème de l'explosion combinatoire en mémoire, cette technique fut rapidement abandonnée en faveur de la recherche avec retour arrière (DPLL Davis *et al.* [1962]). Notons tout de même que l'élimination de variables reste une étape de prétraitement quasi-indispensable dans tous les solveurs modernes. L'élimination d'une variable consiste simplement à effectuer toutes les résolutions possibles sur cette variable, puis à retirer de la formule toutes les clauses contenant cette variable, et enfin à ajouter toutes les clauses des résolutions. En d'autres termes, l'élimination de la variable x dans la formule Σ se résume à appliquer ce pseudo-algorithme : D'abord, partitionner Σ en 3 ensembles : $\Sigma_{\setminus x}$ l'ensemble des clauses ne contenant pas la variable x, Σ_x l'ensemble des clauses contenant le littéral x et $\Sigma_{\neg x}$ l'ensemble des clauses contenant le littéral $\neg x$. Puis, supprimer toutes les occurrences de x et $\neg x$ dans Σ_x et $\Sigma_{\neg x}$, et enfin reconstruire $\Sigma = \Sigma_{\setminus x} \wedge (\Sigma_x \vee \Sigma_{\neg x})$. Pendant l'étape de prétraitement, l'élimination de variables est usuellement appliquée sur toutes les variables pour lesquelles le nombre de clauses de la formule n'augmente pas lorsque celles-ci sont éliminées. Ce processus préserve la satisfaisabilité de la formule originale. Plus de détails et d'exemples sur l'élimination de variables sont fournis dans la partie 3.3. Depuis l'introduction de la technique d'apprentissage de clauses par Silva et Sakallah [1997] et son perfectionnement par Moskewicz *et al.* [2001], la grande majorité des algorithmes de résolution du problème SAT s'appliquant aux problèmes issus du monde réel sont des solveurs CDCL (*Conflict-Driven Clause Learning*), aussi appelés solveurs « modernes ».

2.1.2 CDCL

Les solveurs CDCL sont en partie basés sur DPLL, c'est-à-dire que l'idée général est la même : Les variables sont affectées au fur et à mesure, jusqu'à ce qu'une affectation des variables qui satisfait la formule soit trouvée, si elle

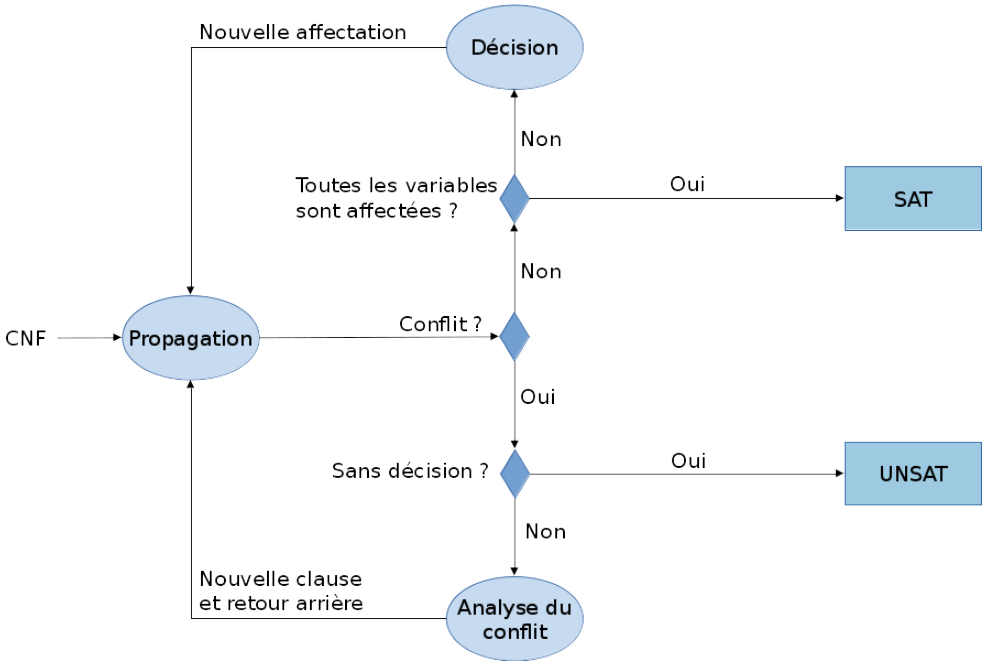


FIGURE 2.1 – Description du fonctionnement interne des solveurs CDCL.

existe. Dans le cas où l'affectation courante, qui est partielle car toutes les variables ne sont pas encore affectées, produit une clause vide (*i.e.*, tous les littéraux de la clause sont faux), le solveur revient en arrière et change l'affectation des variables. Ces affectations partielles sont construites à l'aide de décisions et de propagations. Dans un solveur CDCL, lorsqu'une affectation partielle produit un conflit, *i.e.*, une contradiction sur l'affectation d'une variable, une analyse de ce conflit est déclenchée. Une nouvelle clause est alors déduite (apprise) à partir de cette analyse. Celle-ci permettra par la suite au solveur de ne pas reproduire d'affectation partielle similaire. Ainsi, si le solveur réussit à construire une affectation complète (*i.e.*, affecter une valeur à chaque variable de la formule), qui n'a donc pas généré de conflit, alors la formule est satisfaisable. Autrement dit, en substituant chaque variable par la valeur de l'affectation, toutes les clauses sont satisfaites. A l'inverse, si un conflit se produit alors qu'aucune décision n'a été prise, la formule est insatisfaisable. Le fonctionnement des solveurs CDCL peut donc se résumer à l'enchaînement de 3 grandes étapes : décision, propagation, et analyse de conflit. Le schéma 2.1 représente sommairement le fonctionnement interne des CDCL.

Décision

L'étape de décision consiste simplement à sélectionner une variable dans l'ensemble des variables de la formule qui n'ont pas encore été affectées, puis à lui associer une valeur, et enfin à l'ajouter à l'affectation courante. S'il n'y

a plus de variables à affecter (elles ont déjà toutes une affectation), alors la formule est satisfaisable et le solveur retourne l'affectation courante. Cette affectation correspond à un modèle pour cette formule.

Propagation

Le fait que la formule en entrée du solveur soit en CNF permet d'avoir la propriété suivante : pour satisfaire la formule, il faut qu'au moins un des littéraux de chaque clause soit vrai. L'étape de propagation consiste alors à parcourir l'ensemble des clauses de la formule à la recherche des clauses ne contenant plus qu'un seul littéral non affecté et tel que tous les autres sont faux[1]. Par conséquent, pour satisfaire la formule ce littéral doit nécessairement être vrai. Plus précisément, lors de l'étape de propagation une clause peut être dans 3 états : (i) La clause est déjà satisfaite, *i.e.*, contient un littéral qui est vrai dans l'affectation courante. (ii) La clause contient au moins 2 littéraux non affectés. (iii) La clause ne contient plus qu'un littéral non affecté, ou qui a été affecté dans la polarité inverse par le processus de propagation. Les clauses étant dans l'un des deux premiers cas ne déclenchent pas de propagation. Pour le troisième cas, deux situations se présentent : 1. La variable correspondant au dernier littéral de la clause n'a pas encore été affectée. Le processus de propagation force alors l'affectation du littéral restant de manière à satisfaire la clause. 2. La variable correspondant au dernier littéral de la clause est déjà affectée dans la polarité inverse. Un conflit se produit, c'est-à-dire qu'il y a une contradiction sur l'affectation de la variable. Le solveur doit donc revenir en arrière et changer cette affectation.

Exemple 2.1 (Solveur CDCL)**.** Voici le déroulement possible d'un solveur CDCL lors de la résolution de la formule φ suivante :

$\varphi = \underbrace{(a \vee b)}_{c_1} \wedge \underbrace{(a \vee \neg b)}_{c_2} \wedge \underbrace{(\neg a \vee b \vee c)}_{c_3}$ avec $\vartheta = \{\}$ l'affectation courante.

1. Propagation : Rien à propager, $\vartheta = \{\}$
2. Décision : $\vartheta = \{\neg a\}$, $\varphi = \underbrace{(a \vee b)}_{c_1} \wedge \underbrace{(a \vee \neg b)}_{c_2} \wedge \underbrace{(\neg a \vee b \vee c)}_{c_3}$
3. Propagation : $c_1 \rightarrow b$, $\vartheta = \{\neg a, b\}$
4. Propagation : $c_2 \rightarrow \neg b$, $\vartheta = \{\neg a, b, \neg b\}$, conflit sur l'affectation de b
5. Analyse : Ajout de c_4[2], $\varphi = \underbrace{(a \vee b)}_{c_1} \wedge \underbrace{(a \vee \neg b)}_{c_2} \wedge \underbrace{(\neg a \vee b \vee c)}_{c_3} \wedge \underbrace{(a)}_{c_4}$
   Retour arrière : $\vartheta = \{\}$

1. En pratique, il est très important que le solveur puisse ne pas avoir à parcourir l'ensemble des clauses. Grâce à une structure de donnée paresseuse, il parcourt uniquement un sous ensemble des clauses contenant l'opposé du littéral venant d'être affecté.
2. L'apprentissage de clauses à partir de l'analyse de conflit est expliqué dans la partie suivante.

6. Propagation : $c_4 \rightarrow a$, $\vartheta = \{a\}$
7. Décision : $\vartheta = \{a, \neg b\}$
8. Propagation : $c_3 \rightarrow c$, $\vartheta = \{a, \neg b, c\}$
9. Retour du solveur : SAT$(a, \neg b, c)$

L'étape de propagation est efficacement implémenté à l'aide d'une technique appelée *two watched literals* (Moskewicz *et al.* [2001]). Cette technique vient de l'observation que quelque soit la taille d'une clause, tant que deux des littéraux de celle-ci ne sont pas affectés alors il n'est pas nécessaire de la propager. Cela permet une gestion paresseuse de la propagation, ce qui fournit aux solveurs une méthode pour traiter efficacement des problèmes contenant un très grand nombre de clauses, pouvant même être très longues.

Apprentissage

L'apprentissage est un élément clef des solveurs CDCL. Cette étape est déclenchée lorsqu'un conflit sur l'affectation d'une variable se produit, lors de la propagation (comme pour l'exemple ci-dessus). L'idée est alors de déduire un sous-ensemble des affectations courantes qui ont mené à ce conflit et qui mèneront toujours à ce conflit, quelles que soient les autres affectations. La négation de ce sous-ensemble d'affectations, qui correspond à une clause, est ensuite ajoutée à la formule. Ainsi, grâce à l'étape de propagation, cette affectation ne pourra plus se reproduire. Comme ces clauses sont ajoutées à la formule, le retour en arrière des affectations peut s'effectuer de manière non chronologique (par opposition à DPLL). Autrement dit, à chaque nouvelle clause apprise et ajoutée à la formule, le solveur pourrait redémarrer à zéro (sans changer la complétude de l'algorithme).

Pour calculer ce sous-ensemble de l'affectation qui mène au conflit, les solveurs construisent un graphe d'implication. Il s'agit d'un graphe orienté, où chaque nœud correspond à une variable, associée à son niveau de décision, et chaque arête correspond à une clause. Plus exactement, il existe une arête d'un nœud n_1 vers un nœud n_2 si n_2 a été propagé par la clause c et tel que $n_1 \in c$.

Exemple 2.2 (Graphe d'implication)**.** Supposons la formule CNF suivante :

$$\begin{aligned}\varphi &= c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \\ &= (\neg a \vee \neg b) \wedge (b \vee c \vee \neg d) \wedge (b \vee d) \wedge (d \vee f) \wedge (\neg e \vee \neg f \vee g)\end{aligned}$$

et tel que les décisions d'affecter $g = \bot$ et $c = \bot$ ont été faites par le solveur au niveau 1 et 2, et tel que la décision courante (niveau 3) est $a = \top$. Le graphe d'implication résultant est illustré dans la figure 2.2. Un conflit se produit car la clause c_5 n'est pas satisfaite alors que tous ses littéraux sont affectés.

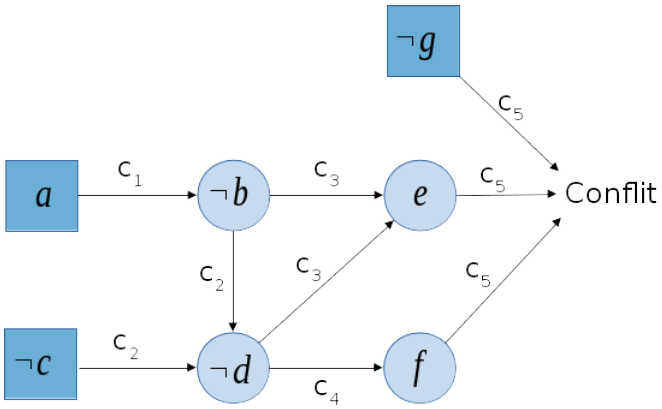


FIGURE 2.2 – Graphe d'implication.

À partir du graphe d'implication, il existe différentes stratégies pour extraire une nouvelle clause. Par exemple, la plus simple des méthodes consiste à extraire la clause qui correspond à l'inverse de toutes les décisions prises par le solveur. Sur l'exemple précédent, la clause apprise correspondante serait $\neg a \vee c \vee g$. Néanmoins, lorsque le graphe d'implication est de grande taille, les clauses générées par cette méthode simple sont généralement beaucoup trop grandes et probablement trop proches des décisions pour que la clause apprise puisse se généraliser au plus grand nombre possible de cas similaires. On préférera ainsi, intuitivement, construire une clause expliquant la contradiction, mais la plus proche possible du conflit. L'objectif est également de générer des clauses de petites tailles, qui sont en général plus efficaces pour accélérer la recherche. Le FUIP (*First Unique Implication Point*, Zhang *et al.* [2001]; Dershowitz *et al.* [2007]) ou 1-UIP est une autre stratégie pour générer une clause apprise à partir d'un graphe d'implication. Cette stratégie est considérée comme étant globalement la plus performante et c'est également celle utilisée dans `Glucose`. Un UIP est un nœud dans le graphe ayant un niveau de décision égal au niveau de décision courant, et tel que tous les chemins, partant de la décision courante et terminant au conflit, passent par ce nœud. Dans l'exemple précédent, il y a deux UIP : le nœud de la décision elle-même (a), ainsi que le nœud correspondant au littéral $\neg b$. Le FUIP correspond au premier UIP rencontré en partant du conflit, lors de l'analyse de conflit. La clause générée est alors composée de l'inverse du FUIP et de l'inverse de tous les littéraux affectés aux niveaux supérieurs (par décision ou propagation) et vus lors de l'analyse de conflit. Dans l'exemple précédent, le FUIP est $\neg b$, le premier UIP rencontré en partant du conflit, dit simplement l'UIP le plus proche du conflit. $\neg c$ et $\neg g$ sont deux décisions qui peuvent atteindre le conflit sans passer par le FUIP. La clause apprise en utilisant la méthode du FUIP serait donc : $b \vee c \vee g$.

Ce graphe d'implication indique le sous-ensemble de clauses de la formule

sur lequel en effectuant des résolutions, on peut déduire la nouvelle clause apprise. Pour bien s'en rendre compte, il suffit par exemple pour inférer la clause apprise d'effectuer l'élimination de l'ensemble des variables qui apparaissent dans le graphe d'implication entre les nœuds des littéraux de la clause apprise et le conflit. Sur l'exemple précédent, si l'on applique la règle de résolution sur e, f et d, on obtiendra la clause $b \vee c \vee g$, ce qui correspond à la clause apprise en utilisant le FUIP. Si en plus on élimine la variable b, on obtiendra alors la clause $\neg a \vee c \vee g$.

Des étapes additionnelles de résolutions, appelées minimisation des clauses apprises, peuvent également être effectuées afin de réduire la taille des clauses apprises (voir Biere *et al.* [2009] pour une référence complète).

Heuristique de sélection des variables et de la polarité

Les heuristiques de sélection de variables permettent de déterminer sur quelle variable le solveur doit brancher lorsqu'il est dans l'étape de décision. Nous ne détaillerons pas l'ensemble des différentes techniques existantes, mais seulement celle utilisée dans `Minisat` (et donc dans `Glucose`) : VSIDS (*Variable State Independent Decaying Sum*, introduite dans Moskewicz *et al.* [2001]). Le fonctionnement de la méthode originale (dans Chaff) est le suivant : Un score d'activité est associé à chaque littéral. Ce score est incrémenté à chaque fois que le littéral apparaît dans une clause apprise. Le score de tous les littéraux est régulièrement divisé par une constante. Le solveur branche sur le littéral ayant le plus grand score d'activité (et aléatoirement en cas d'égalité). Ainsi, le solveur va sélectionner en priorité les littéraux les plus présents dans les derniers conflits.

Dans `Minisat`, cette heuristique est légèrement différente. Un score est associé aux variables plutôt qu'aux littéraux. Au lieu d'incrémenter seulement les littéraux apparaissant dans la clause apprise, tous les scores des variables apparaissant dans l'analyse de conflits sont incrémentés (variables apparaissant dans la clause apprise et variables sur lesquelles une résolution a été effectuée lors de l'analyse). De plus, les scores sont plus régulièrement réduits (tous les conflits) ce qui augmente la réactivité de la méthode (en pratique, le poids avec lequel les scores des variables sont augmenté est multiplié par une constante strictement plus grande que 1 après chaque conflit (en pratique on utilise 1.05). Il augmente donc exponentiellement, ce qui a pour effet d'oublier les variables vues il y a longtemps). Puis, comme précédemment, la variable ayant le plus grand score est choisie. Il reste cependant encore à déterminer sa polarité. La polarité est simplement la valeur, vrai ou faux, que le solveur affecte à la variable choisie par l'heuristique de sélection. Il existe également différentes heuristiques pour la polarité : aléatoire, toujours à une même valeur, etc. Celle utilisée dans `Minisat` (`Glucose`) consiste à se souvenir de la valeur précédemment affectée à la même variable (*Phase saving*, Pipatsrisawat et Darwiche

[2007]). L'idée est d'augmenter les chances de profiter des dernières clauses apprises.

Redémarrage

Le redémarrage est une technique qui a d'abord été introduite pour éviter le phénomène dit *Heavy-tailed distributions*. Aujourd'hui les raisons de son succès dépassent cette motivation initiale. Intuitivement, le temps d'exécution pour une même formule peut énormément varier en fonction des choix de l'heuristique de sélection des variables en haut de l'arbre de recherche. Par exemple, le solveur peut parfois nécessiter beaucoup de temps pour résoudre une formule alors que s'il avait branché sur une variable différente, le temps de résolution aurait été bien plus court. Le redémarrage consiste alors à effectuer un retour en arrière au niveau de décision 0, *i.e.*, revenir à une affectation vide. Cela permet de changer complètement l'arbre de recherche, en effet, comme des clauses ont été apprises et que le score des variables a été mis à jour, l'ordre des décisions sera certainement très différent, même si les polarités sont conservées. Les redémarrages sont généralement effectués tous les X conflits, où X est incrémenté à chaque redémarrage pour garantir la complétude de l'algorithme. Il existe différentes stratégies de redémarrage, la plus connue étant certainement la séquence de Luby *et al.* [1993] qui est théoriquement optimale dans un contexte d'exploration aveugle. Dans `Glucose`, la stratégie de redémarrage est légèrement différente, car elle est basée sur le score LBD des clauses, que nous décrivons dans la partie suivante. Intuitivement, si la qualité (le score LBD) des dernières clauses apprises est moins bonne que la qualité moyenne de l'ensemble des clauses apprises, alors un redémarrage est déclenché.

Gestion des clauses apprises et LBD

Un solveur CDCL peut apprendre plus de 5000 clauses par seconde, c'est pourquoi, il est rapidement devenu crucial de mettre en place des stratégies de gestion pour les bases de données des clauses apprises. Dans *Minisat*, Eén et Sörensson [2003a] ont proposé de supprimer les clauses qui étaient les moins vues lors des dernières analyses de conflit, à l'aide d'un score d'activité sur les clauses (comme pour les variables dans VSIDS). Avec l'idée de mesurer la distance entre les blocs de littéraux (LBD, *Literal Block Distance*), Audemard et Simon [2009] ont introduit une nouvelle façon de classer et donc de supprimer les clauses. Le système de score LBD est relativement simple. Le score d'une clause correspond au nombre de niveaux de décisions distincts de chacun des littéraux de la clause apprise. Par exemple, le score LBD de la clause de l'exemple 2.2 est de 2, a et c ont des niveaux de décisions distincts [3].

3. Une clause binaire a toujours un score LBD égal à 2

Dans `Glucose`, les clauses de LBD 2 sont appelées des *clauses glues* (ou *glues clauses*), et ne sont jamais supprimées de la base de données des clauses apprises. Le reste des clauses apprises sont régulièrement supprimées de la base de données, en fonction de leur score LBD. Les clauses ayant le plus grand score LBD sont supprimées en priorité. Afin de donner un ordre de grandeur, lors d'une exécution de `Glucose`, 95% des clauses apprises peuvent être supprimées au cours de la résolution. A noter que pour rester complet, les solveurs doivent conserver de plus en plus de clauses.

Algorithme CDCL

Pour clarifier le déroulement de l'ensemble de ces étapes, nous indiquons ci-dessous en pseudo-code un exemple d'algorithme CDCL. Pour ne pas ajouter trop d'éléments pouvant nuire à la lisibilité du pseudo-code, nous n'avons pas spécifié comment le redémarrage et la gestion des clauses apprises sont effectués.

```
while 1 do
    conflit := PROPAGATION( )
    if conflit == ∅ then
        if NOUVELLEDECISION( ) == ∅ then
            return SAT
    else
        if NIVEAUDECISION( ) == 0 then
            return UNSAT
        clauseApprise, niveauRetour := ANALYSECONFLIT(conflit)
        AJOUTCLAUSE(clauseApprise)
        ANNULEDECISION(niveauRetour)
```

Algorithme 1 – CDCL.

Voici le descriptif des différentes fonctions :

— PROPAGATION effectue l'étape de propagation. Si une clause vide est rencontrée (tous ses littéraux sont déjà affectés à faux) alors cette clause est retournée. Si aucun conflit ne se produit durant l'étape de propagation, cette fonction retourne $\emptyset$.

— NIVEAUDECISION retourne le niveau de décision courant.

— ANALYSECONFLIT analyse le conflit et retourne la nouvelle clause apprise ainsi que le niveau auquel il faut effectuer un retour arrière.

— AJOUTCLAUSE ajoute simplement la clause en paramètre dans la base de données des clauses apprises, qui sont considérées exactement comme des clauses originales pour notre algorithme.

— annuleDecision annule l'ensemble des décisions et propagation jusqu'au niveau passé en paramètre.

— nouvelleDecision incrémente le compteur de décision courant et ajoute un littéral (en fonction de l'heuristique de branchement) aux décisions. Si toutes les variables sont déjà affectées alors cette fonction retourne $\emptyset$.

2.1.3 Incrémental SAT

La résolution incrémentale est une technique qui s'applique lorsqu'un problème nécessite de vérifier la satisfaisabilité de plusieurs formules similaires, *i.e.*, contenant un sous-ensemble de clauses communes. Plutôt que de vérifier la satisfaisabilité de chacune des formules une à une, l'idée est de profiter des clauses apprises qui concernent la partie commune à ces formules lors des appels successifs. Eén et Sörensson [2003b] ont alors proposé une approche ne nécessitant qu'une légère modification des solveurs : permettre à l'utilisateur de forcer l'affectation de variables, et tel que ces affectations se comportent virtuellement comme des décisions en haut de l'arbre de recherche, donc toujours prises avant toute vrai décision. Ainsi, en ajoutant des littéraux d'activation aux clauses qui diffèrent entre les formules similaires, la résolution incrémentale est entièrement gérée par le processus d'apprentissage de clauses, car toute clause apprise dépendant d'une décision « virtuelles » contiendra le littéral concerné par cette décision. Ces littéraux spéciaux ont été nommés « littéraux d'activation ».

Un littéral d'activation est un littéral qui est ajouté à une clause ou un ensemble de clauses afin de permettre de modifier aisément leur satisfaisabilité, tout en laissant intact le reste de la formule. Par exemple, supposons un solveur S, contenant une clause $c = \{\neg act, x, y\}$, dans laquelle le littéral $\neg act$ a été ajouté. Il est alors facile d'activer c en forçant act à $\top$ dans S. Le solveur doit alors satisfaire le reste de la clause, *i.e.*, $x \vee y$. A l'inverse, en forçant l'affectation de act à $\bot$, la clause c est directement satisfaite, elle est désactivée, et le solveur peut retourner un modèle avec $x = \bot, y = \bot$.
Le principal avantage de cette méthode est que les clauses apprises qui dépendent des clauses activées vont donc également contenir les littéraux d'activation utilisés, grâce au processus d'apprentissage, comme évoqué précédemment. En effet, comme ces littéraux d'activation sont affectés comme des décisions, si une clause contenant un de ces littéraux est utilisée dans l'analyse de conflit alors le littéral d'activation qui sera nécessairement affecté à un niveau de décision plus petit sera ajouté à la nouvelle clause apprise.

2.1.4 Structure en communautés

Si l'on peut, comme on l'a vu, décrire les solveurs SAT assez rapidement, il est important de noter que les raisons de leur efficacité pratique ne sont pas

encore bien connus. Nous avons proposé d'étudié la structure des instances de model checking sur lesquels les solveurs SAT sont particulièrement efficaces pour tenter de répondre à cette question. Pour cela, il nous faut définir dès maintenant quelques notions de base permettant d'appréhender certaines structures dans les graphes.

Un graphe a une structure en communautés (ou en grappes), si pour une partition de ses nœuds en communautés, la plupart des arêtes connectent des nœuds qui sont à l'intérieur d'une même communauté (intra-communautaire) et que très peu d'arêtes relient des nœuds de communautés différentes (inter-communautaire). Afin d'évaluer la qualité d'une telle partition, la structure en communautés est habituellement quantifié à l'aide d'un score. Le score le plus couramment utilisé est la modularité (Newman et Girvan [2004]; Fortunato [2010]). L'idée du score de modularité est de comparer la proportion d'arêtes dans chaque communauté avec la proportion d'arêtes qu'il y aurait dans un graphe aléatoire, c'est-à-dire dans un graphe qui n'a pas une structure en communautés par construction. Ainsi, un graphe aléatoire aura un score de modularité s'approchant de 0, tandis qu'un graphe ayant une forte structure en communautés aura un score de modularité proche de 1.

Récemment, il a été montré par Ansótegui *et al.* [2012] que les formules CNF de la plupart des instances SAT utilisées dans les compétitions SAT, lorsqu'elles sont représentées sous forme de graphe, ont une structure en communautés bien marquée, *i.e.*, un haut score de modularité. La technique de Ansótegui *et al.* [2012] consiste à représenter les formules CNF sous forme d'un VIG (*Variable-Incidence Graph*). Un VIG est un graphe pondéré, où les nœuds correspondent aux variables de la CNF, et où il y a une arête entre deux nœuds, si les deux variables correspondantes apparaissent dans une même clause. Le poids d'une arête est calculé en fonction de la taille des clauses. Puis, à l'aide de la méthode de Louvain (Blondel *et al.* [2008]), une partition des variables de la CNF ainsi qu'une borne inférieure de la valeur maximale de la modularité sont calculés.

2.2 And-Inverter Graphs (AIG)

Après l'introduction des notions des solveurs SAT, la prochaine étape avant d'effectuer la vérification est de choisir une structure pour représenter les modèles. Comme notre point d'entrée est juste après l'étape de bit-blast (ou à partir d'un circuit), nous avons besoin d'une représentation pour les fonctions booléennes. Il en existe de nombreuses, *e.g.*, circuit, BDD, CNF. Ces structures ont chacune leurs avantages et inconvénients, par exemple : les circuits sont compacts et faciles à construire (car de nombreux opérateurs sont autorisés) ; les BDDs sont canoniques (détection gratuite des nœuds équivalents) mais requièrent en général beaucoup d'espace mémoire pour les modèles complexes et sont dépendants de l'ordre des variables ;les CNFs restent de taille raisonnable

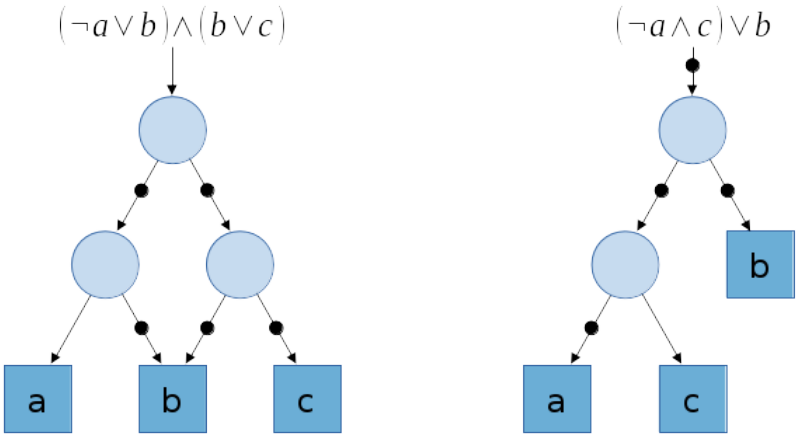


FIGURE 2.3 – Exemple de deux AIGs fonctionnellement équivalents mais structurellement différents. Chaque nœud interne représente un « et » logique ($\wedge$) et chaque point sur une arête indique une négation ($\neg$).

si l'introduction de variables est possible et il s'agit du format d'entrée des solveurs SAT, cependant, la structure du problème est plus difficile à retrouver (Ostrowski *et al.* [2002]). C'est pourquoi, le *And-Inverter Graph* (AIG) est généralement un bon compromis : plus facile à maintenir qu'un circuit, plus compact qu'un BDD, plus facile à optimiser qu'une CNF. Un AIG est un graphe permettant de représenter les fonctions booléennes en utilisant uniquement deux opérateurs : « et » et « non ». Il est utilisé comme point d'entrée dans de nombreux model checker (voir par exemple les participants de la dernière compétition de model checking HWMCC'17, Biere *et al.* [2017]), ou également comme représentation intermédiaire avant la transformation en CNF (comme dans le solveur SMT `Boolector` par exemple, Brummayer et Biere [2009]). AIGER est un format associé à une librairie et également un ensemble d'outils développé par Biere [2007]; Biere *et al.* [2011], pour les AIG. C'est très clairement aujourd'hui le format de référence. Dans le format AIGER, il est également possible d'ajouter aux AIGs des informations nécessaires pour le model checking, par exemple, des propriétés multiples ou des contraintes.

Un AIG est un graphe orienté acyclique composé de deux types de nœuds :

1. les noeuds internes, qui représentent l'opérateur « et » logique ($\wedge$), avec deux arêtes sortantes correspondant aux opérandes de ce $\wedge$.
2. les feuilles, qui sont étiquetées par des variables.

Toutes les arêtes peuvent être marquées, indiquant ainsi qu'il s'agit alors d'une négation ($\neg$). Par exemple, le schéma 2.3 représente deux AIGs structurellement différents mais fonctionnellement équivalents. Cet exemple illustre le principal inconvénient des AIGs par rapport aux structures qui sont canoniques (*e.g.*, les BDDs). Autrement dit, ces deux AIGs pourraient être présent dans un AIG, *i.e.*, être deux sous-graphes d'un plus grand graphe, constituant

alors une redondance d'informations, alors que l'un pourrait être remplacé par l'autre, car ils sont fonctionnellement équivalents.

Systèmes séquentiels

Comme nous souhaitons vérifier des systèmes séquentiels et pas seulement combinatoires, il est nécessaire d'introduire la notion de « mémoire ». Un système séquentiel est un système pour lequel les sorties ne dépendent pas uniquement des entrées (comme pour un système combinatoire), mais également de l'état dans lequel le système se trouve (défini dans la partie 2.3.1). Le format AIGER permet nativement de définir des variables mémoires.

Nous décrivons ci-dessous de façon moins formelle les modèles au format AIGER que nous utiliserons comme point d'entrée. Intuitivement, un modèle peut être défini comme un ensemble d'entrées, de mémoires et de définitions.

Entrée (*Input*)

Une entrée correspond simplement à une variable libre, c'est-à-dire sans contrainte. Ainsi, à chaque cycle, une entrée peut être affecté à n'importe quelle valeur de son domaine (ici, $\top$ ou $\bot$).

Mémoire (*Latch*)

Une mémoire est une variable définie par une valeur initiale et une valeur « suivante ». Cette mémoire vaut donc sa valeur initiale à l'initialisation du système, puis à chaque début de cycle suivant, elle prend la valeur correspondant à sa valeur « suivante » du cycle précédent. Supposons par exemple, m une mémoire, avec $\mathrm{I}(m) := \bot$ sa valeur initiale et $\mathrm{X}(m) := \neg m$ sa valeur suivante. Nous utiliserons également la notation $m := \bot, \neg m$. Ainsi, à l'initialisation du système, autrement dit au cycle 0, m vaudra $\bot$, puis au cycle 1, m vaudra $\top$, et ainsi de suite. Généralement, on parle indifférent de variables mémoires ou de variables d'état.

Définition « ET »

Une définition est l'affectation d'une opération à une variable intermédiaire. Dans un AIG, une opération est toujours un « et » logique ($\wedge$) entre 2 opérandes (littéraux), *e.g.*, $x := a \wedge \neg b$.

Exemple 2.3. Modèle AIG avec 2 entrées a, b, une mémoire x et 2 définitions y, z. Le dépliage suivant est équivalent à celui qui est effectué lorsque l'on

effectue BMC. On notera qu'il n'y a ici aucune propriété à vérifier.

Modèle		Cycle 0	Cycle 1	Cycle 2	...
a;	$\rightarrow$	a_0	a_1	a_2	...
b;	$\rightarrow$	b_0	b_1	b_2	...
$x := \bot,\ z$	$\rightarrow$	$\bot$	z_0	z_1	...
$y := \neg a \wedge x$	$\rightarrow$	$y_0 \Leftrightarrow \neg a_0 \wedge \bot$	$y_1 \Leftrightarrow \neg a_1 \wedge z_0$	$y_2 \Leftrightarrow \neg a_2 \wedge z_1$	...
$z := \neg y \wedge b$	$\rightarrow$	$z_0 \Leftrightarrow \neg y_0 \wedge b_0$	$z_1 \Leftrightarrow \neg y_1 \wedge b_1$	$z_2 \Leftrightarrow \neg y_2 \wedge b_2$	...

En pratique, dans la plupart des approches effectuant un dépliage, les variables mémoires n'apparaissent pas (ou plus). En effet, elles servent uniquement à *connecter* les cycles successifs entre eux.

2.3 Model Checking

Le Model Checking est une technique automatique et exhaustive de vérification formelle, qui détermine si un *modèle* vérifie une *propriété* donnée. Les propriétés expriment un comportement particulier du système que l'on souhaite vérifier, et sont généralement définies en logique temporelle. Rappelons que dans le cadre de ce document, nous considérons uniquement les modèles représentant des systèmes de transitions booléens à états finis. Ces systèmes sont généralement utilisés pour représenter les circuits séquentiels, ou les systèmes définis dans la théorie des vecteurs de bits sans quantificateur (QF_BV) après *bit-blasting*. En outre, nous nous limitons aux propriétés sous la forme d'assertion.

2.3.1 Système de transitions à états finis

Un système de transitions à états finis est un 4-uplet $\mathcal{M} = \langle V, I, T, P \rangle$, où V est un ensemble de variables booléennes, $I(V)$ et $P(V)$ sont des formules sur V représentant respectivement l'ensemble des états initiaux et l'ensemble des états vérifiant la propriété. $T(V, V')$ est la relation de transition, *i.e.*, une formule sur V et V' caractérisant les transitions acceptées du système. $V' = \{v' \mid v \in V\}$ est la version prime de l'ensemble V, généralement utilisée pour représenter les variables au cycle suivant. De la même façon, lorsque nous déplions la relation de transition plusieurs fois, nous utilisons $V_i = \{v_i \mid v \in V\}$ pour définir les mêmes ensembles que V avec renommage des variables. Un état du système est une affectation des variables de V. Les affectations satisfaisant une formule sur V représentent donc un ensemble d'états du système. Nous utilisons $Bad(V) = \neg P(V)$ pour exprimer l'ensemble des *mauvais* états du systèmes, c'est-à-dire les états qui contredisent le comportement désiré (la propriété). Un système admet un *contre-exemple* s'il est possible d'atteindre un mauvais état, en partant des états initiaux et en appliquant répétitivement la

relation de transition. A l'inverse, on dit que la propriété est vérifiée ou parfois *prouvée*, s'il n'existe pas de chemin partant des états initiaux et terminant dans un mauvais état. Autrement dit, l'ensemble des états atteignables du système est disjoint de l'ensemble des mauvais états.

2.3.2 BMC

Le model checking borné (BMC) est un algorithme paramétré orienté vers la recherche de contre-exemples, introduit par Biere *et al.* [1999]. Intuitivement, l'idée est de construire une formule qui est satisfaisable si le système peut atteindre un mauvais état en k transitions. BMC est très efficace pour trouver des contre-exemples, mais il est difficile de trouver un k suffisamment grand pour garantir que la propriété est vérifiée pour tous les états atteignables du système, tout en étant suffisamment petit pour être résolu en pratique[4]. La formule à vérifier est la suivante :

$$\text{BMC}_k = I(V_0) \wedge \bigwedge_{i=0}^{k-1} T(V_i, V_{i+1}) \wedge Bad(V_k)$$

Comme évoqué précédemment, cette formule est ensuite encodée en CNF et sa satisfaisabilité est déterminée à l'aide d'un solveur SAT. Cependant, cette définition ne permet pas de s'assurer qu'il n'existe pas de contre-exemple de profondeur strictement inférieur à k. C'est pourquoi BMC est généralement exécuté de façon incrémentale, *i.e.*, en résolvant BMC_k pour $k = 0, 1, \ldots \infty$, jusqu'à ce qu'un contre-exemple soit trouvé ou que les ressources de calcul disponibles soient épuisées. Une autre approche, moins utilisée, consiste à étendre $Bad(V_k)$ à $\bigvee_{i=0}^{k} Bad(V_i)$.

2.3.3 Induction Temporelle

Pour vérifier une propriété dans un système de transitions, le principe d'induction nécessite 2 étapes :

1. S'assurer que les états initiaux vérifient la propriété (cas de base) :

$$I(V) \Rightarrow P(V)$$

ou en *formulation SAT* :

$$I(V) \wedge \neg P(V) \models \bot$$

4. Le nombre d'états du système est évidemment suffisant mais généralement impossible à résoudre en pratique.

2. Puis, s'assurer qu'en appliquant la relation de transition sur l'ensemble des états vérifiant la propriété, les états suivants vérifient également la propriété (hérédité) :

$$P(V) \wedge T(V, V') \Rightarrow P(V')$$

ou en *formulation SAT* :

$$P(V) \wedge T(V, V') \wedge \neg P(V') \models \bot$$

Intuitivement, l'induction, qui est équivalente à la 1-induction, montre que si la propriété est vraie pour un cycle donné, alors il n'est pas possible d'atteindre un mauvais état en une transition. En d'autres termes, la propriété P est un ensemble dit *inductif* dans le système de transition, *i.e.*, pour tout état vérifiant P, l'ensemble des états atteignables en une transition vérifie également P. Ainsi, si P est vrai pour les états initiaux alors, par récurrence, P est vrai pour l'ensemble du système.

Cependant, les propriétés sont rarement inductives, ce qui signifie que la formule d'hérédité est satisfaisable. Dans ce cas, aucune conclusion ne peut être tirée. L'hypothèse de récurrence doit alors être *renforcée*. La k-induction, introduite par Sheeran *et al.* [2000], propose de la renforcer en augmentant le nombre de cycles consécutifs pour lesquels P est vrai. BMC est alors utilisé comme *base* du raisonnement, assurant que la propriété est vraie pour les k premiers cycles du système. Puis, la k-induction (l'hérédité) consiste à vérifier que si la propriété est vraie pendant k cycles consécutifs alors elle reste vraie au cycle suivant. Ainsi, par récurrence la propriété est vérifiée pour l'ensemble du système. La formule de l'hérédité est définie comme suit :

$$k\text{-induction} = \bigwedge_{i=0}^{k-1} \left[P(V_i) \wedge T(V_i, V_{i+1})\right] \wedge Bad(V_k)$$

2.3.4 Algorithme ZigZag

L'algorithme *ZigZag*, introduit par Eén et Sörensson [2003b], consiste simplement à effectuer l'induction temporelle, *i.e.*, combiner BMC et la k-induction, dans un seul solveur SAT. En effet, les formules générées par BMC et la k-induction possèdent une grande partie commune, et le solveur peut ainsi profiter, en partie, des clauses apprises lors des appels précédents, tout en conservant les scores des heuristiques de branchement. L'algorithme se résume donc à résoudre consécutivement k-induction et BMC_k, avec $k = 0, 1, \ldots \infty$, jusqu'à ce qu'une formule BMC devienne satisfaisable — il existe alors un contre-exemple — ou qu'une formule de k-induction soit insatisfaisable, dans ce cas la propriété est vérifiée pour le système. L'algorithme 2 décrit le déroulement de ZigZag en pseudo-code. Par souci de clarté, nous utilisons I_i, P_i pour

représenter $I(V_i)$, $P(V_i)$ et $T_{i,j}$ pour $T(V_i, V_j)$. AJOUTERCLAUSES et SOLVE sont deux fonctions du solveur SAT, utilisé ici comme un oracle, qui servent respectivement à ajouter des clauses dans le solveur et à résoudre l'ensemble des clauses du solveur. Les paramètres de la fonction SOLVE servent à forcer l'affectation des littéraux d'activation.

Fonction ZIGZAG(I, T, P)
  AJOUTERCLAUSES($I_0 \vee \neg act\_I$)
  for $i \in 0 \dots \infty$ **do**
    AJOUTERCLAUSES($\neg P_i \vee \neg act\_Bad_i$)
    if SOLVE(act_Bad_i) == UNSAT **then**
      return $\bot$ ▷ P est vérifiée
    if SOLVE(act_I, act_Bad_i) == SAT **then**
      return $\top$ ▷ Contre-exemple
    AJOUTERCLAUSES(P_i)
    AJOUTERCLAUSES($T_{i,i+1}$)

Algorithme 2 – ZigZag.

Afin de bien illustrer le déroulement de l'algorithme, voici, dans l'ordre, les premiers appels consécutifs au solveur.

$$
\begin{aligned}
\text{0-ind} &: \quad \big[Bad_0\big] \\
\text{BMC}_0 &: \quad \big[I_0\big] \wedge \big[Bad_0\big] \\
\text{1-ind} &: \quad P_0 \wedge T_{0,1} \wedge \big[Bad_1\big] \\
\text{BMC}_1 &: \quad \big[I_0\big] \wedge P_0 \wedge T_{0,1} \wedge \big[Bad_1\big] \\
\text{2-ind} &: \quad P_0 \wedge T_{0,1} \wedge P_1 \wedge T_{1,2} \wedge \big[Bad_2\big] \\
\text{BMC}_2 &: \quad \big[I_0\big] \wedge P_0 \wedge T_{0,1} \wedge P_1 \wedge T_{1,2} \wedge \big[Bad_2\big] \\
&\vdots
\end{aligned}
$$

Les crochets sont utilisés pour mettre en avant les parties de la formule qui sont temporaires. Ce sont les parties qui sont *activées/désactivées* à l'aide des littéraux d'activation. Cela permet de mettre en évidence que les parties concernant P et T restent toujours actives dans le solveur lorsque ZigZag est effectué. Ainsi, les clauses apprises par le solveur sur cette partie statique et permanente peuvent être réutilisées lors de tous les appels successifs au solveur.

2.3.5 Contraintes de chemin simple

Les contraintes de chemin simple, aussi appelées *contraintes sans boucle*, sont ajoutées à la formule de k-induction afin de rendre l'algorithme complet (Sheeran *et al.* [2000]). Elles sont utilisées pour spécifier que chaque état du

système dans le chemin construit pour la k-induction doit être différent [5]. Cela permet notamment d'éviter le phénomène décrit dans l'exemple 2.4 ci-dessous.

Exemple 2.4 (Non terminaison de l'algorithme ZigZag sans l'ajout des contraintes de chemin simple)**.** Soit c une entrée de type booléen, x une mémoire de type entier encodé sur 4 bits ($x \in [-8, 7]$), et u, v deux définitions de même type que x. Le modèle et les premiers appels de k-induction sont alors définis comme suit :

Modèle

$$
\begin{aligned}
&c;\\
&x \ := 0,\ v\\
&u \ := (x = 3)\ ?\ 0\ \ : x + 1\\
&v \ := c\ ?\ x\ \ : u\\
&\mathrm{P} : x < 5
\end{aligned}
$$

$$
\begin{aligned}
&\text{0-ind :} && (x_0 \geq 5)\\
&&& \hookrightarrow SAT\{x_0 = 5\}\\
&\text{1-ind :} && (x_0 < 5) \wedge (u_0 = (x_0 = 3)\ ?\ 0\ \ : x_0 + 1) \wedge (v_0 = c_0\ ?\ x_0\ \ : u_0) \wedge (v_0 \geq 5)\\
&&& \hookrightarrow SAT\{x_0 = 4,\ u_0 = 5,\ c_0 = \bot,\ v_0 = 5\}\\
&\text{2-ind :} && (x_0 < 5) \wedge (u_0 = (x_0 = 3)\ ?\ 0\ \ : x_0 + 1) \wedge (v_0 = c_0\ ?\ x_0\ \ : u_0) \wedge\\
&&& (v_0 < 5) \wedge (u_1 = (v_0 = 3)\ ?\ 0\ \ : v_0 + 1) \wedge (v_1 = c_1\ ?\ v_0\ \ : u_0) \wedge (v_1 \geq 5)\\
&&& \hookrightarrow SAT\{x_0 = 4,\ u_0 = 5,\ c_0 = \top,\ v_0 = 4,\ u_1 = 5,\ c_1 = \bot,\ v_1 = 5\}
\end{aligned}
$$

Il est relativement clair que même si la relation de transition est dépliée un grand nombre de fois, les formules de k-induction seront toujours satisfaisable. En effet, il suffit d'affecter tous les v_i à 4, les u_i à 5, les c_i à $\top$ à l'exception de la dernière transition où c_{k-1} est affecté à $\bot$ et v_{k-1} à 5, et ce, pour toutes profondeurs k. Les contraintes de chemin simple permettent alors d'empêcher ce phénomène en forçant les v_i à être différents.

Les variables de V peuvent être divisées en 3 ensembles : e les variables d'entrées, m les variables mémoires, et t les variables des définitions (temporaires). Avec l'ajout des contraintes de chemin simple, la formule de la k-induction devient :

$$k\text{-induction} = \bigwedge_{i=0}^{k-1} \big[P(V_i) \wedge T(V_i, V_{i+1})\big] \wedge \bigwedge_{0 \leq i < j < k} \big(m_i \neq m_j\big) \wedge Bad(V_k)$$

Ces contraintes génèrent beaucoup de clauses, même lorsque l'introduction de variables est permise. Eén et Sörensson [2003b] ont alors proposé de générer

5. Seules les variables représentant les mémoires sont considérées.

ces contraintes *à la demande*. Autrement dit, lorsque le solveur fournit une affectation des variables satisfaisant une formule de k-induction, celle-ci est examinée. Ainsi, s'il existe deux états identiques dans l'affectation fournie, une contrainte est ajoutée et le solveur peut reprendre sa recherche.

Exemple 2.5 (Ajout des contraintes de simple chemin sur l'exemple 2.4)**.** Dans l'exemple précédent, lorsque le solveur retourne le modèle satisfaisant la formule 2-ind, ce modèle est maintenant examiné.

$$\begin{aligned}\text{2-ind}: \quad & (x_0 < 5) \wedge (u_0 = (x_0 = 3)\ ?\ 0\ : x_0 + 1) \wedge (v_0 = c_0\ ?\ x_0\ : u_0) \wedge \\ & (v_0 < 5) \wedge (u_1 = (v_0 = 3)\ ?\ 0\ : v_0 + 1) \wedge (v_1 = c_1\ ?\ v_0\ : u_0) \wedge v_1 \geq 5 \\ & \hookrightarrow \text{SAT}\{x_0 = 4,\ u_0 = 5,\ c_0 = \top,\ v_0 = 4,\ u_1 = 5,\ c_1 = \bot,\ v_1 = 5\}\end{aligned}$$

Comme $x_0 = v_0$ et que v_0 est équivalent à x_1 une contrainte est ajoutée, et le solveur est relancé :

$$\begin{aligned}\text{2-ind'}: \quad & (x_0 < 5) \wedge (u_0 = (x_0 = 3)\ ?\ 0\ : x_0 + 1) \wedge (v_0 = c_0\ ?\ x_0\ : u_0) \wedge \\ & (v_0 < 5) \wedge (u_1 = (v_0 = 3)\ ?\ 0\ : v_0 + 1) \wedge (v_1 = c_1\ ?\ v_0\ : u_0) \wedge \\ & v_1 \geq 5 \wedge x_0 \neq v_0 \\ & \hookrightarrow \text{UNSAT}\end{aligned}$$

La formule devient alors insatisfaisable, et la propriété est vérifiée.

Chapitre 3

Relation entre model checker et solveur

Cette partie est organisée de la manière suivante : nous proposons d'abord une courte introduction permettant de synthétiser l'approche proposée dans ce chapitre, en la contextualisant par rapport à l'état de l'art et aux approches usuelles du domaine. Nous proposons ensuite une liste des travaux les plus importants en lien avec notre approche. Enfin, nous décrivons les méthodes que nous avons mises en place.

3.1 Vue d'ensemble

Comme nous l'avons déjà en partie évoqué dans le chapitre précédent, la résolution des problèmes de model checking à l'aide de solveurs SAT [1] peut se résumer en 4 étapes majeures :

1. Simplification de la représentation du système donnée en entrée, *e.g.*, optimisation de l'AIG
2. Encodage en CNF du modèle en fonction de l'algorithme utilisé, *e.g.*, BMC, IC3
3. Simplification de la formule CNF
4. Résolution à l'aide d'un solveur SAT

Les étapes 1 et 2 sont parfois effectuées dans un ordre un peu différent. Par exemple pour BMC, une première approche, la plus fréquemment utilisée, consiste à simplifier le modèle, puis à le transformer en clauses lors du dépliage de la relation de transition. Une seconde approche, plus rare, consiste à simplifier le modèle, puis à d'abord déplier l'AIG, pour ensuite à appliquer les simplifications sur cet AIG déplié, et enfin à le transformer en clauses. Les

1. L'approche reste très similaire pour les autres approches basé sur des décisions de procédures, *e.g.*, BDD ou SMT.

étapes 3 et 4 sont elles aussi parfois entrelacées dans certains solveurs : les méthodes de simplification de la CNF sont effectuées avant et pendant la résolution, ce qui est généralement appelé *inprocessing* (Järvisalo *et al.* [2012b]).

Deux grands axes d'amélioration se dessinent alors :

1. Une approche haut niveau (étapes 1 et 2), où l'objectif est d'améliorer les algorithmes de model checking existants (*e.g.*, les CTGs (*Counterexamples To Generalization*) pour PDR), ou de proposer de nouveaux algorithmes (*e.g.*, IC3), mais également d'améliorer l'étape de simplification des modèles (*e.g.*, les techniques de prétraitement développées par l'équipe d'`ABC`, Brayton et Mishchenko [2010]). Dans cette approche, le solveur est généralement considéré comme une boîte noire fournissant l'ensemble des fonctionnalités suivantes :
   - production d'un modèle, *i.e.*, une affectation des variables lorsque la formule est satisfaisable, utilisée notamment pour fournir les contre-exemples et contre-exemples d'induction.
   - résolution incrémentale, notamment avec les littéraux d'activation, très fréquemment utilisée pour les approches utilisant BMC, k-induction ou PDR.
   - production d'une preuve d'insatisfaisabilité, utilisée par exemple pour la génération d'interpolants ou pour la généralisation dans PDR.
2. Une approche bas niveau (étapes 3 et 4), où l'origine du problème initial n'est pas exploitée explicitement, le point de départ étant la CNF. L'objectif est alors d'améliorer les performances du solveur quelque soit le problème donné, en simplifiant les formules CNF, en améliorant les structures de données, en améliorant ou proposant de nouveaux composants (*e.g.*, apprentissage, redémarrage, LBD), etc.

À noter qu'il s'agit là d'une découpe volontairement grossière, bien que les deux approches soient légèrement différentes, les deux communautés (Model Checking / SAT) ont un objectif commun et restent très connectées. Il existe notamment certaines approches qui visent à retrouver à partir de la CNF la sémantique du problème original (*e.g.*, les portes logiques (Fu et Malik [2007]) ou les contraintes de cardinalité (Biere *et al.* [2014])), afin de proposer des méthodes plus performantes pour certains problèmes.

À travers nos observations, nous avons constaté ces deux points suivants :

1. À l'exception de la résolution incrémentale, dans la grande majorité des approches, la structure du problème n'est pas exploitée.
2. Le dialogue entre les différentes étapes est toujours à sens unique, à savoir, du modèle vers le solveur.

Autrement dit, nous pensons qu'une interconnexion plus importante entre le model checker et le solveur SAT pourrait être bénéfique. Nous avons donc travaillé autour d'un ensemble d'idées permettant de remédier, au moins en partie, à ce cloisonnement :

1. Utilisation de la structure du problème : effectuer les simplifications aux endroits les plus appropriés, différentes approches de dépliage de la relation de transition pour profiter au mieux de la résolution incrémentale, duplication des clauses lorsque la vérification est effectuée avec des algorithmes qui déplient la relation de transition, étude sur la relation entre la profondeur des variables, les communautés, les clauses apprises et le score LBD.

2. Dialogue entre le solveur SAT et le model checker : extraire les déductions faites par le solveur de façon à les transférer au niveau du modèle et à les exploiter lors de la simplification du modèle dans un contexte incrémental.

Première étape : conserver un modèle déplié à haut niveau — Comme nous souhaitons instaurer un dialogue simple entre le solveur et le model checker, il nous faut conserver une représentation homogène entre haut et bas niveau. Or, comme évoqué précédemment, la plupart des approches qui font de l'induction temporelle commencent par simplifier le modèle en entrée (règle de réécriture, etc.), puis le déplient tout en le transformant en CNF. Autrement dit, le modèle haut niveau déplié n'est pas conservé. Ici, nous proposons de conserver cette représentation du modèle déplié à haut niveau. Le déroulement détaillé de l'approche est le suivant : le modèle donné en entrée est simplifié, puis ce modèle simplifié est déplié et conservé, ensuite une nouvelle étape d'optimisation est appliquée sur ce modèle déplié, et enfin cette représentation dépliée et simplifiée est transformée en CNF. L'exemple suivant illustre la différence entre ces deux approches :

Exemple 3.1 (Conservation du modèle déplié)**.** Soit un modèle constitué de a : une entrée, x : une mémoire, et u : une définition.

Modèle
a;
$x := 0,\ u$
$u := a \wedge x$
PO : $\neg x$

La première approche consiste à transformer le problème en CNF lors du dé-

pliage de la relation de transition :

$$\begin{aligned}
\text{Init} &: \neg x_0 \\
\text{Def} &: \neg u_0 \vee a_0 \\
&\quad\ \neg u_0 \vee x_0 \\
&\quad\ u_0 \vee \neg a_0 \vee \neg x_0 \\
\text{Bad} &: [x_0] \\
\text{Remplacer} &: x_1 \leftrightarrow u_0 \\
\text{Def} &: \neg u_1 \vee a_1 \\
&\quad\ \neg u_1 \vee u_0 \\
&\quad\ u_1 \vee \neg a_1 \vee \neg u_0 \\
\text{Bad} &: [u_0]
\end{aligned}$$

Dans cette approche, c'est le solveur qui gère les simplifications liées aux contraintes des états initiaux. La seconde approche déplie d'abord le modèle, propage les contraintes des états initiaux, puis si nécessaire transforme le modèle déplié en CNF :

a_0;
$x_0 := 0$
$u_0 := a_0 \wedge x_0 (\Leftrightarrow 0)$
[Le modèle se réduit à des constantes, le solveur n'est pas utilisé.]
a_1;
$x_1 := u_0 (\Leftrightarrow 0)$
$u_0 := a_1 \wedge x_1 (\Leftrightarrow 0)$
[Le modèle se réduit à des constantes, le solveur n'est pas utilisé.]

Cette seconde approche permet également de conserver une représentation homogène entre le modèle déplié haut niveau et la formule CNF ce qui nous permettra d'établir un dialogue entre le model checker et le solveur SAT. À noter néanmoins que cette approche est plus coûteuse en mémoire, car les définitions dépliées sont conservées, et ce, même après l'ajout des clauses correspondantes à ces définitions dans le solveur.

Première barrière : la simplification des CNFs — Des techniques de simplification des formules CNF permettent de simuler les simplifications haut niveaux, notamment l'élimination de variables et les clauses « bloquées » (*blocked clauses*, Järvisalo *et al.* [2012a], décrit dans la partie 3.2.3). Cela présente deux avantages majeurs : 1. le solveur conserve de bonnes performances sur les formules *mal* encodées, et 2. ces simplifications peuvent être effectuées lors de

l'*inprocessing*. Généralement, il est cependant moins coûteux d'appliquer ces simplifications à haut niveau lorsque cela est possible, et cela devient primordial lorsque les modèles sont de grandes tailles. Par exemple, il est très peu coûteux de calculer le COI au niveau du modèle mais cela est beaucoup moins aisé au niveau de la CNF. De plus, il y a des avantages à effectuer certaines simplifications au niveau du modèle, par exemple l'élimination de variables bornée (BVE, *Bounded Variable Elimination*) par distribution devient BVE par substitution (définition) comme nous le verrons par la suite. Enfin, de façon général la simplification des CNFs ne se combine pas bien avec la résolution incrémentale (nous donnons des détails dans la suite). Néanmoins, tout n'est pas faisable à haut niveau sans étendre l'expressivité de la représentation en AIG, qui se limite en l'état à un ensemble de définitions correspondant à un « et » logique binaire et ses 2 opérandes. Il n'est par exemple pas possible d'éliminer une entrée du modèle au niveau de l'AIG alors que c'est aisé au niveau de la CNF.

Pour résumer, le problème qui se pose est le suivant : les méthodes de simplification des CNFs sont très performantes, et plus particulièrement l'élimination de variables qui est une étape quasi-obligatoire pour avoir une solution globale efficace. Cependant, cette élimination ne se combine pas bien avec la résolution incrémentale et n'est pas compatible avec notre objectif de conserver une représentation homogène entre le modèle déplié et la CNF. En effet, certaines variables présentes à haut niveau sont éliminées au niveau de la CNF.

Ainsi, pour conserver l'élimination de variables dans notre approche, nous proposons d'éliminer les variables uniquement au niveau du modèle. Cela nous permet donc de ne plus le faire au niveau du solveur, et ainsi nous gardons une représentation parfaitement homogène entre le modèle et le solveur, à savoir, une variable du modèle a toujours une variable équivalente dans la CNF et réciproquement. Par conséquent, nous pouvons partager simplement des informations du solveur vers le model checker.

Extraction et exploitation des déductions — Avec une représentation homogène entre le haut et le bas niveau, nous proposons une stratégie simple pour exploiter les déductions du solveur SAT (*i.e.*, utiliser des clauses apprises au niveau du modèle). Nous avons décidé d'extraire les assertions (clauses unitaires) et les équivalences (deux clauses binaires contenant les mêmes variables) apprises par le solveur et de les importer dans le model checker afin de simplifier la représentation haut niveau. Cela peut être utile dans un contexte incrémental, à savoir, avant chaque appel au solveur, les informations qui ont été déduites lors des appels précédents sont importées avant la simplification, mais cela pourrait également être effectué comme un inprocessing haut niveau.

Duplication — Nous avons voulu retravailler une idée déjà proposé au début de BMC par Strichman [2000, 2001, 2004], qui consiste à exploiter la structure

symétrique de la formule en dupliquant les clauses. Plus récemment, cette technique a été étendue à l'induction temporelle par Yin *et al.* [2014]. Néanmoins, il y a généralement trop de clauses dupliquées ayant pour effet de ralentir le solveur. Nous proposons pour l'induction temporelle de limiter la duplication au niveau du model checker, ce qui permet de réduire le nombre de clauses dupliquées et par la même occasion d'avoir une méthode qui ne modifie pas l'algorithme initial et par conséquent facilite la mise en œuvre. Puis, pour limiter encore le nombre de clauses, nous proposons de sélectionner les clauses à dupliquer, en se basant sur la taille et le score LBD des clauses.

Sens de dépliage — Nous avons ensuite voulu étoffer un travail amorcé par Eén et Sörensson [2003b] sur le dépliage de la relation de transition pour l'induction temporelle. En effet, le sens dans lequel les transitions sont dépliées peut avoir un impact sur la résolution dans un contexte incrémental. Nous avons alors d'abord effectué une expérimentation comparative afin d'observer la différence entre un dépliage vers l'avant et vers l'arrière, puis nous avons proposé une méthode hybride pour tenter de profiter de l'avantage du dépliage dans les deux sens.

3.1.1 Organisation

Nous passons d'abord en revue les travaux connexes pour chacune des étapes énoncées dans la partie ci-dessus. Puis, nous décrivons l'élimination de variables au niveau du modèle dans la partie 3.3. Dans la partie 3.4, nous indiquons la technique d'extraction et d'exploitation de déductions que nous avons mise en place. Ensuite, nous présentons la technique de duplication des clauses que nous avons définie, dans la partie 3.5. Pour finir, nous présentons les différentes stratégies de dépliage que nous avons expérimentées dans la partie 3.6.

3.2 Travaux connexes

3.2.1 Simplification du modèle

La simplification des modèles est un domaine très large, qui englobe par exemple toutes les optimisations faites par les compilateurs, l'analyse de la valeur des variables par interprétation abstraite, ... C'est pourquoi, comme pour l'ensemble de ce document, nous nous limitons aux AIGs et aux simplifications qui sont couramment appliquées sur ces derniers.

— Brummayer et Biere [2006] fournissent un sous-ensemble des simplifications locales usuellement appliquées, comme la contradiction ($a \wedge \neg a \Leftrightarrow$

$\bot$), ou l'idempotence ($(a \wedge b) \wedge a \Leftrightarrow a \wedge b$). *Local* signifie que les règles de simplification se limitent aux fils et petit-fils du nœud considéré, *i.e.*, 2 niveaux. En outre, ce sous-ensemble de règles a la propriété de ne pas accroître le nombre total de nœuds de l'AIG.

— Mishchenko *et al.* [2006] proposent des techniques inspirées de l'optimisation des circuits pour la simplification des modèles. Les simplifications sont également locales, *e.g.*, énumération des coupes des nœuds de taille maximale 4. L'idée consiste à utiliser une table pré-calculée de toutes les combinaisons des fonctions à 4 entrées (recherche rapide grâce à un encodage astucieux de la table de vérité). Puis, pour chaque coupe de chaque nœud, les structures fonctionnellement équivalentes sont récupérés dans la table. Ensuite, à l'aide d'une méthode efficace de remplacement de l'arbre courant par ceux de la table, toutes les structures d'arbres équivalentes sont testées. Enfin, s'il existe une meilleure configuration, *i.e.*, qui réduit le nombre total de nœuds de l'AIG, alors le sous-arbre courant est remplacé par celle-ci. Ils entremêlent également cette technique avec d'autres simplifications, pour la plupart également issues de l'optimisation des circuits : par exemple l'équilibrage (Cortadella [2003]), une technique dont l'objectif est de réduire le délai maximum d'un circuit. Ces techniques changent la structure de l'arbre, tout en préservant son aspect fonctionnel. Cela permet généralement de trouver des équivalences structurelles supplémentaires ou de réduire encore plus le nombre de nœuds de l'AIG avec la méthode décrite précédemment.

3.2.2 Encodage CNF

Toute formule propositionnelle quelconque peut être transformée en forme normale conjonctive. Cependant, la transformation peut mener à une explosion de la taille de la formule. De très nombreux travaux proposent des encodages pour transformer une formule propositionnelle en CNF avec différentes propriétés, *e.g.*, une complexité linéaire ou un nombre de clauses minimal. Nous passons rapidement en revue certaines des méthodes qui peuvent être utilisées pour transformer un AIG en CNF :

— Tseitin [1968] introduit des variables intermédiaires pour toutes sous formules (définitions), ce qui permet une transformation linéaire en CNF.

— Plaisted et Greenbaum [1986] ont montré qu'il était possible d'omettre une partie de la bi-implication d'une définition si celle-ci n'est utilisé que dans une seule polarité.

— Boy de la Tour [1992] propose de sélectionner le sous-ensemble des formules à renommer, *i.e.*, pour lesquelles une variable est introduite, de façon à minimiser le nombre de clauses générées.

— Velev [2004] fusionne les portes logiques, à partir d'une liste de configurations, au moment de la transformation en CNF pour réduire le nombre de variables et de clauses.

— Een *et al.* [2007] proposent de nouveau l'utilisation de techniques inspirées de l'optimisation des circuits pour l'encodage en CNF (*Technology Mapping*). Ils calculent l'ensemble des nœuds internes pour lesquels ils vont générer une variable en utilisant un algorithme d'optimisation, où l'objectif est de minimiser le nombre de clauses générées.

— Manolios et Vroon [2007]; Chambers *et al.* [2009] introduisent les *NICE DAG* (Negation, If-then-else, Conjunction, Equivalence), une structure normalisée pour représenter les expressions booléennes. L'ajout de variables intermédiaires, lors de la transformation en CNF, est basé sur une heuristique cherchant cette fois à minimiser le nombre de littéraux.

3.2.3 Simplification des formules CNF

La dernière étape avant la résolution consiste à simplifier les formules CNF.

— Subbarayan et Pradhan [2004] proposent d'utiliser l'élimination (bornée) de variables par distribution des clauses en étape de pré-traitement.

— Eén et Biere [2005] introduisent 3 techniques de simplification : subsomption, résolution par subsomption, et élimination (bornée) de variables par substitution. La subsomption est simplement l'inclusion d'une clause dans une autre. La résolution par subsomption est l'inclusion d'une clause dans une autre mais où un seul littéral est dans la polarité inverse. L'élimination par substitution est plus efficace que celle par distribution car elle génère moins de clauses mais plus difficile à mettre en place car il faut détecter des configurations de clauses. `SatElite` a mis en place cette détection et seulement pour les définitions du type $x = a_1 \wedge \ldots \wedge a_n$, ce qui fonctionne très bien lorsque les définitions de l'AIG sont encodées avec Tseitin [1968]). Biere [2009] a été plus loin en détectant les définitions de types ITEs ou XORs.

— Les clauses bloquées (*blocked clauses*, Kullmann [1999]; Järvisalo *et al.* [2010, 2012a]) sont des clauses qui peuvent être éliminée sans changer la satisfaisabilité de la formule. Une clause est dite bloquée si elle contient un littéral qui la bloque. Un littéral bloque une clause si toutes les résolutions entre cette clause et toutes les clauses contenant la négation de ce littéral sont des tautologies. Cette élimination des clauses bloquées, combinée avec l'élimination de variables, permet de simplifier les formules CNF, comme le ferait certaines simplifications qui se font habituellement au niveau du modèle, *e.g.*, Plaisted-Greenbaum ou le cône d'influence. C'est donc très utile lorsque la CNF a été *mal* encodée.

3.2.4 Résolution

Ici, nous ne parlons pas des techniques internes aux solveurs, mais plutôt de l'utilisation des informations haut niveaux du modèle pour aider, *i.e.*, accélérer la résolution dans le solveur.

— Guerra e Silva *et al.* [1999] proposent de fournir des informations structurelles haut niveaux sur les variables pour aider le solveur lors de la résolution, *e.g.*, modification de l'heuristique de décisions.

— Ganai *et al.* [2002] proposent un algorithme hybride qui combine deux solveurs SAT l'un étant basé sur les CNF et l'autre sur les circuits.

— Strichman [2000, 2001, 2004] propose 3 techniques pour BMC : ordre statique des décisions basé sur le modèle, réutilisation des clauses apprises (résolution incrémentale) et duplication des clauses apprises.

— Eén et Sörensson [2003b] généralisent la résolution incrémentale pour la k-induction avec l'utilisation des littéraux d'activation. Ils proposent également l'ajout des contraintes de simple chemin à la demande.

3.3 Élimination de variables

L'élimination de variables par distribution est une méthode très efficace de prétraitement des formules CNFs lorsqu'elle est limitée aux variables qui n'augmentent pas le nombre total de clauses de la formule lors de leur élimination, afin d'éviter l'explosion combinatoire de la méthode. Elle est rapidement devenue une étape incontournable avant la résolution. La quasi-totalité des solveurs effectue cette élimination. Il est donc nécessaire de la prendre en compte dans notre approche. Néanmoins, cette méthode élimine des variables et transforme donc la formule, *i.e.*, elle n'est plus équivalente à la formule originale. Cette transformation préserve tout de même la satisfaisabilité. On dit alors que la formule originale et la version simplifiée sont équisatisfaisables. Comme nous souhaitons instaurer un dialogue simple du solveur vers le model checker, nous ne pouvons conserver cette étape en l'état, car les deux ne communiqueraient plus sur le même ensemble de variables. Nous proposons d'effectuer uniquement l'élimination au niveau du modèle afin de conserver une représentation équivalente entre les deux niveaux. Cela permet de bénéficier des gains apportés par l'élimination de variables, tout en autorisant la résolution incrémentale.

Dans cette partie, nous commençons par rappeler rapidement le principe de l'élimination de variables au niveau du solveur, et précisons les limitations que cela implique dans un contexte incrémental. Nous présentons aussi les approches qui ont alors été proposées pour tenter de remédier à ce problème. Puis, nous donnons un exemple complet pour bien illustrer et mettre en avant

les différences entre l'élimination aux différents niveaux. Ensuite, nous précisons la faisabilité de l'élimination en fonction du type des variables. Enfin, nous décrivons l'approche que nous avons mise en place.

3.3.1 Principe

L'élimination d'une variable par distribution consiste simplement à effectuer toutes les résolutions possibles sur cette variable, puis à retirer de la formule toutes les clauses contenant cette variable, et enfin à ajouter toutes les clauses produites par résolution dans l'étape précédente. Pendant l'étape de pré-traitement dans le solveur SAT, l'élimination de variables est usuellement appliquée sur toutes les variables pour lesquelles le nombre de clauses de la formule n'augmente pas lorsque celles-ci sont éliminées. On parle alors d'élimination de variables bornée (BVE, *Bounded Variable Elimination*). A noter que le résultat de l'élimination dépend de l'ordre dans lequel les variables sont éliminées. Par exemple dans `Minisat` (et donc dans `Glucose`), la variable x ayant le plus petit score $|x| \times |\neg x|$ est éliminée en priorité, avec $|x|$ et $|\neg x|$ correspondant respectivement aux nombres d'occurrences des littéraux x et $\neg x$. A noter également qu'il s'agit là de l'élimination de variable par distribution de clauses (Subbarayan et Pradhan [2004]), et non l'élimination de variable par substitution, appelée aussi élimination de variable par définition (`SatElite`, Eén et Biere [2005]).

De façon générale, les méthodes de simplifications des formules CNF, qui transforment la formule originale en une formule qui n'est plus équivalente mais équisatisfaisable, nécessitent certaines précautions lorsqu'elles sont effectuées dans un contexte incrémental. Concernant l'élimination de variables, les variables qui peuvent être réintroduites lors d'appels successifs ne doivent en aucun cas être éliminées, sous peine de changer la satisfaisabilité de l'instance. Certains choisissent alors soit de sacrifier le prétraitement en faveur de la résolution incrémentale, soit de sacrifier l'approche incrémentale et de simplifier la formule à l'aide de techniques de prétraitement des CNFs.

Dans notre cas précis, lors de la résolution de formules encodant des problèmes BMC ou d'induction temporelle, à chaque nouveau dépliage de la relation de transition, des variables ayant potentiellement été éliminées peuvent être réintroduites. Il faut donc généralement choisir entre la résolution incrémentale et l'élimination. Néanmoins, lorsqu'il est possible de connaître à l'avance les variables qui seront réintroduites dans les appels suivants au solveur, l'élimination de certaines variables peut être proscrite, ce qui est généralement appelé : *geler* une variable. Par exemple, pendant les itérations d'induction temporelle avec contraintes de chemin simple, toutes les variables mémoires sont gelées pour éviter qu'elles soient éliminées. Kupferschmid *et al.* [2011] proposent quant à eux de geler seulement les variables mémoires du dernier cycle pour BMC. Cette méthode permet alors de profiter de la résolution

incrémentale, tout en profitant d'une grande partie des bénéfices apportés par l'élimination.

Cependant, il n'est pas toujours possible de déterminer à l'avance les variables qui seront réintroduites, notamment lorsque les méthodes d'optimisation sont appliquées au niveau du modèle déplié. Voici par exemple quelques simplifications au niveau du modèle déplié qui ne sont pas effectuables si l'élimination de variables est activée au niveau du solveur :

— L'équivalence structurelle (et fonctionnelle) entre différents cycles ; supposons par exemple qu'il y ait une définition $x = a \wedge b$ au cycle 2, et une définition $y = a \wedge b$ au cycle 3, alors y ne peut pas être remplacé par x car x a potentiellement été éliminé lors de l'appel précédent au solveur.

— Le k-COI, qui équivaut à effectuer le COI sur le modèle déplié ; Soit $x = a \wedge b$ et $y = x \wedge c$ deux définitions dépliées au cycle 2 avec x qui appartient au 2-COI et y appartient au 3-COI mais pas au 2-COI, alors x peut être éliminé avant l'introduction de y.

À noter qu'il existe également des techniques qui profitent de la résolution incrémentale et du prétraitement sans connaître à l'avance les variables qui seront réintroduites. Nadel *et al.* [2012, 2014], notamment, proposent d'annuler l'élimination, *i.e.*, de reconstruire la formule telle qu'elle l'était avant l'élimination de la variable à réintroduire, ou de rééliminer la variable qui doit être réintroduite, à la volée. Ils proposent également une technique qui permet de considérer les littéraux d'activation non plus comme des décisions mais comme des clauses unitaires, ce qui permet de réduire encore plus la formule (en combinant cette méthode avec la propagation et l'élimination de variables). Ces techniques sont rendues possibles en conservant un historique des éliminations effectuées, des clauses propagées à l'aide d'un littéral d'activation, etc.

3.3.2 Exemple illustratif

Dans cette partie, nous illustrons sur un exemple le résultat de l'élimination par distribution d'une variable en fonction de la représentation du modèle. L'élimination peut être effectuée à différents niveaux, lorsque le modèle est encore sous forme de définitions ou lorsqu'il est sous forme de clauses, mais aussi à différents moments, à savoir, avant ou après dépliage de la relation de transition.

Nous commençons d'abord par donner le modèle d'entrée de notre exemple sous forme de définitions et son équivalent sous forme clausale. Ce modèle est

composé de deux entrées a, b, d'une mémoire m et de trois définitions x, y, z :

Modèle		Forme clausale
a, b (entrées)		
$m := \bot,\ y$		
$x := a \wedge m$	$\rightarrow$	$\neg x \vee a$ $\neg x \vee m$ $x \vee \neg a \vee \neg m$
$y := b \wedge \neg x$	$\rightarrow$	$\neg y \vee b$ $\neg y \vee \neg x$ $y \vee \neg b \vee x$
$z := b \wedge x$	$\rightarrow$	$\neg z \vee b$ $\neg z \vee x$ $z \vee \neg b \vee \neg x$

Ensuite, nous effectuons deux itérations de dépliage du modèle ci-dessus avec la contrainte des états initiaux, comme lors du dépliage pour BMC, *i.e.*, la mémoire est initialisée à sa valeur initiale (mais non propagée). Comme précédemment, nous donnons les deux représentations, à savoir, le modèle déplié sous forme de définitions et son équivalent sous forme clausale :

Modèle déplié	Forme clausale
$m_0 := \bot$	$\neg m_0$
$x_0 := a_0 \wedge m_0$	$\neg x_0 \vee a_0$ $\neg x_0 \vee m_0$ $x_0 \vee \neg a_0 \vee \neg m_0$
$y_0 := b_0 \wedge \neg x_0$	$\neg y_0 \vee b_0$ $\neg y_0 \vee \neg x_0$ $y_0 \vee \neg b_0 \vee x_0$
$z_0 := b_0 \wedge x_0$	$\neg z_0 \vee b_0$ $\neg z_0 \vee x_0$ $z_0 \vee \neg b_0 \vee \neg x_0$
$m_1 := y_0$	
$x_1 := a_1 \wedge y_0$	$\neg x_1 \vee a_1$ $\neg x_1 \vee y_0$

	$x_1 \vee \neg a_1 \vee \neg y_0$
$y_1 := b_1 \wedge \neg x_1$	$\neg y_1 \vee b_1$
	$\neg y_1 \vee \neg x_1$
	$y_1 \vee \neg b_1 \vee x_1$
$z_1 := b_1 \wedge x_1$	$\neg z_1 \vee b_1$
	$\neg z_1 \vee x_1$
	$z_1 \vee \neg b_1 \vee \neg x_1$

L'élimination peut alors être effectuée aux quatre niveaux suivants : modèle, modèle sous forme clausale, modèle déplié, modèle déplié sous forme clausale. Pour chacun de ces niveaux, nous indiquons les clauses qui seront générées lorsque l'élimination de x est effectuée. Puis, nous discutons les différences entre ces approches.

Élimination de x au niveau du modèle

Modèle	Après élimination de x	Clauses
a, b (entrées)	a, b (entrées)	
$m = \bot,\ y$	$m = \bot,\ y$	
		$\neg y \vee b$
$x = a \wedge m$		$\neg y \vee \neg a \vee \neg m$
		$y \vee \neg b \vee a$
$y = b \wedge \neg x$	$y = b \wedge (\neg a \vee \neg m)$	$y \vee \neg b \vee m$
		$\neg z \vee b$
$z = b \wedge x$	$z = b \wedge a \wedge m$	$\neg z \vee a$
		$\neg z \vee m$
		$z \vee \neg b \vee \neg a \vee \neg m$

Élimination de x au niveau du modèle sous forme clausale

Modèle	Clauses	Après élimination de x
	$\neg x \vee a$	$\neg y \vee \neg z$
$x = a \wedge m$	$\neg x \vee m$	$y \vee \neg b \vee z$
	$x \vee \neg a \vee \neg m$	
	$\neg y \vee b$	$\neg y \vee b$
$y = b \wedge \neg x$	$\neg y \vee \neg x$	$\neg y \vee \neg a \vee \neg m$
	$y \vee \neg b \vee x$	$y \vee \neg b \vee a$
		$y \vee \neg b \vee m$
	$\neg z \vee b$	$\neg z \vee b$
$z = b \wedge x$	$\neg z \vee x$	$\neg z \vee a$
	$z \vee \neg b \vee \neg x$	$\neg z \vee m$
		$z \vee \neg b \vee \neg a \vee \neg m$

Élimination de x au niveau du modèle déplié

Modèle après élimination de x_0 et x_1	Clauses
$m_0 = \bot$	$\neg m_0$
	$\neg y_0 \vee b_0$
	$\neg y_0 \vee \neg a_0 \vee \neg m_0$
$y_0 = b_0 \wedge (\neg a_0 \vee \neg m_0)$	$y_0 \vee \neg b_0 \vee a_0$
	$y_0 \vee \neg b_0 \vee m_0$
	$\neg z_0 \vee b_0$
	$\neg z_0 \vee a_0$
$z_0 = b_0 \wedge a_0 \wedge m_0$	$\neg z_0 \vee m_0$
	$z_0 \vee \neg b_0 \vee \neg a_0 \vee \neg m_0$
$m_1 = y_0$	
	$\neg y_1 \vee b_1$
	$\neg y_1 \vee \neg a_1 \vee \neg y_0$
$y_1 = b_1 \wedge (\neg a_1 \vee \neg y_0)$	$y_1 \vee \neg b_1 \vee a_1$
	$y_1 \vee \neg b_1 \vee y_0$
	$\neg z_1 \vee b_1$
	$\neg z_1 \vee a_1$

$z_1 = b_1 \wedge a_1 \wedge y_0$ $\quad\quad$ $\neg z_1 \vee y_0$
$z_1 \vee \neg b_1 \vee \neg a_1 \vee \neg y_0$

Élimination de x au niveau du modèle déplié sous forme clausale

Clauses	Clauses après élimination de x_0 et x_1
$\neg m_0$	$\neg m_0$
$\neg x_0 \vee a_0$	
$\neg x_0 \vee m_0$	$\neg y_0 \vee \neg z_0$
$x_0 \vee \neg a_0 \vee \neg m_0$	$y_0 \vee \neg b_0 \vee z_0$
	$\neg y_0 \vee b_0$
$\neg y_0 \vee b_0$	$\neg y_0 \vee \neg a_0 \vee \neg m_0$
$\neg y_0 \vee \neg x_0$	$y_0 \vee \neg b_0 \vee a_0$
$y_0 \vee \neg b_0 \vee x_0$	$y_0 \vee \neg b_0 \vee m_0$
	$\neg z_0 \vee b_0$
$\neg z_0 \vee b_0$	$\neg z_0 \vee a_0$
$\neg z_0 \vee x_0$	$\neg z_0 \vee m_0$
$z_0 \vee \neg b_0 \vee \neg x_0$	$z_0 \vee \neg b_0 \vee \neg a_0 \vee \neg m_0$
$\neg x_1 \vee a_1$	
$\neg x_1 \vee y_0$	$\neg y_1 \vee \neg z_1$
$x_1 \vee \neg a_1 \vee \neg y_0$	$y_1 \vee \neg b_1 \vee z_1$
	$\neg y_1 \vee b_1$
$\neg y_1 \vee b_1$	$\neg y_1 \vee \neg a_1 \vee \neg y_0$
$\neg y_1 \vee \neg x_1$	$y_1 \vee \neg b_1 \vee a_1$
$y_1 \vee \neg b_1 \vee x_1$	$y_1 \vee \neg b_1 \vee y_0$
	$\neg z_1 \vee b_1$
$\neg z_1 \vee b_1$	$\neg z_1 \vee a_1$
$\neg z_1 \vee x_1$	$\neg z_1 \vee y_0$
$z_1 \vee \neg b_1 \vee \neg x_1$	$z_1 \vee \neg b_1 \vee \neg a_1 \vee \neg y_0$

Discussion

— Les clauses en bleu dans les exemples précédents correspondent à des clauses supplémentaires générées uniquement lorsque l'élimination de x

est effectuée au niveau de la CNF. Ces clauses sont directement impliquées par les autres clauses, et ne sont donc pas nécessairement utiles. Cependant, elles peuvent l'être pour la propagation. L'élimination par substitution (définition) permet notamment de ne pas générer ces clauses additionnelles lors de l'élimination.

— Effectuer l'élimination après le dépliage fournit les mêmes résultats, cependant l'élimination d'une même variable doit être fait pour chaque profondeur.

— Comme évoqué précédemment, l'ordre d'élimination des variables peut entraîner des résultats différents. En effet, dans l'exemple ci-dessus le résultat serait différent si y_0 était éliminé avant. L'ordre d'élimination est donc important même lorsque l'élimination de variables est effectuée à haut niveau et après le dépliage.

— Comme abordé précédemment, l'élimination est effectuée de façon bornée, *i.e.*, de façon à ne pas augmenter le nombre de clauses. Ainsi, si une méthode génère moins de clauses, alors cela permet d'éliminer plus de variables. Par exemple, considérons l'élimination de x pour les 3 définitions suivantes :

$$\begin{aligned} x &= a \wedge b \quad \rightarrow \quad 3 \text{ clauses} \\ y &= x \wedge c \quad \rightarrow \quad 3 \text{ clauses} \\ z &= x \wedge d \quad \rightarrow \quad 3 \text{ clauses} \\ &\downarrow \\ y &= a \wedge b \wedge c \quad \rightarrow \quad 4 \text{ clauses} \\ z &= a \wedge b \wedge d \quad \rightarrow \quad 4 \text{ clauses} \end{aligned}$$

L'élimination de x au niveau du modèle entraîne donc une diminution du nombre de clauses (passage de 9 à 8 clauses). En suivant l'heuristique habituellement mise en œuvre dans les solveurs, la variable x serait donc éliminée. Or, en effectuant l'élimination au niveau du solveur, l'élimination de x entraînerait une augmentation du nombre de clauses (passage de 9 à 10 clauses) et x ne serait donc pas éliminé (en admettant une telle limite fixe).

3.3.3 Type de variables

Dans l'exemple précédent, nous éliminons la variable d'une définition, ce qui se prête bien à l'élimination au niveau du modèle. Cependant, ce n'est le cas ni des entrées ni des mémoires. Nous précisons alors le comportement de l'élimination en fonction du type de variables :

— **Les variables des définitions**, comme nous l'avons vu dans l'exemple précédent, peuvent être éliminées aussi bien au niveau du modèle qu'au

niveau du solveur, même si l'élimination à haut niveau a généralement l'avantage de générer moins de clauses.

— **Les variables mémoires :** Lors du dépliage d'un modèle, ces variables sont généralement virtuelles, *i.e.*, elles n'apparaissent pas dans le modèle déplié ou dans la CNF, elles servent de lien entre les variables du cycle précédent et celles du cycle courant. Par conséquent, une variable mémoire n'est rien d'autre en pratique qu'une référence sur une variable d'une définition ou d'une entrée. À noter qu'une variable mémoire peut aussi référencer une autre mémoire. Ces variables référencées par des mémoires ne sont généralement pas éliminées, que ce soit à haut ou bas niveau, car elles sont nécessaires pour la validité des algorithmes de model checking, notamment pour ajouter les contraintes de chemin simple pour la k-induction. Néanmoins, pour BMC, ces variables référencées par des mémoires peuvent être éliminées à l'exception de celles du dernier cycle, comme proposé par Kupferschmid *et al.* [2011]. Cela ne peut donc pas être fait au niveau du modèle (qu'il soit sous forme de définitions ou sous forme clausale), mais uniquement au niveau du modèle déplié ou de la CNF encodant une relation de transition dépliée. Comme il s'agit juste d'une référence sur une autre variable, l'avantage d'effectuer cette élimination au niveau du modèle déplié lorsqu'il s'agit d'une variable d'une définition est le même que précédemment, à savoir, moins de clauses seront générées. On peut noter cependant qu'effectuer cette élimination sur le modèle haut niveau déplié revient à éliminer des variables qui ont normalement [2] déjà été ajoutées dans le solveur.

— **Les variables d'entrées** sont éliminées de la même façon que les variables des définitions lorsque l'élimination est effectuée sur le modèle sous forme clausale, et qui plus est sans distinction. Cependant, l'élimination d'une variable d'entrée n'est pas *exprimable* dans la représentation sous forme de définition, cela nécessite généralement des implications dans un sens uniquement, comme illustré sur l'exemple 3.2 ci-dessous. Les entrées ne sont donc pas éliminées lorsque l'élimination est effectuée au niveau du modèle. C'est une autre différence majeure avec l'élimination au niveau de la CNF.

Exemple 3.2. x, y sont des définitions ; a, b et c des entrées ; a est

2. Si elles appartiennent au k-COI d'un des cycles précédents.

éliminé.

Modèle	Forme clausale	a éliminé	
	$\neg x \vee a$		
$x = a \wedge b$	$\neg x \vee b$	$\neg x \vee b$	$(x \Rightarrow b)$
	$x \vee \neg a \vee \neg b$	$x \vee \neg y \vee \neg b$	$(y \wedge b \Rightarrow x)$
	$\neg y \vee a$		
$y = a \wedge c$	$\neg y \vee c$	$\neg y \vee c$	$(y \Rightarrow c)$
	$y \vee \neg a \vee \neg c$	$y \vee \neg x \vee \neg c$	$(x \wedge c \Rightarrow y)$

Après l'élimination de a, l'ensemble des clauses n'est plus exprimable sous forme de définitions.

— On peut remarquer que la variable référencée par la propriété (ou le mauvais état) n'est évidemment pas éliminée, tout comme l'ensemble des variables *particulières*, *i.e.*, littéraux d'activation, contraintes, etc.

3.3.4 Élimination des variables à haut niveau

Comme évoqué dans l'introduction de ce chapitre, nous proposons de remplacer entièrement l'élimination de variables bornée par distribution faite au niveau du solveur par une élimination effectuée au niveau du model checker. Cela doit nous permettre de conserver une correspondance entre les variables haut et bas niveau, pour exploiter facilement les déductions faites par le solveur. Il y a néanmoins deux inconvénients majeurs : 1. les variables d'entrées ne sont plus éliminées, et 2. la structure des définitions doit être étendue pour pouvoir conserver les définitions après élimination, ce qui complexifie une partie des structures de données et algorithmes du model checker. Dans la plupart des approches, l'élimination haut niveau est effectuée lors de la génération de la CNF, ils n'ont donc pas à conserver les définitions après élimination.

Dans la suite, nous montrons que cette technique seule, *i.e.*, en désactivant complètement l'élimination au niveau du solveur et sans mettre en place un dialogue entre le model checker et le solveur, ne dégrade pas les performances en terme de nombres d'instances résolues et en temps de résolution. Au contraire, nous observons une légère amélioration des performances dans l'outil que nous avons implémenté.

Extension des définitions

Pour simplifier, nous considérons les modèles comme un ensemble de variables d'entrées, un ensemble de variables mémoires et leur état suivant, ainsi

qu'un ensemble de définitions. La première étape pour permettre cette élimination haut niveau est d'étendre l'expressivité actuelle des définitions [3]. Nous étendons alors les définitions qui sont actuellement limités à un « et » logique binaire, à toutes les expressions qui vérifient cette définition récursive :

$$
\begin{aligned}
def ::= \ & var \\
| \ & \neg var \\
| \ & def \ \wedge \ def \\
| \ & def \ \vee \ def
\end{aligned}
$$

où var est une variable d'entrée, une variable mémoire, ou une variable d'une autre définition.

À noter que les négations ne sont autorisées qu'au niveau des variables. Cela signifie que les négations sont poussées aux feuilles lors de l'élimination. On remarquera également que les constantes ne sont plus acceptées car elles sont propagées. Autrement dit, cela revient à étendre les définitions à toute formule propositionnelle constituée uniquement des opérateurs $\wedge$ et $\vee$, sans constante logique et où les négations sont autorisées seulement sur les variables.

Génération des clauses

L'étape suivante consiste à déterminer comment transformer les définitions en formules CNF, c'est-à-dire générer des clauses à partir de la structure des définitions. Une définition est de la forme $lhs \Leftrightarrow rhs$, où le lhs est simplement une variable (un identifiant), alors que le rhs correspond à une expression vérifiant la définition donnée dans la partie ci-dessus. Ces expressions peuvent également être représentées sous forme d'arbres, dans lesquels chaque feuille est un littéral et chaque nœud interne un opérateur $\wedge$ ou $\vee$. Alors, à l'aide d'un algorithme qui va des feuilles vers la racine (*bottom-up*), les deux ensembles de clauses correspondant aux deux implications de l'équivalence sont calculés. Autrement dit, il faut calculer les ensembles de clauses correspondant à $lhs \Rightarrow rhs$ et $lhs \Leftarrow rhs$. Cela revient donc à calculer $\neg lhs \vee rhs$ et $lhs \vee \neg rhs$. Il suffit alors de calculer les deux ensembles des clauses correspondant à rhs et $\neg rhs$, puis d'ajouter par la suite dans toutes les clauses des deux ensembles, les littéraux $\neg lhs$ et lhs respectivement.

L'algorithme de construction de l'ensemble de clauses pour rhs est le suivant : Chaque feuille est associée à un ensemble composé d'un singleton contenant le littéral de la feuille. Puis, en remontant des feuilles vers la racine, l'ensemble associé à chaque nœud interne est calculé en fonction de l'opérateur correspondant et des deux ensembles de ses fils. Si l'opérateur d'un nœud interne est un « $\wedge$ » alors il suffit de faire l'union des ensembles des fils de ce

3. Les entrées et les mémoires restent identiques.

nœuds. Si l'opérateur est un « $\vee$ », alors l'ensemble associé au nœud correspond au produit cartésien des ensembles de ces fils. En effet, pour transformer $\text{CNF}_1 \vee \text{CNF}_2$ en CNF, il faut distribuer, alors que $\text{CNF}_1 \wedge \text{CNF}_2$ est déjà en CNF.

En suivant le même principe pour construire $\neg rhs$, il suffit de prendre la négation des littéraux pour les feuilles et d'inverser les deux opérations pour les nœuds internes, *i.e.*, effectuer l'union pour « $\vee$ » et le produit cartésien pour « $\wedge$ ».

Nous décrivons cette génération dans l'algorithme 3 ci-dessous. La fonction GENERECLAUSES prend en paramètre le rhs d'une définition et retourne deux ensembles de clauses correspondants aux deux sens de l'implication de la définition auxquels il faut ajouter la variable du lhs.

```
Fonction GENERECLAUSES(def)
    if OPERATEUR(def) == ∅ then                          ▷ def est une feuille
        return {VAR(def)}, {¬ VAR(def)}
    else                                                 ▷ def est un noeud interne
        clsFG, nclsFG := GENERECLAUSES(FILSGAUCHE(def))
        clsFD, nclsFD := GENERECLAUSES(FILSDROIT(def))
        if OPERATEUR(def) == ∨ then
            return clsFG × clsFD, nclsFG ∪ nclsFD
        else if OPERATEUR(def) == ∧ then
            return clsFG ∪ clsFD, nclsFG × nclsFD
```

La fonction OPERATEUR retourne $\emptyset$ si le paramètre est un littéral, et l'opérateur courant sinon, *i.e.*, « $\wedge$ » ou « $\wedge$ ». La fonction VAR retourne la variable du paramètre qui est un littéral. Les fonctions FILSGAUCHE et FILSDROIT retournent respectivement le fils gauche et droit du paramètre, à savoir les opérandes de l'opérateur courant.

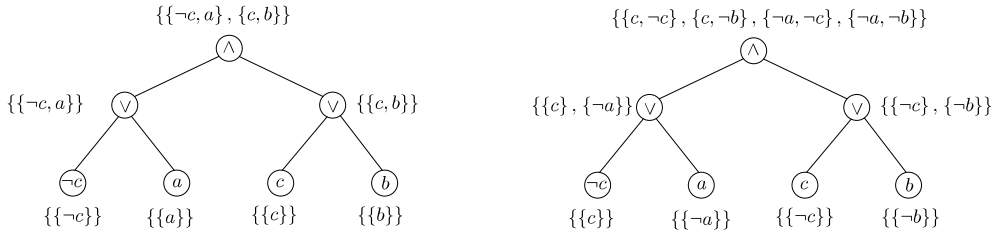
Algorithme 3 – Génération de clauses à partir d'une définition.

Afin de bien illustrer cette génération, nous détaillons sur l'exemple ci-dessous l'application de cet algorithme.

Exemple 3.3. Supposons que l'on cherche à générer les clauses pour la définition suivante : $x = ITE(c, a, b) \Leftrightarrow x = (\neg c \vee a) \wedge (c \vee b)$. Alors, à partir de l'algorithme qui va des feuilles vers la racine, les deux ensembles de clauses qui correspondent à $x \Rightarrow ITE(c, a, b)$ et $x \Leftarrow ITE(c, a, b)$ sont construits, comme illustré dans la figure 3.1.

La figure 3.1a correspond à $x \Rightarrow ITE(c, a, b) \Leftrightarrow \neg x \vee ITE(c, a, b)$. L'algorithme décrit ci-dessus est appliqué, *i.e.*, produit cartésien lorsque l'opérateur du nœud interne est un « $\vee$ » et l'union lorsque c'est un « $\wedge$ ».

La figure 3.1b correspond à $x \Leftarrow ITE(c, a, b) \Leftrightarrow x \vee \neg ITE(c, a, b)$. L'inverse des opérations est donc appliqué, *i.e.*, produit cartésien lorsque l'on rencontre



(a) Génération de clauses ($\Rightarrow$). (b) Génération de clauses ($\Leftarrow$).

FIGURE 3.1 – Génération de clauses pour $x = ITE(c, a, b)$.

un $\wedge$ et l'union lorsque c'est un $\vee$. La polarité de chaque littéral doit également être inversée.

En ajoutant x et $\neg x$ aux ensembles calculés, l'ensemble des clauses suivantes sont obtenues pour la définition $x = ITE(c, a, b)$:

$$\neg x \vee \neg c \vee a \qquad x \vee c \vee \neg c \qquad x \vee \neg c \vee \neg a$$
$$\neg x \vee c \vee b \qquad x \vee c \vee \neg b \qquad x \vee \neg a \vee \neg b$$

La clause en rouge est une tautologie. Celle en bleue est optionnelle, *i.e.*, elle n'est pas nécessaire pour la validité de l'algorithme mais elle améliore la propagation (voir la partie 5 de Een *et al.* [2007]). À noter qu'avec cette méthode de génération, il manque la clause $\neg x \vee a \vee b$, qui est également optionnelle mais généralement ajoutée lors de l'encodage des ITEs.

Il est important de préciser que cet algorithme est applicable en pratique car il ne génère pas un nombre de clauses exorbitant, en raison du critère d'élimination que nous définissons ci-dessous.

Critère d'élimination

Dans la partie précédente, nous avons montré comment générer les clauses pour une définition. Maintenant, nous devons définir quand et comment effectuer l'élimination des variables des définitions. Le point d'entrée de l'algorithme d'élimination est un ensemble de définitions de la forme $lhs := rhs$. L'objectif est alors de minimiser le nombre de définitions, *i.e.*, éliminer les variables lhs, en substituant les occurrences des variables lhs dans les autres définitions par leur rhs. Néanmoins, il faut veiller à ne pas augmenter considérablement le nombre total de clauses générées par l'ensemble des définitions. Ainsi, de la même façon que dans le solveur SAT, nous choisissons d'éliminer une variable si le nombre total de clauses générées par l'ensemble des définitions, lorsque cette variable est remplacée par sa définition, est inférieur ou égal au nombre total en gardant sa définition.

L'idée générale est similaire à celle appliquée dans les solveurs SAT, et se résume en 4 étapes :

1. Pour une définition d de la forme $d_{lhs} := d_{rhs}$, récupérer toutes les définitions qui référencent d_{lhs}, à savoir, toutes les définitions qui contiennent dans leur rhs la variable d_{lhs}.
2. Calculer le nombre de clauses générées par la définition d et toutes les clauses qui la référencent.
3. Calculer le nombre de clauses générées par toutes les définitions qui référencent d_{lhs} tout en substituant, à la volée, d_{lhs} par d_{rhs}.
4. Si le nombre de clauses calculées à l'étape 3 est inférieur à celui de l'étape 2, effectuer l'élimination de d.

Il est possible de calculer à la volée les clauses générées par une définition qui contient une variable que l'on souhaite éliminer. Par exemple, si l'on souhaite éliminer la définition $x = a \wedge b$ référencée par $y = \neg x \wedge c$, on est capable de calculer le nombre de clauses générées par la définition y tout en remplaçant x par sa définition dans y. Il suffit d'abord de calculer les ensembles de clauses pour la définition x et de les fournir lors du calcul des ensembles de clauses de y à chaque occurrence de x. En d'autres termes, on remplace virtuellement toute occurrence de x dans y par sa définition.

À noter que nous avons développé deux méthodes : une précise qui construit les clauses et qui est donc capable de calculer le nombre exact car elle peut supprimer les clauses triviales, et une moins précise qui ne construit pas les clauses mais qui compte juste leur nombre et leur taille max. La méthode qui construit les clauses est évidemment plus coûteuse. Notons également que nous avons spécifié une limite sur la taille maximale des clauses pouvant être générées, comme ce qui est fait lors de l'élimination dans les solveurs SAT. Pour nos expérimentations, nous avons utilisé les mêmes critères que dans les solveurs SAT (`Minisat` ou `Glucose`) à savoir, une variable est éliminée si le nombre de clauses n'augmente pas, et si aucune des clauses générées par l'élimination ne dépasse la taille maximale de 20 littéraux. Dans la suite de nos travaux, cela pourrait être intéressant de tester différents paramètres. Néanmoins, il faut avoir conscience que l'étape de génération peut rapidement exploser en mémoire si ces paramètres sont trop grands.

Ordre d'élimination

Comme pour l'élimination dans le solveur, l'élimination que nous effectuons au niveau du modèle est également dépendante de l'ordre dans lequel les variables sont éliminées. Il n'est cependant pas aisé d'anticiper l'impact qu'aura une élimination plutôt qu'une autre sur les performances du solveur. Nous avons donc choisi l'approche la plus simple : les variables sont éliminées

(selon les critères donnés ci-dessus) en suivant l'ordre des définitions du modèle, c'est-à-dire que l'on parcourt la liste des définitions dans l'ordre de leur profondeur, *i.e.*, des feuilles (les définitions ne référençant que des variables d'entrées ou des variables mémoires) jusqu'à la racine (la propriété). C'est cet ordre que nous utilisons dans l'ensemble des expérimentations de cette thèse.

Nous avons quand même testé de manière empirique différentes heuristiques (indiquées ci-dessous), même si nous n'avons pas observé de différences majeures entre elles, que ce soit en nombre de clauses générées ou en temps de résolution. Voici les autres heuristiques que nous avons expérimentées pour l'ordre d'élimination à haut niveau :

— Lorsque l'élimination est effectuée dans le solveur, le critère d'ordre est le suivant : la variable x ayant le plus petit score $|x| \times |\neg x|$, est éliminée en priorité. Autrement dit, les variables les « moins équilibrées » sont sélectionnées en premier (moins équilibré signifie qu'il existe plus d'occurrences de la variable dans une polarité que dans l'autre). Nous avons donc appliqué la même heuristique au niveau des définitions, à savoir les variables les « moins équilibrés » sont sélectionnées en premier.

— Une approche basé sur l'inverse de l'ordre du modèle, *i.e.*, de la racine jusqu'aux feuilles.

— Une dernière approche inspirée de Velev [2004], où l'idée est de faire plusieurs itérations (passages sur l'ensemble des définitions) de façon à appliquer d'abord les « meilleures » éliminations, et ensuite des éliminations de moins en moins « efficaces ». Nous disons d'une élimination qu'elle est « meilleure » ou « plus efficace » qu'une autre si elle permet de générer moins de clauses. Voici un exemple d'une heuristique d'élimination composée de quelques règles que nous avons établie :

  — D'abord, effectuer l'élimination de toutes les variables référencées une seule fois et de polarité positive. Cela revient à construire ce qui est parfois appelé les « *big ands* », *i.e.*, une conjonction de plusieurs opérandes. Chaque élimination dans ce premier passage entraîne donc un gain d'une variable et de 2 clauses.

  — Ensuite, encore pour construire les *big ands*, effectuer l'élimination de toutes les variables référencées au plus 2 fois et toujours de polarité positive. Néanmoins, la taille de la définition qui est éliminée ne doit pas dépasser 3 opérandes, pour ne pas augmenter le nombre de clauses. Si la taille de la définition éliminée est de 2 opérandes alors il y a un gain d'une variable et d'une clauses. Si la définition contient 3 opérandes alors le nombre de clauses reste inchangé.

  — Puis, toujours pour la construction des *big ands*, effectuer l'élimination de toutes les variables référencées au plus 3 fois et toujours de polarité positive, et cette fois-ci avec une taille maximale de 2 opé-

randes. Cette règle d'élimination ne réduit pas le nombre de clauses générées.

— Lors d'un quatrième passage, les variables qui correspondent à une partie de la définition d'une équivalence ou d'un *XOR* (« OU » exclusif) sont éliminées, par exemple :

$$\begin{aligned} x &= a \wedge \neg b \\ y &= \neg a \wedge b \\ z &= \neg x \wedge \neg y \\ &\downarrow \\ z &= (\neg a \vee b) \wedge (a \vee \neg b) \end{aligned}$$

— Enfin, un dernier passage pour éliminer les variables qui correspondent à une condition (ITE).

À noter que ces deux derniers passages servent principalement pour le prétraitement, notamment pour l'équivalence structurelle. On notera également qu'il ne s'agit que d'un exemple, il est possible de définir de nombreuses heuristiques d'élimination composées de règles comme celles indiquées ci-dessus.

Expérimentations

Nous avons ensuite vérifié de manière empirique si cette élimination haut niveau était une alternative viable à l'élimination au niveau de la CNF. Nous avons alors défini 3 niveaux où l'élimination peut être effectuée :

— « ll » signifie que l'élimination est réalisée à bas niveau, c'est-à-dire dans le solveur SAT. Il s'agit de l'élimination par distribution, effectuée par défaut dans `Glucose` ou `Minisat`.

— « hl » indique que l'élimination est effectuée à haut niveau, à savoir au niveau du modèle comme défini ci-dessus.

— « uhl » spécifie que les variables sont éliminées au niveau du modèle déplié, ce qui permet de profiter de simplifications supplémentaires (*e.g.*, propagation des états initiaux pour BMC).

Notons que ces différents niveaux d'application de l'élimination peuvent aussi se combiner. Nous avons alors souhaité observer l'impact de ces différentes stratégies d'élimination pour les algorithmes BMC et ZigZag. Afin d'observer uniquement l'impact de l'élimination, aucune simplification autre que la propagation de constantes et d'alias n'est effectué après le dépliage. Néanmoins, il y a tout de même une différence importante à noter, lorsque l'élimination bas niveau est activée (ll) pour BMC, le k-COI est désactivé pour éviter de réintroduire une variable qui a déjà été éliminée, comme évoqué précédemment [4].

4. Le k-COI est équivalent au COI pour la k-induction s'il n'y a pas de simplification sur le modèle déplié.

	ll	hl	uhl	hl+ll	hl+uhl+ll
IND	409	409	409	**408**	**408**
CEX	124	124	124	**125**	**125**
PRO	14	14	14	14	14

TABLE 3.1 – Résultats des différentes stratégies d'élimination des variables sur l'ensemble des instances HWMCC15 pour BMC.

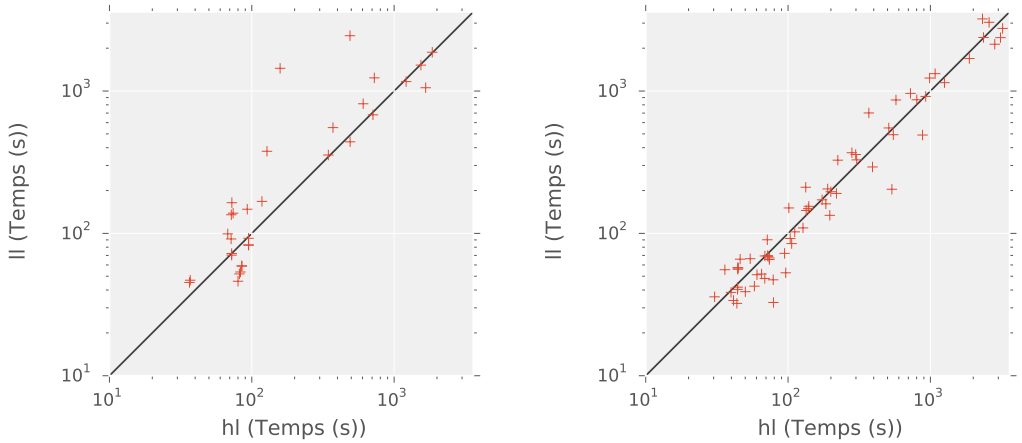
	ll	hl	uhl	hl+ll	hl+uhl+ll
IND	335	**333**	335	**333**	334
CEX	118	118	118	**119**	118
PRO	94	**96**	94	95	95

TABLE 3.2 – Résultats des différentes stratégies d'élimination des variables sur l'ensemble des instances HWMCC15 pour ZigZag.

Conditions d'expérimentation — Nous avons défini et exécuté différentes stratégies d'élimination sur l'ensemble des 547 instances de la compétition de model checking 2015 (HWMCC'15). Les résultats pour BMC et ZigZag sont reportés dans les tableaux 3.1 et 3.2 respectivement. La ligne IND correspond au nombre d'instances pour lesquelles le résultat reste indéterminé après un temps d'exécution supérieur à 3600s ou après avoir dépassé le seuil de 8Go de mémoire. La ligne CEX dénombre les instances pour lesquelles un contre-exemple a été trouvé. La ligne PRO indique le nombre d'instances pour lesquelles la propriété est vérifiée. À noter que pour l'élimination bas niveau de BMC, nous avons utilisé la technique de Kupferschmid *et al.* [2011], qui consiste à ne geler que les variables qui seront réintroduites dans le prochain appel. Dans le tableau de résultat de BMC, lorsque une propriété est vérifiée, il s'agit des cas où la propriété s'évalue statiquement à vrai après prétraitement.

Analyse — Pour BMC, les résultats sont globalement équivalents, avec une instance résolue en plus pour la combinaison de l'élimination à haut niveau et à bas niveau. En regardant en détail, nous observons que lorsque l'élimination est effectuée à haut niveau (hl ou uhl) ou à bas niveau (ll), les instances résolues sont les mêmes. hl+ll et hl+uhl+ll trouvent chacun un CEX unique supplémentaire : `oski15a10b14s` pour hl+ll et `6s13` pour hl+uhl+ll. Les résultats qui nous intéressent le plus concernent les deux stratégies hl et ll, car nous souhaitions savoir si nous pouvions remplacer l'élimination à bas niveau par l'élimination à haut niveau sans perte de performance. Nous avons souhaité

regarder plus précisément ces résultats car s'il y a de réelles différences entre les méthodes, elles peuvent ne pas être visibles avec le tableau seul. Nous avons donc illustré, sur la figure 3.2a, une comparaison du temps mis par les deux stratégies (hl et ll) pour résoudre les mêmes problèmes. Nous avons seulement sélectionné les instances nécessitant plus de 30 secondes pour être résolues. Nous observons alors que, globalement, pour les problèmes nécessitant plus de



(a) BMC. (b) ZigZag.

FIGURE 3.2 – Comparaison du temps de résolution entre hl et ll.

100 secondes de résolution, l'élimination à haut niveau semble plus efficace. Il est néanmoins important de noter que le k-COI peut grandement réduire la taille du problème, ce qui pourrait en partie expliquer ces résultats.

Pour ZigZag, des conclusions similaires peuvent être tirées, à savoir le nombre d'instances résolues par les différentes stratégies est presque équivalent : les deux meilleurs stratégies sont hl, qui est le seul à réussir à vérifier une instance de plus, et hl+ll qui trouvent un CEX de plus. De la même façon que pour BMC, nous avons alors illustré sur la figure 3.2b une comparaison du temps mis par les deux stratégies hl et ll pour résoudre les mêmes problèmes difficiles, *i.e.*, nécessitant plus de 30 secondes pour être résolus. Le graphe est cette fois-ci plus équilibré, ce qui peut être lié au fait que le k-COI n'a pas d'impact pour ZigZag dans cette expérimentation.

Conclusion

Nous avons défini une nouvelle stratégie pour effectuer l'élimination de variables uniquement au niveau du modèle. Nous avons montré que cette stratégie est en pratique au moins aussi efficace que lorsque l'élimination est effectué dans le solveur, au niveau de la formule CNF. Cependant, si l'on analyse plus finement les résultats, nous observons que notre élimination est plus coûteuse

que celle implémentée dans les solveurs sur les instances simples. En effet, sur les instances résolues en moins de 2 minutes l'élimination bas niveau est généralement plus rapide. Cependant, sur les instances plus difficiles notre approche est globalement plus rapide. Cela peut être lié au fait que l'élimination n'est effectué qu'une seule fois, et non à chaque appel au solveur. Le solveur a donc plus de temps pour la résolution. Cela peut également provenir de la différence entre l'élimination par distribution et celle par substitution. Effectuer l'élimination à haut niveau permet également de ne plus avoir à se soucier des problèmes relatifs à l'élimination de variables dans un contexte incrémental. Avec cette approche, nous pouvons par exemple effectuer le k-COI, l'équivalence structurelle globale, etc. Enfin, cela nous permet d'obtenir une représentation homogène entre le MC et le solveur SAT. Notre approche a néanmoins certains inconvénients. Tout d'abord, on retrouve les même limitations que l'élimination au niveau de la CNF : notre méthode est dépendante de l'ordre dans lequel les variables sont éliminées. Ensuite, comme l'expressivité des définitions est étendue, l'ensemble des structures et algorithmes gérant ces définitions sont plus complexes. Le dernier désavantage concerne les variables d'entrée qui ne peuvent pas être éliminées.

Dans de futurs travaux, nous pensons qu'il serait intéressant de réfléchir à d'autres ordres d'élimination ou encore de spécialiser l'ordre d'élimination en fonction de l'algorithme de vérification qui sera utilisé par exemple. A l'inverse, il aurait été intéressant de comparer les performances de l'élimination au niveau de la CNF avec le même ordre d'élimination que nous utilisons lors de l'élimination dans le model checker. Il serait également intéressant d'étendre encore l'expressivité à haut niveau afin de permettre l'élimination des entrées pour observer l'impact de leur élimination. Enfin, nous aurions souhaité avoir plus de temps pour implémenter la méthode d'élimination d'`ABC` pour comparer les deux approches.

3.4 Extraction des déductions et dialogue

La limitation de l'élimination de variables au niveau du modèle nous permet d'avoir une représentation homogène entre le modèle haut niveau et la formule CNF. Ainsi, nous pouvons maintenant établir un dialogue simple du solveur vers le model checker. Dans cette partie, nous commençons par donner quelques motivations, illustrées autour d'un exemple complet, puis nous décrivons les déductions faites par le solveur que nous avons choisies d'extraire pour notre expérimentation.

3.4.1 Motivation

Comme évoqué précédemment, nous pensons qu'il est possible d'exploiter encore plus les déductions faites par le solveur. Afin de préserver la résolution incrémentale, les simplifications applicables au niveau de la CNF sont restreintes. Or, en important les déductions faites par le solveur au niveau du modèle, et en appliquant des simplifications sur le modèle déplié, nous pouvons profiter des déductions du solveur pour simplifier le problème. Pour résumer, cela permet d'effectuer une étape d'*inprocessing* dans un contexte incrémental. Voici ci-dessous un exemple montrant l'intérêt de cette méthode.

Exemple 3.4. En entrée, nous prenons un modèle AIG optimisé (propagation de constantes, règles de réécriture, COI, etc.) et où l'élimination de variables a été effectuée comme décrit dans la partie précédente. Supposons un modèle contenant 3 entrées a, b, et c, 2 mémoires x, y, et 5 définitions k, l, m, n et o, tel que :

$$
\begin{aligned}
x &= \bot, m \\
y &= \bot, n \\
k &= (\neg a \vee x) \wedge (a \vee c) && \Leftrightarrow ITE(a, x, c) \\
l &= (\neg a \vee \neg y) \wedge (a \vee \neg c) && \Leftrightarrow ITE(a, \neg y, \neg c) \\
m &= (\neg b \vee k) \wedge (b \vee l) && \Leftrightarrow ITE(b, k, l) \\
n &= (\neg b \vee \neg l) \wedge (b \vee \neg k) && \Leftrightarrow ITE(b, \neg l, \neg k) \\
o &= (\neg m \vee n) \wedge (m \vee \neg n) && \Leftrightarrow m = n \\
Bad &: \neg o
\end{aligned}
$$

Dépliage $k = 0$ pour BMC :

$$
\begin{aligned}
&[x_0 = \bot,\ y_0 = \bot] \\
&k_0 = (\neg a_0 \vee \bot) \wedge (a_0 \vee c_0) \rightarrow k_0 = \neg a_0 \wedge c_0 \\
&l_0 = (\neg a_0 \vee \top) \wedge (a_0 \vee \neg c_0) \rightarrow l_0 = a_0 \vee \neg c_0 \\
&m_0 = (\neg b_0 \vee k_0) \wedge (b_0 \vee l_0) \\
&n_0 = (\neg b_0 \vee \neg l_0) \wedge (b_0 \vee \neg k_0) \\
&o_0 = (\neg m_0 \vee n_0) \wedge (m_0 \vee \neg n_0) \\
&Bad : \neg o_0
\end{aligned}
$$

En supposant que lors de l'application des simplifications sur le modèle déplié, l'équivalence $l_0 = \neg k_0$ ne soit pas détectée[5], et que lors de la résolution, le solveur apprenne les clauses :

$$\neg m_0 \vee n_0 \text{ et } m_0 \vee \neg n_0$$

5. Cela permet d'illustrer les bénéfices de l'extraction d'information, tout en gardant un exemple relativement simple.

En observant toutes les clauses binaires du solveur, nous pouvons alors extraire l'information $n_0 = m_0$ du solveur et l'importer dans la représentation haut niveau. Ainsi, lors de l'application des règles d'optimisation sur la représentation haut niveau, lors du dépliage suivant, ici $k = 1$, n_0 peut être substitué par m_0, ce qui simplifiera le modèle, et par conséquent réduira le nombre de clauses passées au solveur.

Notons que les clauses ($\neg m_0 \vee n_0$ et $m_0 \vee \neg n_0$) seront dans le solveur même si celui-ci ne les apprend pas. En effet, après chaque appel à BMC la clause unitaire correspondant à la négation de Bad peut être ajoutée au solveur. Ainsi, comme la transformation de la définition $o_0 = (\neg m_0 \vee n_0) \wedge (m_0 \vee \neg n_0)$ va générer les clauses $\neg o_0 \vee \neg m_0 \vee n_0$ et $\neg o_0 \vee m_0 \vee \neg n_0$. En ajoutant la clause unitaire $\neg Bad \Leftrightarrow o_0$, ces deux clauses seront après propagation équivalentes aux deux clauses ci-dessus.

3.4.2 Description

L'idée général de notre approche se résume à profiter des déductions du solveur pour simplifier la représentation haut niveau afin d'accélérer potentiellement la résolution lors de futurs appels au solveur. Pour tester l'efficacité de cette approche, nous avons décidé d'extraire deux types d'information du solveur, les constantes (*e.g.*, $x = \top$) et les équivalences (*e.g.*, $x = \neg y$), car elles peuvent facilement être extraites à partir des clauses apprises et peuvent directement être importées dans la représentation haut niveau. La figure 3.3 illustre le déroulement général de notre approche, lors de l'exécution de BMC avec extraction des déductions du solveur. Voici la description des différentes

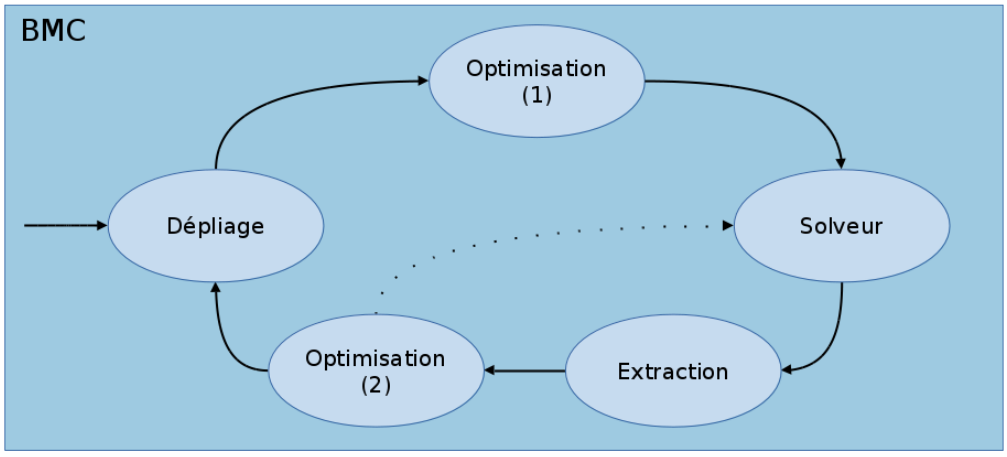


FIGURE 3.3 – Déroulement de BMC avec dialogue entre solveur et MC

étapes :

— **Dépliage :** À partir d'un AIG optimisé, le dépliage du modèle est effectué et la représentation du modèle déplié est conservé à haut niveau, *i.e.*, le modèle n'est pas traduit en clause à la volée, lors du dépliage.

— **Optimisation (1) :** Des règles d'optimisation sont ensuite appliquées sur ce modèle déplié.

— **Solveur :** Le modèle déplié et simplifié est ensuite transformé en CNF, et après ajout des assertions nécessaires pour vérifier si les mauvais états sont atteignables, le problème est résolu avec le solveur SAT.

— **Extraction :** S'il n'existe pas de CEX, après chaque appel au solveur, les constantes et équivalences déduites par le solveur sont extraites et importées au niveau de la représentation du modèle déplié.

— **Optimisation (2) :** Les règles d'optimisation sont alors réappliquées sur le modèle déplié. Si de nouvelles simplifications concernant des parties du modèle déplié déjà transformées en CNF sont trouvées, alors elles sont transmises au solveur sous forme de nouvelles clauses.

— Le processus est alors répété pour la profondeur suivante.

Il nous faut maintenant décrire comment effectuer l'extraction des déductions faites dans le solveur.

3.4.3 Extraction des constantes

L'équivalent d'une constante, comme $x = \bot$, au niveau du modèle est une clause unitaire ($\neg x$) au niveau du solveur. Une clause unitaire force un littéral à être affecté à une valeur dans le solveur, *i.e.*, dans toutes les affectations possibles. Comme nous avons une relation d'équivalence entre le modèle haut niveau et les clauses, ces clauses unitaires peuvent directement être introduites dans la représentation haut niveau. Dans `Minisat` (et par conséquent dans `Glucose`), les clauses unitaires sont gérées comme des décisions de niveau 0, c'est-à-dire avant tout branchement par l'heuristique de sélection de variables. Pour les extraire, il suffit alors de conserver toutes les clauses qui sont décidées et propagées au niveau 0, ce qui fournit un avantage supplémentaire que nous décrivons dans la sous-partie ci-dessous. À noter que la majorité des clauses unitaires du solveur sont des clauses apprises, car comme nous effectuons les propagations de constantes à haut niveau, très peu de clauses unitaires sont passées au solveur. Néanmoins, il peut y avoir des exceptions avec la propriété, les contraintes, les littéraux d'activation, ou encore les nouvelles optimisations [6] qui peuvent être propagées au solveur.

Il faut cependant s'assurer que ce processus conserve la validité de l'algorithme, et donc déterminer si les clauses unitaires qui sont importées sont bien

6. Il s'agit des optimisations qui sont faites sur des parties du modèle qui ont déjà été transformées en clauses.

indépendantes de l'itération courante. En d'autres termes, est-ce qu'il est correcte d'ajouter ces contraintes dans la représentation haut niveau indéfiniment, *i.e.*, pour toutes itérations de BMC.

La validité de ce point découle directement de la résolution incrémentale en présence de littéraux d'activation. En effet, si une clause était dépendante de la propriété de l'itération courante (dans BMC) ou des états initiaux (dans ZigZag), alors le processus d'apprentissage du solveur aurait ajouté un littéral d'activation à la clause unitaire. Par conséquent, toute clause unitaire apprise est vraie pour toute itération (de BMC ou de ZigZag), lorsque l'algorithme est effectué incrémentalement avec littéraux d'activation.

Profiter de la propagation du solveur

Usuellement, la propagation de constantes et plus généralement l'ensemble des simplifications effectuées à haut niveau ne s'effectue que dans un sens, c'est-à-dire des entrées vers les sorties. Cela peut parfois limiter les simplifications qui pourraient être appliquées avec les informations extraites. Prenons par exemple la définition $x = a \wedge b$. Si la déduction $x = \top$ est extraite du solveur, aucune information sur a et b n'est propagée. Or, il est clair que $a = \top$ et $b = \top$. De ce fait, afin d'éviter l'ajout de règles de simplifications qui consistent à propager des informations vers les opérandes d'une définition, nous proposons d'utiliser la propagation du solveur. Par exemple, si l'on considère la définition $x = a \wedge b$ sous forme de clause :

$$\begin{aligned} &\neg x \vee a \\ &\neg x \vee b \\ &x \vee \neg a \vee \neg b \end{aligned}$$

Il apparaît clairement que lors de l'apprentissage de la clause unitaire x, immédiatement après, a et b seront propagés. Cependant, comme nous importons dans le model checker toutes les clauses propagées au niveau 0 dans le solveur, au lieu de surveiller l'ensemble des clauses apprises et de sauvegarder uniquement les clauses unitaires, nous profitons ainsi de cette propagation « gratuite » faite par le solveur.

3.4.4 Extraction des équivalences

L'extraction des équivalences est un peu moins trivial, car 2 clauses sont nécessaires pour exprimer $x = y$ en CNF (Warners et Van Maaren [1998]; Li [2003]). La détection de l'équivalence entre deux littéraux au niveau des clauses consiste alors à rechercher deux configurations (*patterns*) de 2 clauses

binaires :

$$\begin{array}{lllll} \neg x \vee y & \text{et} & x \vee \neg y & \rightarrow & x = y \\ \neg x \vee \neg y & \text{et} & x \vee y & \rightarrow & x = \neg y \end{array}$$

Autrement dit, à chaque étape d'extraction des déductions, si dans l'ensemble des clauses du solveur (*i.e.*, clauses originales et clauses apprises) il existe des occurrences des deux configurations ci-dessus, alors les équivalences correspondantes sont ajoutées dans la représentation haut niveau. Ainsi, lors de l'application des simplifications, les définitions correspondantes à des équivalences sont remplacées comme pour la propagation d'un alias. À noter qu'en raison de la gestion paresseuse des propagations (*watched literals*) dans `Glucose`, lorsque une clause unitaire est apprise, les clauses contenant la négation de ce littéral ne sont pas modifiées. Ainsi, il est également nécessaire d'observer les clauses qui ont une taille supérieure à 2 mais contenant des littéraux qui sont toujours faux en raison de clauses unitaires. Plus précisément, on définit la longueur effective d'une clause c comme étant $|c \setminus \bar{u}|$ où u est l'ensemble des littéraux unitaires dans le solveur et $\bar{u}$ correspond à la négation de chaque élément de u. On effectue alors la recherche de configurations sur l'ensemble des clauses d'une longueur effective égale à 2.

Comme il serait trop coûteux de rechercher ces configurations de clauses en parcourant l'ensemble des clauses, nous utilisons simplement une table de hachage. La fonction de hachage est la suivante : nous ignorons la polarité des littéraux, les 2 variables sont ordonnées, puis nous effectuons l'addition entre ces deux variables, qui sont au préalable multipliées par un nombre premier pour limiter les collisions (*e.g.*, une fonction de hachage f pour 2 entiers x, y : $f(x, y) = x * 31 + y * 111$). Ainsi, avec cette fonction toute clause binaire contenant les 2 mêmes variables aura la même signature, quel que soit la polarité des littéraux (bien entendu d'autres fonctions de hachage pourraient convenir). Par conséquent, pour repérer les équivalences, a chaque fois qu'une clause binaire est ajoutée au solveur (clause original ou apprise), il suffit de regarder si une autre clause binaire composée des mêmes variables est déjà dans la table. Si ce n'est pas le cas, nous ajoutons cette nouvelle clause dans la table. Sinon, en regardant les deux clauses en détail, nous déduisons la nouvelle équivalence à intégrer dans la représentation haut niveau. On notera que cette méthode permet également de détecter des constantes lorsque la configuration suivante se produit :

$$x \vee y \qquad \text{et} \qquad x \vee \neg y \qquad \rightarrow \qquad x = \top$$

L'importation de ces équivalences au niveau du modèle déplié est également valide grâce au processus d'activation des littéraux. En effet, si une clause est dépendante d'une partie temporaire, elle contiendra le littéral d'activation correspondant. En outre, aucune variable du modèle ne peut être équivalente à

un littéral d'activation, donc on ne risque pas d'importer un littéral d'activation dans le modèle déplié.

3.4.5 Exportation des simplifications

La dernière étape du dialogue consiste à fournir au solveur les simplifications qui ont été effectuées au niveau de la représentation du modèle déplié sur les parties qui ont déjà été transformées en clauses. Nous avons choisi de ne transmettre au solveur que les simplifications « importantes », à savoir, les simplifications qui s'expriment par une clause unitaire (constantes) ou par deux clauses binaires (équivalences). Autrement dit, les simplifications qui n'engendrent pas l'élimination d'une définition ne sont pas propagées au solveur.

3.4.6 Expérimentations

Nous avons expérimenté trois versions de BMC sur l'ensemble des instances d'HWMCC'15, avec les mêmes conditions que précédemment, à savoir, un temps de calcul limité à 1h et 8Go de mémoire maximum :

1. « Ref » correspond à notre version de référence, *i.e.*, BMC avec élimination des variables à haut niveau et k-COI mais sans appliquer pas de règles de simplification sur le modèle déplié.
2. « Prep » est identique à « Ref » mais effectue en plus une étape de prétraitement à chaque dépliage de la relation de transition sur le modèle déplié (détaillé dans la partie 5.7).
3. « Import » est la même version que « Prep » où en plus nous importons les déductions du solveur et appliquons les simplifications sur le modèle déplié après chaque appel, puis les nouvelles simplifications trouvées sont exportées vers le solveur.

Ces trois versions trouvent le même nombre de CEX (124), et semblent obtenir des résultats proches lorsque l'on observe les résultats globalement. Donc, afin de les comparer, nous avons reporté dans la première partie du tableau 3.3 les résultats détaillés obtenus par ces différentes versions de BMC, pour les 11 instances les plus difficiles, contenant un contre-exemple. Plus précisément, nous avons sélectionné les 10 instances nécessitant le plus de temps de calcul pour la version de référence, les 10 instances les plus difficiles pour « Prep » et de même pour « Import ». Nous avons ainsi obtenu 11 instances distinctes. Dans la deuxième partie du tableau, nous avons reporté 14 instances pour lesquelles toutes les versions ont atteint le temps limite (« TO » signifie *Time Out*) sans dépasser le seuil de mémoire autorisé. Plus précisément, nous avons sélectionné toutes les instances les plus durs, *i.e.*, pour lesquelles aucune version n'a réussi à atteindre une profondeur supérieure à 100, et où nous avons observé

Instances	Ref		Prep		Import	
	Temps	Résultat	Temps	Résultat	Temps	Résultat
beembrptwo6b1	492.2s	CEX	504.9s	CEX	**440.8s**	CEX
6s20	541.2s	CEX	**305s**	CEX	514.8s	CEX
6s184	**586.7s**	CEX	615.1s	CEX	630.8s	CEX
6s309b034	747.7s	CEX	**651.7s**	CEX	682.5s	CEX
oski3b0i	969.5s	CEX	**837.2s**	CEX	929.2s	CEX
6s7	**1236.8s**	CEX	1495.9s	CEX	1313.8s	CEX
intel036	1355.5s	CEX	**897.4s**	CEX	1359.8s	CEX
beemprdcell2f1	1515.5s	CEX	1306.8s	CEX	**903.9s**	CEX
intel009	**1591.1s**	CEX	2106.4s	CEX	2012.5s	CEX
6s122	1663.5s	CEX	1786.2s	CEX	**1156s**	CEX
oski2ub0i	1923.9s	CEX	**1712.2s**	CEX	1983.8s	CEX
6s322rb646	TO	9	TO	13	TO	**14**
6s387rb291	TO	13	TO	19	TO	**20**
eijkbs6669	TO	20	TO	**23**	TO	21
beemskbn2b1	TO	**26**	TO	22	TO	24
6s31	TO	29	TO	**33**	TO	31
beemrshr4b1	TO	**39**	TO	36	TO	36
oski15a14b16s	TO	39	TO	**43**	TO	42
6s343b31	TO	41	TO	45	TO	**54**
pdtvisns3p02	TO	42	TO	**47**	TO	41
intel047	TO	45	TO	**51**	TO	49
intel044	TO	**49**	TO	41	TO	39
6s339rb22	TO	52	TO	58	TO	**64**
oski15a14b10s	TO	**79**	TO	44	TO	52
pdtpmsns2	TO	88	TO	91	TO	**94**

TABLE 3.3 – Résultats des différentes versions de BMC.

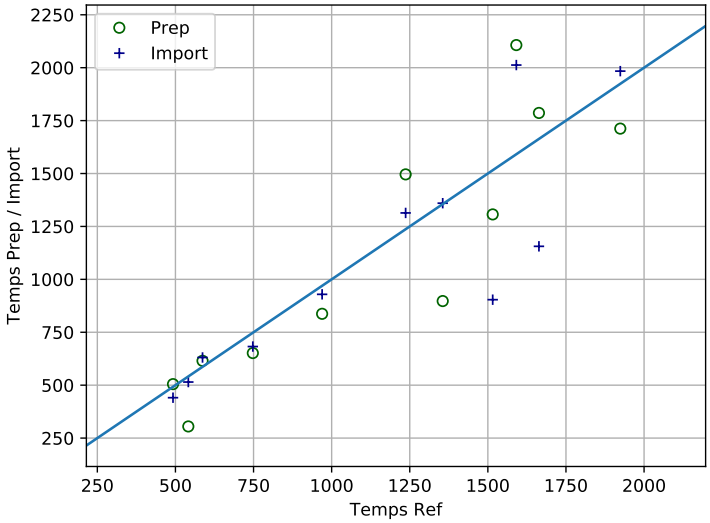


FIGURE 3.4 – Comparaison du temps (en s) nécessaire pour trouver un CEX sur les problèmes les plus difficiles, entre la version de référence et les deux autres version : Prep et Import.

une différence de profondeur strictement supérieure à 2 entre au moins deux versions. Les colonnes « résultats » du tableau indiquent « CEX » lorsqu'un contre-exemple a été trouvé. Dans le cas contraire, elles indiquent la profondeur qui a été atteinte.

Première partie du tableau (CEX) — Sur les 11 instances les plus difficiles pour les 3 stratégies, la version de référence est la plus rapide sur 3 instances, la version avec prétraitement sur 5 instances, et enfin la version avec importation sur 3 instances. La figure 3.4 compare le temps de résolution sur ces problèmes entre la version de référence et les deux autres versions : Prep et Import.

Deuxième partie du tableau (TO) — Sur les 14 instances les plus difficiles pour les 3 stratégies, la version de référence atteint la plus grande profondeur sur 4 instances, la version avec prétraitement sur 5 instances, et enfin la version avec importation sur 5 instances. La figure 3.5 compare les profondeurs atteintes par la version de référence et les deux autres versions : Prep et Import.

Discussions

— Les résultats sont mitigés, que ce soit entre la version de référence et la version avec prétraitement, ou entre cette dernière et la version avec importation des déductions. En effet, il n'y a pas de versions qui se démarquent réellement sur l'ensemble des instances.

— Nous observons pour certains problèmes un surcoût lié au prétraitement

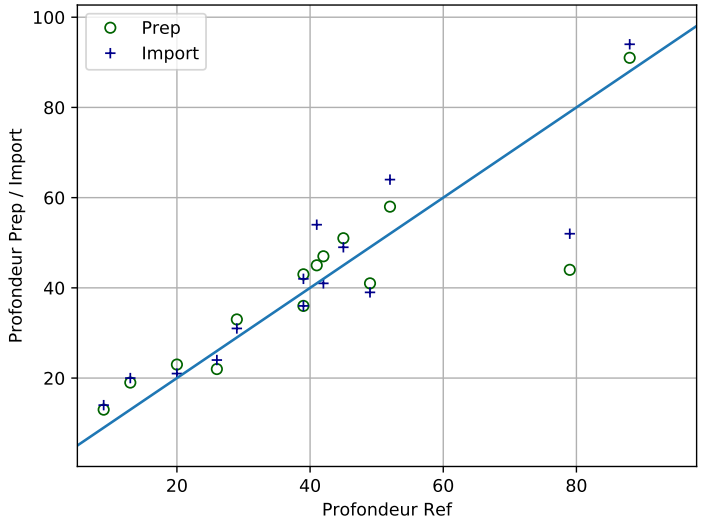


FIGURE 3.5 – Comparaison des profondeurs atteintes sur les problèmes les plus difficiles, entre la version de référence et les deux autres versions : Prep et Import.

du modèle déplié qui peut nécessiter un certain temps lorsque le nombre de définitions est grand ou lorsque le modèle a été déplié un grand nombre de fois, et que très peu d'informations sont déduites des règles de simplification.

— Nous pensons qu'il est nécessaire d'améliorer les déductions importées du solveur et l'étape de simplification du modèle pour observer un réel gain en performance (Les règles d'optimisation utilisées dans cette expérimentation sont détaillées dans la partie 5.7).

3.4.7 Travaux futurs

Il y a plusieurs autres axes d'amélioration possible. Par exemple, nous pensons que pour les problèmes difficiles, où il y a peu d'appels au solveur dans le temps imparti, c'est-à-dire que chaque appel nécessite un temps de résolution important, il faudrait mettre en place un dialogue plus fréquent, en autorisant par exemple le solveur à se mettre en pause pendant le calcul pour communiquer avec le model checker les informations qu'il a déjà pu déduire. Cette idée est détaillée dans la partie 5.10. Une autre piste d'amélioration serait d'étendre les déductions du solveur qu'il est possible d'extraire, comme par exemple la détection de portes logiques (Ostrowski *et al.* [2002] ($\wedge$, $\vee$ et $\Leftrightarrow$), Biere [2009] (ITEs, XORs)). Néanmoins, il est nécessaire d'avoir une méthode efficace pour la détection des configurations, sinon le surcoût pourrait être facilement su-

périeur au gain apporté. Aussi, au regard de ces résultats partagés il parait tout à fait naturel d'envisager une parallélisation de ces trois méthodes. Enfin, avec une vision encore plus globale, il serait intéressant d'imaginer une approche avec un troisième niveau de représentation, *i.e.*, avant le bit-blast, où les déductions du solveur seraient donc importées dans une représentation de même niveau que QF_BV par exemple, où il est possible d'effectuer d'autres simplifications, comme des optimisations arithmétiques.

3.5 Duplication

3.5.1 Motivation : Structure symétrique et duplication

Comme évoqué plus haut, les solveurs SAT sont basés sur l'apprentissage de clauses. Ces clauses permettent de généraliser les raisons d'un conflit pour ne pas le retrouver plus tard dans la recherche. L'idée de cette partie est de tenter de généraliser ces clauses grâce à la connaissance de la structure répétitive de la formule encodée en SAT. Intuitivement, si une clause apprise ne dépend que de clauses qui sont, par construction, répétées à différents temps, alors on peut également répété la clause apprise à ces autres temps. La méthode de la duplication de clauses se base donc sur la structure symétrique des formules. En effet, en raison du dépliage successif de la relation de transition, les formules générées par BMC contiennent une partie symétrique :

$$I_0 \wedge \underbrace{T_{0,1} \wedge T_{1,2} \wedge \cdots \wedge T_{k-1,k}}_{\textit{Partie symétrique}} \wedge Bad_k$$

En d'autres termes, si on ignore les parties concernant les états initiaux et les mauvais états, la formule est parfaitement symétrique. Cela signifie que si le solveur CDCL déduit une nouvelle clause en ne faisant des résolutions que sur la partie symétrique, alors cette nouvelle clause peut être dupliquée à d'autres cycles en renommant simplement les variables. Ces clauses dupliquées empêcheront alors peut-être la même affectation partielle de se produire dans un autre cycle, et accélérera ainsi potentiellement la résolution du problème.

Nous devons introduire une nouvelle notation mettant en relation chaque clause avec le temps duquel elle dépend. La notation $C_{[i,j]}$ indique ainsi que la clause C a été déduite à l'aide de résolutions ne faisant intervenir que des clauses provenant des relations de transition tel que le plus petit (respectivement le plus grand) indice de cycle de l'ensemble des variables utilisées est i (respectivement j). Supposons par exemple, que lors de la résolution de BMC_3, le solveur apprenne la clause $C_{[0,1]}$ à partir de clauses provenant uniquement des relations de transition $T_{0,1}$ et $T_{1,2}$, alors celle-ci peut être dupliquée pour

$T_{1,2}$ et $T_{2,3}$ en renommant ses variables :

$$I_0 \wedge \underbrace{T_{0,1} \wedge \overbrace{T_{1,2}}^{C_{[1,2]}} \wedge T_{2,3}}_{C_{[0,1]}} \wedge Bad_3$$

avec $C_{[0,1]} = \{x_0, \neg y_1, z_1\}$ et $C_{[1,2]} = \{x_1, \neg y_2, z_2\}$. La première étape de la duplication se produit donc directement à l'intérieur du processus d'apprentissage du solveur SAT. Ainsi, le solveur est maintenant capable d'apprendre plusieurs clauses par conflit.

En outre, comme BMC est effectué dans un contexte incrémental, une nouvelle relation de transition sera ajoutée au solveur après chaque appel résolu. En sauvegardant les clauses duplicables, ces clauses peuvent être dupliquées pour cette nouvelle relation de transition avant même de déterminer la satisfaisabilité de la formule BMC suivante. En reprenant notre exemple précédent, la clause $C_{[2,3]}$ peut être ajoutée, avant d'établir la satisfaisabilité de BMC_4 :

$$I_0 \wedge \underbrace{T_{0,1} \wedge \overbrace{T_{1,2}}^{C_{[1,2]}}}_{C_{[0,1]}} \wedge \underbrace{T_{2,3}}_{C_{[2,3]}} \wedge T_{3,4} \wedge Bad_4$$

Cette seconde étape requiert un plus haut niveau de supervision, *i.e.*, extérieur au solveur SAT, et s'effectue alors directement dans l'algorithme de model checking. La principale difficulté de cette méthode réside donc dans la caractérisation des clauses apprises, à savoir, est-ce qu'une clause est duplicable et à quels cycles elle peut être dupliquée. Il est évident qu'il ne faut surtout pas dupliquer par erreur une clause qui ne serait pas duplicable, *i.e.*, qui aurait été déduite à partir de résolution faisant intervenir les parties statiques de la formule (les états initiaux ou les mauvais états). Intuitivement, cela correspondrait à ajouter des contraintes des états initiaux ou des mauvais états à différents cycles de la formule, pouvant ainsi entraîner un changement de satisfaisabilité de la formule, et par conséquent l'algorithme ne serait plus correct.

3.5.2 État de l'art de la duplication de clauses

Conditions de duplication

Dans les premières tentatives de duplication (Strichman [2000]), une approche consistait à *inciter* l'apprentissage des mêmes clauses à des cycles différents en forçant l'affectation des variables pour mener à des conflits ayant comme conséquence l'apprentissage des clauses souhaitées. Une autre méthode consistait à dupliquer les clauses même si c'est celles-ci n'étaient pas duplicables et de vérifier lors d'un résultat insatisfaisable, à l'aide de la méthode ci-dessus, si les clauses dupliquées n'avaient pas changé la satisfaisabilité de

l'instance (Strichman [2000]). À noter que la duplication de clauses non duplicables ne peut entraîner que des faux négatifs [7].

Dans la suite des travaux proposés autour de la duplication des clauses (Strichman [2001, 2004]), l'approche principalement suivie a consisté à *marquer* les clauses avec 3 informations additionnelles :

1. un marqueur pour déterminer si la clause est duplicable, *i.e.*, ne dépend pas des états initiaux ou des mauvais états et n'a pas été déduite à partir de clauses non duplicables.
2. deux marqueurs pour définir le plus petit et le plus grand cycle dont la clause dépend.

Ainsi, lors d'un conflit, si toutes les clauses utilisées lors de l'analyse du conflit sont marquées comme étant duplicables, alors la nouvelle clause l'est aussi. En outre, le plus petit (respectivement grand) cycle de la nouvelle clause correspond au minimum (respectivement maximum) des cycles des clauses impliquées dans le conflit.

La démonstration de la validité de la duplication consiste simplement à montrer que toutes les clauses permettant d'inférer la clause dupliquée se trouvent également dans la formule. En d'autres termes, toutes les clauses utilisées dans l'analyse de conflit pour apprendre la clause que l'on souhaite dupliquer, sont également présentes (modulo renommage) aux cycles où la clause est dupliquée. Il est donc correct de dupliquer une clause dans ces conditions.

La duplication de clauses est, cependant, considérée comme une méthode trop coûteuse, c'est-à-dire que même si un gain en performance est apporté par les clauses dupliquées, il est en pratique annihilé par le surcoût de la mise en place de cette méthode ou en raison d'un trop grand nombre de clauses dupliquées.

Duplication des clauses pour l'induction temporelle

Plus récemment, Yin *et al.* [2014] ont proposé d'utiliser les littéraux d'activation pour caractériser les clauses duplicables. Pour cela, ils ont suggéré d'ajouter des littéraux d'activation act_{T_i} à toutes les relations de transitions :

$$\left[\neg act_I \vee I_0\right] \wedge \bigwedge_{i=0}^{k-1} \left[\neg act_{T_i} \wedge P_i \wedge T_{i,i+1}\right] \wedge \left[\neg act_B \vee Bad_k\right]$$

Ainsi, grâce au processus d'analyse de conflit, les parties de la formule à partir desquelles la nouvelle clause apprise est déduite peuvent être retrouvées en regardant simplement les variables de cette clause. La condition de duplication des clauses est alors entièrement gérée de la même façon que la résolution

7. En effet, l'ajout de contraintes ne peut pas rendre la formule satisfaisable si elle ne l'était pas.

incrémental en pratique. Une clause apprise à partir d'une partie spécifique de la formule contiendra toujours le littéral d'activation associé qui est utilisé pour activer cette partie dans le solveur. Autrement dit, si une clause apprise contient un des littéraux d'activation des états initiaux ou des mauvais états, alors la clause est considéré non-duplicable. Sinon, la clause peut être dupliquée en fonction des indices minimums et maximums des variables contenu dans cette clause. Par exemple, supposons que lors de la résolution d'une instance de BMC_4, la clause $c_{[1,2]} = \{\neg act_{T_1}, \neg act_{T_2}, x_1, y_2\}$ est apprise. Comme elle ne contient ni act_I ni act_B, la clause est considérée comme duplicable avec 1 et 2 comme indice minimum et maximum respectivement, en raison de la présence de act_{T_1} et act_{T_2} dans la clause. Par conséquent, $c_{[1,2]}$ peut être dupliquée en $c_{[0,1]}$ et $c_{[2,3]}$. Plus généralement, pour une profondeur k et une nouvelle clause apprise duplicable $c_{[m,n]}$, les clauses suivantes sont ajoutées à la formule : $\forall_{0<i\leq m}\ c_{[m-i,n-i]}$ et $\forall_{0<i\leq k-n}\ c_{[m+i,n+i]}$. Ces nouvelles clauses sont simplement obtenues en décalant chaque indice des variables par $-i$ ou $+i$. Enfin, en suivant le même mécanisme, chaque fois qu'une nouvelle relation de transition est dépliée, le processus de duplication externe (au niveau du model checker) est effectué, *i.e.*, toutes les clauses duplicables qui ont été apprises depuis la résolution du premier appel au solveur sont dupliquées pour la dernière relation de transition (en fonction de la profondeur courante et des indices de la clauses). L'avantage principal de cette méthode est qu'elle ne nécessite qu'une modification minime du solveur. Il suffit de parcourir chaque clause apprise pour déterminer si elle est duplicable et calculer ses indices. Cependant, l'ajout de ces littéraux d'activation n'augmente pas seulement la taille de toutes les clauses des relations de transition, mais aussi celles de toutes les clauses apprises. Par conséquent, un littéral additionnel est ajouté aux clauses apprises pour chaque relation de transition utilisée lors de l'analyse de conflit. Par exemple, comme toutes les parties de la formule sont ajoutées avec des littéraux d'activation, plus aucune clause unitaire n'est apprise par le solveur. Enfin, Yin *et al.* [2014] ne suggère pas non plus de méthodes pour réduire le nombre de clauses dupliquées, ce qui reste le problème principal de la duplication comme nous le montrons dans la partie 3.5.4, où nous présentons les résultats de nos expérimentations.

3.5.3 Approche proposée

Duplication et variables élimination

Il est important de noter que, malgré son importance pratique dans la résolution des problèmes du monde réel, l'élimination de variables n'est pas considérée dans les travaux évoqués ci-dessus. Ainsi, elle n'était pas encore utilisée lors des travaux de Strichman [2000, 2001, 2004] et, de manière plus problématique, elle n'est pas évoquée dans les travaux de Yin *et al.* [2014].

Nous supposons qu'elle est désactivée ou au moins limitée pour être valide dans un contexte incrémental. En effet, l'élimination de variables au niveau du solveur peut changer la symétrie de la formule. Par exemple, cela peut engendrer des situations où une clause contenant une variable qui est éliminée dans d'autres transitions de relation est dupliquée, ce qui entraîne la création de clauses trivialement satisfaisables. Dans notre approche, comme nous effectuons l'élimination uniquement au niveau du modèle, nous n'avons pas de cas particuliers à gérer. Il s'agit d'un des points forts de notre approche.

Duplication et contraintes de chemin simple à la demande

Comme évoqué précédemment, les contraintes de chemin simple sont primordiales pour rendre l'algorithme d'induction temporelle complet. Il est donc nécessaire de les prendre en compte dans notre contexte de duplication. Comme nous ajoutons ces contraintes *à la demande*, de nouvelles variables sont ajoutées à la formule de façon « asymétrique », *i.e.*, ces nouvelles variables n'ont pas nécessairement d'équivalents dans les autres profondeurs. Alors, pour préserver la validité de notre algorithme nous avons choisi de ne pas dupliquer les clauses contenant une variable introduite pour les contraintes de chemin simple. La condition de duplication est alors étendue à : une clause apprise est duplicable si elle ne contient ni un littéral d'activation des états initiaux ou des mauvais états, ni une variable introduite par les contraintes de chemin simple.

Limiter la duplication au niveau du model checker

Nous avons observé une augmentation non négligeable du temps de calcul lors de la vérification des modèles d'HWMCC'15 avec l'algorithme ZigZag lorsque des littéraux d'activation sont ajoutées pour chaque relation de transition (voir la partie 3.5.4). Nous avons alors décidé de limiter le processus de duplication au niveau du model checker seulement, permettant ainsi de se débarrasser de ces littéraux d'activation supplémentaires.

Cette limitation fournit une méthode de duplication *gratuite*, *i.e.*, il n'y a pas de surcoût lié au processus de duplication à cause de l'ajout de littéraux d'activation aux relations de transition, et l'implémentation est très aisée : il suffit de regarder si une clause apprise contient un littéral d'activation pour déterminer si elle est duplicable ou non. Puis, à chaque dépliage d'une nouvelle relation de transition l'ensemble des clauses duplicables sont dupliquées en les décalant d'un cycle.

Enfin, cette limitation permet de réduire le nombre de clauses dupliquées, ce qui est souhaité comme nous le montrons dans la partie 3.5.4. Elle agit comme une première sélection des clauses à dupliquer. Autrement dit, au lieu de dupliquer les clauses pour toutes profondeurs possibles, cela revient à *décaler*

l'ensemble des clauses précédemment apprises qui ont mené à une solution ou qui constituent la preuve de non satisfaisabilité.

Sélection des clauses à dupliquer

Néanmoins, malgré cette première limitation, nous avons tout de même observé une diminution des performances de notre model checker liée à la duplication (voir 3.5.4). Nous avons alors décidé de limiter encore le nombre de clauses dupliquées, en dupliquant uniquement les *bonnes* clauses. Nous avons défini une clause comme étant de *bonne qualité* si elle est de petite taille ou si elle a un petit score LBD. Nous avons également souhaité observer l'impact du processus de suppression agressive des clauses apprises mis en place dans `Glucose`. Nous avons alors défini un ensemble de stratégies de duplication basé sur ces deux critères :

1. Choix de la base de données des clauses : duplication dans la base de données des clauses apprises ou dans la base de données dans clauses originales (dans ce dernier cas, nous avons l'assurance que les clauses dupliquées ne seront pas effacées par le solveur) ;

2. Limiter la duplication uniquement aux *bonnes* clauses, *i.e.*, ayant une taille ou un score LBD inférieur à un certain seuil.

Résumé de notre approche

L'approche de duplication que nous avons mise en place pour ZigZag se résume aux critères suivants :

— Condition de duplication : Si une clause apprise contient le littéral d'activation des états initiaux ou un littéral introduit par les contraintes de chemin simple, alors elle n'est pas duplicable. En outre, cette clause n'est répliquée que si elle est de *bonne* qualité.

— Duplication limitée au niveau du model checker : Aucune duplication de clauses n'est effectuée dans le solveur. Après chaque nouveau dépliage de la relation de transition et avant l'appel au solveur, l'ensemble des clauses duplicables sont copiées, en renommant simplement l'ensemble des variables, comme suit : Pour une clause duplicable qui a été apprise lors de la résolution d'une des deux formules (BMC ou k-induction) à une profondeur i et où la prochaine profondeur à résoudre est j, chaque variable v_l de la clause à dupliquer est renommée en $v_{l+(j-i)}$.

Exemple 3.5. Duplication pour ZigZag

$$
\begin{array}{rc}
 & \vdots \\
\text{1-ind} : & P_0 \wedge T_{0,1} \wedge \left[Bad_1\right] \\
\text{BMC}_1 : & \left[I_0 \vee \neg act_I\right] \wedge P_0 \wedge T_{0,1} \wedge \left[Bad_1\right] \\
\text{2-ind} : & P_0 \wedge T_{0,1} \wedge P_1 \wedge T_{1,2} \wedge \left[Bad_2\right] \\
\text{BMC}_2 : & \left[I_0 \vee \neg act_I\right] \wedge P_0 \wedge T_{0,1} \wedge P_1 \wedge T_{1,2} \wedge \left[Bad_2\right] \\
 & \vdots
\end{array}
$$

L'ensemble des clauses ayant été apprises lors des deux appels où $k = 1$ (*i.e.*, 1-ind et BMC_1) et ne contenant pas le littéral act_I sont dupliquées après le dépliage de la relation de transition $T_{1,2}$ et avant la résolution de 2-ind et BMC_2. Ensuite, après le dépliage de $T_{2,3}$, l'ensemble des clauses ayant été apprises lors des appels où $k = 1$ et $k = 2$ et ne contenant pas act_I sont dupliquées, et ainsi de suite.

3.5.4 Expérimentations

Nous proposons maintenant la comparaison expérimentale des différentes stratégies, décrites ci-dessous, sur l'ensemble des problèmes provenant de la compétition de model checking sur les circuits 2015 (HWMCC'15) :

— « Ref » correspond à l'algorithme ZigZag de référence (sans duplication). On notera qu'il ne s'agit pas exactement de la même version que dans les parties précédentes. Afin de simplifier l'implémentation du processus du duplication, à chaque cycle nous déplions l'ensemble des variables mémoires et ajoutons des contraintes d'équivalences entre ces nouvelles variables et les variables du cycle précédent. Cela nous permet d'obtenir une relation simple pour retrouver la même variable à une profondeur différente. Ainsi, pour un modèle avec n variables (entrées, mémoires, définitions), si l'on souhaite obtenir la même variable x_i de profondeur i à la profondeur j, il suffit de calculer son indice comme suit : $x_{i+n\times(j-i)}$. Cette version de ZigZag est donc moins performante mais nous permet d'évaluer notre stratégie de duplication. À noter qu'il est également possible d'implémenter la duplication sur la version de ZigZag précédente, il suffit d'implémenter une méthode qui, à partir d'une variable, retourne son équivalent à un cycle donné.

— « T » signifie que nous avons utilisé la méthode définie par Yin *et al.* [2014], *i.e.*, des littéraux d'activation sont ajoutés aux relations de transition. Ainsi, la duplication peut être effectuée non seulement au niveau du model checker mais aussi au niveau du solveur. À noter qu'avec la notation « T » signifie juste que des littéraux d'activation sont ajoutés

TABLE 3.4 – Résultats des différentes méthodes de duplication.

	Ref	T	TL	TLG10	OG3	OG5	LG5
IND	337	340	340	337	333	**329**	332
CEX	117	115	113	117	119	**121**	118
PRO	93	92	94	93	95	**97**	**97**

aux relations de transition, mais qu'aucune duplication n'est effectuée par défaut. Notons également que nous avons ajouté l'élimination haut niveau et les contraintes de simple chemin à leur méthode.

— « O » et « L » signifient que les clauses dupliquées sont conservées dans la base de données des clauses originales et apprises, respectivement.

— « G » signifie que les clauses glues, *i.e.*, qui ont un LBD égal à 2, sont dupliquées.

— $X \in \{3, 5, 10\}$, signifie que les clauses ayant une taille inférieure ou égal à X sont dupliquées.

Le tableau 3.4 récapitule le nombre de problèmes pour lesquels un contre-exemple a été trouvé (CEX), la propriété a été vérifiée (PRO), et où le résultat n'a pas pu être déterminé (IND) dans les mêmes conditions imparties que précédemment (*i.e.*, 1h et 8Go), pour un ensemble représentatif de stratégie parmi l'ensemble des stratégies exécutées.

Analyse des résultats

— T : Avec l'ajout des littéraux d'activation aux relations de transition, 3 problèmes ne sont plus résolus par rapport à l'algorithme de référence. Cela correspond aux problèmes qui nécessitent presque la totalité du temps imposé pour être résolus par l'algorithme de référence. Avec l'ajout des littéraux d'activation qui ralentissent la résolution comme illustrée sur la figure 3.6, ces problèmes ne sont alors plus résolus dans le temps imparti.

— TL : Le nombre de clauses dupliquées étant important, nous avons voulu voir si en dupliquant toutes les clauses duplicables comme décrit dans Yin *et al.* [2014], tout en utilisant la gestion agressive des clauses apprises, nous étions en mesure d'améliorer le nombre d'instances résolues. Nous n'obtenons pas d'amélioration en nombre d'instances résolues, néanmoins en regardant les résultats en détail, nous avons observé des gains sur le temps de résolution pour certains problèmes du même ordre que ceux décrit dans Yin *et al.* [2014]. Cependant, il y a également de nombreux problèmes pour lesquels les performances sont réduites. Les résultats détaillés sont représentés dans la figure 3.7. La figure de gauche compare le temps de résolution des problèmes résolus par les deux algorithmes

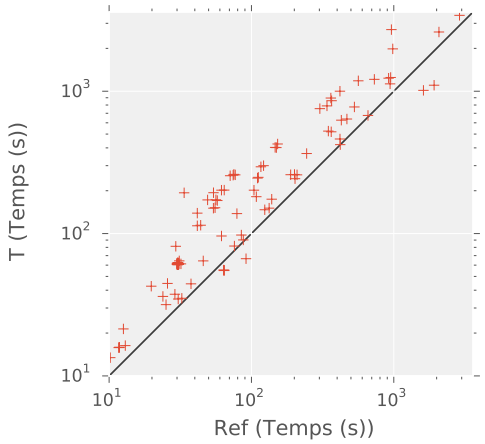


FIGURE 3.6 – Comparaison du temps de résolution entre « Ref » et « T ».

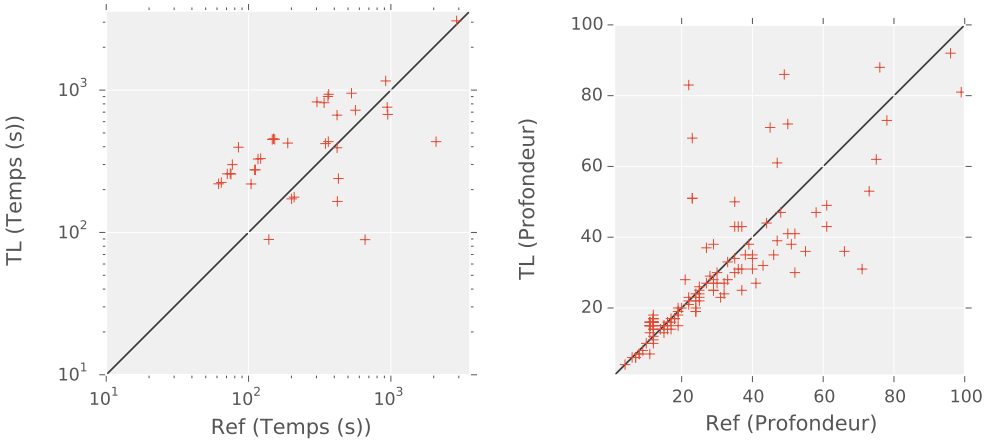


FIGURE 3.7 – Comparaison du temps de résolution et de la profondeur atteinte par l'algorithme ZigZag de référence et la version de Yin *et al.* [2014].

(Ref et TL). Seuls les problèmes intéressants, *i.e.*, nécessitant plus d'une minute de résolution, ont été considérés. Les points au dessus de la diagonale signifient que la méthode avec duplication a mis plus de temps que l'algorithme de référence. La figure de droite, quant à elle, compare la profondeur atteinte lorsque les deux méthodes n'ont pas réussi à déterminer si la propriété est vérifiée ou s'il existe un contre-exemple dans le temps imparti. Seuls les problèmes ayant atteints une profondeur inférieur à 100 sont considérés, au-delà cela signifie que les appels au solveur sont relativement simples et il n'y a donc pas de réel intérêt à utiliser la duplication. Les points en dessous de la diagonale signifient que l'algorithme de référence a atteint une plus grande profondeur. On observe alors que même si dans certains cas, la duplication apporte un gain de performance (*i.e.*, que ce soit en terme de temps de résolution ou de profondeur atteinte). Dans la majorité des cas, ce processus détériore les performances.

— TLG10 : Nous avons également tenté d'améliorer la méthode de Yin *et al.* [2014] en limitant le nombre de clauses dupliquées. Le meilleur résultat que nous avons obtenu consiste à dupliquer les glues clauses et les clauses de taille inférieure ou égale à 10. Néanmoins, le nombre d'instances résolues n'est pas supérieur à la version de référence. En regardant le détail des analyses, nous avons constaté que les graphes (non reportés ici) sont similaires à ceux de la figure 3.7 mais plus équilibrés.

— OG5 : La duplication externe des clauses glues et des clauses de taille inférieure ou égale à 5 est la meilleure stratégie que nous ayons trouvé. Avec cette stratégie, nous réussissons à résoudre 8 problèmes de plus (4 contre-exemple et 4 propriété vérifiée en plus). Cela peut paraître peu, mais cela montre qu'il est encore possible d'améliorer les techniques actuelles, même dans un domaine ultra-concurrentiel comme le model checking sur des instances de référence. De plus, avec le détail des résultats représentés sur la figure 3.8, on observe que, dans la majorité des cas, cette stratégie est plus rapide pour les problèmes résolus par les deux méthodes. Elle atteint une plus grande profondeur pour les instances indéterminées. À noter également que cette version est plus performante que la version des parties précédentes, ce qui signifie qu'il est potentiellement possible d'obtenir encore de meilleurs résultats en effectuant cette stratégie de duplication sur les versions de ZigZag des parties ci-dessus.

— OG3, LG5, et autres stratégies : Nous n'avons reporté ici qu'un échantillon de l'ensemble des stratégies que nous avons évalué (même si nous avons testé par exemple les stratégies O5, O8, OG8, L3, LG3, L5, L8, LG8, etc.). Toutes ces stratégies fournissent globalement de moins bonnes performances en terme d'instances résolues que OG5. Il y a certainement diverses raisons, *e.g.*, trop de clauses dupliquées, la duplication des

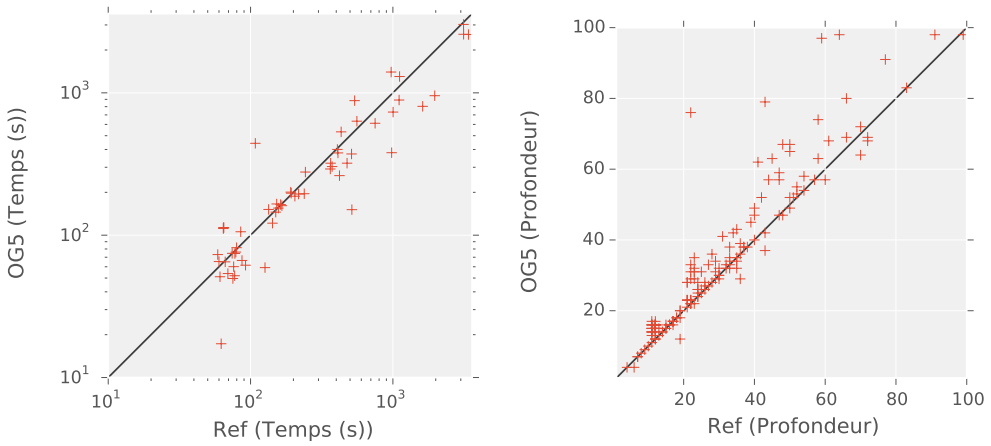


FIGURE 3.8 – Comparaison du temps de résolution et de la profondeur atteinte par l'algorithme ZigZag de référence (sans duplication), et une version où la duplication externe est effectuée (OG5).

clauses dans la base de données des clauses apprises ne s'accorde peut-être pas avec les paramètres par défaut de `Glucose` (nombre de conflits entre chaque suppression et augmentation de la taille de la base après chaque suppression). Ainsi, `Glucose` n'a pas été exhaustivement testé pour le cas où un grand nombre de clauses pouvait être ajouté en un seul coup (cela pourrait avoir des conséquences par exemple sur les stratégies de redémarrage du solveur).

3.5.5 Clauses dupliquées

Dans une dernière expérimentation, nous avons voulu observer le nombre de clauses dupliquées par rapport au nombre de clauses apprises (conflits). La figure 3.9 compare le nombre de conflits et le nombre de clauses dupliquées en distinguant les cas en fonction du résultat : propriété vérifiée, contre-exemple trouvé, résultat indéterminé, en utilisant la stratégie OG5.

On observe que dans la majorité des cas, le nombre de clauses dupliquées ne dépasse pas le nombre de conflits ou reste proche. Le solveur n'a alors pas à gérer un nombre « trop important » de clauses, ce qui pourrait expliquer en partie les bons résultats obtenus par la stratégie OG5.

3.5.6 Tentative pour BMC

Nous avons également voulu regarder si cette méthode était applicable pour BMC. Pour se faire, nous avons utilisé la même stratégie de détection des clauses duplicables que précédemment. Nous avons alors utiliser un littéral

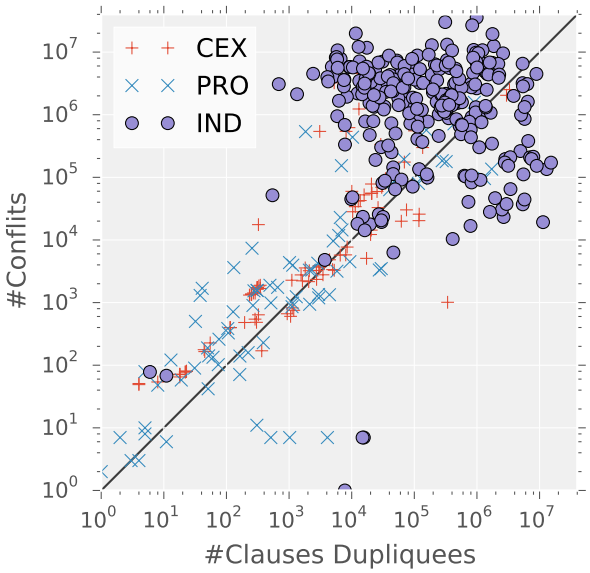


FIGURE 3.9 – Comparaison entre le nombre de clauses dupliquées et le nombre de conflits.

d'activation pour la contrainte des états initiaux. Il est important de noter que même si ce littéral est activé pour chaque appel, cela a un effet négatif sur les performances pour BMC : de nombreuses propagations ne sont pas effectués, toutes les clauses apprises dépendantes des états initiaux contiennent un littéral de plus. Afin d'observer le comportement de la duplication sur BMC, nous avons lancé 4 stratégies sur l'ensemble des problèmes de HWMCC'15 :

1. une version de référence, *i.e.*, BMC avec élimination de variables à haut niveau (121 CEX).
2. une version pour mesurer l'impact de l'ajout du littéral d'activation (119 CEX), c'est-à-dire la version (1.) avec un littéral d'activation pour les états initiaux mais toujours sans duplication, *i.e.* :

$$\left[\neg act_I \vee I_0\right] \wedge \bigwedge_{i=0}^{k-1} T_{i,i+1} \wedge \left[\neg act_B \vee Bad_k\right]$$

3. la version (2.) avec la stratégie de duplication LG5 (118 CEX)
4. la version (2.) avec la stratégie de duplication OG5 (120 CEX)

Discussion

La version de référence (1.) trouve 3 CEX de moins que les versions de référence de BMC des parties précédentes pour les mêmes raisons que pour la version de ZigZag avec duplication. À savoir, nous utilisons un dépliage

simplifié pour avoir un facteur simple pour retrouver les mêmes variables à des cycles différents. L'analyse des résultats montre que la version (4.) trouve 2 CEX non trouvé par la version (1.). Cela signifie donc que la duplication peut avoir un impact important sur certaines instances. Néanmoins, nous pouvons globalement conclure que les gains de performance apportés par la duplication ne couvre pas les pertes dues à la non propagation de la contrainte des états initiaux.

3.5.7 Travaux futurs sur la duplication

Comme évoqué précédemment, la première amélioration possible est d'intégrer la duplication avec un processus de dépliage optimisé. Dans un second temps, nous avons également pensé à dupliquer le score d'activité des variables pour VSIDS. Nous avons tenté plusieurs stratégies : affecter aux nouvelles variables dépliées un score égal à celui de la même variable au cycle précédent, décaler l'ensemble des scores d'un cycle à chaque dépliage, etc. Ces stratégies ont eu des résultats variés mais aucune n'a permis d'améliorer nettement les performances du model checker. Il pourrait donc être possible de continuer sur cette voie. Une autre piste pourrait être d'améliorer l'algorithme techniquement, par exemple, définir une base de données spéciale pour les clauses dupliquées, ou encore surveiller les clauses dupliquées en *one-watched*, etc. Il est également possible d'imaginer d'autres critères de sélection des clauses à dupliquer : aléatoire, différence de profondeur (comme dans le chapitre 4), etc. Une autre idée que nous souhaitions essayer est d'évaluer l'« utilité » des clauses dupliquées. Par exemple, si une clause est dupliquée depuis plusieurs cycles mais n'est jamais apparue dans un conflit alors on pourrait la retirer de l'ensemble des clauses duplicables. Pour BMC, il serait également intéressant d'implémenter la version de Strichman [2004] (marquage des clauses) tout en limitant le nombre de clauses dupliquées comme nous le faisons ici. Cela permettrait de combiner le dépliage optimisé et la duplication.

3.6 Dépliage de la relation de transition

Eén et Sörensson [2003b] sont les premiers à évoquer la possibilité et l'intérêt de déplier la relation de transition dans les deux sens lors de la vérification par induction temporelle. Dans leurs expérimentations, ils ont arbitrairement choisi d'effectuer le dépliage vers l'avant lorsqu'ils effectuent ZigZag, en suivant l'intuition que la contrainte des états initiaux est plus forte que celle des mauvais états. De plus, comme il est plus naturel de déplier la relation de transition vers l'avant, la plupart des model checkers la déplient de cette façon. Cependant, aucune étude sur l'impact du sens de dépliage n'est disponible dans la littérature. Nous avons alors comparé les deux techniques de dépliage, c'est-

à-dire effectuer l'algorithme ZigZag en dépliant la relation de transition vers l'avant ou bien vers l'arrière. Enfin, nous avons également voulu voir s'il était profitable de déplier la relation dans les deux sens alternativement. Nous avons alors défini une méthode de dépliage hybride, que nous avons implémentée et dont nous reportons dans la suite l'étude expérimentale.

3.6.1 Dépliage arrière

Pour bien illustrer la différence entre les deux dépliages, nous indiquons ci-dessous, les premiers appels au solveur de l'algorithme ZigZag lorsque le dépliage est effectué en arrière :

$$\begin{aligned}
\text{0-ind} &: & Bad_0 \\
\text{BMC}_0 &: & \left[I_0\right] \wedge Bad_0 \\
\text{1-ind} &: & P_1 \wedge T_{1,0} \wedge Bad_0 \\
\text{BMC}_1 &: & \left[I_1\right] \wedge P_1 \wedge T_{1,0} \wedge Bad_0 \\
\text{2-ind} &: & P_2 \wedge T_{2,1} \wedge P_1 \wedge T_{1,0} \wedge Bad_0 \\
\text{BMC}_2 &: & \left[I_2\right] \wedge P_2 \wedge T_{2,1} \wedge P_1 \wedge T_{1,0} \wedge Bad_0
\end{aligned}$$

Il y a deux changements majeurs à observer par rapport à la version avec dépliage vers l'avant :

1. La partie statique, à savoir, la partie qui reste toujours activée à la même profondeur, n'est pas la même. Lorsque le dépliage est effectué vers l'avant, ce sont les contraintes des états initiaux qui restent définies à la profondeur 0. Avec le dépliage vers l'arrière, ce sont les contraintes des mauvais états qui restent statiques.
2. En outre, en dépliant la relation de transition vers l'arrière, la partie statique reste activée pour chaque appel au solveur. Quand le dépliage est effectué vers l'avant, la partie statique (*i.e.*, les états initiaux) n'est activée que pour BMC.

Les détails sur l'implémentation de ce dépliage sont fournis dans la partie 5.9.

3.6.2 Dépliage dans les deux directions

Après avoir effectué des expérimentations sur différents modèles, nous avons observé le comportement suivant. L'algorithme ZigZag semble plus performant pour trouver des CEXs lorsque la relation de transition est dépliée vers l'avant. À l'inverse, il est généralement plus à rapide à vérifier les modèles où la propriété est vraie lorsque la relation de transition est dépliée vers l'arrière. Cela semble logique au regard de la partie qui reste statique dans les deux cas. Nous avons donc voulu voir s'il était possible de profiter de ces deux avantages

en effectuant un dépliage hybride : la moitié des relations de transition sont dépliées vers l'avant et l'autre moitié vers l'arrière, tout en dépliant alternativement dans les deux sens. De cette façon, toutes les clauses apprises qui sont dépendantes d'une des contraintes (états initiaux ou mauvais états), *i.e.*, la partie non statique, ne sont pas nécessairement perdues pour les appels suivants au solveur. Par soucis de clarté, nous découpons la formule du dépliage hybride en 3 parties comme suit. Nous définissons d'abord la partie concernant le dépliage de la relation de transition vers l'avant :

$$T_{av}(k) \ := \bigwedge_{i=0}^{(k-1)/2} \big(P_{2i} \wedge T_{2i,2i+2}\big)$$

Puis, celle du dépliage vers l'arrière :

$$T_{ar}(k) \ := \bigwedge_{i=0}^{(k-2)/2} \big(P_{2i+3} \wedge T_{2i+3,2i+1}\big)$$

Ensuite, il faut définir la partie de la formule qui joint ces deux parties, c'est-à-dire celle qui nous permet de connecter ces deux dépliages ensembles. Comme précédemment, m représente les variables mémoires, *i.e.*, un sous-ensemble de V sans les entrées et les définitions.

$$EQ(k) \ := m_{2.\lfloor \frac{k+1}{2} \rfloor} = m_{2.\lceil \frac{k+1}{2} \rceil - 1}$$

Pour un k donné, la formule du dépliage hybride est donc la suivante :

$$\big[I_0\big] \wedge T_{av}(k) \wedge \big[EQ(k)\big] \wedge T_{ar}(k) \wedge Bad_1$$

Les parties entre crochets représentent les parties qui sont activées et désactivées à l'aide de littéraux d'activation, que nous avons omis dans la formule pour plus de lisibilité. I_0 est uniquement activé pour les appel BMC (*i.e.*, désactivé pour les appels de k-induction). Les contraintes $EQ(k)$ sont activées et désactivées au fur et à mesure du dépliage, c'est-à-dire que pour une profondeur k, la contrainte $EQ(k)$ correspondante est activée et toutes les autres sont désactivés. Par exemple, $EQ(0)$ est activé lors de la résolution des formules de BMC_0 et 0-ind. Puis, cette contrainte est désactivée de façon permanente (*i.e.*, pour tous les appels suivants) avant la résolution de la profondeur suivante (k=1). Profondeur à laquelle la contrainte $EQ(1)$ est à son tour activée, et ainsi de suite. Pour mieux illustrer ce dépliage hybride, les premiers appels au

TABLE 3.5 – Résultats des différentes méthodes de dépliage.

	Avant	Arrière	Hybride
IND	333	327	332
CEX	118	118	118
PRO	96	102	97

solveur sont données ci-dessous :

$$
\begin{aligned}
\text{0-ind} &: [m_0 = m_1] \wedge Bad_1 \\
\text{BMC}_0 &: [I_0] \wedge [m_0 = m_1] \wedge Bad_1 \\
\text{1-ind} &: P_0 \wedge T_{0,2} \wedge [m_2 = m_1] \wedge Bad_1 \\
\text{BMC}_1 &: [I_0] \wedge P_0 \wedge T_{0,2} \wedge [m_2 = m_1] \wedge Bad_1 \\
\text{2-ind} &: P_0 \wedge T_{0,2} \wedge [m_2 = m_3] \wedge P_3 \wedge T_{3,1} \wedge Bad_1 \\
\text{BMC}_2 &: [I_0] \wedge P_0 \wedge T_{0,2} \wedge [m_2 = m_3] \wedge P_3 \wedge T_{3,1} \wedge Bad_1 \\
\text{3-ind} &: P_0 \wedge T_{0,2} \wedge P_2 \wedge T_{2,4} \wedge [m_4 = m_3] \wedge P_3 \wedge T_{3,1} \wedge Bad_1 \\
\text{BMC}_3 &: [I_0] \wedge P_0 \wedge T_{0,2} \wedge P_2 \wedge T_{2,4} \wedge [m_4 = m_3] \wedge P_3 \wedge T_{3,1} \wedge Bad_1 \\
\text{4-ind} &: P_0 \wedge T_{0,2} \wedge P_2 \wedge T_{2,4} \wedge [m_4 = m_5] \wedge P_5 \wedge T_{5,3} \wedge P_3 \wedge T_{3,1} \wedge Bad_1
\end{aligned}
$$

Ainsi, toutes les clauses apprises impliquant les relations de transition dépliées vers l'avant et dépendantes des états initiaux sont conservées pour chaque appel BMC successif au solveur. De la même façon, toutes les clauses apprises impliquant les relations de transition dépliées vers l'arrière et dépendantes des mauvais états sont gardées pour tous les appels au solveur.

3.6.3 Expérimentations

Dans les mêmes conditions que précédemment, à savoir, un temps de calcul limité à 1h et un plafond à 8Go de mémoire, nous avons exécuté ces différents dépliages sur l'ensemble des problèmes d'HWMCC'15. Le tableau 3.5 résume les résultats pour les différentes stratégies de dépliage. Pour avoir une vision plus détaillée des résultats, nous avons comparé dans la figure 3.10 les résultats obtenus par les différentes stratégies de dépliage sur l'ensemble des problèmes nécessitant plus de 10s pour être résolus par les 3 techniques.

Analyse des résultats

Nous observons une amélioration lorsque le dépliage est effectué vers l'arrière : 6 instances de plus sont résolues par rapport au dépliage vers l'avant. En outre, la partie gauche de la figure 3.10 nous montre que l'approche vers l'arrière semble globalement plus rapide pour les problèmes difficiles, à l'exception de certains problèmes contenant un CEX. Les résultats pour le dépliage

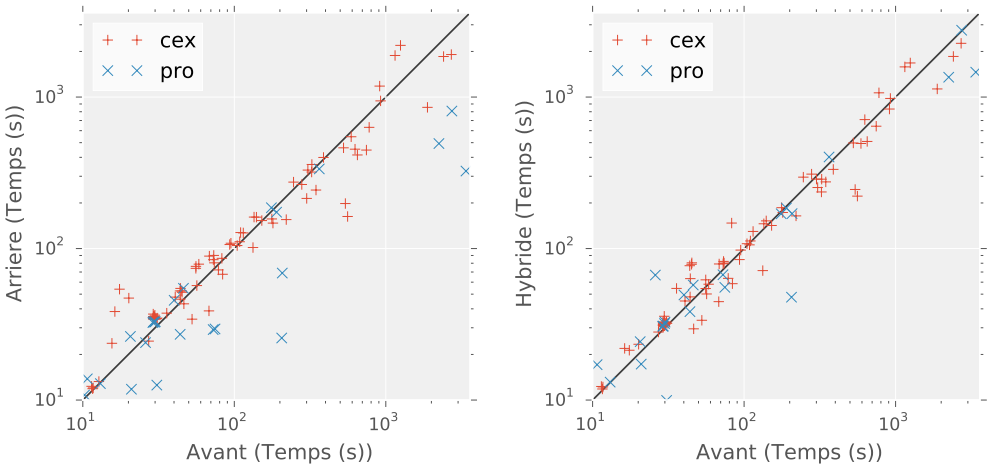


FIGURE 3.10 – Comparaison du temps de résolution entre, à gauche, le dépliage vers l'avant et vers l'arrière, et à droite entre le dépliage vers l'avant et le dépliage hybride.

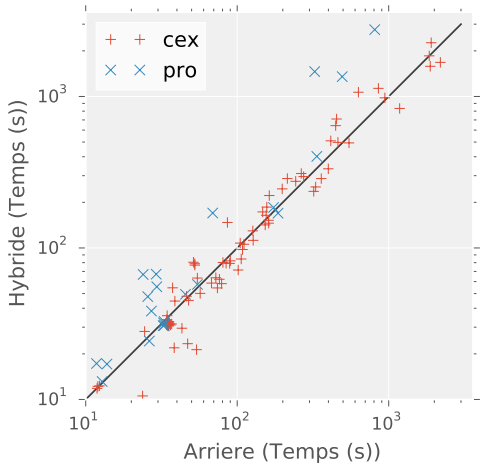


FIGURE 3.11 – Comparaison du temps de résolution entre le dépliage vers l'arrière et le dépliage hybride.

hybride semble plus mitigés. Il y a une instance supplémentaire résolue et globalement un temps de résolution légèrement meilleur pour les problèmes difficiles (partie droite de la figure 3.10). Néanmoins, les écarts en faveur du dépliage hybride sont moins importants qu'avec le dépliage vers l'arrière. Qui plus est, lorsque l'on compare le dépliage arrière et hybride (illustré sur la figure 3.11), le dépliage hybride semble moins performant que le dépliage vers l'arrière pour les problème difficile.

Conclusion et travaux futurs

D'après nos expérimentations sur l'ensemble des problèmes de la compétition, il semble que le dépliage vers l'arrière soit la meilleure stratégie à utiliser lorsque l'on effectue l'algorithme ZigZag. Il serait alors intéressant de combiner ces dépliages avec la duplication des clauses. Pour le dépliage vers l'arrière, la même stratégie de duplication que celle utilisée pour le dépliage vers l'avant est applicable. Nous avons effectué quelques expérimentations préliminaires et la duplication semble également améliorer les performances avec le dépliage arrière. Il faudrait donc procéder à une implémentation stable et une expérimentation complète pour confirmer ces résultats. Pour le dépliage hybride, il est nécessaire de définir une autre stratégie de duplication, par exemple un ensemble de clauses à dupliquer pour les relations de transitions dépliées vers l'avant et un autre pour celles dépliées vers l'arrière.

Chapitre 4

Analyse des formules CNFs générées par BMC

Ce chapitre décrit une étude que nous avons effectuée sur la structure des formules CNFs générées lors de la vérification de modèle avec BMC. Le chapitre précédent suppose ainsi, en filigrane, qu'une certaine structure, due au déroulements successifs de la relation de transition, pourrait être exploité efficacement par le solveur SAT. Ce travail a été réalisé en collaboration avec Jesús Giráldez-Cru. L'objectif de ce travail est donc de voir si nous pouvions spécialiser notre solveur SAT afin d'améliorer ses performances sur les instances BMC.

Parmi toutes les études ayant été proposées pour étudier expérimentalement les performances des solveurs SAT, il a été mis en avant la forte modularité des instances industrielles sur lesquelles les solveurs SAT sont particulièrement efficaces. Cette modularité met en avant l'existence de communautés dans l'instance encodée en CNF. Nous avons donc voulu observer s'il existait un lien entre l'encodage haut niveau et la structure en communautés de formules CNF encodant des instances de BMC, ainsi qu'entre l'encodage haut niveau et l'ensemble des mécanismes mis en œuvre dans le solveur lors de la résolution. Pour se faire, nous avons marqué chaque variable avec sa profondeur lors de l'encodage, *i.e.*, la transition de relation pendant laquelle elle est ajoutée au solveur. Puis, nous avons observé comment les variables étaient réparties dans les communautés et comment ces variables se comportaient lors de la résolution. Par exemple, nous avons regardé si les décisions se faisaient plus au « milieu » de la formule ou encore si les clauses apprises pouvaient contenir des variables provenant d'un large ensemble de transitions comme nous le détaillons par la suite.

Afin de ne pas introduire de biais dans les formules, pour ce chapitre nous avons utilisé une génération simple : pour un modèle et une profondeur donnée, nous générons une formule CNF correspondant à l'instance BMC pour cette profondeur sur laquelle nous n'effectuons aucune élimination, aucune simplifi-

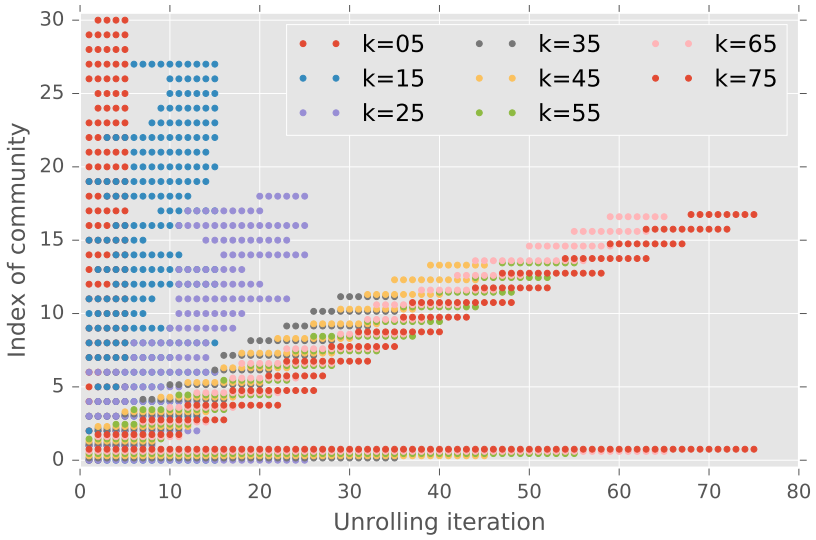


FIGURE 4.1 – Relation entre la profondeur de dépliage et la structure en communautés pour l'instance `6s7`, pour différentes valeurs de k.

cation et aucune propagation des états initiaux.

4.1 Communautés et profondeur de dépliage

L'origine de la structure en communautés des formules SAT provenant du milieu industriel reste indéfinie, même si la structure en communauté ne fait pas de doutes. Dans de précédents travaux, l'ensemble hétérogène des instances industrielles utilisé dans les compétitions SAT a été analysé sans s'intéresser aux problèmes qu'ils encodent à plus haut niveau. En outre, il a été montré que la grande majorité de ces instances possèdent une structure en communautés. Nous avons alors décidé d'analyser plus précisément les relations entre la structure en communautés de formules BMC et la structure haut niveau du problème. L'idée étant toujours de spécialiser notre model checker, c'est-à-dire aider et améliorer les performances de notre solveur SAT à l'aide d'informations plus haut niveau.

Pour une instance donnée, nous pouvons générer différentes formules CNF représentant BMC_k, en jouant sur la profondeur k. Ensuite, sur chacune des formules, nous appliquons l'algorithme de Louvain (Blondel *et al.* [2008]). Nous obtenons alors un score de modularité et un partitionnement. Ainsi, pour chaque variable nous avons sa profondeur de dépliage x et son indice de communautés y, *i.e.*, l'indice de la partition à laquelle elle appartient. À noter qu'il est possible que deux ou plusieurs variables soient caractérisées par les mêmes coordonnées (x, y), lorsqu'elles appartiennent à la même profondeur et qu'elles ont été affectées à la même communauté.

Dans la figure 4.1, nous représentons la relation entre la profondeur de dépliage et la structure en communautés des formules BMC_k encodant l'instance `6s7` (HWMCC'15), pour les valeurs suivantes $k \in \{5, 15, 25, 35, 45, 55, 65, 75\}$. Nous utilisons les coordonnées (x, y) définies ci-dessus : pour chaque profondeur, il y a un point en (x, y) s'il y a au moins une variable dépliée à la profondeur x appartenant à la communauté y. Pour certain k, nous avons légèrement décalé les valeurs de y pour améliorer la clarté de la figure. Nous pouvons alors observer que lorsque le nombre de dépliage est petit (*e.g.*, $k = 5$), la structure en communautés ne semble pas corrélée à la profondeur de dépliage. En effet, la plupart des communautés contiennent des variables qui proviennent de toutes les profondeur (on notera que le nombre de communautés est plus grand que k). Cependant, lorsque k augmente (plus précisément à partir de $k = 35$), un motif semble se dessiner suggérant la présence d'une corrélation entre la profondeur et les communautés.

Néanmoins, chaque point représenté sur la figure précédente peut être le fait d'une seule variable. Il est donc nécessaire d'utiliser une mesure afin de vérifier si cette corrélation existe bel et bien. Pour cela, nous calculons le coefficient de corrélation de Pearson [1895] r entre ces deux variables X (Profondeur) et Y (Communauté) pour tous les points (x, y). Ce coefficient est défini comme suit :

$$r_{X,Y} = \frac{\sigma_{XY}}{\sigma_X \ \sigma_Y} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^{n} (y_i - \bar{y})^2}}$$

où n correspond à la taille de l'échantillon (nombre de variables dans la CNF), *i.e.*, à la taille des ensembles de points $X = \{x_1, \ldots, x_n\}$ et $Y = \{y_1, \ldots, y_n\}$. Les notations $\bar{x}$, σ_X, $\bar{y}$ et σ_Y correspondent respectivement à la moyenne des variables et leur écart type et σ_{XY} représente la covariance. Lorsque r est proche de 1, les variables X et Y sont très corrélées. À l'inverse lorsque r est proche de 0, les variables ne sont pas corrélées.

Par exemple, pour l'instance de la figure 4.1, nous obtenons les résultats suivants : $r = 0.423$ pour $k = 5$, $r = 0.769$ pour $k = 25$, et $r = 0.968$ pour $k = 45$. Ces résultats sont une première confirmation que la structure en communautés des formules BMC semble représenter le processus de dépliage.

Nous vérifions maintenant si ce constat (*i.e.*, cette corrélation) se retrouve dans toutes les formules BMC. Pour cela, nous avons analysé les 513 problèmes (sans les instances `intel`) provenant d'HWMCC'15. Pour chacune de ces instances, nous avons généré différentes formules BMC_k, avec $k \in \{5, 10, 20, 40, 60\}$. Puis, pour chacune de ces formules, nous avons partitionné la formule en communautés et calculé le coefficient de corrélation r décrit ci-dessus. Dans cette expérimentation, nous avons omis les instances pour lesquelles le calcul du partitionnement nécessitait plus de 5000 secondes (À noter que certaines formules sont très grandes lorsque la profondeur k est importante). Néanmoins, l'ensemble des instances restantes reste conséquent : 509 formules lorsque $k = 5$ et 400 formules lorsque $k = 60$.

k	$\overline{r}$	σ_r	med_r	$\max_r$	$P_{5\%}$	$\min_r$
5	0.594	0.304	0.667	0.995	0.055	0.000
10	0.677	0.304	0.808	0.997	0.108	0.023
20	0.856	0.170	0.918	0.999	0.492	0.108
40	0.892	0.155	0.956	0.999	0.580	0.109
60	0.904	0.152	0.973	0.999	0.643	0.131

TABLE 4.1 – Statistiques du coefficient de corrélation r entre les communautés et la profondeur de dépliage sur l'ensemble des problèmes provenant de HWMCC'15, pour différentes valeurs de k.

Le tableau 4.1 reporte les résultats de cette analyse. Pour chacune des profondeurs k, nous avons calculé la moyenne ($\overline{r}$), l'écart type (σ_r), la médiane (med_r), le maximum ($\max_r$), le minimum ($\min_r$), et le percentile 5% ($P_{5\%}$) des coefficients r pour toutes les instances.

Les résultats sont sans équivoque, la corrélation entre la profondeur et la structure en communautés est fortement établie : la moyenne et la médiane sont supérieurs à 0.5 et tendent vers 1 lorsque k augmente, qui plus est avec un écart type assez faible. Nous observons également qu'il existe certaines instances où cette corrélation est très faible (minimum). Cependant, cela semble être le cas pour seulement un nombre réduit de problèmes comme le montre le percentile 5% lorsque $k \geq 20$.

Cette analyse confirme alors que la structure en communautés des formules BMC provient du processus de dépliage de la relation de transition. Cela peut sembler être un résultat anecdotique : il parait évident que le dépliement va induire une structure sur la formule encodée. Mais, cette étude est importante à deux titre. Tout d'abord, même si le résultat n'est pas surprenant, il s'agit de la première étude, à notre connaissance, qui confirme formellement ce lien. Ensuite, comme nous le verrons dans la suite, les choses ne sont pas si simples et les communautés sont en fait à cheval sur un petit nombre d'itérations. La figure 4.1 suggère ainsi que les communautés construites par l'algorithme de regroupement (*Clustering*) associent des variables de plusieurs profondeurs contiguës. Par conséquent dans la partie suivante, nous avons souhaité vérifier si cette observation (*i.e.*, les communautés sont étendues sur plusieurs profondeurs) reste vraie pour l'ensemble des formules BMC.

4.1.1 Les communautés s'étendent sur plusieurs profondeurs

Pour vérifier cette observation, nous avons partitionné l'ensemble des variables de chaque formule tel que chaque profondeur forme une communauté, c'est-à-dire que toutes les variables dépliés à la même profondeur appartiennent

	Q					
k	$\overline{Q}$	σ_Q	med_Q	$\min_Q$	$\max_Q$	$P_{5\%}$
5	**0.850**	0.053	0.857	0.681	0.970	0.747
10	**0.874**	0.047	0.877	0.701	0.977	0.786
20	**0.887**	0.043	0.894	0.731	0.981	0.813
40	**0.906**	0.038	0.907	0.759	0.986	0.837
60	**0.917**	0.036	0.917	0.776	0.988	0.856
	Q_d					
k	$\overline{Q_d}$	σ_{Q_d}	med_{Q_d}	$\min_{Q_d}$	$\max_{Q_d}$	$P_{5\%}$
5	0.616	0.083	0.607	0.480	0.826	0.387
10	0.687	0.085	0.676	0.549	0.901	0.450
20	0.728	0.086	0.717	0.588	0.945	0.488
40	0.751	0.086	0.739	0.609	0.968	0.509
60	0.761	0.094	0.767	0.614	0.976	0.516

TABLE 4.2 – Statistiques des modularités Q et Q_d sur l'ensemble des problèmes de HWMCC15, pour différentes valeurs de k. La plus grande valeur entre $\overline{Q}$ et $\overline{Q_d}$ est mise en gras pour chaque k.

à la même communauté. Puis, nous avons calculé la modularité Q_d de ce partitionnement, que nous avons comparée avec la modularité Q de la partition générée par l'algorithme de regroupement.

Le tableau 4.2 reporte les moyennes, écarts types, médianes, min, max, et percentiles 5% des modularité Q et Q_d. Comme attendu, les modularités Q et Q_d augmentent lorsque la profondeur k augmente. Les résultats nous montrent que la modularité Q est significativement plus grande que Q_d, indépendamment de la profondeur k. Notons que lors du calcul des communautés avec l'algorithme de regroupement avec un k relativement grand (*e.g.*, $k = 60$), le nombre de communautés est généralement plus petit que k. Nous remarquons également que la modularité fluctue moins que le coefficient de corrélation.

Nous pouvons tirer deux conclusions de cette analyse :

1. La structure en communautés des formules BMC est issue du processus de dépliage de la relation de transition. En effet, la modularité est élevée, et ce, même en regroupant simplement les variables par profondeur (voir Q_d). Nous constatons également que plus la profondeur est grande, plus la structure en communautés est définie.
2. L'algorithme de regroupement identifie des communautés qui regroupent des variables de plusieurs profondeurs contiguës. On peut alors se demander si ce nombre de profondeurs regroupés dans une même communauté peut représenter une certaine dépendance temporelle du modèle et s'il est possible de l'exploiter. Nous n'avons néanmoins pas d'éléments de

réponse à apporter pour l'instant.

4.2 Profondeurs des décisions, des résolutions et des clauses apprises

Dans la partie précédente, nous avons analysé statiquement la structure des formules CNF encodant des problèmes BMC. Nous proposons maintenant de regarder l'aspect dynamique, *i.e.*, ce qui se produit lors de la résolution par le solveur. Nous nous intéressons plus particulièrement à la profondeur des variables intervenant dans les décisions, les résolutions (se produisant dans les analyses de conflits) et dans les clauses apprises.

Pour cette expérimentation, nous avons déplié les problèmes de HWMCC'15 jusqu'à obtenir des formules satisfaisables, ou des formules n'étant pas été résolues en moins de 6600 secondes. Pour les formules satisfaisables, nous avons également pris les formules à la profondeur précédente. Les instances faciles (*i.e.*, résolues en moins d'une minute) ont été écartées. Nous avons alors obtenu un ensemble de 106 formules BMC, que nous avons donné en entrée à une version modifiée de `Glucose` permettant d'observer les variables et leurs profondeurs. Nous avons ensuite calculé trois ensembles de variables caractérisant l'état du solveur à chaque conflit :

1. L'ensemble *dec* contenant toutes les variables de décisions. À noter que certaines de ces décisions ne sont pas nécessairement liées au conflit courant.
2. L'ensemble *res* contenant toutes les variables utilisées lors de l'analyse du conflit.
3. L'ensemble *learnt* contenant toutes les variables de la clause apprise.

On notera que *res* est disjoint des variables de décisions et des variables de la clause apprise ($dec \cup learnt$).

Pour chacun de ces ensembles et à chaque conflit, nous avons regardé la taille des ensembles ainsi que certaines statistiques sur la profondeur des variables de ces ensembles : moyenne, médiane, écart type, minimum, maximum, MAD (*Median Absolute Deviation*), et le coefficient de dissymétrie. Sur les problèmes générant des millions de conflits, il n'est pas possible de résumer simplement toutes ces données (chaque donnée étant liée à un seul conflit). Nous avons alors décidé de regrouper ces valeurs par paquets de 10,000. Tous les 10,000 conflits, nous calculons les mêmes statistiques (moyenne, médiane, etc) sur les mesures précédentes. Nous obtenons alors par exemple : la médiane des moyennes, le maximum des écarts types, etc. Après avoir investigué l'ensemble de ces données, nous avons remarqué que la moyenne des moyennes pour chacun des ensembles (*dec*, *res* et *learnt*) permet de tirer quelques conclusions. Cela permet d'estimer les valeurs mesurées, tout en restant facilement

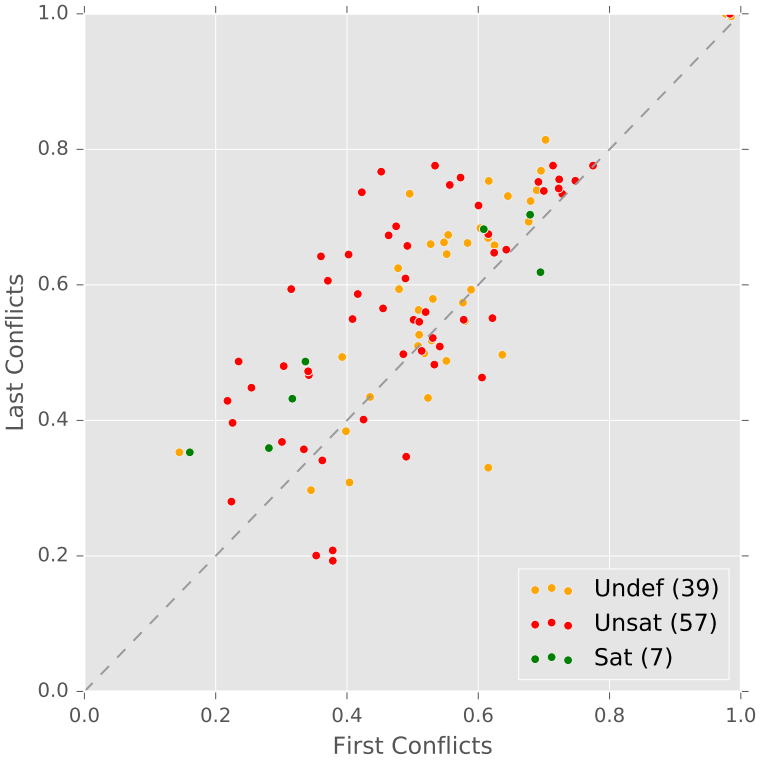


FIGURE 4.2 – Comparaison entre la profondeur des variables de décisions lors des premiers conflits et des derniers. Les valeurs sont normalisées avec la profondeur de dépliage de chacune des formules.

compréhensible. Nous avons alors utilisé dans les parties suivantes les valeurs moyennes du nombre moyen de profondeurs trouvé dans chaque ensemble. À noter que ces valeurs ont été calculées pour des instances satisfaisables et insatisfaisables mais aussi pour les instances qui ne terminent pas dans le temps imparti. Précisément, il y a 7 formules satisfaisables, 61 insatisfaisables et 39 indéterminées.

4.2.1 Relation entre la profondeur et la progression du solveur

Dans notre première expérience, nous regardons s'il existe une progression dans les décisions et les résolutions du solveur par rapport à la profondeur des variables. Autrement dit, est-ce que le solveur commence par travailler à certaines profondeurs pour ensuite se déplacer vers d'autres ? Nous avons donc comparé les valeurs de *dec*, *res*, et *learnt* lors des premiers conflits avec

celles mesurées lors des derniers conflits. La figure 4.2 représente cette comparaison pour l'ensemble des décisions *dec*. Les premiers conflits sont calculés entre les 10,000ème et 20,000ème conflits. Cela permet d'éviter tout biais qui pourrait être introduit par la phase d'initialisation de VSIDS (décrit dans la partie 2.1.2) lors des premiers conflits, où le score d'activité des variables n'est pas encore bien défini. Les derniers conflits correspondent simplement à ceux rencontrés avant la fin de l'exécution du solveur (solution trouvée ou non). Les résultats sont normalisés avec la profondeur k de chacune des instances BMC_k. Une valeur proche de 1 signifie que la moyenne des moyennes des profondeurs de l'ensemble des variables de *dec* est proche des derniers dépliages. À l'inverse, une valeur proche de 0 signifie que la moyenne est proche des premiers dépliages. Comme nous le voyons sur la figure 4.2, pour la majorité des instances, les profondeurs impliquées dans les premiers conflits sont plus petits que ceux des derniers conflits, et cela quel que soit le résultat de l'instance. Cela suggère que le solveur a tendance à commencer la recherche dans de plus petites profondeurs de dépliage que celles dans lesquelles ils continuent par la suite. Nous n'avons reporté ici que le cas de l'ensemble des décisions. Cependant, le même phénomène se produit pour les variables de résolutions (res) et les clauses apprises ($learnt$), avec un nuage de points très similaire.

Pour résumer, il semblerait que les solveurs CDCL sur les instances BMC suivent le comportement suivant : les efforts du solveur (décisions, analyses de conflits, et clauses apprises) semblent se déplacer vers des profondeurs plus grandes au fur et à mesure de la recherche.

4.2.2 Profondeur et LBD

Newsham *et al.* [2014] ont montré une corrélation entre le score LBD et le nombre de communautés sur les instances industrielles de la compétition SAT de 2013 pour une majorité des cas. Ici, nous voulons savoir si le LBD est corrélé avec la profondeur des variables dans les formules BMC. La figure 4.1 suggère que les communautés sont définies sur un petit nombre de profondeurs contiguës. Nous avons testé plusieurs hypothèses sur la corrélation entre le LBD et les profondeurs des variables de la clause correspondante. La plus grande corrélation que nous avons trouvée met en relation le LBD et le $(max - min)$ des profondeurs des variables de la clause. Nous pouvons représenté cette relation avec une *heat map* pour chaque instance. La figure 4.3 illustre cette relation pour l'instance `beemprdcell2f1` et $k = 103$. Il s'agit d'une instance où la *heat map* fonctionne bien, c'est-à-dire où la corrélation est bien marquée.

Cependant, même si nous l'avons observé pour une majorité d'instance, ce n'est pas le cas pour toutes les formules. Afin de résumer tous les résultats, nous avons calculé pour chaque problème le coefficient de corrélation de Pearson r entre le LBD et la valeur $(max - min)$ de toutes les clauses apprises. Le coefficient de corrélation de Pearson mesure des relations linéaires. Or, nous

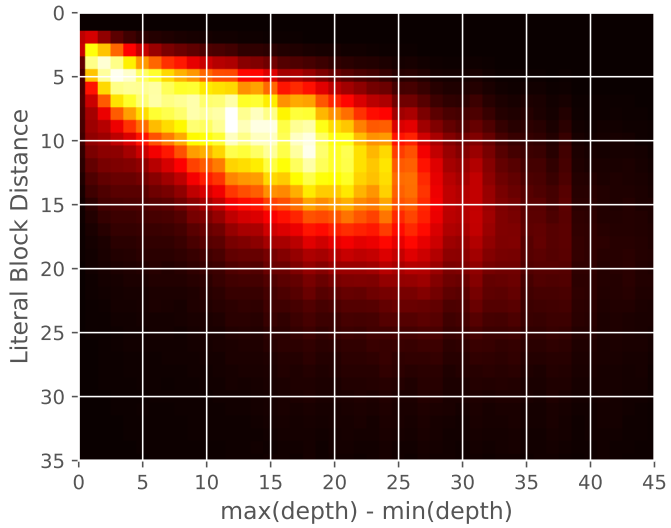


FIGURE 4.3 – *Heat map* montrant la corrélation entre le score LBD (partie 2.1.2) de la clause et la valeur $(max - min)$ des profondeurs de ses variables. Le coefficient de corrélation de Pearson est seulement de 0.315 sur cet exemple, ce qui montre que même les petites valeurs sont des bons indicateurs sur notre ensemble de problème.

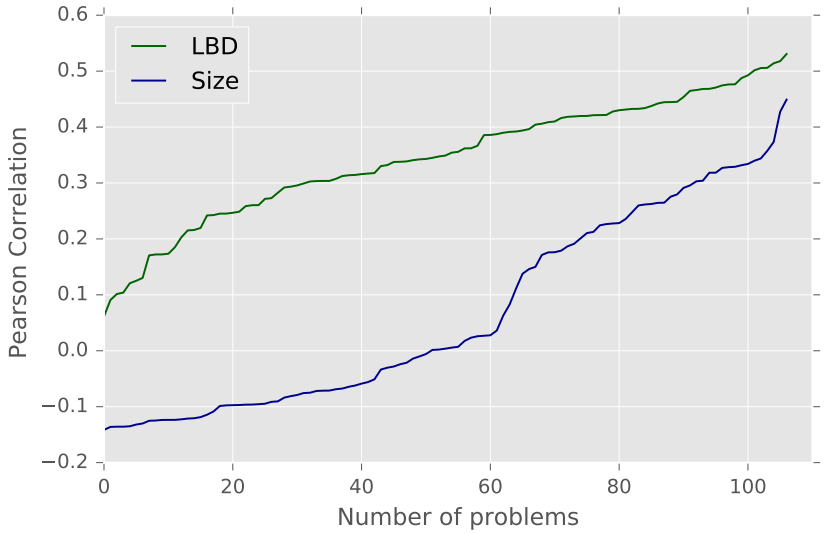


FIGURE 4.4 – Fonction de répartition du coefficient de corrélation de Pearson entre le $(max - min)$ des profondeurs des variables et le LBD des clauses. Pour comparer, nous avons représenté cette fonction en remplaçant le LBD par la taille des clauses.

voyons immédiatement sur la figure 4.3 que les points ne sont pas répartis autour d'une ligne. Par conséquent, nous ne nous attendions pas à obtenir des scores supérieurs à 0.5, mais nous souhaitions juste être en mesure de pouvoir observer si une tendance générale se dessinait. Nous avons alors représenté sur la figure 4.4 la fonction de répartition de r (en vert), ainsi que la fonction de répartition du coefficient de Pearson en utilisant la taille des clauses à la place du LBD (en bleu). Cette figure nous indique que dans la plupart des cas, le coefficient de Pearson de r est suffisamment élevé pour montrer une corrélation entre le LBD et la profondeur. Cependant, cette corrélation pourrait être un simple artefact dû à la longueur des clauses apprises. Autrement dit, une clause de plus grande taille possède généralement un plus grand nombre de niveau de décisions, et donc potentiellement, un plus grand nombre de variables avec différentes profondeurs, ce qui pourrait avoir comme effet d'augmenter la mesure $(max - min)$. Néanmoins, sur la figure 4.4, la fonction de répartition du coefficient de Pearson montre une corrélation plus faible en utilisant la taille des clauses.

Ces résultats suggèrent que le LBD est une mesure qui semble mieux capturer la structure haut niveau du problème qu'une simple mesure comme la taille des clauses, permettant ainsi potentiellement au solveur de l'exploiter.

4.2.3 Relation entre décisions, résolutions et clauses apprises

Nous avons ensuite regardé la valeur $(max - min)$ des profondeurs pour l'ensemble des variables de décisions (dec), des variables de résolutions (res) et pour les variables des clauses apprises ($learnt$). En se basant sur les observations précédentes, nous pouvons présumer que dec possède la plus petite valeur $(max - min)$, tandis que res possède la plus grande, c'est-à-dire :

$$(max - min)_{dec} \leq (max - min)_{learnt} \leq (max - min)_{res}$$

Comme précédemment, nous avons un très grand nombre de données à analyser (pour chaque conflit). Nous avons donc utilisé la même stratégie que dans la partie précédente. À savoir, nous avons regroupé toutes les données tous les 10,000 conflits. La figure 4.5 représente le pourcentage d'échantillons pour lesquels la relation ci-dessus est vérifiée. Nous observons que cet ordre est bien maintenu dans une très grande majorité des cas analysés. Les variables de décisions concernent un plus petit ensemble de profondeurs que les variables des clauses apprises. Ces dernières s'étendent elles aussi sur moins de profondeur que les variables des résolutions. La fonction de répartition montre clairement que cette hypothèse est vérifiée pour presque tous les échantillons observés. Ce résultat est aussi surprenant parce que les variables de dec sont choisies avec VSIDS, qui augmentent l'activité des variables de res et $learnt$.

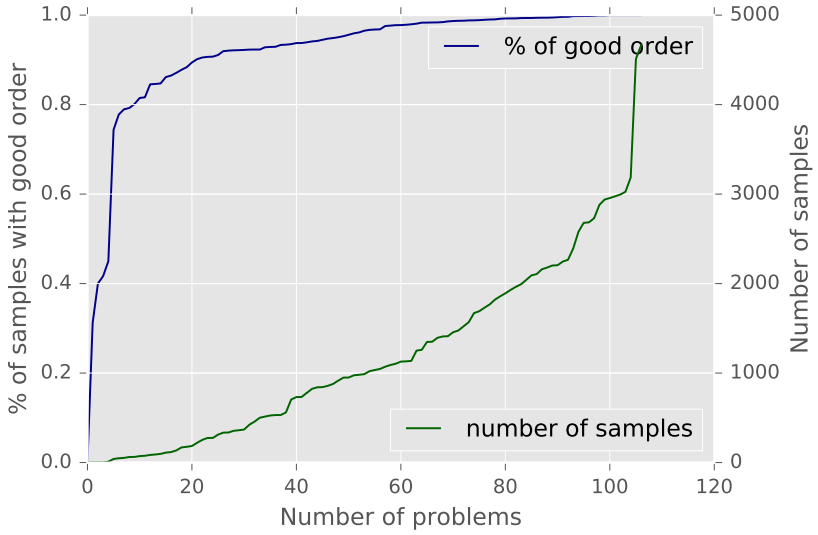


FIGURE 4.5 – Fonction de répartition du pourcentage des échantillons qui vérifient : $(max - min)_{dec} \leq (max - min)_{learnt} \leq (max - min)_{res}$ (axe de gauche). Nous avons également reporté la fonction de répartition du nombre d'échantillons par problème (axe de droite).

Ce phénomène est potentiellement dû aux relations entre variables et à la propagation unitaire, qui doit avoir tendance à impliquer des variables de profondeurs proches les unes des autres lorsque le solveur descend dans l'arbre de recherche. Cependant, nous n'avons pas été capable de trouver une connexion entre ce phénomène et le problème haut niveau (BMC).

4.2.4 Relation entre la profondeur des variables et l'élimination de variables

Le prétraitement utilisé dans `Minisat`, et par conséquent dans `Glucose`, est essentiellement composé du processus d'élimination de variables. Comme évoqué précédemment, VE est crucial pour beaucoup de problèmes SAT, et particulièrement pour les formules BMC encodées en utilisant la transformation de Tseitin. Nous avons voulu regarder où l'élimination de variable s'effectue principalement. Notre hypothèse de départ est que dans la plupart des cas les variables sont éliminées dans une seule profondeur (comme ce que nous faisons dans l'élimination au niveau du model checker). Nous construisons alors l'expérience suivante : pour chaque variable éliminée, nous mesurons la profondeur maximale moins la profondeur minimale $(max - min)$ de toutes les variables de toutes les clauses produites par l'élimination. Ainsi, une valeur $(max - min)$ égale à 0 signifie que l'élimination s'est effectuée uniquement à

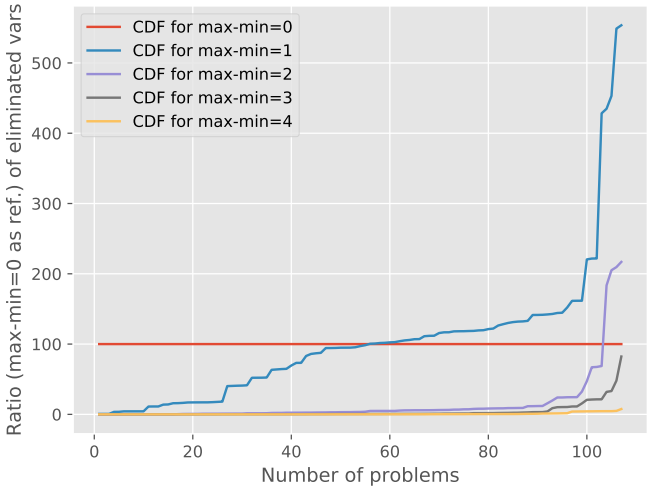


FIGURE 4.6 – Fonction de répartition du pourcentage de variables éliminées à la même profondeur, à une différence de profondeur 1, 2, 3 et 4. Le nombre de variables éliminées à la même profondeur est pris en référence pour la mise à l'échelle de tous les problèmes.

l'intérieur d'une seule profondeur. Les résultats sont reportés sur la figure 4.6. Nous observons que sur environ 50 problèmes (là où la ligne bleu, représentant $(max - min) = 1$, croise la ligne rouge représentant $(max - min) = 0$), il y a plus de variables éliminées à l'intérieur d'une seule profondeur que sur deux profondeurs contiguës. Nous pouvons également voir que sur presque tous les problèmes, il n'y a que très peu d'éliminations de variables qui génèrent des clauses avec une grande valeur $(max - min)$. Néanmoins, il apparaît que pour un petit nombre de problèmes, l'élimination de variables génère des clauses qui contiennent des variables de plusieurs profondeurs ($(max - min) = 2$ croise la ligne rouge).

4.3 Tentatives d'amélioration du solveur

L'idée de cette étude étant de spécialiser le solveur SAT afin d'améliorer ses performances sur les problèmes de type BMC, nous avons tenté de proposer quelques modifications du solveur qui utilisent la profondeur de dépliage des variables. Néanmoins, l'ensemble de nos propositions ne fournissent pas de résultats suffisamment encourageant, *i.e.*, elles n'améliorent pas ou même dégradent les performances du solveur.

4.3.1 Branchement sur les variables structurellement équivalentes

Dans un premier temps, nous avons essayé de forcer le solveur à choisir des variables structurellement équivalentes à différentes profondeurs. Pour se faire, après chaque redémarrage et jusqu'au premier conflit rencontré, nous avons modifié l'heuristique de sélection des variables par les différentes stratégies suivantes : à chaque fois qu'une variable est sélectionnée, nous prenons la même variable (1) une profondeur après, (2) à la dernière profondeur, (3) une profondeur avant, (4) à la première profondeur. L'idée est que une variable ayant un grand score avant le redémarrage est une variable beaucoup vu lors des derniers conflits. Il est ainsi possible de pousser le solveur à privilégier cette même variable mais plus profonde (ou moins profonde) pour forcer le solveur à travailler sur des conflits similaires mais plus ou moins profonds dans le dépliement. Nous avons alors observé une dégradation des performances pour l'ensemble de ces expériences à l'exception de la modification (2) qui reste compétitive avec la version non modifiée de Glucose (même nombre de problèmes résolus). Ce résultat peut paraître plutôt négatif mais nous aimerions mettre en avant ici le fait qu'il s'agit de la première fois, à notre connaissance, qu'une heuristique exploitant des informations haut niveau ne dégrade pas l'heuristique VSIDS. Elle montre donc une possible voie d'amélioration de l'heuristique de branchement, ou met en avant une conclusion possible : comme le fait de forcer le solveur à chercher des conflits le plus profondément possible ne dégrade pas ses performances, on peut émettre l'hypothèse que c'est en fait une caractéristique native de VSIDS. Elle pourrait nativement privilégier les conflits les plus profonds.

4.3.2 Utilisation du score max-min des profondeurs de dépliage

En raison de l'observation de la relation entre le LBD et le $(max - min)$ des profondeurs des variables, nous avons voulu essayer de remplacer le score LBD des clauses par le score $(max - min)$ de l'ensemble des variables de la clause. Nous avons néanmoins observé une petite dégradation des performances. À noter que dans `Glucose`, le score LBD est non seulement important pour évaluer les clauses, mais aussi pour déclencher les redémarrages. Cela suggère donc qu'il faudrait peut-être également modifier l'heuristique de redémarrage pour obtenir des performances au moins comparable.

4.3.3 Modification de l'ordre pour l'élimination de variables

Dans une dernière expérimentation, nous avons tenté de changer l'ordre d'élimination des variables (dans la phase de prétraitement du solveur), en éliminant d'abord les variables n'appartenant qu'à une seule profondeur, puis les variables des profondeurs successives, ensuite les variables appartenant à 3 profondeurs, etc. Pour détecter le nombre de profondeurs dans lesquelles une variable est présente, il suffit de regarder le $(max - min)$ des clauses résultantes de l'élimination de cette variable. Comme attendu, en raison de la structure des formules BMC et de l'encodage que nous avons choisi [1], nous avons observé que la grande majorité des variables sont locales à une seule profondeur, *i.e.*, presque toutes les éliminations de variables se font lors du premier passage. D'un point de vue performance, cette version dégrade malheureusement légèrement, encore une fois, le nombre d'instances résolues par rapport à la version originale.

1. Les variables référencées par des mémoires sont les seules à pouvoir être présentes à différentes profondeurs.

Chapitre 5

Outil de vérification de modèles

Cette partie donne une vue synthétique et compacte de l'outil que nous avons développé au cours de la thèse. Nous pensons que la production d'un outil open-source, permettant de comparer les différentes approches état de l'art et codant nos techniques de duplication de clauses est une contribution importante du travail présenté. On notera également que cette partie est relativement indépendante pour le lecteur spécialiste du domaine. Il pourra y trouver la description du travail d'implémentation effectué ainsi que les liens avec le code source associé au manuscrit. Nous mettons en particulier l'accent sur les techniques introduites dans la partie précédente qui peuvent nécessiter une implémentation particulière afin d'être efficace. Nous indiquons aussi deux tentatives d'amélioration : (i) un redémarrage à haut niveau, et (ii) une réinitialisation des clauses. Néanmoins, malgré nos efforts et le bien fondé de l'idée initiale, ces deux méthodes ne sont malheureusement pas efficaces en l'état. Nous pensons qu'elles nécessitent encore du travail pour le devenir.

5.1 Modèle en entrée

Les outils que nous avons développés acceptent en entrée des modèles AIGs, au format AIGER (Biere *et al.* [2011]). Nous avons donc utilisé la bibliothèque associée à AIGER, qui fournit toutes les fonctions nécessaires pour la lecture et la gestion des AIGs. Sauf cas particulier, les outils que nous avons développés ont certaines restrictions :

— Seules les propriétés de sûreté sont acceptées (pas de propriétés de vivacité, ni d'équité [1]).

— Une seule propriété par modèle est acceptée, *i.e.*, un modèle ne peut pas contenir plusieurs propriétés à vérifier en même temps. Néanmoins, il suffit de créer, au préalable, une définition correspondant à la conjonction des propriétés si nécessaire.

1. Un comportement du système se répétera une infinité de fois.

— Les contraintes permettent en général de représenter l'environnement physique du système. Par exemple, une aiguille (ferroviaire) ne peut pas physiquement être à gauche et à droite en même temps. Ainsi, on ajoutera au modèle une contrainte spécifiant que ces deux entrées ne peuvent pas être vraies en même temps. Ces contraintes ne sont pas acceptés dans l'outil actuel. De la même façon que les propriétés multiples, il suffit au préalable de créer une définition où les contraintes correspondent aux prémisses et la propriété à la conclusion d'une implication.

— Toutes les variables mémoires sont initialisées à $\bot$.

5.2 Prétraitement

Comme abordé précédemment, l'étape de prétraitement consiste à optimiser le modèle donné en entrée pour la vérification, c'est-à-dire avant de le soumettre aux algorithmes de model checking. Dans une première approche, nous avions choisi d'utiliser le processus de prétraitement disponible avec l'outil `ABC` (Brayton et Mishchenko [2010]) qui est très efficace et facile à utiliser. Néanmoins, pour avoir plus de contrôle sur ce qui est fait lors de cette étape de prétraitement et pour ne pas être dépendant d'un outil externe (*i.e.*, ne pas avoir à importer une bibliothèque d'une taille imposante ($\approx$ 120Mo)), nous avons décidé de développer notre propre module de prétraitement. Cela nous permet également d'avoir un outil complet open source. Nous avons néanmoins choisi d'implémenter un prétraitement relativement simple [2] tout en essayant de limiter les ressources (temps de calculs et mémoire) qui sont utilisées lors de ce processus, afin de pouvoir traiter des modèles de *grandes* tailles avec *peu* de ressources.

La première étape du prétraitement consiste à calculer le cône d'influence (COI), ce qui permet notamment d'éviter d'effectuer des optimisations sur les parties du modèle qui n'influencent aucunement la propriété. Le COI est un simple calcul de dépendance syntaxique qui, en partant de la propriété, marque chaque définition, chaque entrée du modèle et chaque variable mémoire qui sont accessibles depuis cette propriété. Ainsi, chaque élément du modèle qui n'est pas marqué après ce calcul peut être retiré du modèle. Cependant, pour la construction d'un chemin d'exécution lorsqu'il existe un CEX, il est important de se souvenir des entrées qui ont été éliminées si l'on souhaite simuler ce CEX sur le modèle original.

Nous décrivons dans la suite de cette partie les optimisations que nous effectuons sur l'ensemble des définitions, puis sur l'ensemble des mémoires.

2. La durée de la thèse étant limitée nous ne voulions pas passer un temps trop important à implémenter ce module déjà existant dans les outils développés par `SafeRiver`.

5.2.1 Optimisation des définitions

Les optimisations sont effectuées lors de la première étape du model checker. Cela signifie que le modèle est encore sous forme AIG (*i.e.*, l'élimination haut niveau n'a pas encore été effectuée). Par conséquent, le modèle peut être vu comme une liste de définitions contenant l'opérateur $\wedge$ avec 2 opérandes (*e.g.*, $x = a \wedge b$), et chaque variable ne possédant pas de définition (combinatoire) est soit une entrée, soit une variable mémoire. Lors de cette étape d'optimisation, une définition peut se simplifier et devenir une constante ou un « alias ». Dans cette situation, elle est propagée puis supprimée de la liste des définitions. Les simplifications sont appliquées sur la liste des définitions jusqu'à ce qu'un point fixe soit atteint. Autrement dit, les règles d'optimisation sont appliquées itérativement sur l'ensemble des définitions dans un ordre donné, jusqu'à ce que plus aucune règle ne s'applique. La liste est ordonnée par dépendance, c'est-à-dire que si une définition référence la variable de la partie gauche d'une définition alors la définition de la variable est inférieure dans l'ordre d'application des optimisation, *e.g.* :

$$\begin{aligned} &x = a \wedge b \\ &y = x \wedge c \end{aligned} \qquad \rightarrow \qquad x < y$$

Il s'agit seulement d'un ordre partiel, certaines définitions n'ont pas de relations entre elles (même transitivement), par exemple :

$$\begin{aligned} &x = a \wedge b \\ &y = c \wedge d \\ &z = x \wedge y \end{aligned} \qquad \rightarrow \qquad x < z \quad \text{et} \quad y < z$$

Ici, x et y n'ont pas de relations, ces deux définitions sont interchangeables sans distinction (en respectant leurs relations respectives) dans l'ordre des définitions. L'absence de dépendance cyclique nous assure néanmoins que l'ordre est strict pour les définitions dépendantes. Autrement dit, on ne peut pas avoir la situation suivante :

$$\begin{aligned} x = a \wedge y \quad &\rightarrow \quad y < x \\ y = b \wedge x \quad &\rightarrow \quad x < y \end{aligned}$$

À noter qu'il ne s'agit pas d'une relation d'ordre, cette relation n'est ni réflexive ni antisymétrique.

Règle de réécriture et simplification

Voici la liste des règles de simplifications que nous effectuons sur les définitions :

— La première étape d'optimisation est la propagation de constantes ou d'alias. Lorsqu'une définition se réduit à une constante ($\top$ ou $\bot$) ou à une autre variable (*e.g.*, $x = \top$ ou $x = \neg y$), alors toutes ses occurrences dans les autres définitions sont remplacées par sa partie droite. Par exemple :

$$\begin{aligned} x &= \top \\ y &= \neg b \\ z &= x \wedge \neg y \end{aligned} \qquad \rightarrow \qquad z = \top \wedge b$$

— La seconde partie des optimisations consiste à appliquer un ensemble de règles locales. À noter que la plupart des règles indiquées ici proviennent ou sont inspirées de Brummayer et Biere [2006]. Pour une définition donnée, le terme « local » signifie que les informations nécessaires à l'application d'une règle sont limitées soit dans la définition elle-même, soit dans les définitions de ses opérandes. Voici la liste de l'ensemble des règles que nous appliquons, où nous donnons qu'un seul exemple par règle :

- Constante (1.1) :

$$\begin{aligned} x &= a \wedge \bot \quad &\rightarrow \quad x &= \bot \\ x &= a \wedge \top \quad &\rightarrow \quad x &= a \end{aligned}$$

- Contradiction (1.1) :

$$x = a \wedge \neg a \qquad \rightarrow \qquad x = \bot$$

- Contradiction (1.2) :

$$\begin{aligned} x &= a \wedge b \\ y &= x \wedge \neg a \end{aligned} \qquad \rightarrow \qquad y = \bot$$

- Contradiction (1.3) :

$$\begin{aligned} x &= a \wedge b \\ y &= c \wedge \neg a \\ z &= x \wedge y \end{aligned} \qquad \rightarrow \qquad z = \bot$$

- Idempotence (1.1) :

$$x = a \wedge a \qquad \rightarrow \qquad x = a$$

- Idempotence (1.2) :

$$\begin{aligned} x &= a \wedge b \\ y &= x \wedge a \end{aligned} \qquad \rightarrow \qquad y = x$$

- Idempotence (1.3) :

$$\begin{aligned} x &= a \wedge b \\ y &= a \wedge b \\ z &= x \wedge y \end{aligned}$$

  Cette situation est couverte par l'équivalence structurelle qui propagera la définition $y = x$, ainsi la règle idempotence (1.1) sera appliquée.

- Subsomption (1.1) :

$$\begin{aligned} x &= \neg a \wedge b \\ y &= \neg x \wedge a \end{aligned} \qquad \rightarrow \qquad y = a$$

- Subsomption (1.2) :

$$\begin{aligned} x &= \neg a \wedge b \\ y &= a \wedge c \\ z &= \neg x \wedge y \end{aligned} \qquad \rightarrow \qquad z = y$$

- Résolution (1.1) :

$$\begin{aligned} x &= a \wedge b \\ y &= a \wedge \neg b \\ z &= \neg x \wedge \neg y \end{aligned} \qquad \rightarrow \qquad z = \neg a$$

- Substitution (1.1) :

$$\begin{aligned} x &= a \wedge b \\ y &= a \wedge \neg x \end{aligned} \qquad \rightarrow \qquad y = a \wedge \neg b$$

- Substitution (1.2) :

$$\begin{aligned} x &= a \wedge b \\ y &= a \wedge c \\ z &= \neg x \wedge y \end{aligned} \qquad \rightarrow \qquad z = \neg b \wedge y$$

Pour rester concis, nous n'avons pas indiquées les permutations applicables à la plupart des règles. Par exemple, la règle « Substitution (1.1) » est également applicable dans le cas suivant :

$$\begin{aligned} x &= a \wedge b \\ y &= b \wedge \neg x \end{aligned} \qquad \rightarrow \qquad y = \neg a \wedge b$$

— L'étape d'optimisation suivante est l'équivalence structurelle. Cette étape consiste simplement à détecter (à l'aide d'une table de hachage) si deux définitions sont structurellement équivalentes. L'une des définitions devient alors un alias ce qui entraîne sa propagation. Par exemple :

$$\begin{array}{lll} x = a \wedge b & & \\ y = a \wedge b & \rightarrow & y = x \end{array}$$

Comme les règles sont appliquées dans l'ordre de dépendance des définitions, la définition qui sera éliminée est toujours celle qui se trouve le plus loin dans la liste des définitions.

— Ensuite, nous construisons pour chaque définition deux listes l_a et l_s correspondant, respectivement, à la conjonction des opérandes de cette définition avec et sans variables intermédiaires. Autrement dit, en effectuant un calcul de dépendances à partir d'une définition, nous remontons à l'aide des opérandes jusqu'aux variables apparaissant négativement ou n'étant pas une définition. Par exemple, voici les listes obtenues pour les définitions suivantes :

$$\begin{array}{llrll} x = a \wedge b & \rightarrow & x_{l_s} = \{a, b\} & \text{et} & x_{l_a} = \{a, b\} \\ y = x \wedge c & \rightarrow & y_{l_s} = \{a, b, c\} & \text{et} & y_{l_a} = \{a, b, x, c\} \\ z = x \wedge \neg y & \rightarrow & z_{l_s} = \{a, b, \neg y\} & \text{et} & z_{l_a} = \{a, b, x, \neg y\} \end{array}$$

Nous appliquons alors un ensemble de règles basées sur les mêmes principes que celles décrites précédemment. Par exemple :

- Contradiction (2.1) :

$$z = x \wedge y \qquad \text{avec } z_{l_a} = \{\ldots, x, \ldots, a, \neg a, \ldots y\} \quad \rightarrow \quad z = \bot$$

Avec l'utilisation du bit de poids faible comme marqueur de signe [3] et en ordonnant la liste l_a, il suffit de comparer les éléments contigus pour détecter les contradictions (un seul parcours de la liste en ordonnant les éléments à l'insertion).

- Subsomption (2.1) :

$$z = a \wedge \neg x \qquad \text{avec } x_{l_a} = \{\ldots, \neg a, \ldots\} \qquad \rightarrow \qquad z = a$$

- Substitution (2.1) :

$$\begin{array}{lll} y = \neg x \wedge b & \text{avec } x_{l_a} = \{\ldots, \neg a, \ldots\} & \\ z = a \wedge y & & \rightarrow \quad z = a \wedge b \end{array}$$

3. Les variables dans la polarité positive sont représentées par des indices pairs et un indice impair indique une polarité négative.

La liste l_s permet quant à elle de détecter l'équivalence fonctionnelle entre les définitions. Autrement dit, si deux définitions possèdent la même liste l_s alors l'une peut être substituée par l'autre, comme sur l'exemple suivant :

$$\begin{array}{lll} u = a \wedge b & \text{avec } u_{l_s} = \{a, b\} & \\ v = u \wedge c & \text{avec } v_{l_s} = \{a, b, c\} & \\ w = v \wedge d & \text{avec } w_{l_s} = \{a, b, c, d\} & \\ x = a \wedge d & \text{avec } x_{l_s} = \{a, d\} & \\ y = b \wedge c & \text{avec } y_{l_s} = \{b, c\} & \\ z = x \wedge y & \text{avec } z_{l_s} = \{a, b, c, d\} & \rightarrow \quad z = w \end{array}$$

À noter que la liste l_s permet de détecter uniquement l'équivalence fonctionnelle entre les listes de conjonctions. Ainsi, seulement une sous partie de toutes les équivalences fonctionnelles sont détectées. Dans l'exemple suivant, v et z sont fonctionnellement équivalents avec des listes l_s différentes :

$$\begin{array}{lll} t = a \wedge b & \text{avec } t_{l_s} = \{a, b\} & \\ u = \neg t \wedge c & \text{avec } u_{l_s} = \{\neg u, c\} & \Leftrightarrow \quad u = (\neg a \vee \neg b) \wedge c \\ v = \neg u \wedge d & \text{avec } v_{l_s} = \{\neg u, d\} & \Leftrightarrow \quad v = ((a \wedge b) \vee \neg c) \wedge d \\ w = \neg a \wedge c & \text{avec } w_{l_s} = \{\neg a, c\} & \\ x = \neg b \wedge c & \text{avec } x_{l_s} = \{\neg b, c\} & \\ y = \neg w \wedge \neg x & \text{avec } y_{l_s} = \{\neg w, \neg x\} & \Leftrightarrow \quad y = (a \vee \neg c) \wedge (b \vee \neg c) \\ z = y \wedge d & \text{avec } z_{l_s} = \{\neg w, \neg x, d\} & \Leftrightarrow \quad y = (a \vee \neg c) \wedge (b \vee \neg c) \wedge d \end{array}$$

Équilibrage

L'équilibrage est une méthode de réécriture qui est habituellement utilisé pour l'optimisation des délais dans les circuits (Cortadella [2003]). Mishchenko *et al.* [2006] ont proposé de l'utiliser pour l'optimisation des AIGs. Cela permet de modifier la structure de l'AIG tout en restant fonctionnellement équivalent, et ainsi de potentiellement détecter plus d'équivalences structurelles. Nous n'avons pas implémenté la même technique que celle utilisée dans `ABC`, mais une heuristique plus simple basée sur la détection de configurations.

La première étape consiste à calculer le niveau de chaque nœud. Le niveau des feuilles (des entrées) et des variables mémoires est 0. Chaque nœud interne a un niveau égal au maximum des niveaux de ses opérandes plus 1. L'exemple 5.1 ci-dessous illustre le niveau de chaque nœud d'un AIG (indiqué entre parenthèse).

La seconde étape consiste à appliquer des règles de réécriture locales. Pour cela, nous utilisons la fonction $niv(x)$ qui, pour un nœud x retourne son niveau

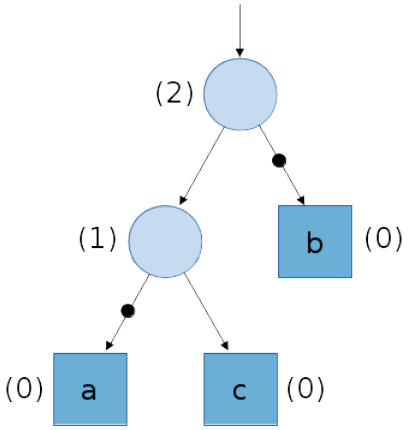
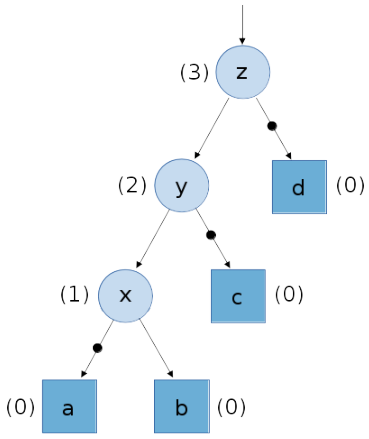
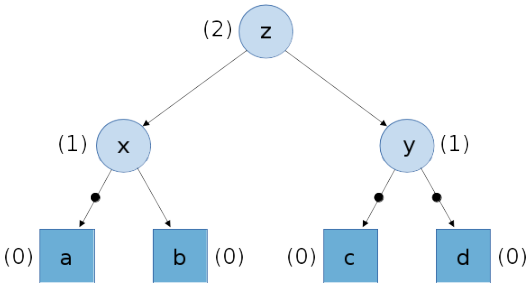


FIGURE 5.1 – Exemple d'un AIG avec le niveau de chaque nœud



(a) AIG avant équilibrage



(b) AIG après équilibrage

FIGURE 5.2 – Équilibrage

dans l'AIG. Puis, nous appliquons la règle de réécriture suivante :

	Si y n'est référencée que par z		
$y = x \wedge c$	avec $niv(c) < niv(x)$	$\rightarrow$	$y = c \wedge d$
$z = y \wedge d$	et $niv(d) < niv(x)$	$\rightarrow$	$z = x \wedge y$

Notons que les permutations applicables à la règle sont omises pour rester concis. L'application de cette règle d'équilibrage est illustrée sur la figure 5.2, dans laquelle nous avons utilisé les mêmes variables que ci-dessus.

5.2.2 Optimisation des mémoires

Les seules optimisations que nous effectuons pour les variables mémoires sont la détection et la propagation des constantes, ainsi que l'équivalence struc-

turelle. Comme toutes les mémoires sont initialisées à $\bot$, il n'y a que 2 cas à vérifier pour détecter les constantes :

$$\begin{array}{lll} x = \bot,\ \bot & \rightarrow & x = \bot \\ x = \bot,\ x & \rightarrow & x = \bot \end{array}$$

Pour la même raison (initialisation à $\bot$), la détection des équivalences structurelles se fait en comparant l'opérande « suivant ». Par exemple :

$$\begin{array}{lll} x = \bot,\ a & & \\ y = \bot,\ a & \rightarrow & y = x \end{array}$$

Simulation ternaire

La simulation ternaire, introduit par Bryant et Seger [1990], est une méthode qui peut être utilisée lors du prétraitement du modèle pour détecter les variables mémoires qui sont constantes. Cela est réalisé en simulant le modèle avec 3 valeurs : 0,1 et « X ». La valeur « X » permet de donner une valeur aux variables d'entrées et à toutes les variables pour lesquelles il n'est pas possible d'en déterminer la valeur. Le modèle est alors simulé, jusqu'à atteindre un point fixe, en ajoutant les règles suivantes à la sémantique binaire habituelle ($0 \wedge 0 = 0$, $0 \wedge 1 = 0$, etc) :

$$\begin{aligned} X \wedge 0 &= 0 \\ X \wedge 1 &= X \\ X \wedge X &= X \\ \neg X &= X \end{aligned}$$

Ainsi, toutes les variables mémoires qui gardent la même valeur ($\neq X$) pour tous les états de la simulation jusqu'à ce que le point fixe soit atteint, peuvent être remplacées par cette valeur. L'exemple 5.1 ci-dessous illustre ce processus.

Exemple 5.1. Soit a et b des entrées, m et n des mémoires, et x, y, z des définitions.

Modèle	Simulation init	Simulation 1	Simulation 2 (Point fixe)
a;	$a = X$	$a = X$	$a = X$
b;	$b = X$	$b = X$	$b = X$
$m = 0,\ y$	$m = 0$	$m = 0$	$m = 0$
$n = 0,\ z$	$n = 0$	$n = X$	$n = X$
$x = a \wedge m$	$x = (X \wedge 0) = 0$	$x = 0$	$x = 0$
$y = x \wedge b$	$y = (0 \wedge X) = 0$	$y = 0$	$y = 0$
$z = a \wedge \neg n$	$z = (X \wedge 1) = X$	$z = X$	$z = X$

La variable mémoire m est alors détectée comme étant une constante ($= 0$). Ainsi, en la propageant, les définitions x et y seront également réduites à des constantes.

5.2.3 Discussion

Comme évoqué précédemment, ce module de prétraitement des modèles AIGs est assez basique. Un grand nombre de règles d'optimisation pourrait encore être ajouté pour minimiser le plus possible la représentation (*e.g.*, notamment les méthodes de *sweeping*). Il serait également intéressant d'effectuer une comparaison empirique entre ce module et le prétraitement effectué par `ABC`. En particulier, cela permettrait d'observer le nombre de définitions éliminées et d'identifier l'impact de ces deux prétraitement lors de la vérification, et ainsi de proposer des améliorations pour ce module.

5.3 Choix de l'algorithme

Après la phase de prétraitement, le modèle simplifié est le point d'entrée pour les algorithmes de Model Checking. Les étapes suivantes dépendront du choix de l'algorithme et de ses options. Par exemple, les étapes d'élimination et de dépliage seront différentes si l'on souhaite utiliser BMC avec élimination à bas niveau ou si la vérification est effectuée avec PDR. C'est également lors de cette étape que la gestion d'une stratégie de parallélisation est le plus facile à mettre en place. Par exemple, une technique simple pourrait consister à associer une stratégie par *thread*, *e.g.*, un *thread* pour l'induction temporelle, un *thread* pour l'interpolation et un *thread* pour PDR. Ainsi, dès que l'un des algorithmes trouve la solution alors les autres sont arrêtés et le résultat est retourné.

5.4 Élimination des variables à haut-niveau

Rappelons brièvement la méthode d'élimination de variables à haut-niveau, décrite dans la partie 3.3, que nous avons implémentée. La première mesure que nous avons prise est d'étendre la structure des définitions. En effet, habituellement, une définition représente toujours un « et » logique entre deux opérandes. Or, pour effectuer l'élimination à haut niveau, il faut être capable de gérer des définitions pouvant contenir de multiples littéraux et également enrichir les définitions en ajoutant la gestion de l'opérateur « ou » logique (car la négation est propagée aux littéraux). Pour cela, nous avons utilisé une structure (récursive) d'arbre binaire où les nœuds internes représentent les opérateurs ($\wedge$ ou $\vee$) et les feuilles les littéraux. En outre, pour simplifier les étapes suivantes, nous limitons la négation aux littéraux (*i.e.*, aux feuilles)

dans cette structure. Ainsi, lors de l'élimination d'une définition les négations sont propagées aux feuilles.

Ensuite, en utilisant cette structure nous avons pu définir une méthode qui permet d'éliminer la variable d'une définition, c'est-à-dire substituer, dans toutes les définitions référençant cette variable, toutes les occurrences de cette variable par sa définition (*i.e.*, par sa partie droite, son *rhs*). L'idée étant de calculer le nombre de clauses générées par la définition courante et toutes les définitions qui la référencent, puis de calculer à nouveau le nombre de clauses générées lorsque la variable de la définition est virtuellement éliminée. Ainsi, si les clauses avant et après élimination répondent aux critères d'élimination que nous avons spécifiés, la définition est éliminée. Nous avons utilisé les mêmes critères d'élimination que dans `Glucose` (ou `Minisat`), à savoir, une variable (d'une définition) est éliminée si le nombre de clauses générée après l'élimination est inférieur ou égal au nombre de clauses avant élimination, et également si la taille des clauses générées après élimination ne dépasse pas 20 littéraux. Nous avons proposé deux variantes à cette méthode d'élimination : une qui est précise, c'est-à-dire qui construit explicitement les ensembles de clauses qui seront générées si la variable est éliminée et qui est par conséquent plus coûteuse, et une autre moins précise qui compte simplement le nombre de clauses générées. Dans cette deuxième version, les clauses générées par l'élimination et qui sont des tautologies (*e.g.*, $a \vee \neg a$) ne sont pas détectées. Elles sont donc comptabilisées à tort lors de l'évaluation des critères d'élimination.

5.5 Dépliage de la relation de transition

Après l'étape d'élimination, nous obtenons la représentation que nous allons pouvoir déplier pour les algorithmes qui déroulent la relation de transition. Le dépliage s'effectue de la même façon que ce soit avec des définitions constituées uniquement de 2 opérandes ou après élimination avec des définitions plus complexes, sous forme d'arbres. Il y a cependant deux différences dans notre implémentation avec les approches « habituelles » : (i) Nous ne générons pas les clauses à la volée. Nous conservons les définitions dépliées dans une structure. (ii) Nous effectuons une étape de propagation de constantes et d'alias lors du dépliage (*e.g.*, pour BMC comme toutes les mémoires sont initialisées à $\bot$, de nombreuses définitions sont éliminées par la propagation lors du dépliage). L'exemple 5.2 ci-dessous illustre ce processus de dépliage pour BMC.

Exemple 5.2. Soit a, b, c, d, e des entrées, m une mémoire, et x, y, z des

définitions.

$$
\begin{array}{llll}
m = 0,\ z & m_0 = 0 & m_1 = z_0 & \dots \\
x = (a \wedge b) \vee \neg m & x_0 = (a_0 \wedge b_0) \vee \neg m_0 & x_1 = (a_1 \wedge b_1) \vee \neg m_1 & \dots \\
y = m \vee x \vee \neg c & y_0 = m_0 \vee x_0 \vee \neg c_0 & y_1 = m_1 \vee x_1 \vee \neg c_1 & \dots \\
z = e \wedge (d \vee \neg y) & z_0 = e_0 \wedge (d_0 \vee \neg y_0) & z_1 = e_1 \wedge (d_1 \vee \neg y_1) & \dots \\
\underbrace{\qquad\qquad}_{\text{Modèle}} & \underbrace{\qquad\qquad}_{\text{Initialisation}} & \underbrace{\qquad\qquad}_{\text{Cycle 1}} & \\
 & \downarrow & \downarrow & \\
 & x_0 = 1 & x_1 = (a_1 \wedge b_1) \vee \neg z_0 & \\
 & y_0 = 1 & y_1 = z_0 \vee x_1 \vee \neg c_1 & \\
 & z_0 = e_0 \wedge d_0 & z_1 = e_1 \wedge (d_1 \vee \neg y_1) &
\end{array}
$$

La partie « Modèle » correspond au modèle après élimination de variables. « Initialisation » correspond au premier dépliage, *i.e.*, toutes les variables mémoires sont initialisé à $\bot$. La partie « Cycle 1 » correspond au second dépliage, à savoir, les variables mémoires sont substituées par les variables correspondant à la valeur suivante de ces mémoires au cycle précédent. La partie inférieure correspond à l'ensemble des définitions après propagation, à savoir, la représentation qui est utilisée après le dépliage.

5.6 Mise à plat

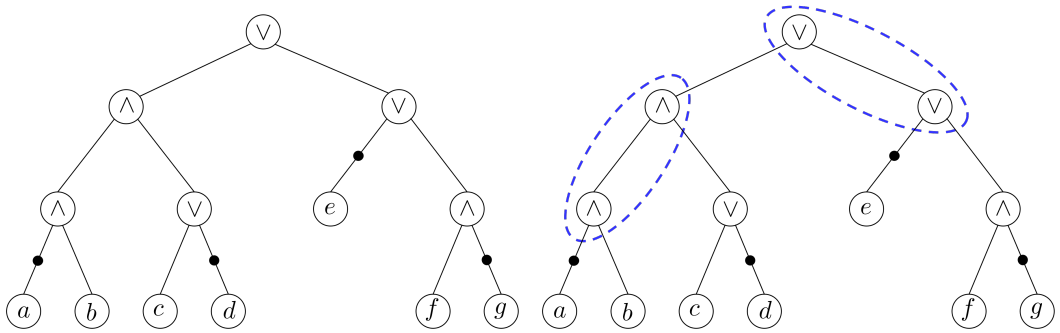
Les définitions dépliées sont ensuite mises à plat pour faciliter l'étape de simplification. La structure d'arbre est étendue, la structure binaire (2 fils par nœuds internes) est rompu, chaque nœud peut alors posséder plus de 2 fils. Ainsi, il est possible de fusionner les nœuds internes voisins représentant le même opérateur, comme illustré sur la figure 5.3 de l'exemple 5.3 ci-dessous.

Exemple 5.3.

$$x = (\neg a \wedge b \wedge (c \vee \neg d)) \vee \neg e \vee (f \wedge \neg g)$$

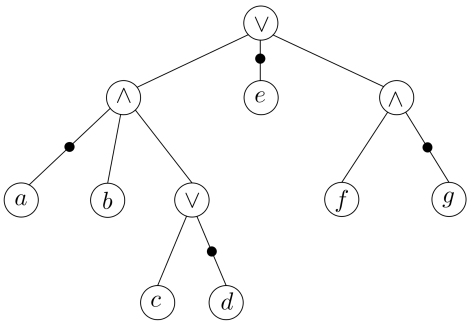
La figure 5.3a illustre l'arbre de la définition ci-dessus, avant la mise à plat. La figure 5.3b met en avant les nœuds qui sont fusionnés lors de la mise à plat. La figure 5.3c représente l'arbre de la définition après la mise à plat.

Afin d'améliorer l'étape de simplification, notamment l'efficacité de l'équivalence structurelle, lors de la mise à plat, les fils de chaque nœuds sont ordonnés du plus petit indice de variable au plus grand, puis de la plus petite sous-formule à la plus grande (*i.e.*, la taille d'une formule correspond à son nombre d'opérandes). Cela permet d'obtenir une forme normalisée (mais non canonique), et ainsi d'augmenter les chances de détecter les équivalences structurelles.



(a) Avant la mise à plat

(b) Nœuds à fusionner



(c) Après la mise à plat

FIGURE 5.3 – Mise à plat des définitions dépliées.

5.7 Simplification

Une fois les définitions mises à plat, nous effectuons une nouvelle étape de simplification sur les définitions dépliées. Comme lors de l'étape de prétraitement du modèle en entrée, les définitions équivalentes à des constantes ou à des alias sont propagées puis éliminées. Nous donnons ci-dessous la liste des règles que nous appliquons :

— Des règles « simples » sont appliquées en premier. Ce sont des règles similaires à celles appliquées au niveau du modèle d'entrée mais cette fois-ci sur les structures en arbre des définitions. Comme précédemment, nous n'indiquons pas toutes les permutations possibles applicables à ces règles :

$$
\begin{array}{lcll}
x = a \wedge \ldots \wedge \bot & \rightarrow & x = \bot & (1) \\
x = a \vee \ldots \vee \top & \rightarrow & x = \top & (2) \\
x = a_1 \wedge \ldots \wedge \top \wedge \ldots \wedge a_n & \rightarrow & x = a_1 \wedge \ldots \wedge a_n & (3) \\
x = a_1 \vee \ldots \vee \bot \vee \ldots \vee a_n & \rightarrow & x = a_1 \vee \ldots \vee a_n & (4) \\
x = a_1 \wedge \ldots \wedge a_i \wedge \neg a_i \wedge \ldots \wedge a_n & \rightarrow & x = \bot & (5) \\
x = a_1 \vee \ldots \vee a_i \vee \neg a_i \vee \ldots \vee a_n & \rightarrow & x = \top & (6) \\
x = (a \wedge b) \vee (a \wedge \neg b) & \rightarrow & x = a & (7) \\
x = (a \vee b) \wedge (a \vee \neg b) & \rightarrow & x = a & (8)
\end{array}
$$

Comme les éléments sont ordonnés, la détection de ces règles ne requiert qu'un simple parcours. Par exemple pour les règles (5) et (6), il suffit de regarder les éléments contigus. En outre, ces règles sont aussi appliquées récursivement dans les sous-arbres d'une définition. Par exemple :

$$x = a_1 \wedge \ldots \wedge (b \vee \ldots \vee \top) \wedge \ldots \wedge a_n \quad \rightarrow \quad x = a_1 \wedge \ldots \wedge a_n$$

— Nous avons ensuite implémenté une méthode d'optimisation que nous appelons « simplification par domination [4] ». Cette simplification revient à effectuer les substitutions et les subsomptions à l'intérieur des définitions. La figure 5.4 illustre cette simplification. Chaque littéral de même niveau qu'un nœud interne, et ayant le même nœud père, domine toutes les feuilles (littéraux) du sous-graphe ayant pour racine ce nœud interne (autrement dit, de niveau inférieur). Par exemple, dans la figure 5.4, le littéral a de plus haut niveau (fils de la racine) domine l'ensemble de tous les littéraux de l'arbre. Ensuite, en fonction de l'opérateur du nœud parent du littéral dominant, toutes les occurrences de ce littéral qui sont dominées sont alors substituées par une constante : $\top$ pour l'opérateur

4. Cette notion de domination n'a pas de rapport avec les ensembles dominants dans le domaine des graphes.

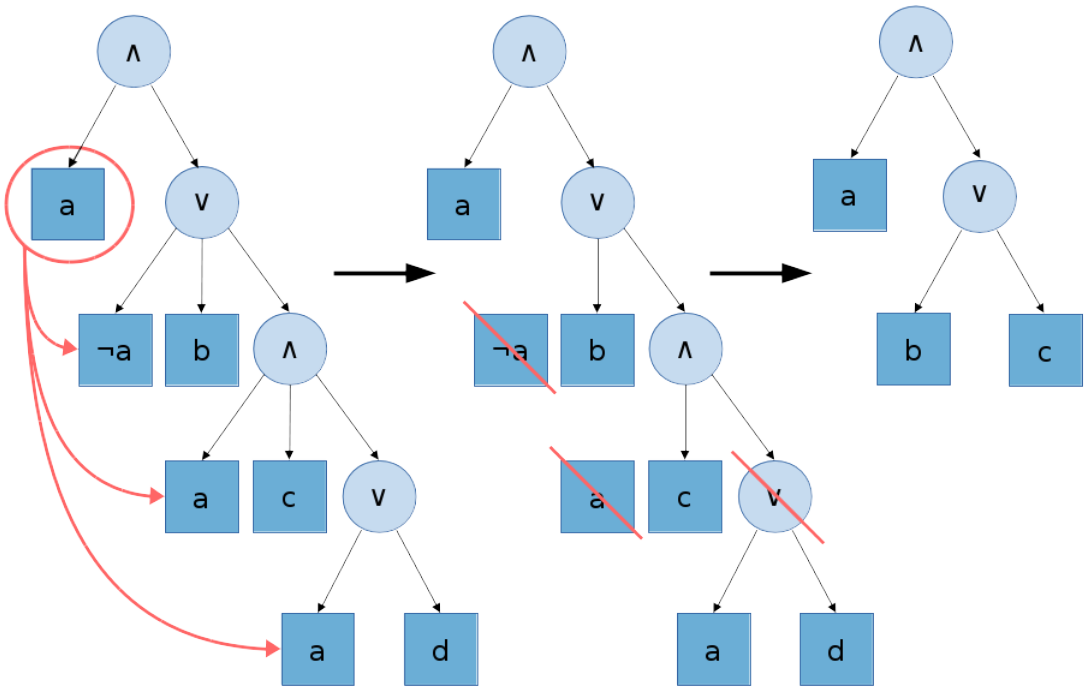


FIGURE 5.4 – Simplification par domination de $a \wedge (\neg a \vee b \vee (a \wedge c \wedge (a \vee d)))$

$\wedge$ et $\bot$ pour $\vee$. Dans l'exemple 5.4, l'opérateur du nœud parent du littéral dominant est $\wedge$ (*i.e.*, la racine). Les littéraux dominés sont alors substitués comme suit : $\neg a$ par $\bot$ (*i.e.*, $\neg\top$) et a par $\top$.

— Nous avons également implémenté la détection des équivalences et différences structurelle à deux niveaux : local et global. « Local » signifie que la détection des équivalences et des différences structurelles des définitions a lieu dans le même cycle (intra-cycle), et « global » signifie que la détection s'effectue entre différents cycles (inter-cycle). Par exemple :

L'indice des variables indique le cycle auquel celles-ci appartiennent.

$$x_1 = a_1 \wedge (b_1 \vee c_1)$$
$$y_1 = a_1 \wedge (b_1 \vee c_1) \quad \rightarrow \quad y_1 = x_1 \text{ (local)}$$
$$z_2 = a_1 \wedge (b_1 \vee c_1) \quad \rightarrow \quad z_2 = x_1 \text{ (global)}$$

La différence structurelle est simplement l'équivalence avec la négation d'une définition, *e.g.* :

$$x = a \wedge (b \vee c)$$
$$y = \neg a \vee (\neg b \wedge \neg c) \qquad \rightarrow \qquad y = \neg x$$

La détection des équivalences structurelles avec des définitions à 2 opérandes ou avec un arbre s'effectue de la même manière, *i.e.*, avec une table

de hachage. La détection des différences structurelles s'effectue avec un surcoût minime : il n'y a pas besoin de table additionnelle, le calcul de la négation de la valeur de hachage de la définition et la négation de la définition se font à la volée.

— Les simplifications ne sont habituellement effectuées que sur la liste des nouvelles définitions venant d'être introduites. En effet, une fois le point fixe de l'application des règles de simplification atteint, il n'est pas nécessaire de tenter de simplifier à nouveau ces définitions. Sans information supplémentaire aucune règle supplémentaire ne s'appliquera. Cependant, l'importation des déductions du solveur SAT modifie ce comportement. Alors, afin de profiter le plus possible des déductions du solveur, les simplifications sont appliquées sur toutes les définitions de l'initialisation au cycle courant lorsque l'importation est activé. En pratique, nous marquons dans quels cycles des informations ont été ajoutées et nous effectuons les optimisations uniquement entre le plus petit cycle où une information a été ajouté et le cycle courant.

5.8 Génération des clauses

La génération de clauses s'effectue simplement en un parcours de la liste des définitions tout en appliquant la méthode de génération décrite dans la partie 3.3.4. Ce parcours est implémenté par une fonction récursive. Cependant, comme la taille des définitions est limitée par les critères d'élimination, il n'y a pas de problème de dépassement de la pile à chaque appel de la fonction.

Quand cela est possible, la génération peut également être effectuée en appliquant le COI (k-COI) : seules les définitions référencées transitivement par la définition de la propriété ou du mauvais état courant sont générées à l'aide d'un calcul de dépendance syntaxique. Les définitions qui n'apparaissent pas dans le k-COI ne sont pas pour autant supprimées, car elles peuvent appartenir au COI des cycles suivants. Dans la pratique, comme le nombre de définitions n'est pas borné et qu'il est peut être très grand, nous utilisons une structure pour stocker au fur et à mesure les définitions qui doivent être transformées en CNF (par exemple une pile) pour ne pas utiliser de récursion et ainsi éviter un dépassement de la taille de la pile.

5.9 Dépliage vers l'arrière

Le dépliage vers l'arrière et par conséquent le dépliage hybride de la relation de transition (décrit dans la partie 3.6) ne sont pas aussi simples à mettre en place que la dépliage vers l'avant. Il y a deux solutions possibles. Une première solution, la version « simple », consiste à ajouter des contraintes d'équivalences

pour chaque variable mémoire connectant les différentes profondeurs, comme illustré sur l'exemple suivant :

Modèle			Cycle 2	Cycle 1	Cycle 0
$x := \bot,\ \neg u$	$\rightarrow$	$\ldots$	x_2	x_1	x_0
$y := \bot,\ u$	$\rightarrow$	$\ldots$	y_2	y_1	y_0
$u := x \wedge y$	$\rightarrow$	$\ldots$	$u_2 := x_2 \wedge y_2$	$u_1 := x_1 \wedge y_1$	$u_0 := x_0 \wedge y_0$
		$\ldots$	$x_1 \Leftrightarrow \neg u_2$	$x_0 \Leftrightarrow \neg u_1$	
		$\ldots$	$y_1 \Leftrightarrow u_2$	$y_0 \Leftrightarrow u_1$	

Le dépliage s'effectue de la droite vers la gauche. Le cycle 0 correspond au premier dépliage, il s'agit également du cycle dans lequel les propriétés ou les mauvais états seront activés. Le dépliage pour ce cycle 0 est le même que pour le dépliage vers l'avant. Ensuite, pour déplier la prochaine relation de transition (cycle 1), il faut considérer le cycle 0 comme le cycle suivant et non le précédent. Il faut donc que les valeurs « suivantes » des variables mémoires du cycle 0 correspondent aux variables du cycle précédent, *i.e.*, aux variables du cycle 1. Dans cette version, nous générons une variable pour chaque définition, ici u_1. Il faut alors ajouter des contraintes qui permettent de conserver la sémantique du modèle, pour notre exemple : $x_0 \Leftrightarrow \neg u_1$ et $y_0 \Leftrightarrow u_1$. Ce même processus est ensuite répété pour chaque nouveau dépliage.

Cette première solution a l'inconvénient de rajouter de nombreuses contraintes d'équivalences (une contrainte pour chaque variable mémoire), et donc de nombreuses clauses. La seconde solution consiste à ajouter ces contraintes seulement lorsque cela est nécessaire, comme illustré ci-dessous, en reprenant l'exemple précédent :

Modèle			Cycle 2	Cycle 1	Cycle 0
$x := \bot,\ \neg u$	$\rightarrow$	$\ldots$	x_2	x_1	x_0
$y := \bot,\ u$	$\rightarrow$	$\ldots$	y_2	y_1	y_0
$u := x \wedge y$	$\rightarrow$	$\ldots$	$\neg x_1 := x_2 \wedge y_2$	$\neg x_0 := x_1 \wedge y_1$	$u_0 := x_0 \wedge y_0$
		$\ldots$	$y_1 \Leftrightarrow \neg x_1$	$y_0 \Leftrightarrow \neg x_0$	
		$\ldots$			

L'idée consiste à remplacer la variable d'une définition par la variable mémoire déjà dépliée qui la référence. Dans l'exemple ci-dessus, la variable u_1 est remplacé par $\neg x_0$ au cycle 1. Lorsque plusieurs mémoires référencent la même variable, il est tout de même nécessaire de rajouter certaines contraintes : ici comme les mémoires x et y référencent toutes les deux la variable u, il faut ajouter la contrainte $y_0 \Leftrightarrow \neg x_0$ au cycle 1 . En outre, avec ce dépliage, il faut également prendre en compte la possibilité que la variable d'une définition soit dans la polarité négative, comme dans l'exemple ci-dessus. Cela affecte notamment la gestion des propagations, les règles de simplification, ou encore la transformation en CNF.

5.10 Redémarrage haut-niveau

Dans certains cas, la vérification d'une propriété requiert *beaucoup* de temps pour être résolue dès les premiers appels au solveur. Dans d'autres cas, il existe des modèles où tous les premiers appels au solveur sont triviaux jusqu'à un certain cycle donné (*e.g.*, résolus par simplification) à partir duquel la résolution requiert énormément de temps. Dans ces situations, la résolution incrémentale perd de son intérêt, et généralement les approches qui privilégient le prétraitement des formules CNFs (notamment avec l'inprocessing) à la résolution incrémentale sont plus performantes sur ce type de problèmes. Nous avons alors voulu essayer une méthode permettant de concilier l'approche incrémentale et l'inprocessing, en assistant le solveur lorsque certains modèles nécessitent un temps de calcul important sans inprocessing : le redémarrage haut niveau.

Nous avons alors mis en place une stratégie simple pour essayer cette approche : lorsque le solveur SAT redémarre, si le nombre de clauses unitaires ou binaires apprises depuis le dernier redémarrage haut niveau dépasse un certain seuil, alors un redémarrage haut-niveau s'effectue. Cela signifie que le solveur est mis en « pause », autrement dit la résolution est arrêtée, et la main est repassée au model checker. Les informations apprises par le solveur sont alors importées dans la représentation haut-niveau. Les simplifications haut-niveaux sont effectuées, et les conséquences de ces simplifications sont à leur tour exportées vers le solveur. Puis, ce dernier est redémarré. Cela revient donc globalement à mettre en place une méthode d'inprocessing haut niveau.

Cependant, les expérimentations préliminaires que nous avons effectuées sur les problèmes d'HWMCC'15 n'ont pas été concluantes : aucun CEX supplémentaire n'a été trouvé, aucune propriété en plus n'a été vérifiée, et nous avons observé une légère augmentation du temps de résolution sur certains problèmes. Les raisons sont certainement les mêmes que pour la stratégie de dialogue instauré entre le solveur et le model checker, c'est-à-dire qu'il faudrait importer plus d'informations du solveur, et/ou appliquer plus de règles de simplification à haut niveau pour que cela soit plus efficace.

5.11 Réinitialisation du solveur (des clauses)

Sur les modèles de tailles importantes (*e.g.*, contenant plus de $500,000$ définitions), le nombre de clauses générées par cycle est généralement considérable. Cela entraîne un ralentissement global du processus de propagation, et ce, même avec la gestion paresseuse implémentée dans les solveurs. On observe alors naturellement une baisse des performances du solveur sur ce type de problème. Or, d'un cycle à l'autre, le k-COI peut varier et donc certaines clauses qui ont été ajoutées lors des cycles précédents ne sont plus utiles. Ce phénomène est illustré sur le schéma 5.5 ci-dessous. Sur cette figure, nous

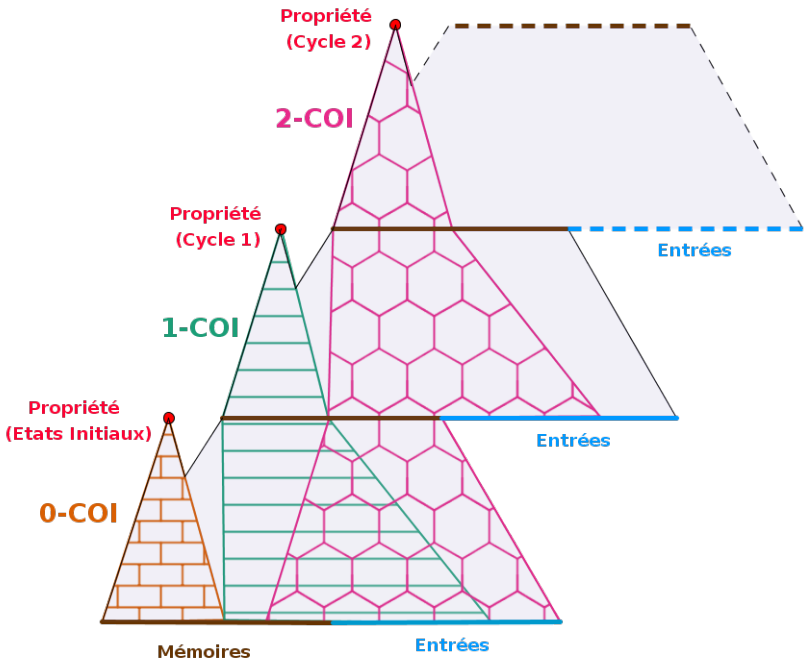


FIGURE 5.5 – Évolution du k-COI

avons volontairement exagéré les différences entre les COIs à chaque profondeur. Ainsi, on peut bien voir qu'entre chaque k-COI, il peut y avoir des clauses qui ont été passées au solveur alors qu'elles n'appartiennent plus aux k-COI suivants. En pratique, les COIs successifs ont généralement une majorité de définitions communes.

Ce phénomène est amplifié avec l'importation des déductions du solveur. L'importation au niveau du model checker d'une équivalence va entraîner l'élimination de l'une des définitions ainsi que toutes les définitions référencées par celle-ci. Or, ces informations restent sous formes de clauses dans le solveur. Nous avons alors tenté une approche qui consiste à supprimer toutes les clauses qui ont été passées au solveur, tout en conservant les clauses apprises pour conserver l'apport de la résolution incrémentale. Ensuite, les définitions appartenant au k-COI suivant sont transformées en CNF, puis ajoutées au solveur. Ainsi, les anciennes clauses qui ne sont pas dans le k-COI courant ne seront plus dans le solveur. Pour savoir quand effectuer cette opération de réinitialisation du solveur, nous calculons la différence entre le nombre de définitions qui ont été passées au solveur et le nombre de définitions qu'il y a dans le k-COI courant. Si ce nombre est supérieur à un seuil fixé, *e.g.*, $30,000$ définitions, alors nous réinitialisons le solveur. Sur l'ensemble des problèmes d'HWMCC'15, le surcoût lié à l'application de cette méthode, à savoir, le calcul du nombre de définitions, la suppression de l'ensemble des clauses originales du solveur et la transformation des définitions du k-COI courant en

CNF, ne permet pas d'améliorer les performances. Une autre piste pourrait consister à réinitialiser d'autres composants du solveur (*e.g.*, le score d'activité des variables) ou encore à réinitialiser les paramètres de gestion de la base de donnée des clauses apprises (*e.g.*, nombre de conflits entre chaque suppression des clauses apprises).

5.12 Description de l'outil développé

5.12.1 Reproductibilité

Pour la reproductibilité des résultats, nous avons figé la version de chaque outil utilisé dans nos expérimentations. L'ensemble de ces outils est disponible à l'adresse suivante : Bitbucket [2018]. Les noms des archives des différents outils sont relativement explicites :

— `tool_thesis_ve_3.3` correspond à l'outil utilisé pour l'élimination de variables dans l'expérimentation de la partie 3.3,

— `tool_thesis_inprep_3.4` correspond à l'expérimentation de la partie 3.4,

— `tool_thesis_dupli_3.5` de la partie 3.5,

— `tool_thesis_dupli_bmc_3.5.6` de la partie 3.5.6,

— `tool_thesis_bwd_3.6` et `tool_thesis_hybrid_3.6` correspondent, respectivement, aux outils effectuant le dépliage vers l'arrière et le dépliage hybride de la partie 3.6.

5.12.2 Description des fichiers

Nous décrivons rapidement les fonctionnalités implémentés dans chaque fichier :

— `aigprep` correspond à l'étape de prétraitement de l'AIG. Ce fichier contient donc ce que nous avons décrit dans la partie 5.2 ci-dessus.

— `utils` détaille les structures de données utilisées pour les représentations utilisés dans l'ensemble du processus de vérification.

— `hl_ve` contient l'algorithme d'élimination de variables à haut niveau (partie 5.4).

— `unroll` fournit les fonctions d'initialisation de la représentation interne, ainsi que les fonctions de dépliage (partie 5.5) et de mise à plat des définitions (partie 5.6).

— `un_simp` contient l'implémentation du prétraitement des définitions dépliés, décrit dans la partie 5.7.

— `solver` contient toutes les fonctions d'interface avec le solveur, *i.e.*, initialisation du solveur, génération et ajout des clauses (partie 5.8), importation des déductions, etc.

Le nom des fichiers restants sont explicites. Ils implémentent les algorithmes correspondant à leur noms en utilisant les fonctions des différents fichiers ci-dessus, *i.e.*, `bmc`, `zigzag`, `zigzag_bwd`, `dual`, `pdr`.

5.12.3 Compilation

La compilation de tous les outils s'effectue de la même manière. Après extraction de l'archive, il faut dans un premier temps générer les librairies de `Glucose` :

```
$ cd glucose_syrup_mod/simp
$ make libr
$ cd ../..
```

Puis, compiler la librairie `AIGER` :

```
$ cd aiger-1.9.4
$ ./configure
$ make
$ cd ..
```

Enfin, compiler l'outil en question :

```
$ make
```

5.12.4 Exécution

La commande de lancement est quasi identique entre les différents outils. Le format est le suivant :

```
$ ./mc [-options] model.aig
```

Pour avoir la liste des options d'un outil en particulier, il suffit d'utiliser l'option « -h ». Voici quelques exemples des lignes de commande utilisées dans nos script de lancement des outils avec les options correspondantes :

— ZigZag avec dépliage vers l'arrière (`tool_thesis_bwd_3.6`) :

```
$ ./mc -b -simple_path_od -bwd $bench
```

L'option « -b » indique au model checker d'utiliser la sortie de l'AIG comme mauvais état. L'option « -simple_path_od » spécifie qu'il faut ajouter les contraintes de chemin simple à la demande. L'option « -bwd » indique que le relation de transition est dépliée vers l'arrière. L'option « $bench » est la liste de modèles AIG.

— ZigZag avec duplication (`tool_thesis_dupli_3.5`) :

```
$ ./repMC -b -simple_path_od -replicate -rep_sz 5 -repG
   $bench
```

L'option « -replicate » signifie que les clauses sont dupliquées dans la base de données des clauses originales. L'option « -rep_sz 5 » spécifie que seules les clauses de tailles inférieures ou égales à 5 sont dupliquées. L'option « -repG » indique que toutes les clauses glues sont dupliquées.

5.13 Discussion

Globalement, les outils sont encore au stade de prototypes, ils ont été développés avec des contraintes de temps assez fortes et sont donc encore susceptible de contenir certains bugs non couverts par nos expérimentations. Le déroulement de la thèse s'est également effectué en parallèle du développement de maquettes implantant les différentes idées pour en juger l'intérêt pratique. Ainsi, certains modules peuvent nécessiter une certaine refonte pour être plus facilement réutilisables et plus performants, notamment l'étape de prétraitement à laquelle il faudrait également ajouter les méthodes de *sweeping*. Il faudrait aussi améliorer le module de prétraitement du modèle déplié qui est relativement basique. Cela permettrait entre autres d'améliorer l'efficacité du dialogue.

Parmi les méthodes de MC usuels, nous n'avons pas implémenté d'algorithmes basés sur l'interpolation. Nous n'avons pas non plus mis en place de techniques d'abstraction. Il serait alors intéressant de les implémenter, particulièrement si l'on souhaite définir une stratégie de parallélisation avec un *portefeuille* (*portfolio*) de chacun des algorithmes.

Nous avons également implémenté une version de PDR très inspirée de la version originale (*i.e.*, celle décrite dans Een *et al.* [2011]) dans laquelle nous n'avons pas encore ajouté les dernières idées innovantes (*e.g.*, Hassan *et al.* [2013]; Ivrii et Gurfinkel [2015]). Elle nécessite donc des améliorations pour être réellement compétitive.

Conclusion

Dans cette thèse, nous nous sommes concentrés sur la vérification de modèles provenant de circuits électroniques ou sur la représentation de systèmes logiciels critiques obtenue après l'étape de *bit-blasting*. Nous avons proposé des méthodes qui visent à améliorer l'algorithme d'induction temporelle lorsque la satisfaisabilité des formules générées est déterminée à l'aide d'un solveur SAT. Plus particulièrement, nous avons proposé des idées dont l'objectif était d'augmenter l'efficacité de notre model checker en enrichissant les connexions entre ce dernier et le solveur.

Résumé

Dans le chapitre 3, nous avons dans un premier temps proposé une élimination des variables à haut niveau pour remplacer celle effectuée dans le solveur. Cela nous a permis de conserver les bénéfices apportés par cette méthode tout en gardant une représentation homogène entre le model checker et le solveur. Ainsi, nous avons pu établir un dialogue simple entre le solveur et le model checker. Dans ce but, nous avons défini les informations que nous souhaitions extraire depuis l'ensemble des clauses du solveur, ainsi qu'une stratégie d'importation de ces informations. Les simplifications importantes qui sont par la suite effectuées au niveau du model checker sur des parties déjà transformées en clauses sont transmises au solveur sous forme de clauses supplémentaires. Ensuite, nous avons proposé une approche pour exploiter la structure symétrique des formules générées afin de dupliquer certaines clauses. Cette approche a l'avantage de ne nécessiter aucune modification des formules originales. Elle requiert simplement l'ajout d'une surcouche très élémentaire qui regarde les littéraux de chaque clause apprise pour déterminer si elles sont duplicables ou non. Enfin, nous avons observé l'impact du dépliage de la relation de transition vers l'arrière et proposé un dépliage hybride.

Le chapitre 4 présente une étude que nous avons effectuée avec Jesús Giráldez Crú. L'objectif de ce travail est légèrement différent de celui du chapitre précédent, dans le sens où il visait à comprendre la structure de la CNF et à observer le comportement du solveur pendant la résolution. La finalité, étroitement liée à l'idée « fil rouge » de notre travail, était de spécialiser le solveur pour

les instances BMC, *i.e.*, améliorer ses performances pour ce type de formules, simplement en ajoutant des informations lors de l'encodage de la formule mais sans modifier le model checker. L'objectif était encore une fois l'amélioration de la communication entre le model checker et le SAT solveur. Nous avons d'abord montré qu'il existait une relation entre la structure en communauté des formules et la profondeur des variables. Puis, toujours en observant la profondeur des variables nous avons examiné le comportement du solveur lors de son exécution, *i.e.*, la profondeur des variables de décisions, la profondeur des variables participants à la résolution, et la profondeur des variables des clauses apprises.

Le dernier chapitre (5) décrit en détail les méthodes que nous avons implémentées dans l'outil développé au cours de cette thèse. Plus précisément, nous énumérons les règles de simplification que nous appliquons sur le modèle d'entrée, ainsi que sur le modèle après dépliage. Nous décrivons aussi l'implémentation de certaines fonctionnalités internes telle que le dépliage (vers l'avant et vers l'arrière), la mise à plat des définitions après élimination, la génération des clauses, etc.

Perspectives

En plus des travaux présentés dans cette thèse, il y a de nombreuses méthodes que nous n'avons pas pu implémenter faute de temps. Comme évoqué précédemment, nous aurions souhaité étendre encore plus l'expressivité de la représentation interne du model checker afin de pouvoir exprimer de simples implications (et non uniquement des définitions) ce qui nous aurait permis par exemple d'éliminer les entrées à haut niveau, d'importer facilement beaucoup plus d'informations depuis le solveur, etc. Néanmoins, pour avoir une méthode efficace il aurait également fallu définir de nombreuses autres règles de simplification mixant définitions et simples implications.

Concernant la duplication, nous espérons que nos résultats encourageront d'autres travaux, que ce soit des améliorations techniques pour la gestion des clauses dupliquées, des stratégies de sélection plus élaborées, ou encore une duplication des clauses dans d'autres algorithmes. Aussi, comme abordé dans la partie 3.6, intégrer et observer l'impact de la duplication lorsque la relation de transition est dépliée vers l'arrière ou de façon hybride pourrait constituer un travail intéressant.

En ce qui concerne la spécialisation du solveur pour BMC, il y a également plusieurs pistes pour continuer ou améliorer ces travaux. Par exemple, nous aurions aimé observer l'impact sur la structure en communautés ou pendant la résolution, lorsque l'on utilise des techniques modifiant les formules générées (*e.g.*, le prétraitement du modèle en entrée et du modèle dépliée ou l'élimination de variables). Une autre idée pourrait consister à effectuer une analyse

similaire mais cette fois en prenant en compte un contexte de résolution incrémentale. Naturellement, il serait également intéressant de reproduire le même type d'analyse pour d'autres problèmes : CSP (*Constraint Satisfaction Problem*), problèmes cryptographiques, etc.

Discussion

Les expérimentations effectuées nous ont permis d'identifier différents axes d'amélioration ayant pour but d'augmenter les performances de notre model checker pour l'analyse de modèles complexes et de grandes tailles. Globalement, l'ensemble des approches que nous avons proposé semble améliorer légèrement les performances de l'induction temporelle dans un seul solveur (algorithme ZigZag). La prochaine grande étape est maintenant de combiner et d'implémenter correctement ces techniques afin de pouvoir les intégrer dans l'outil de vérification développé au sein de `SafeRiver`. Cette intégration permettrait d'étudier l'efficacité des algorithmes proposés dans cette thèse sur un grand nombre d'instances de différents domaines industrielles (*e.g.*, ferroviaire, automobile, etc.). En effet, ces problèmes couvrent plusieurs catégories de modèles allant de simples systèmes numériques de contrôle-commande à des algorithmes de traitement de signal, des algorithmes d'ordonnancement, et également des fonctions cryptographiques complexes. La deuxième grande étape que nous aurions aimé réaliser si nous avions eu plus temps est l'intégration d'autres algorithmes, et plus particulièrement PDR. Ce dernier n'a malheureusement pas encore été réellement intégré aux outils industriels effectuant la vérification de systèmes logiciels. Il est donc nécessaire d'effectuer des expérimentations afin de montrer son efficacité sur les modèles industriels. Nous avons commencé les expérimentations avec notre version de Een *et al.* [2011], mais nous n'avons pas encore pu essayer et évaluer expérimentalement certaines améliorations comme, par exemple, l'ajout des CTGs (*Counterexample To Generalization*, Hassan *et al.* [2013]).

Bibliographie

Ansótegui, Carlos, Giráldez-Cru, Jesús et Levy, Jordi, 2012. The community structure of SAT formulas. Dans *Proc. of SAT'12*, pages 410–423.

Arnold, André, Point, Gérald, Griffault, Alain et Rauzy, Antoine, 1999. The altarica formalism for describing concurrent systems. *Fundam. Inform.*, 40(2-3) :109–124.

Audemard, Gilles et Simon, Laurent, 2009. Predicting learnt clauses quality in modern sat solvers. Dans *IJCAI*, tome 9, pages 399–404.

Audemard, Gilles et Simon, Laurent, 2013. The glucose sat solver.

Barrett, Clark, Conway, Christopher L, Deters, Morgan, Hadarean, Liana, Jovanović, Dejan, King, Tim, Reynolds, Andrew et Tinelli, Cesare, 2011. Cvc4. Dans *International Conference on Computer Aided Verification*, pages 171–177. Springer.

Barrett, Clark, Stump, Aaron, Tinelli, Cesare *et al.*, 2010. The smt-lib standard : Version 2.0. Dans *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, tome 13, page 14.

Barrett, Clark W, Sebastiani, Roberto, Seshia, Sanjit A, Tinelli, Cesare *et al.*, 2009. Satisfiability modulo theories. *Handbook of satisfiability*, 185 :825–885.

Biere, Armin, 2007. The aiger and-inverter graph (aig) format. *Available at fmv. jku. at/aiger*.

Biere, Armin, 2009. P {re, i} cosat@ sc'09. *SAT*, 4 :41–43.

Biere, Armin, 2016. Collection of combinational arithmetic miters submitted to the sat competition 2016. *Proc. of SAT Competition*, pages 65–66.

Biere, Armin, Artho, Cyrille et Schuppan, Viktor, 2002. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2) :160–177.

Biere, Armin, Cimatti, Alessandro, Clarke, Edmund et Zhu, Yunshan, 1999. *Symbolic model checking without BDDs*. Springer.

Biere, Armin, Heljanko, Keijo et Wieringa, Siert, 2011. Aiger 1.9 and beyond.

Biere, Armin, Heule, Marijn et van Maaren, Hans, 2009. *Handbook of satisfiability*, tome 185. IOS press.

Biere, Armin, Le Berre, Daniel, Lonca, Emmanuel et Manthey, Norbert, 2014. Detecting cardinality constraints in cnf. Dans *International Conference on Theory and Applications of Satisfiability Testing*, pages 285–301. Springer.

Biere, Armin, van Dijk, Tom et Heljanko, Keijo, 2017. Hardware model checking competition 2017. Dans *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 9–9. FMCAD Inc.

Bitbucket, 2018. Outils utilisés pour les expérimentations.
URL https://bitbucket.org/GuillaumeBB/thesis/downloads/

Blondel, Vincent D, Guillaume, Jean-Loup, Lambiotte, Renaud et Lefebvre, Etienne, 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics : Theory and Experiment*, 2008(10) :P10008.

Boy de la Tour, Thierry, 1992. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14(4) :283–301.

Bradley, Aaron R, 2011. Sat-based model checking without unrolling. Dans *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer.

Brayton, Robert et Mishchenko, Alan, 2010. Abc : An academic industrial-strength verification tool. Dans *Computer Aided Verification*, pages 24–40. Springer.

Brummayer, Robert et Biere, Armin, 2006. Local two-level and-inverter graph minimization without blowup. *Proc. MEMICS*, 6 :32–38.

Brummayer, Robert et Biere, Armin, 2009. Boolector : An efficient smt solver for bit-vectors and arrays. Dans *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer.

Bryant, Randal E, 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8) :677–691.

Bryant, Randal E et Seger, Carl-Johan H, 1990. Formal verification of digital circuits using symbolic ternary system models. Dans *International Conference on Computer Aided Verification*, pages 33–43. Springer.

Burch, Jerry R, Clarke, Edmund M, McMillan, Kenneth L, Dill, David L et Hwang, Lain-Jinn, 1990. Symbolic model checking : 10 20 states and beyond. Dans *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439. IEEE.

Chambers, Benjamin, Manolios, Panagiotis et Vroon, Daron, 2009. Faster sat solving with better cnf generation. Dans *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1590–1595. European Design and Automation Association.

Champion, Adrien, Delmas, Rémi et Dierkes, Michael, 2013. Generating property-directed potential invariants by backward analysis. *arXiv preprint arXiv :1301.0039*.

Cimatti, Alessandro, Griggio, Alberto, Schaafsma, Bastiaan Joost et Sebastiani, Roberto, 2013. The mathsat5 smt solver. Dans *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer.

Clarke, Edmund, Grumberg, Orna, Jha, Somesh, Lu, Yuan et Veith, Helmut, 2000. Counterexample-guided abstraction refinement. Dans *Computer aided verification*, pages 154–169. Springer.

Clarke, Edmund M., Emerson, E. Allen et Sistla, A. Prasad, 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263.

Clarke, Edmund M, Grumberg, Orna et Peled, Doron, 1999. *Model checking*. MIT press.

Colón, Michael A, Sankaranarayanan, Sriram et Sipma, Henny B, 2003. Linear invariant generation using non-linear constraint solving. Dans *Computer Aided Verification*, pages 420–432. Springer.

Conchon, Sylvain, Contejean, Evelyne, Kanig, Johannes et Lescuyer, Stéphane, 2008. Cc (x) : Semantic combination of congruence closure with solvable theories. *Electronic Notes in Theoretical Computer Science*, 198(2) :51–69.

Cortadella, Jordi, 2003. Timing-driven logic bi-decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6) :675–685.

COUDERT, Olivier, BERTHET, Christian et MADRE, Jean Christophe, 1989. Verification of synchronous sequential machines based on symbolic execution. Dans *International Conference on Computer Aided Verification*, pages 365–373. Springer.

COUSOT, Patrick et COUSOT, Radhia, 1977. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Dans *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM.

CRAIG, William, 1957. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03) :269–285.

CYTRON, Ron, FERRANTE, Jeanne, ROSEN, Barry K, WEGMAN, Mark N et ZADECK, F Kenneth, 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4) :451–490.

DABNEY, James B et HARMAN, Thomas L, 2004. *Mastering simulink*. Pearson.

DAVIS, Martin, LOGEMANN, George et LOVELAND, Donald, 1962. A machine program for theorem-proving. *Communications of the ACM*, 5(7) :394–397.

DAVIS, Martin et PUTNAM, Hilary, 1960. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3) :201–215.

DERSHOWITZ, Nachum, HANNA, Ziyad et NADEL, Alexander, 2007. Towards a better understanding of the functionality of a conflict-driven sat solver. *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 287–293.

DORMOY, Francois-Xavier, 2008. Scade 6 : a model based solution for safety critical software development. Dans *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*, pages 1–9.

EÉN, Niklas et BIERE, Armin, 2005. Effective preprocessing in sat through variable and clause elimination. Dans *Theory and Applications of Satisfiability Testing*, pages 61–75. Springer.

EEN, Niklas, MISHCHENKO, Alan et BRAYTON, Robert, 2011. Efficient implementation of property directed reachability. Dans *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125–134. IEEE.

EEN, Niklas, MISHCHENKO, Alan et SÖRENSSON, Niklas, 2007. Applying logic synthesis for speeding up sat. Dans *International Conference on Theory and Applications of Satisfiability Testing*, pages 272–286. Springer.

EÉN, Niklas et SÖRENSSON, Niklas, 2003a. An extensible sat-solver. Dans *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer.

EÉN, Niklas et SÖRENSSON, Niklas, 2003b. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4) :543–560.

EMERSON, E Allen, 1990. Temporal and modal logic. *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, 995(1072) :5.

ERNST, Michael D, COCKRELL, Jake, GRISWOLD, William G et NOTKIN, David, 2001. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2) :99–123.

FORTUNATO, Santo, 2010. Community detection in graphs. *Physics Reports*, 486(3-5) :75 – 174.

FU, Zhaohui et MALIK, Sharad, 2007. Extracting logic circuit structure from conjunctive normal form descriptions. Dans *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*, pages 37–42. IEEE.

GANAI, Malay K, ZHANG, Lintao, ASHAR, Pranav, GUPTA, Aarti et MALIK, Sharad, 2002. Combining strengths of circuit-based and cnf-based algorithms for a high-performance sat solver. Dans *Design Automation Conference, 2002. Proceedings. 39th*, pages 747–750. IEEE.

GUERRA E SILVA, Luis, SILVEIRA, L Miguel et MARQUES-SILVA, Joöa, 1999. Algorithms for solving boolean satisfiability in combinational circuits. Dans *Proceedings of the conference on Design, automation and test in Europe*, page 107. ACM.

GUPTA, Ashutosh et RYBALCHENKO, Andrey, 2009. Invgen : An efficient invariant generator. Dans *Computer Aided Verification*, pages 634–640. Springer.

HALBWACHS, Nicholas, CASPI, Paul, RAYMOND, Pascal et PILAUD, Daniel, 1991. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320.

HASSAN, Zyad, BRADLEY, Aaron R et SOMENZI, Fabio, 2013. Better generalization in ic3. Dans *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 157–164. IEEE.

Ivrii, Alexander et Gurfinkel, Arie, 2015. Pushing to the top. Dans *Formal Methods in Computer-Aided Design (FMCAD), 2015*, pages 65–72. IEEE.

Järvisalo, Matti, Biere, Armin et Heule, Marijn, 2010. Blocked clause elimination. Dans *Tools and Algorithms for the Construction and Analysis of Systems*, pages 129–144. Springer.

Järvisalo, Matti, Biere, Armin et Heule, Marijn JH, 2012a. Simulating circuit-level simplifications on cnf. *Journal of Automated Reasoning*, 49(4) :583–619.

Järvisalo, Matti, Heule, Marijn JH et Biere, Armin, 2012b. Inprocessing rules. Dans *International Joint Conference on Automated Reasoning*, pages 355–370. Springer.

Jhala, Ranjit et McMillan, Kenneth L, 2005. Interpolant-based transition relation approximation. Dans *International Conference on Computer Aided Verification*, pages 39–51. Springer.

Knuth, DE, 2009. The art of computer programming : Bitwise tricks & techniques ; binary decision diagrams, volume 4, fascicle 1.

Kroening, Daniel et Strichman, Ofer, 2008. *Decision procedures*, tome 5. Springer.

Kullmann, Oliver, 1999. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96 :149–176.

Kupferman, Orna et Vardi, Moshe Y, 2001. Model checking of safety properties. *Formal Methods in System Design*, 19(3) :291–314.

Kupferschmid, Stefan, Lewis, Matthew, Schubert, Tobias et Becker, Bernd, 2011. Incremental preprocessing methods for use in bmc. *Formal Methods in System Design*, 39(2) :185–204.

Li, Chu-Min, 2003. Equivalent literal propagation in the dll procedure. *Discrete Applied Mathematics*, 130(2) :251–276.

Luby, Michael, Sinclair, Alistair et Zuckerman, David, 1993. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4) :173–180.

Manolios, Panagiotis et Vroon, Daron, 2007. Efficient circuit to cnf conversion. Dans *International Conference on Theory and Applications of Satisfiability Testing*, pages 4–9. Springer.

McMillan, Kenneth L, 2003. Interpolation and sat-based model checking. Dans *Computer Aided Verification*, pages 1–13. Springer.

McMillan, Kenneth L, 2006. Lazy abstraction with interpolants. Dans *International Conference on Computer Aided Verification*, pages 123–136. Springer.

McMillan, Kenneth L et Amla, Nina, 2003. Automatic abstraction without counterexamples. Dans *TACAS*, tome 3, pages 2–17. Springer.

Mishchenko, Alan, Chatterjee, Satrajit et Brayton, Robert, 2006. Dag-aware aig rewriting : A fresh look at combinational logic synthesis. Dans *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 532–535. IEEE.

Monniaux, David, 2016. A survey of satisfiability modulo theory. Dans *International Workshop on Computer Algebra in Scientific Computing*, pages 401–425. Springer.

Moskewicz, Matthew W, Madigan, Conor F, Zhao, Ying, Zhang, Lintao et Malik, Sharad, 2001. Chaff : Engineering an efficient sat solver. Dans *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM.

Nadel, Alexander, Ryvchin, Vadim et Strichman, Ofer, 2012. Preprocessing in incremental sat. Dans *SAT*, pages 256–269. Springer.

Nadel, Alexander, Ryvchin, Vadim et Strichman, Ofer, 2014. Ultimately incremental sat. Dans *International Conference on Theory and Applications of Satisfiability Testing*, pages 206–218. Springer.

Nelson, Greg et Oppen, Derek C, 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2) :245–257.

Newman, M. E. J. et Girvan, M., 2004. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2) :026113.

Newsham, Zack, Ganesh, Vijay, Fischmeister, Sebastian, Audemard, Gilles et Simon, Laurent, 2014. Impact of community structure on sat solver performance. Dans *Proc. of SAT'14*, pages 252–268.

Niemetz, Aina, Preiner, Mathias et Biere, Armin, 2015. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9 :53–58.

Niemetz, Aina, Preiner, Mathias et Biere, Armin, 2016. Precise and complete propagation based local search for satisfiability modulo theories. Dans *International Conference on Computer Aided Verification*, pages 199–217. Springer.

Nieuwenhuis, Robert, Oliveras, Albert et Tinelli, Cesare, 2006. Solving sat and sat modulo theories : From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6) :937–977.

Ostrowski, Richard, Grégoire, Éric, Mazure, Bertrand et Sais, Lakhdar, 2002. Recovering and exploiting structural knowledge from cnf formulas. Dans *International Conference on Principles and Practice of Constraint Programming*, pages 185–199. Springer.

Pearson, Karl, 1895. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58 :240–242.

Pipatsrisawat, Knot et Darwiche, Adnan, 2007. A lightweight component caching scheme for satisfiability solvers. *SAT*, 4501 :294–299.

Plaisted, David A et Greenbaum, Steven, 1986. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3) :293–304.

Sheeran, Mary, Singh, Satnam et Stålmarck, Gunnar, 2000. Checking safety properties using induction and a sat-solver. Dans *Formal Methods in Computer-Aided Design*, pages 127–144. Springer.

Shostak, Robert E, 1984. Deciding combinations of theories. *Journal of the ACM (JACM)*, 31(1) :1–12.

Silva, João P Marques et Sakallah, Karem A, 1997. Grasp—a new search algorithm for satisfiability. Dans *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227. IEEE Computer Society.

Simon, Laurent et Katsirelos, George, 2012. Eigenvector centrality in industrial sat instances. Dans *Proc. of CP'12*, pages 348–356.

Strichman, Ofer, 2000. Tuning sat checkers for bounded model checking. Dans *International Conference on Computer Aided Verification*, pages 480–494. Springer Berlin Heidelberg.

Strichman, Ofer, 2001. Pruning techniques for the sat-based bounded model checking problem. Dans *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 58–70. Springer Berlin Heidelberg.

Strichman, Ofer, 2004. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1) :5–24.

Subbarayan, Sathiamoorthy et Pradhan, Dhiraj K, 2004. Niver : Non-increasing variable elimination resolution for preprocessing sat instances. Dans *International conference on theory and applications of satisfiability testing*, pages 276–291. Springer.

Tinelli, Cesare, 2002. A dpll-based calculus for ground satisfiability modulo theories. Dans *European Workshop on Logics in Artificial Intelligence*, pages 308–319. Springer.

Tseitin, Grigorii Samuilovich, 1968. On the complexity of proof in prepositional calculus. *Zapiski Nauchnykh Seminarov POMI*, 8 :234–259.

Velev, Miroslav N, 2004. Efficient translation of boolean formulas to cnf in formal verification of microprocessors. Dans *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pages 310–315. IEEE Press.

Vizel, Yakir et Grumberg, Orna, 2009. Interpolation-sequence based model checking. Dans *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 1–8. IEEE.

Vizel, Yakir et Gurfinkel, Arie, 2014. Interpolating property directed reachability. Dans *Computer Aided Verification*, pages 260–276. Springer.

Vizel, Yakir, Weissenbacher, Georg et Malik, Sharad, 2015. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11) :2021–2035.

Warners, Joost P et Van Maaren, Hans, 1998. A two-phase algorithm for solving a class of hard satisfiability problemsfn1. *Operations research letters*, 23(3-5) :81–88.

Welp, Tobias, 2013. Program verification with property directed reachability.

Welp, Tobias et Kuehlmann, Andreas, 2013. Qf bv model checking with property directed reachability. Dans *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 791–796. EDA Consortium.

Wetzler, Nathan, Heule, Marijn JH et Hunt, Warren A, 2014. Drat-trim : Efficient checking and trimming using expressive clausal proofs. Dans *International Conference on Theory and Applications of Satisfiability Testing*, pages 422–429. Springer.

Yin, Liangze, He, Fei, Gu, Ming et Sun, Jiaguang, 2014. Clause replication and reuse in incremental temporal induction. Dans *19th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 108–115. IEEE.

ZHANG, Lintao, MADIGAN, Conor F, MOSKEWICZ, Matthew H et MALIK, Sharad, 2001. Efficient conflict driven learning in a boolean satisfiability solver. Dans *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press.