Modernisation de la duplication de clauses

Guillaume Baud-Berthier[12] **Laurent Simon**[1]

[1] Université de Bordeaux, LaBRI
[2] SafeRiver
guillaume.baud-berthier@safe-river.com lsimon@labri.fr

Résumé

Le Model Checking est une méthode de vérification formelle très populaire dans le milieu industriel, en raison de son automatisation et de sa faculté à produire une trace d'exécution courte lorsqu'il existe un comportement non désiré. Le model checking borné (BMC) et la k-induction sont deux méthodes paramétrées, allant de pair. Elles sont toutes les deux très dépendantes de l'efficacité de leur procédure de décision sous-jacente, généralement un solveur SAT/SMT, en raison notamment de leur propension à générer des formules de plus en plus grandes. Ces formules ont une caractéristique particulière : elles peuvent être trivialement partitionnées en 3 ensembles, dont l'un possède une structure symétrique. Dans cet article, nous proposons d'exploiter cette structure symétrique en dupliquant les clauses apprises par le solveur. Cette idée fut déjà proposée peu après l'introduction de BMC, mais rapidement abandonnée à cause de sa tendance à surcharger le solveur SAT avec un nombre trop important de clauses. Nous proposons d'étendre la duplication pour la k-induction et suggérons une méthode simplifiée, i.e. sans incidence sur le solveur, pour détecter les clauses duplicables. En outre, nous montrons qu'en sélectionnant soigneusement les clauses à dupliquer et à quels temps les dupliquer, nous obtenons des gains intéressants dans notre model checker, ce qui va à l'encontre de l'idée communément admise.

Abstract

Model Checking is a well-established formal verification method in industry, mainly because of its automation and its ability to produce short execution trace when a bug is found. Bounded Model Checking (BMC) and k-induction are both parameterized methods, working together, and relying almost entirely on their underlying decision procedure, e.g. SAT/SMT solver. The formulas generated by those methods can be trivially partitioned into 3 sets, and such that one of these set is perfectly symmetric. In this paper, we propose to exploit this symmetry to replicate learnt clauses. This was already suggested in the early years of BMC, but quickly left aside because of its tendency to overburden the solver with too many clauses. In this paper, we extend replication to k-induction and propose a simpler but sound strategy to detect replicable clauses. Moreover, we show that by carefully selecting learnt clauses to replicate and where to replicate them, we can observe an interesting progress in our model checker.

1 Introduction

Le model checking est une technique de vérification formelle très utilisée dans le milieu industriel, pour vérifier les circuits électroniques, ou plus généralement les systèmes informatiques dits *critiques*. Ceci est principalement dû à son caractère automatique et à sa capacité de générer de courtes traces d'exécution menant à un comportement non désiré (*bug*). Le model checking borné (BMC) [4] est une méthode paramétrée qui vérifie exhaustivement, dans un système, l'absence de chemins d'une taille donnée menant à un *mauvais* état. Il s'agit d'une des premières applications industrielles et certainement du premier succès historique des solveurs SAT (avec la planification). Cependant, BMC est un algorithme orienté vers la recherche de *contre-exemple* et n'est généralement pas utilisé pour en prouver l'absence [1], ce qui a conduit à l'introduction de la *k-induction* [12], qui est basée sur le principe de récurrence, et ajoute donc un critère de terminaison : l'idée est de montrer que, lorsque le système vérifie le comportement souhaité pendant un nombre de cycle donné (cette hypothèse est fournie par BMC), alors on ne peut plus atteindre un mauvais état. Les performances de ces deux méthodes de model checking sont très dépendantes des performances de la procédure de décision sous-jacente, ici le solveur SAT.

1. En effet, il est très difficile de déterminer la profondeur séquentielle d'un système.

Ces méthodes, générant des formules SAT extraordinairement grandes (pouvant dépasser plusieurs milliards de clauses et des millions de variables), ou demandant des milliers d'appels SAT sur une même instance, ont poussé un changement de paradigme dans la conception des solveurs SAT. Les solveurs SAT "modernes", ou Conflict-Driven Clause Learning (CDCL) [11] ont été initialement conçus pour résoudre de telles instances.

Ce papier s'inscrit dans une lignée de travaux visant à interconnecter le model checker et le solveur SAT. Plus précisement, l'objectif est d'améliorer les performances du solveur en utilisant des éléments d'informations de plus haut niveaux, i.e. disponible au niveau du model checker. Aux prémices de BMC, plusieurs idées ont émergé [13, 14, 16] :

1. Réutilisation des clauses apprises lors des appels successifs BMC avec une profondeur de plus en plus grande : *SAT incrémental*.
2. Remplacement de l'heuristique de sélection des variables dans le solveur par un ordre statique basé sur le modèle, e.g. brancher en priorité sur les entrées.
3. Exploiter la structure symétrique de la formule BMC pour dupliquer les clauses apprises par le solveur.

La résolution incrémentale est, de nos jours, utilisée dans presque tous les model checkers effectuant BMC et la k-induction. La sélection des variables fut abandonnée en faveur de VSIDS [11] ayant une plus grande adaptabilité, fournissant ainsi de meilleurs performances de manière générale. Dans cet article, nous nous intéressons plus particulièrement à la duplication de clauses. Cette dernière fut abandonnée en raison de sa propension à générer beaucoup de clauses dupliquées, ce qui entraine une surcharge du solveur et par conséquent ralentit la propagation. Notre regain d'intérêt pour cette technique provient de deux nouvelles caractéristiques des solveurs SAT récents : la suppression aggressive de la base de données des clauses apprises et le score *LBD (Literal Block Distance)* permettant d'évaluer la *qualité* des clauses [1]. Les contributions de ce papier sont les suivantes :

— Extension de la duplication des clauses à l'algorithme ZigZag (BMC et k-induction), tout en considérant les contraintes de chemin simple.
— Simplification de la méthode de caractérisation des clauses duplicables, en d'autres mots, comment déterminer si une clause est duplicable ou non.
— Sélection des clauses à dupliquer en fonction de leur taille et de leur score LBD, ainsi que le choix de la base de données dans laquelle ajouter ces clauses dupliquées.

Le reste de l'article est organisé comme suit. Dans un premier temps, nous rappelons les principes généraux des solveurs SAT et des algorithmes de vérification de modèle. Dans la partie 3, nous présentons de façon informelle la duplication et nous faisons un état de l'art des critères de duplicabilité. Dans la partie 4, nous proposons une nouvelle approche pour détecter les clauses duplicable et appliquons cette méthode à l'algorithme ZigZag. Dans la partie 5, nous vérifions expérimentalement l'efficacité de cette approche. Enfin, nous concluons dans la dernière partie.

2 Définitions

Le problème de satisfaisabilité (SAT) est un problème NP-complet 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 en 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 également considérer une clause comme un ensemble de littéraux, e.g. $\{x, \neg y, z\}$, et une CNF comme un ensemble de clauses : $\{\{x, \neg y, z\}, \{\neg x\}, \{y, z\}\}$. 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 $\top$ en substituant chaque variable par sa valeur associée.

2.1 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 (en général : certaines approches specialisées ne rentrent pas dans ce cadre). Les solveurs plus récents peuvent également produire une preuve de non-satisfaisabilité. Cependant, pour être capable par la

suite de caractériser si une clause est duplicable ou non, nous avons besoin d'introduire quelques notions essentielles des solveurs modernes.

Un des premiers solveurs SAT, DP [7], éliminait les variables une par une, à l'aide de la *règle de résolution*, jusqu'à trouver la clause vide (UNSAT), ou jusqu'à ce que la formule devienne vide (SAT). Cependant, en raison du problème de l'explosion combinatoire en mémoire, cette technique fut rapidement abandonnée en faveur de la recherche avec saut arrière (DPLL [6]). Notons tout de même que l'élimination de variables reste une étape de pré-traitement 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 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 pré-traitement, 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.

Avec l'introduction des algorithmes modernes de résolution du problème SAT, dit CDCL [8, 11, 15], l'apprentissage de clauses devient l'élément clef des solveurs. Cependant, 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* [8], les auteurs ont proposés de supprimer les clauses qui étaient les moins vues lors des dernières analyses de conflit. Avec l'idée de mesurer la distance entre les blocs de littéraux (LBD), *Glucose* [2] a introduit une nouvelle façon de classer et donc de supprimer les clauses.

Dans les solveurs modernes, lorsqu'une affectation partielle des variables (générée à l'aide de décisions et de propagations) produit un conflit, i.e. une contradiction sur l'affection d'une variable, une analyse de ce conflit est effectuée. A partir de cette analyse, une clause est apprise pour éviter de regénérer la même affectation partielle. Intuitivement, chaque clause apprise est obtenue par des étapes de résolutions sur un sous-ensemble de clauses propagées au dernier niveau de décision. Des étapes additionnelles de résolutions, utilisées pour réduire la taille de la clause apprise, peuvent également être effectuées (voir [5] pour une référence complète). Le système de score LBD est relativement simple. Le score d'une clause est le nombre de niveaux de décisions distincts de chacun des littéraux de la clause. Dans *Glucose*, les clauses de LBD 2 sont appelées *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 d'un classement basé sur leur score LBD. 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.

2.2 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. Dans cet article, nous considérons uniquement les modèles représentant des systèmes de transitions booléens à états finis, généralement utilisés pour représenter les circuits séquentielles. En outre, nous nous limitons aux propriétés sous la forme d'assertion, e.g. de la forme AGp en CTL*.

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, 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 représente donc un ensemble d'états du système. Nous utilisons $Bad(V) = \neg P(V)$ pour exprimer l'ensemble des *mauvais* états du systèmes, i.e. 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 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.

2.2.1 BMC

Le model checking borné est un algorithme paramétré orienté vers la recherche de contre-exemples [4]. 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 [2]. La définition formelle de BMC est la suivante :

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

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 inférieur à k. C'est pourquoi, BMC est généralement exécuté de façon incrémental, 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 disponibles soient epuisées. Une autre approche, moins utilisée, consiste à étendre $Bad(V_k)$ à $\bigvee_{i=0}^{k} Bad(V_i)$.

2.2.2 k-induction

Le principe d'induction pour un système de transitions est relativement simple, il s'effectue en 2 étapes :

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

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

   ou en *formulation SAT* :

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

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

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

   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, 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. Aucune conclusion ne peut donc être tirée. L'hypothèse de récurrence doit alors être *renforcée*. La k-induction, introduite dans [12], 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, l'hérédité en k-induction consiste à vérifier que si la propriété est vraie pendant k cycles consécutifs alors elle reste vraie au cycle suivant. Ainsi, la propriété est prouvée pour l'ensemble du système. Formellement, l'hérédité est définie telle que :

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

2.2.3 Algorithme ZigZag

L'algorithme *ZigZag* introduit dans [9] consiste simplement à 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. 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 k-ind soit insatisfaisable, dans ce cas la propriété est vérifiée pour le système.
Afin de bien illustrer le déroulement de l'algorithme, voici, dans l'ordre, les premiers appels consécutifs du solveur. Par souci de clarté, nous utilisons I_i, P_i pour représenter $I(V_i)$, $P(V_i)$ et $T_{i,j}$ pour $T(V_i, V_j)$.

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

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

Les crochets sont utilisés pour mettre en avant les parties de la formule qui sont temporaires. Ces parties sont généralement *activées/désactivées* à l'aide de littéraux d'activation. Un littéral d'activation est un littéral que l'on ajoute à une ou plusieurs clauses afin de pouvoir utiliser ce littéral pour modifier la prise encompte de ces clauses (en permettant de les satisfaire trivialement). Supposons, par exemple, $C = \{\neg act, x, y\}$ une clause ajoutée à un solveur S, en forçant alors l'affectation de *act* à $\top$ dans S, on oblige le solveur à prendre en compte la contrainte $x \vee y$ et à l'inverse en forçant act à $\bot$, le solveur peut assigner librement $x = \bot, y = \bot$ en même temps.

2.2.4 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. Elles sont utilisées pour spécifier que chaque état du système dans le chemin construit pour la k-induction doit être différent [3] (Voir [12] pour plus de détails.).

En général, les contraintes *tous différents* génèrent beaucoup de clauses, même lorsque l'introduction de variables est permise. Les auteurs de [9] ont alors proposé de générer ces contraintes *à la demande*. Autrement dit, lorsque le solveur nous 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 relancé.

Cette partie décrit succinctement la fonction des contraintes de chemin simple. Ces contraintes sont ajoutées dans presque tous les model checkers effectuant la k-induction, il est donc primordial de les considérer dans notre approche. Néanmoins, le seul point important dans le contexte de ce papier est l'introduction de nouvelles variables pour ces contraintes, qui peuvent interférer lors de la duplication.

3 Duplication

Nous présentons d'abord, intuitivement, le fonctionnement de la duplication dans BMC, puis nous donnons quelques détails sur les approches utilisées.

3.1 Structure symétrique et duplication

La duplication de clauses est basée sur la structure symétrique des formules. Les formules générées par BMC contiennent une partie symétrique :

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

En d'autres mots, si on ignore les parties concernant les états initiaux et les mauvais états, la formule est parfaitement symétrique. En effet, il s'agit simplement de la répétition (k fois), ce que l'on appelle parfois le *dépliage*, de la relation de transition. 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 accèlerera ainsi potentiellement la résolution du problème.
La notation $C_{[i,j]}$ spécifie que la clause C a été déduite à l'aide de résolution ne faisant intervenir que des clauses provenant des relations de transition et tel que le plus petit (respectivement 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 $C_{[0,2]}$ 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 :

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

avec $C_{[0,2]} = \{x_0, \neg y_1, x_2\}$ et $C_{[1,3]} = \{x_1, \neg y_2, x_3\}$. 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 alors 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 $C_{[2,4]}$ peut être ajoutée, avant d'établir la satisfaisabilité de BMC_4 :

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

Cette seconde étape requiert un plus haut niveau de supervision, extérieur au solveur SAT, et s'effectue directement dans l'algorithme de model checking. La

3. Seules les variables représentant les mémoires sont considérées.

principale difficulté réside donc dans la caractérisation des clauses apprises, à savoir est-ce que cette clause est duplicable et à quels cycles peut-elle être dupliquée.

3.2 Conditions de duplication

Dans les premières tentatives de duplication, 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 si les clauses dupliquées n'avaient pas changé la satisfaisabilité de l'instance. A noter que la duplication de clauses non duplicables ne peut entraîner que des faux négatifs [4]. Cependant, ces deux méthodes sont beaucoup trop coûteuses en terme de performance, et donc même si un gain est apporté par les clauses dupliquées, il est annhilé par ce coût.

Par la suite, l'approche qui fut retenue consiste 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. L'ensemble de ces travaux sont décrits dans [13, 14, 16].

4 Duplication pour l'algorithme ZigZag

La duplication de clauses en model checking n'a pour le moment été expérimentée que pour BMC. Or, la structure de la formule de k-induction s'y prête tout autant, voir plus, car la formule ne contient pas la contrainte des états initiaux. Cependant, comme la k-induction est toujours associée à BMC, nous proposons d'étendre la duplication à l'algorithme ZigZag (partie 2.2.3).

La structure de la formule de l'algorithme ZigZag est la même que celle de BMC :

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

Les conditions de duplication sont donc applicables en suivant le même procédé. Cependant, lors de nos premières expérimentations, nous avons observé que la méthode de marquage des clauses avait un effet préjudiciable sur les performances du solveur. Les opérations nécessaires pour calculer les marqueurs sont pourtant élémentaires. Néanmoins, à chaque analyse de conflit les marqueurs de toutes les clauses qui ont mené aux conflits doivent être comparés. Or, comme ces conflits peuvent se produire plus d'un millier de fois par seconde, les effets en deviennent non négligeables.

Nous proposons une nouvelle approche qui utilise les littéraux d'activation comme marqueur de *duplicabilité*. Nous supprimons également la notion de marqueurs de cycles. De cette façon, nous n'avons plus à gérer le moindre marqueur. Par conséquent, nous obtenons une méthode très simple à mettre en place et les performances du solveur ne sont pas impactées.

L'idée consiste à profiter des littéraux d'activation qui sont ajoutés aux clauses des états initiaux et des mauvais états. Lors de l'analyse d'un conflit, le processus d'apprentissage du solveur va obligatoirement ajouter un des littéraux d'activation à la clause apprise si celle-ci est déduite à partir des états initiaux ou des mauvais états. En effet, sans l'affectation des littéraux d'activation, aucun conflit impliquant les états initiaux et les mauvais états ne peut se produire. Ainsi, lorsque le solveur apprend une clause dépendante de ces états non duplicables, elle contiendra forcément au moins un des littéraux d'activation. Le critère de duplicabilité se réduit donc, dans un premier temps, à la présence ou non d'un littéral d'activation dans une clause apprise. En outre, afin de nous libérer des marqueurs de cycle, nous choisissons de nous restreindre à la duplication incrémentale. Cela revient alors à considérer toutes clauses duplicables comme ayant un marqueur minimum l'indice de cycle 0 et comme marqueur maximum l'indice de cycle courant. La duplication s'effectue alors uniquement au niveau du model checker.

Pour résumer, nous proposons donc une approche ne nécessitant qu'une modification infime du solveur. Pour chaque clause apprise, il suffit de vérifier si elle contient un littéral d'activation, si ce n'est pas le cas,

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

```
Fonction ZigZag
    ajouterClauses(I_0 ∨ ¬act_I)
    for i ∈ 0...∞ do
        ajouterClauses(¬P_i ∨ ¬act_Bad_i)
        if solve(act_Bad_i) == UNSAT then
            return 0                          ▷ P est vérifié
        if solve(act_I, act_Bad_i) == SAT then
            return 1                          ▷ Contre-exemple
        ajouterClauses(P_i)
        ajouterClauses(T_{i,i+1})
        dupliquerClauses(i + 1)

Procédure dupliquerClauses(k)
    for all c_l ∈ clauses duplicables do
        ▷ l étant le cycle au cours duquel c est apprise
        c_k ← shiftVariables(c, k − l)
        ajouterClauses(c_k)
```

Algorithme 1: ZigZag avec duplication

elle est duplicable. On sauvegarde alors cette clause en lui associant l'indice du cycle courant auquel elle est apprise. Au niveau du model checker, il est nécessaire d'être en mesure de retrouver pour chaque variable : son cycle et la même variable à un cycle différent. Enfin, entre chaque appel successif au solveur, l'ensemble des clauses sauvegardées est dupliqué comme pour l'exemple suivant : Soit $c_l = \{x_i, y_j\}$ une clause duplicable, l le cycle auquel c a été apprise, et i, j les cycles des variables x, y, et k le cycle auquel on souhaite dupliquer c, on a $i, j \leq l < k$. La version dupliquée de c s'obtient simplement en ajoutant la différence $k - l$ aux indices de cycle des variables : $c_k = \{x_{i+k-l}, y_{j+k-l}\}$. L'algorithme 1 illustre le déroulement de l'approche proposée au niveau du model checker.

La restriction de duplication au niveau du model checker permet de s'émanciper des marqueurs de cycle et fournit également un avantage supplémentaire : elle permet de limiter le nombre de clauses dupliquées, ce qui est le principal inconvénient de la duplication et donc notre objectif premier.

Contraintes de chemin simple à la demande

Comme évoqué précedemment, il est primordial de considérer ces contraintes, si l'on souhaite que la duplication soit adoptée. Cette technique permet l'ajout de nouvelles variables de façon imprévisible, par le fait qu'elles soient générées à la demande. Ces nouvelles variables n'ont donc pas nécessairement d'équivalent dans les autres cycles. La solution que nous avons adoptée pour préserver la satisfaisabilité de la formule consiste à ne pas dupliquer les clauses contenant une variable introduite par ces contraintes. Pour résumer, une clause est considérée comme duplicable si elle ne contient ni une variable introduite par les contraintes de chemin simple, ni un littéral d'activation.

5 Résultats Expérimentaux

Afin de valider expérimentalement notre approche, nous avons implémenté un model checker prenant en entrée des modèles au format standard AIGER [3], et constitué des composants suivants :

- Pré-traitement des modèles AIG fournissant les fonctionnalités habituelles, i.e. propagation de constantes, règles de réécritures (idempotence, contradiction, subsumption, etc.), équivalence structurelle, cône d'influence.
- Elimination des variables à haut-niveau, ce qui permet de désactiver l'élimination des variables au niveau du solveur, afin de ne pas ajouter une clause dupliquée contenant une variable éliminée.
- Algorithme ZigZag (BMC/k-induction)
- Contraintes de chemin simple à la demande
- Duplication des clauses apprises suivant le principe décrit dans la partie 4
- Utilisation du solveur SAT *Glucose* [2] comme procédure de décision

Notre hypothèse est la suivante : la suppression aggressive de clauses et la sélection des clauses à dupliquer permettent de limiter la perte de performance dûe à un trop grand nombre de clauses dupliquées. Afin de la vérifier, nous avons défini un ensemble de stratégies : Ref, O, L, OG5, LG5, O5, L5, tel que :

- Ref correspond à la référence, i.e. sans duplication.
- O et L signifient que les clauses dupliquées sont ajoutées, respectivement, dans la base de données des clauses originales et dans la base des clauses apprises.
- 5 exprime que seules les clauses de tailles inférieures ou égales à 5 sont dupliquées.
- G5 indique qu'en plus des clauses de tailles inférieures ou égales à 5, les glues clauses sont également dupliquées.

A noter que lorsque les clauses sont dupliquées dans la base de données des clauses apprises, nous donnons aux clauses dupliquées le même score LBD que la clause dont elles sont issues, ainsi les glues clauses dupliquées ne sont jamais supprimées.
Nous avons ensuite exécuté toutes ces stratégies sur un ensemble de 513 problèmes (*benchmarks*), provenant de la dernière compétition de model checking hardware (HWMCC 2015), en fixant le temps d'exécution maximal à une heure et la mémoire maximale autori-

	Ref	O	L	OG5	LG5	O5	L5
IND	305	312	308	**300**	**300**	302	302
CEX	108	102	104	**112**	**112**	**112**	110
PRO	92	91	**93**	**93**	**93**	91	**93**
PRF	29	28	31	**34**	31	33	31
TPS	170	255	124	**108**	125	118	114

TABLE 1 – Résultats des différentes stratégies de duplication des clauses apprises sur l'ensemble des instances HWMCC15.

sée à 8Go.

L'ensemble des résultats résumés se trouvent dans le tableau 1. La figure 1 correspond également à ces résultats sous la forme plus classique d'un *cactus plot*, dans lequel nous n'avons pas dessiné toutes les stratégies, par souci de clarté. La ligne *IND* du tableau correspond au nombre de problèmes pour lesquels le résultat reste *indéterminé* après le temps imparti ou lorsque la mémoire nécessaire dépasse le seuil autorisé. *CEX* indique le nombre de modèles pour lesquels notre outil a trouvé un contre-exemple. La ligne *PRO* représente le nombre de modèles pour lesquels la propriété a été prouvée. La ligne *PRF* correspond à la moyenne de la somme des profondeurs atteintes pour les 178 problèmes les plus difficiles, i.e. qui ont atteint la limite de temps sans dépasser une profondeur de 100. Il s'agit là d'une alternative au score utilisé pour la *deep track* lors de la compétition, qui nous paraît plus significative [5]. Enfin, la ligne *TPS* est la moyenne de la somme des temps nécessaires pour résoudre 92 problèmes non triviaux. C'est-à-dire ceux pour lesquels le temps d'exécution est supérieur à 10 secondes et qui ont été résolus par toutes les stratégies, afin de n'en désavantager aucune.

Analyse des résultats

Tout d'abord, on observe une détérioration des performances et du nombre d'instances résolues lorsque la duplication est effectuée pour toutes les clauses. Que celle-ci soit effectuée dans les bases de données des clauses originales ou apprises. Ceci même en restreignant la duplication au niveau du model checker et en n'infligeant aucune pénalité au solveur avec la gestion de la duplication. On peut également noter que la suppression aggressive a tout de même un impact positif lorsque toutes les clauses sont dupliquées.
Ensuite, on remarque que si on limite la duplication aux clauses de taille 5, quelque soit la base de données

5. A noter que l'ordre est semblable avec les scores utilisés pendant la compétition.

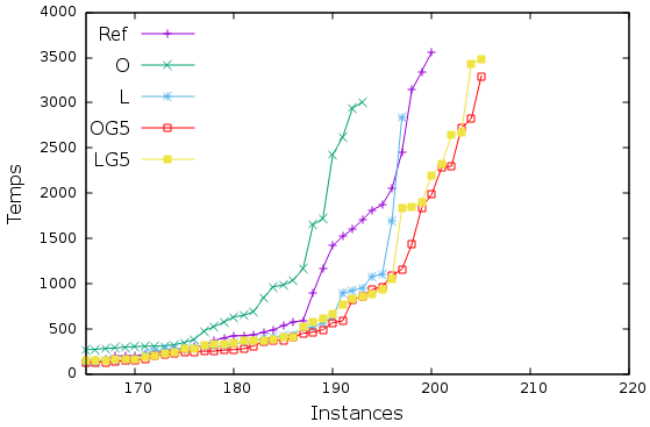

FIGURE 1 – Cactus plot – Nombre d'instances résolues par les différentes stratégies.

dans laquelle on duplique ces clauses, les performances sont meilleures en terme d'instances résolues, de profondeurs moyennes, et de temps moyens.
Enfin, si on ajoute la duplication des glues clauses, le nombre d'instances résolues augmente encore légèrement. De plus, les profondeurs et temps moyens sont encore meilleurs, lorsque l'on effectue la duplication (des glues clauses et des clauses de taille maximale 5) dans la base originale, ce qui est relativement inattendu. Cela signifie probablement que certaines clauses très utiles, ayant un LBD supérieur à 2, sont éliminées lors de la suppression des clauses apprises.
On peut donc en conclure que la meilleure stratégie à adopter consiste à appliquer une sélection drastique des clauses que l'on souhaite dupliquées et de conserver celles-ci indéfiniment, plutôt qu'opter pour une stratégie où plus de clauses sont dupliquées mais où elles risquent d'être supprimées.

6 Conclusion

Dans cet article, nous étendons la duplication pour l'algorithme ZigZag, i.e. une combinaison de BMC et k-induction dans un seul solveur SAT, en considérant les contraintes de chemin simple à la demande qui sont nécessaires pour la complétude de l'algorithme. Nous proposons une solution simple pour la détection des clauses duplicables, très facile à mettre en œuvre. Cette solution a l'avantage de n'avoir aucun impact préjudiciable sur les performances du solveur SAT. Nous montrons également qu'en sélectionnant les clauses à dupliquer à l'aide du score LBD et de leurs tailles, nous sommes capable d'améliorer les performances de notre model checker.
En outre, des améliorations sont encore possibles, notamment : la mise en place d'une politique particu-

lière de suppression des clauses dupliquées, basée par exemple sur leur activité. D'autres critères pourraient également être utilisés pour discriminer les clauses à dupliquer, e.g. SBR (*Size-Bounded Randomized*, [10]). Nous pensons également à étendre la duplication à la méthode d'interpolation ou encore essayer de dupliquer les scores d'activités des variables.

Références

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, volume 9, pages 399–404, 2009.

[2] Gilles Audemard and Laurent Simon. The glucose sat solver, 2013.

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

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

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

[6] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7) :394–397, 1962.

[7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3) :201–215, 1960.

[8] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[9] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4) :543–560, 2003.

[10] Saïd Jabbour, Jerry Lonlac, Lakhdar Sais, and Yakoub Salhi. Revisiting the learned clauses database reduction strategies. *arXiv preprint arXiv :1402.1956*, 2014.

[11] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[12] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.

[13] Ofer Shtrichman. Tuning sat checkers for bounded model checking. In *International Conference on Computer Aided Verification*, pages 480–494. Springer, 2000.

[14] Ofer Shtrichman. Pruning techniques for the sat-based bounded model checking problem. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 58–70. Springer, 2001.

[15] João P Marques Silva and Karem A Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227. IEEE Computer Society, 1997.

[16] Ofer Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1) :5–24, 2004.