Contribution to SAT-based Bounded Model Checking

Anissa Kheireddine

Thèse présentée pour l'obtention du grade de

Docteur de Sorbonne Université

École doctorale EDITE de Paris (ED130)

Informatique, Télécommunication et Éléctronique

Laboratoire de Recherche de l'EPITA (LRE)

Laboratoire d'Informatique de Paris 6 (LIP6)

Contribution to SAT-based Bounded Model Checking

Contribution à la vérification de modèles bornés basée sur la résolution SAT

Anissa Kheireddine

Soutenu le : 19/12/2023, devant le jury composé de :

Rapporteurs :

Vijay GANESH, Professeur des universités, GT, Georgia Institute of Technology
Ahmed BOUNEKKAR, Maître de Conférences, ERIC, Université Claude Bernard Lyon 1

Examinateurs :

Laure PETRUCCI, Professeure des universités, LIPN, Université Sorbonne Paris Nord
Emmanuelle ENCRENAZ, Professeure des universités, LIP6, Sorbonne Université

Sous la supervision de :

Étienne RENAULT, Compilers, Libraries and Tools Team Leader, SiPearl (Co-encadrant)
Souheib BAARIR, Professeur des universités, LRE, EPITA (Directeur de thèse)

Abstract

Keywords: Bounded model checking, Boolean satisfiability, Linear Temporal Logic, parallelism, portfolio, learned clause database, clause sharing

Computer systems have become omnipresent in our daily lives. Ensuring the reliability and robustness of these systems is an absolute necessity. Model-Checking is one of the approaches dedicated to this purpose. Its objective is to either prove the absence of failures or identify potential ones. Model-Checking is declined into several technique. Among these, there is Bounded Model Checking (BMC), a technique that relies on Boolean satisfiability (SAT). The core idea behind BMC is to verify that a model, restricted to executions bounded by some integer k, satisfies its specification, often defined as a set of temporal logic expressions. In this approach, system behaviors are expressed as SAT problems. Unlike other formal verification methods, SAT-based BMC is generally not prone to the state space explosion problem, which can be problematic when dealing with designs involving millions of variables and constraints. However, the trade-off lies in the time complexity, as SAT problems are known to be NP-complete.

Over the past few decades, significant advancements have been made in sequential SAT solving. These developments have mainly focused on utilizing dynamic information, acquired during the solving process (*e.g.*, Learning Binary Clauses), or static information, extracted from the inherent structure of the SAT problem (*e.g.*, community structure). However, less attention has been given to the structural information embedded within the original problem. For instance, when a BMC problem is reduced to SAT, critical information is lost in the translation. As this thesis emphasizes, reintegrating this lost information can greatly enhance the solving process. This work explores ways to improve SAT-based BMC problem-solving, both in sequential and parallel settings, by harnessing and leveraging pertinent information extracted from the problem's inherent characteristics. This may involve improving existing generic heuristics or effectively breaking down the formula into partitions.

Résumé long en français

Context

Les systèmes informatiques, qu'il s'agisse du matériel ou des logiciels, sont devenus une partie fondamentale de notre vie quotidienne. Ils vont des programmes simples qui contrôlent des appareils ménagers tels que les micro-ondes aux logiciels complexes qui gèrent des systèmes critiques, comme les centrales nucléaires, en passant par nos smartphones et nos voitures. Cette omniprésence des systèmes informatiques a suscité une préoccupation majeure : en cas de défaillance de ces systèmes, cela peut entraîner des conséquences graves. Par conséquent, il est essentiel de garantir le bon fonctionnement de ces systèmes critiques. Pour répondre à cette préoccupation, des techniques de vérification formelle ont été développées. Contrairement aux méthodes de test traditionnelles, qui évaluent les performances des systèmes sur un ensemble limité de scénarios, les techniques de vérification formelle sont exhaustives. Elles explorent tous les scénarios possibles, ce qui renforce considérablement la confiance dans la fiabilité de ces systèmes. Par conséquent, la vérification formelle est devenue un outil essentiel dans l'industrie pour éliminer les bugs et renforcer la confiance dans les produits matériels et logiciels.

Le processus de développement des systèmes critiques peut être résumé en trois étapes clés : (i) la modélisation du système, (ii) l'expression des exigences et (iii) la vérification du modèle par rapport à ces exigences. Généralement, le système, ou le *modèle*, est représenté dans un langage formel, tel que le langage SMV [1], VERILOG [2] ou PROMELA [3], décrivant tous les états potentiels du système. Les exigences sont constituées de *propriétés* que le système doit respecter pour garantir son bon fonctionnement. Ces propriétés sont souvent définies à l'aide de logiques temporelles, telles que LTL [4, 5] ou CTL [6], car elles offrent des opérateurs pour exprimer les comportements temporels, ce qui est précieux pour spécifier les exigences. Avec le modèle et un ensemble d'exigences, la conception et la mise en œuvre de haut niveau du système peuvent être vérifiées à l'aide de méthodes formelles. L'une des méthodes de vérification les plus courantes dans l'industrie est le *model checking* (vérification de modèle). Le model checking [7] peut fournir un contre-exemple (CEX) lorsqu'un modèle ne satisfait pas une exigence. Ce CEX correspond à un chemin d'exécution du système, ce qui aide considérablement les concepteurs à comprendre où se situe le problème dans le système. Pour réaliser cette vérification, il est nécessaire d'effectuer une traversée complète de l'espace des états représentant les comportements du modèle. Deux approches ont été utilisées : la vérification de modèle explicite [8] et la vérification de modèle symbolique [9, 10].

Vérification de modèle explicite. Dans la vérification de modèle explicite, le comportement du modèle ainsi que la propriété sont représentés sous forme d'automates : une structure de Kripke pour le modèle [11] et un automate de Büchi pour

la propriété LTL [12, 13]. Ensuite, un produit synchronisé est effectué entre la structure de Kripke et l'automate de la propriété après négation. Si le produit est vide, la propriété est vérifiée ; sinon, elle est violée. Le principal inconvénient de cette technique est le problème de l'explosion de l'espace d'états [14], ce qui signifie que la taille de l'espace d'états du système augmente de manière exponentielle.

Vérification de modèle symbolique. Pour surmonter le problème de l'explosion de l'espace d'états, la vérification de modèle symbolique représente les états de manière implicite en utilisant des fonctions booléennes, telles que les Diagrammes de Décision Binaires (BDD) [10] ou les formules booléennes SAT [15]. Les BDD sont utilisés pour représenter de manière symbolique la relation de transition de l'automate ou des structures de Kripke sous analyse, ainsi que les ensembles d'états manipulés par l'algorithme de vérification de modèle. Depuis leur création, la vérification de modèle symbolique basée sur les BDD a révolutionné la vérification formelle et les méthodes formelles. Elle a permis la vérification pratique de systèmes industriels, commençant par le matériel [16] et s'étendant au logiciel [17].

Dans cette thèse, nous nous concentrons sur les techniques symboliques qui utilisent des procédures de résolution SAT, qui sont devenues un pilier de la vérification de modèle moderne. En effet, les BDD ont ouvert la voie à d'autres formes de vérification de modèle symbolique, principalement les vérificateurs de modèle (model checkers) basés sur SAT [18, 19]. Les solveurs SAT modernes sont devenus la technologie centrale de nombreux model checkers, améliorant considérablement leur capacité par rapport aux model checkers basés sur BDD [20]. En particulier, les procédures SAT sont largement utilisées dans la version *bornée* de la vérification de modèle, notamment pour la vérification des spécifications LTL. Le Bounded Model Checking (BMC) [15, 19, 21] désigne une approche de vérification de modèle où la vérification de la propriété est effectuée à l'aide d'une traversée limitée, c'est-à-dire une traversée d'une représentation symbolique de l'espace d'états qui est bornée par un entier k. Cette approche ne nécessite pas le stockage de l'ensemble de l'espace d'états, ce qui la rend plus souple et utile [18, 22].

Vers une résolution efficace du BMC basée sur le SAT

Ces dernières décennies, de nombreuses améliorations ont été développées dans le domaine de la résolution séquentielle SAT [23–26], pour n'en citer que quelques-unes. Ces approches sont assez génériques et sont basées sur l'exploitation d'informations dynamiques, obtenues à partir de la progression de l'algorithme de résolution lui-même (*e.g.*, LBD [23]), ou d'informations statiques, dérivées de la structure sous-jacente du problème SAT (*e.g.*, structure communautaire [24], symétrie [25, 27, 28]).

Le principal inconvénient de la vérification de modèle symbolique et plus précisément des procédures SAT réside dans le fait qu'un problème BMC est réduit à une formule propositionnelle. Lors de cette transformation, des informations cruciales sont perdues, lesquelles auraient pu être très utiles pour le processus de résolution. L'efficacité des procédures SAT est principalement due aux nombreuses optimisations "génériques" qui ont été développées pour orienter les procédures SAT vers des espaces de recherche prometteurs, réduisant ainsi les temps de résolution. Une

optimisation particulièrement remarquable implique la génération et l'utilisation d'informations de haute qualité (apprentissage) provenant des solveurs SAT, à base d'apprentissage par conflit [29, 30] (Conflict Driven Clause Learning algorithm, CDCL), ce qui permet l'élagage de sous-espaces inutiles. Il est essentiel de s'assurer que ces précieuses informations apprises ne sont pas rejetées prématurément au cours du processus de résolution [29, 31]. Ces informations apprises ont également prouvé leur efficacité dans des contextes parallèles où plusieurs solveurs partagent dynamiquement les informations qu'ils ont apprises. Ainsi, les optimisations proposées dans les solveurs SAT bien connus [32, 33] et les stratégies de résolution SAT parallèles [34, 35] restent génériques et n'exploitent pas les caractéristiques spécifiques du problème en question.

Comme nous le soulignerons dans cette thèse, lorsqu'elles sont réintégrées, ces informations peuvent considérablement améliorer le processus de résolution. Par conséquent, cette thèse se concentre principalement sur l'ajustement fin et l'amélioration des mécanismes d'apprentissage dans le domaine des problèmes BMC basés sur SAT. Plus précisément, nous avons tenté à différents niveaux d'identifier et de générer des informations pertinentes pour accélérer la résolution d'instances BMC, que ce soit dans un environnement séquentiel ou parallèle. Ces contributions participent collectivement à améliorer de manière significative l'efficacité du BMC pour la vérification de propriétés LTL.

Résumé de nos contributions

La littérature a montré un intérêt significatif pour l'amélioration de la vérification de modèle bornée (BMC) en utilisant des procédures SAT (satisfiabilité). Les chercheurs ont réussi à atteindre cet objectif en adaptant les heuristiques internes des solveurs et en parallélisant la résolution du problème en le divisant en sous-parties spécifiques. Toutes ces contributions ont considérablement accéléré le processus de résolution. Cet intérêt accru a conduit au développement d'approches qui prennent en compte des caractéristiques spécifiques des problèmes, telles que les spécificités des variables des formules propositionnelles et la symétrie des formules, des aspects dont les procédures SAT ne tiennent généralement pas compte.

Cette thèse apporte une petite pierre à cet édifice. Elle se concentre sur l'extraction d'informations de haut niveau qui caractérisent les problèmes BMC dans le but d'identifier et/ou de construire des clauses pertinentes, permettant la suppression efficace des sous-espaces inutiles en injectant ces clauses dans le solveur SAT. La thèse propose diverses techniques orthogonales à cette fin pour améliorer les performances des procédures SAT dans l'évaluation des instances BMC, aussi bien dans des contextes séquentiels que parallèles. Trois axes d'exploration principaux ont été suivis, qui sont totalement indépendants. Chacun a contribué à l'identification ou à la génération de clauses apprises pertinentes. Ces travaux peuvent être combinés afin de contribuer à la création d'un solveur spécialisé dans la résolution d'instances BMC.

Contribution I: Identification de clauses apprises pertinentes

Le concept d'informations pertinentes dans les procédures SAT reste vague. De nombreuses techniques existantes gèrent les bases de clauses apprises en utilisant

des métriques génériques (*e.g.*, LBD [23], Activity [26], *etc.*) qui aident à caractériser potentiellement des clauses apprises de haute qualité. Cependant, dans notre étude, qui se concentre spécifiquement sur la résolution de problèmes BMC, nous n'avons pas trouvé de recherche fournissant une mesure significative permettant de caractériser les clauses apprises intéressantes.

Ce travail a donné lieu à une publication à la conférence CP'2021, et son extension a été publiée ultérieurement dans le journal CONSTRAINT. Notre principal objectif porte sur la caractérisation des "clauses apprises" avec l'intuition que : bien que la métrique générique LBD [23], qui a été utilisée par les meilleurs solveurs SAT à ce jour, identifie efficacement les clauses apprises pour la plupart des types de problèmes, elle peut être fortement ajustée avec des informations structurelles dans le contexte de BMC.

L'objectif est de concevoir une nouvelle méthode pour déterminer la pertinence d'un ensemble de clauses, en se basant sur leur valeur LBD et leur appartenance à une catégorie de clauses C_X, une notion que nous avons introduite au Chapitre 3. Cette classification permet de regrouper les clauses en fonction des variables qui les composent. Plus formellement, pour un ensemble de clauses $\mathcal{F}$ et $V(\omega)$ l'ensemble des variables contenu dans la clause ω:

$$C_X = \{\omega \in \mathcal{F}, |, \forall v \in V(\omega), v \in X\}$$

C_X représente les catégories de clauses, où X représente l'un des ensembles suivants : $\mathcal{P}$ qui est l'ensemble des variables qui interviennent dans l'encodage de la propriété en CNF, $\mathcal{M}$ correspond aux variables encodant le système (à l'exclusion des variables $\mathcal{P}$), $\mathcal{J}$ inclut les variables auxiliaires introduites pour compléter la traduction du modèle en CNF, $\mathcal{PJ}$ est un ensemble qui font intervenir les variables de la propriété et les variables auxiliaires, $\mathcal{PM}$ comprend les variables de la propriété et du modèle, $\mathcal{MJ}$ regroupe les variables du modèle et les variables auxiliaires, ou $\mathcal{PMJ}$ qui englobe les variables de la propriété, du modèle et les variables auxiliaires.

Pour ce faire, nous introduisons la notion de **sélecteur**. Elle représente un vecteur spécifiant la valeur LBD appropriée pour chaque catégorie de clauses C_X. Cela permet d'adapter la politique de suppression de clauses en fonction de la catégorie de clause. Par exemple, un sélecteur pourrait indiquer que les clauses apprises de type $C_{\mathcal{P}}$ doivent être protégées jusqu'à une valeur LBD ≤ 8, $C_{\mathcal{J}}$ avec LBD ≤ 4, et ainsi de suite.

En particulier, nos contributions présentée dans le Chapitre 4 sont les suivantes :

- Nous proposons de nouvelles heuristiques pour le calcul de sélecteurs permettant de préserver des clauses apprises intéressantes contre la suppression lors de la résolution séquentielle SAT.

- Nous utilisons ces sélecteurs pour faciliter l'échange d'informations entre différents solveurs CDCL dans le contexte de la résolution SAT parallèle, en utilisant une stratégie de parallélisme basée sur un portefeuille.

- Nous démontrons l'applicabilité de cette étude à n'importe quel solveur SAT basé sur CDCL en menant des expériences avec nos approches sur les deux

principaux solveurs SAT : MAPLECOMSPS [32] et KISSAT-MAB [33].

Nous avons étudié l'impact de la classification des clauses C_X et sur la base de cette étude, nous avons développons deux heuristiques, H_S et H_{LP}, pour calculer des sélecteurs adaptés permettant l'identification de clauses apprises pertinentes à la résolution du BMC.

La partie expérimentale de cette recherche a renforcé l'importance de prendre en compte les variables composant les clauses apprises. Par conséquent, la méthodologie de classification des clauses peut être appliquée à n'importe quel solveur CDCL et est adaptable aux problèmes pouvant partitionner les variables qui constituent le problème.

Par la suite, nous avons exploré l'application de ces sélecteurs dans un contexte de parallélisme, en présentant notre stratégie de partage pour l'échange de clauses apprises pertinentes entre les solveurs CDCL dans une stratégie parallèle basée sur un portefeuille. Notre stratégie de partage basée sur le selecteur H_{LP} a donné lieu à des résultats prometteurs. Tout ce travail est intégré dans notre outil BSALTIC[1].

Contribution II: Exploiter la structure de la formule BMC

Certaines des techniques SAT les plus avancées pour la BMC présentées dans le Chapitre 3 ont montré des résultats prometteurs, qui ont contribué à guider la procédure SAT vers des espaces de recherche intéressants, réduisant ainsi le temps de résolution. Une telle optimisation implique notamment la décomposition de la formule propositionnelle 1.1 en plusieurs sous-formules.

En tirant parti des caractéristiques uniques des instances BMC, nous avons introduit, dans le Chapitre 5, un schéma de partitionnement qui divise la formule en plusieurs sous-parties indépendantes. Ces sous-parties sont structurées de manière à encapsuler des transitions successifs du système. Ensuite, il s'agit d'enrichir le problème avec des informations qui ne sont pas explicitement encodées. Ces informations sont déduites de l'analyse des relations entre les différentes parties. Cela est accompli grâce à l'exploitation des interpolantions de Craig [36].

Pour atteindre nos objectifs, nous revisitons le concept présenté dans [37], qui décrit un schéma de réconciliation employant les procédures d'interpolation. Leur objectif principal était de relever les défis de la résolution de formules extrêmement grandes dans des environnements distribués au travers d'un découpage aléatoire du problème (LZY-D). Nous avons adapté ce schéma de réconciliation à nos besoins. Tout d'abord, nous introduisons une nouvelle méthode de décomposition spécialement adaptée à la résolution de problèmes BMC (BMC-D). Ensuite, nous utilisons le mécanisme d'interpolation comme moyen de générer des clauses apprises, accélérant ainsi la résolution SAT. Cela nous permet de tirer parti des connaissances acquises grâce à la décomposition BMC dans deux dimensions distinctes :

Les interpolants comme moteur de prétraitement : Notre approche exploite l'introduction d'interpolants avant la résolution, en tant que procédure de prétraitement, dans un contexte séquentiel. Cette nouvelle utilisation des interpolants améliore

[1] For a description of our setup, detailed results and code, see `https://akheireddine.github.io/`

l'efficacité du processus de résolution SAT en introduisant des clauses pertinentes déduites de l'interpolation.

Les interpolants dans un environnement parallèle : Pour exploiter davantage la richesse d'informations fournie par les interpolants, nous étendons l'application des interpolants dans des environnements parallèles en partageant ces interpolants pour guider d'autres solveurs CDCL vers des espaces de recherche prometteurs. Cette approche collaborative entre les solveurs améliore leurs capacités de raisonnement collectif et conduit finalement à une résolution de problème plus efficace.

Notre contribution comprend le développement d'un solveur BMC basé sur la décomposition (BMC-D), un composant polyvalent qui fonctionne comme un générateur de clauses utilisant des techniques d'interpolation à la fois dans des environnements séquentiels et parallèles. BMC-D exploite les propriétés structurelles de la formule BMC en la divisant en plusieurs segments, ce qui permet de générer des interpolants hautement pertinents.

Étant donné que le calcul d'interpolation est généralement consommateur de temps, les interpolants obtenus à partir de l'approche de décomposition aléatoire introduite dans [37] (LZY-D) et notre décomposition basée sur BMC (BMC-D) peuvent être utilisés lors du prétraitement pour améliorer l'efficacité de la résolution séquentielle. De plus, on peut partager ces interpolants au sein d'un portefeuille de solveurs CDCL classiques (qui n'ont pas cette vue en partition) pour améliorer la communication et la collaboration efficaces entre eux. Dans les deux cas, ces interpolants, issus d'une approche de partitionnement prenant en compte la structure du problème, ont amélioré les performances du solveur en comparaison avec un partitionnement aléatoire.

Contribution III: Combiner model-checking explicite et symbolique

Le Chapitre 6 introduit une approche alternative pour incorporer des informations structurelles dans les problèmes SAT, sans nécessiter une plongée profonde dans le code du solveur SAT. Ce concept est connu sous le nom de *résolution SAT programmatique* (Programmatic SAT solving). De manière programmatique, il présente un moyen simplifié d'interagir avec le solveur pour le guider pendant le processus de résolution. Cette interaction se produit via une entité externe, qui est généralement spécialisée dans les connaissances spécifiques au domaine du problème à traiter. Dans notre contexte, cette entité est adaptée aux problèmes de model-checking.

Notre contribution principale consiste à introduire une nouvelle approche pour exploiter la représentation automates de la vérification de modèles afin d'extraire les informations perdues lors de l'encodage du problème BMC original en une formule propositionnelle.

Ces informations, lorsqu'elles sont transformées en un ensemble de clauses, peuvent orienter efficacement le solveur SAT vers des sous-espaces de recherche significatifs. Fondamentalement, nous pouvons générer des informations que le solveur SAT n'avait pas connaissance.

Cette étude a été menée dans le but de combiner les deux mondes du model checking : la représentation explicite du système à l'aide de techniques basées sur des automates et la représentation symbolique via la résolution de formules SAT booléennes. Ceci est réalisé grâce au nouveau composant externe (*boîte noire*). Plus précisément, lorsque le solveur SAT l'invoque, en fournissant l'affectation partielle actuelle α, la boîte noire extrait des faits cachés pour le solveur. Ces faits sont dérivés de l'automate de Büchi représentant le produit synchronisé de la propriété évaluée φ et d'une représentation de l'affectation α fournie par le solveur. Les informations extraites sont ensuite encodées en CNF pour être injectées dans le solveur SAT. Le concept de programmatic SAT a été utilisé pour simplifier la mise en œuvre afin d'incorporer des informations structurelles dans les procédures SAT sans nécessiter une plongée profonde dans le code du solveur SAT. Pour ce faire, nous introduisons un composant externe dans l'algorithme CDCL vu comme une boite noire (black-box). Lorsque cette dernière est invoqué par le solveur SAT, recevant l'affectations des variables actuelles du SAT solver, la boite noire va fonctionner de manière similaire que les algorithmes de vérification de vacuité (emptiness checking) qui construisent l'automate du produit synchronisé entre l'exécution fournie par l'affectation du SAT solver, qui représente le modèle, et la propriété à évaluer. Cela repose sur l'énumération des composants fortement connexes (SCC). L'objectif est d'identifier les SCC qui contiennent des cycles acceptants, indiquant si l'execution fournie mène à une violation de la propriété. En conséquence, la boite noire construit des informations sous forme de clauses apprises, qui sont ensuite injectées dans le solveur SAT.

Cependant, l'inconvénient des procédures basées sur les automates réside dans le temps et la consommation de mémoire que l'invocation de la boîte noire peut entraîner. Cela est principalement dû à la surcharge computationnelle de la construction du produit synchronisé, qui peut devenir substantielle à mesure que la complexité du problème augmente. Pour atténuer cela, nous avons tenté de contrôler le flux des appels à la boîte noire. Cependant, cela nécessite encore une paramétrisation plus fine, qui fera l'objet d'investigations futures.

Jusqu'à présent, aucune expérimentation n'a donné des résultats encourageants. Par conséquent, ce chapitre présente la théorie derrière la combinaison des procédures de vérification de modèles explicites et symboliques dans l'objectif d'exploiter les forces des deux mondes dans la résolution des problèmes BMC.

Remerciements

Je tiens tout d'abord à exprimer ma sincère reconnaissance envers Souheib Baarir et Étienne Renault pour leur soutien indéfectible. Leur disponibilité et leur écoute ont rendu l'élaboration de cette thèse plus qu'agréable.

Je souhaite exprimer ma gratitude envers les personnes suivantes :

- Vijay Ganesh et Ahmed Bounekkar d'avoir accepté d'être rapporteurs de ma thèse. Je remercie également Laure Petrucci et Emmanuelle Encrenaz-Tiphène d'avoir accepté d'être membres de mon jury.

- Yann Thierry-Mieg, qui m'a offert l'opportunité d'intégrer le LIP6 en tant que stagiaire en Licence. Sans ce stage, je n'aurais pas eu l'occasion de rencontrer Souheib.

- Mes enseignants de master, particulièrement Pierre Fouilhoux dont sa passion pour la recherche transparaît durant ses cours et à mes encadrants de stage Thibaut Lust et Carola Doerr, dont les conseils et l'encadrement ont été précieux.

- Tous les collègues du LRE et LIP6 : Isabelle Mounier, Daniela Becker, Alexandre Duret-Lutz, Claude Dutheillet, Philipp Schlehuber, Nicolas Boutry, Joseph Chazalon, Jonathan Fabrizio, Thierry Géraud, Elodie Puybareau, Esteban Baptiste, Michaël Roynard, ...

- Une mention spéciale à mes collègues de bureau : Sabrine Saouli, Vincent Vallade, Antoine Martin, Hao Xu et Florian Renkin (finalement, c'était toi qui m'avait ralenti dans le travail !)

- Et à tous ceux que j'ai pu côtoyer de près ou de loin durant ces 4 années.

Enfin, je ne saurais passer sous silence le rôle essentiel de ma famille. Leur soutien a été déterminant dans la réussite de mes études. Merci à vous !

Contents

List of Figures

List of Tables

Introduction

Context

Computer systems, both hardware and software, have become a fundamental part of our daily lives. They range from simple programs that operate household appliances like microwaves to complex software running critical systems, such as nuclear power plants, passing by our smartphones and cars. This ubiquity of computer systems has brought about a significant concern: if these systems fail, it can lead to severe consequences. Hence, ensuring the flawless behavior of these critical systems is of utmost importance. To address this concern, *formal verification* techniques have been developed. Unlike traditional testing methods, which assess the performance of systems under a limited set of scenarios, formal verification techniques are exhaustive. They explore every possible scenario, which significantly boosts confidence in the reliability of these systems. Therefore, formal verification has become an essential tool in the industry for eliminating bugs and enhancing trust in hardware and software products.

The process of developing critical systems can be broadly summarized into three key steps: (i) system modeling, (ii) expressing requirements, and (iii) verifying the model against these requirements. Usually, the system, or *model*, is represented in a formal language, like SMV [1], VERILOG [2], or PROMELA [3], detailing all potential system states. The requirements consist of *properties* that the system must adhere to, ensuring its proper functioning. These properties are often defined using temporal logics, such as LTL [4, 5] or CTL [6], as they offer operators to express temporal behaviors, which are valuable for specifying requirements. With the model and a set of requirements, the high-level design and implementation of the system can be verified using formal methods. One of the most common verification methods in the industry is *model checking*. Model checking [7] can provide a counterexample (CEX) when a model fails to meet a requirement. This CEX corresponds to an execution path of the system, which provides substantial assistance to designers in understanding where the issue lies in the system. To achieve this verification, a complete traversal of the state-space representing the model's behaviors is required. Two approaches have been used: explicit model checking [8] and symbolic model checking [9, 10].

Explicit model-checking. In explicit model checking, the behavior of the model as well as the property are represented as automata: a Kripke structure for the model [11] and a Büchi automaton for the LTL property [12, 13]. Then, a synchronous product is performed between the Kripke and the automaton of the (negated) property. If the product is empty, the property is verified; otherwise, it is violated. The main drawback of this technique is the state-space explosion problem [14], meaning the size of the system state-space grows exponentially.

Symbolic model-checking. To overcome the state-space explosion issue, symbolic

model checking represents states implicitly using Boolean functions, such as Binary Decision Diagrams (BDD) [10] or Boolean SAT formula [15]. BDDs are used to symbolically represent the transition relation of the automaton or Kripke structures under analysis, and sets of states manipulated by the model checking algorithm. Since its inception, BDD-based symbolic model checking has revolutionized formal verification and formal methods. It has enabled practical verification of industrial systems beginning with hardware [16] and extending to software [17].

In this thesis, we focus on symbolic techniques that use SAT solving procedures, which have become a cornerstone of modern model checking. In fact, BDDs have paved the way for other forms of symbolic model checking, primarily SAT-based model checkers [18, 19]. Modern SATisfiability (SAT) solvers have since become the core technology in many model checkers, greatly improving their capacity compared to BDD-based model checkers [20]. In particular, SAT procedures are extensively applied in the *bounded* version of model checking, specifically for verifying LTL specifications. Bounded model checking (BMC) [15, 19, 21] refers to a model checking approach where the verification of the property is performed using a bounded traversal, meaning a traversal of a symbolic representation of the state-space that is bounded by an integer k. Such an approach does not require storing the entire state-space and is thus found more scalable and useful [18, 22].

Towards an efficient SAT-based BMC solving

These last decades, many improvements have been developed in the field of sequential SAT solving [23–26], to name just a few. These approaches are quite generic and are based on exploiting either dynamic information, obtained from the progress of the solving algorithm itself (e.g., LBD [23]), or static information, derived from the underlying structure of the SAT problem (e.g., community structure [24], symmetry [25, 27, 28]).

The major drawback of symbolic model checking and more precisely SAT procedures arises when a BMC problem is reduced to a propositional SAT formula. In this transformation, crucial information is lost, which could have been very useful for the resolution process. The efficiency of SAT procedures is primarily due to the numerous "*generic*" optimizations that have been developed to guide SAT procedures towards promising search spaces, ultimately reducing solving times. One particularly noteworthy optimization involves the generation and utilization of high-quality (learnt) information from Conflict-Driven Clause Learning SAT solvers [29, 30], which enables the pruning of unuseful subspaces. It's essential to ensure that these valuable learnt information are not prematurely discarded during the solving process [29, 31]. These learnt information have also proven their effectiveness in parallel contexts where multiple solvers dynamically share the information they have learnt. Hence, the optimizations provided in well-known SAT solvers [32, 33] and parallel SAT solving strategies [34, 35] remain generic and do not exploit the specific characteristics of the problem at hand.

As we will emphasize in this thesis, when reintegrated, this information can significantly boost the solving process. Therefore, this thesis primarily focuses on fine-tuning and enhancing the learning mechanisms within the domain of SAT-based

BMC problems. More specifically, we've attempted at different levels to identify and generate relevant information to speed up the solving of BMC instances, whether in a sequential or parallel environment. These contributions collectively contribute to significantly improving in the efficiency and effectiveness of BMC for verifying LTL properties.

Contribution I: Relevant learnt clauses identification

The concept of relevant information in SAT procedures remains ambiguous. Several existing techniques handle learnt clause databases using generic metrics (*e.g.*, LBD [23], Activity [26], *etc.*) that help characterize potentially high-quality learnt clauses.

Our primary focus lies in the characterization of "learnt clauses" with the intuition that although the generic LBD metric [23], used by the best SAT solvers to date, efficiently characterizes learnt clauses for most problem instances, it can be significantly adjusted with structural information in the context of BMC. The goal is to develop a new method to assess the relevance of a set of clauses, based on their LBD value and the new metric we proposed that enables to classify the clauses according to the meaning of their variables.

The experimental part of this research has emphasized the importance of considering variables composing the learnt clauses. Therefore, the clause classification methodology can be applied to any CDCL solver and is adaptable to problems that can partition the problem variables. Subsequently, we explored the application of this clause classification in a parallel context by adapting the sharing policy for exchanging relevant learnt clauses among CDCL solvers in a portfolio-based parallel strategy. Our sharing strategy based on the H_{LP} selector has yielded promising results.

Contribution II: Exploiting the BMC formula structure

Some of the most advanced SAT techniques for BMC involve decomposing the propositional formula 1.1 into multiple sub-formulas. By leveraging the unique characteristics of BMC instances, in Chapter 5, we introduced a partitioning scheme that divides the formula into several independent sub-parts. These parts are structured to encapsulate successive transitions of the system. The next step involves enriching the problem with information not explicitly encoded. This information is deduced from analyzing the relationships between the different parts. This is accomplished through the application of Craig interpolants [36].

We introduced a new decomposition method tailored specifically for solving BMC problems (BMC-D). Then, we used the interpolation mechanism as a means to generate learnt clauses, thereby speeding up SAT resolution. This allowed us to leverage the insights gained from BMC decomposition during the pre-processing phase but also in a parallel context by sharing these interpolants to guide other CDCL solvers toward promising search spaces.

Contribution III: Combine explicit and symbolic model-checking

We investigate in Chapter 6 a new way of generating new constraints from the unique structure of the problem. This approach aims to efficiently resolve BMC by reintegrating high-level information into the SAT solver. To achieve this, we developed a new entity specifically designed to manipulate the automata representation of the problem, to extract the information lost during the encoding of the BMC problem into a Boolean formula.

To achieve this, we utilized the concept of SAT-based programming, allowing a non-expert user with no prior knowledge of SAT algorithms to easily introduce domain-specific information. The modifications are limited to a function call that involves the external entity en question. In the context of BMC, we specialized this entity to generate facts translated into clauses derived from the explicit problem representation. These clauses either aid in terminating the search in a dead-end subspace or complete the path to reach a state that invalidates the property. Fundamentally, we can generate information that the SAT solver was previously unaware of.

This contribution aims to merge the two worlds of model-checking: explicit and symbolic, by reintegrating the structural information of the problem into symbolic techniques.

Manuscript Structure

The structure of this manuscript is organized into six chapters, with the first two serving as an in-depth description and explanation of model checking explicit and symbolic procedures, as well as the SAT solving procedure in both sequential and parallel settings. The third chapter reviews prior work on SAT-based BMC and highlights some features of BMC formulas that will be explored in the remainder of the thesis.

Model Checking. Chapter 1 offers a detailed explanation of both explicit representations of the model using automata-based verification and implicit representations through SAT procedures.

SATisfiability solving. Chapter 2 provides a comprehensive overview of SAT solving, encompassing the historical DPLL algorithm and the nowadays used CDCL algorithm. It also delves into various heuristics and structures embedded in modern efficient SAT sequential solvers. Furthermore, it discusses well-known parallelization techniques for solving SAT problems that can be found in the literature.

SAT-Based BMC - Positioning, Analysis and Benchmarking. Chapter 3 surveys the landscape of SAT-based BMC solving in both sequential and parallel environments. It conducts an in-depth analysis of BMC characteristics, which will serve as the foundation for the subsequent chapters that propose BMC-based heuristics.

Tuning the learnt clause databases. Chapter 4 introduces a methodology for developing SAT heuristics tailored for BMC problems. These optimizations are

achieved by exploiting the structure of BMC problems in the sequential context, which involves characterizing the variables encoding the BMC problem as a Boolean SAT formula. Whereas in the parallel context, a heuristic tailored for solving BMC is presented, specifying the information to be communicated between different solvers. These parallel heuristics are then combined with the sequential ones.

Decomposition-based BMC. Chapter 5 introduces a decomposition-based BMC formula method that utilizes interpolation techniques to reconcile subparts of the whole problem. This technique partitions the BMC formula into segments, facilitating the generation of highly relevant interpolants used in both sequential and parallel environments.

Programmatic SAT-based BMC Chapter 6 gives an introduction to programmatic SAT solving within the context of BMC. It offers an alternative approach for incorporating structural information into SAT problems without requiring an extensive understanding of the SAT solver's code. This structural information takes the form of learnt constraints generated by an external entity (designed by us). The entity utilizes explicit model checking procedures through automaton-based representation to extract hidden information from the SAT solver.

The last chapter 7 concludes this manuscript and discusses various short-term and long-term directions for future research.

Chapter 1

Model checking

Contents

In this chapter, we introduce some necessary definitions for understanding this thesis. We review both explicit [8] and symbolic [9] model checking. We begin by defining the notion of transition system 1.1 that serves to encode the system at hand into a Kripke structure [11]. We review the semantics and the categories of Linear Temporal Logics properties 1.2, and then in Section 1.3, we provide a detailed explanation of the explicit representation of model checking problems using automata-based verification procedures. Section 1.4 presents the bounded version of model checking and Boolean satisfiability problems before delving into the symbolic representation of model checking using SAT-based verification approaches in Section 1.4.2.

1.1 Transition system

A system is a collection of interacting components designed to perform a specific *task*. It consists of variables, typically *finite* in number, that have significance in the execution of the task. These variables change their states, usually within a finite set of values, as the system operates.

To verify the validity of a property, which means checking whether the property is satisfied at every state of the system, it is necessary to build a model of the system.

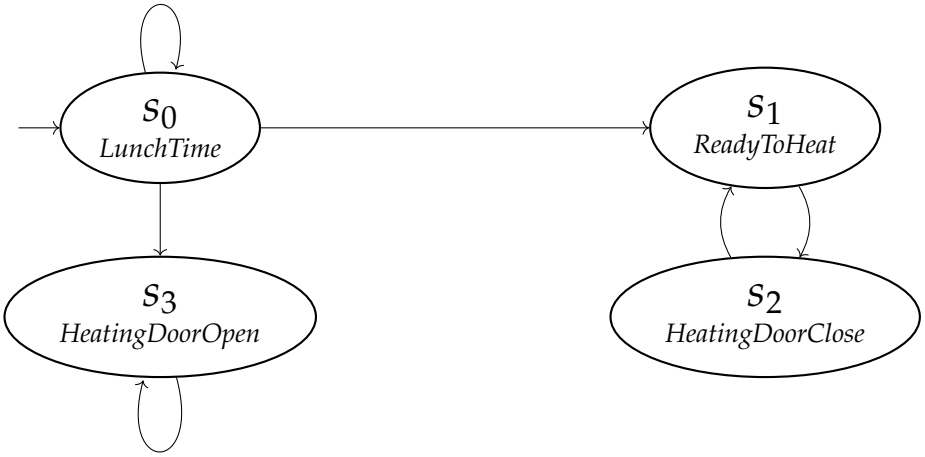


FIGURE 1.1: Transition system $\mathcal{TS}_1$ of Example **1.1**

This can be accomplished using a *transition system* formalism ($\mathcal{TS}$). Transition systems are directed graphs whose vertices symbolically represent the internal states of a system, and whose edges indicate the way in which states can evolve. The states from which such a system can start are referred to as *initial* states (*e.g.*, the initial state of a program when its variables are initialized).

Definition 1.1 (Transition System). A transition system is defined as a triplet $\mathcal{TS} = \langle S, T, I \rangle$ where:

- S is a set of elements called states,
- $T \subseteq S \times S$ is a transition relation,
- $I \subseteq S$ represents the set of initial states.

Such a system $\mathcal{TS}$ is said to be finite if its set of states S is finite.

Example 1.1. Let's consider the following finite transition system $\mathcal{TS}_1 = \langle S, T, I \rangle$ that illustrates a microwave-like system:

- $S = \{\overbrace{LunchTime}^{s_0}, \overbrace{ReadyToHeat}^{s_1}, \overbrace{HeatingDoorClose}^{s_2}, \overbrace{HeatingDoorOpen}^{s_3}\}$,
- $I = \{s_0\}$,
- $T = \{(s_0, s_0), (s_0, s_1), (s_0, s_3), (s_1, s_2), (s_2, s_1), (s_3, s_3)\}$.

and its graphical representation is shown in Figure 1.1

1.1.1 Paths & executions

Consider the transition system $\mathcal{TS} = \langle S, T, I \rangle$. A *finite path* is a finite sequence of states $s_0 s_1 s_2 \dots s_{n-1} s_n$ for $n \in \mathbb{N}$, such that $((s_0, s_1)(s_1, s_2) \dots (s_{n-1}, s_n)) \in T$. Similarly, an *infinite path* is an infinite sequence of states $s_0 s_1 s_2 \dots$ such that $T(s_0, s_1) T(s_1, s_2) \dots$. We say that a path is *initial* if $s_0 \in I$; and it is *maximal* if it is infinite, or if it is finite

and its last state s_n is terminal, *i.e.* it has no successors (no existing path leaving from that state). An *execution* of $\mathcal{TS}$ is an initial and maximal path.

Example 1.2. Let's reconsider the transition system $\mathcal{TS}_1$ of Example **1.1** and displayed in Figure 1.1. The sequence $\rho := s_1 s_2$ is a finite path of $\mathcal{TS}_1$ and the sequence $\rho' := s_0 s_1 s_2 s_1 s_2 \dots$ is an infinite path of $\mathcal{TS}_1$. The path ρ' is an execution since it is initial and infinite (infinite repetition of the sequence $s_1 s_2$). The path ρ is not initial as s_1 is not initial. Neither is ρ maximal since s_2 is not a terminal state, *i.e.*, it has an immediate successor which is s_1. In fact, $\mathcal{TS}_1$ has no terminal state.

1.1.2 Labelled transitions and Kripke Structures

Transition systems can be used to describe the behaviors of a system expressed in a formal language (*e.g.*, SMV [1], VERILOG [2], AIG [38], *etc.*). This formalism extends to Labelled transitions systems (LTS), which provide additional information to reason about the *properties* of these behaviors. In an LTS, each transition is labelled with an *action* (proposition).

Definition 1.2 (Labelled Transition System). An LTS is a system labelled on transitions, $G = \langle S, T, I, Act \rangle$, is defined as follows:

- S is the set of finite states,
- $I \subseteq S$ is the set of initial states,
- Act is the set of *actions*, and
- $T \subseteq S \times Act \times S$ represents the transition relation, given in the form of triplets (origin state, action, destination state), which are referred to as *transitions*. if $(s, \alpha, d) \in T$, we write $s \xrightarrow{\alpha} d$.

Atomic proposition & Languages

A proposition is a combination of *atomic propositions*, that capture a portion of a system's state at a given moment and can take on the values *true* ($\top$) or *false* ($\bot$). The non-empty and finite set of atomic proposition variables is denoted as AP, where 2^{AP} represents the set of subsets of AP. For instance, if $AP = \{a, b\}$, then $2^{AP} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. Each variable in the model can be converted into a set of atomic propositions. For example, an integer variable can simply be represented by a set of atomic propositions, each representing an individual bit.

The set of atomic propositions 2^{AP} can be seen as the *letters* (or *symbols*) of a finite set called an *alphabet*, denoted by Σ. Thus, a word over Σ is a finite (or infinite) sequence of symbols from Σ and can be defined by the function $\sigma : \mathbb{N} \to \Sigma$. A word of size 0 is denoted by ϵ and is called the *empty word*; an infinite word is denoted by ω-word and has a size ω where $\omega \notin \mathbb{N}$ is the infinite ordinal. This gives the following sets: $\Sigma^\star$ represents the set of finite words over Σ, Σ^ω the set of ω-words, and $\Sigma^+ = \Sigma^\star \setminus \epsilon$ the set of non-empty finite words.

The ω-words are expressed using the *ω-regular expressions*, an extension of *regular expressions*. The syntax of ω-regular expressions over the alphabet Σ is defined by

this grammar, where $p \in \Sigma$:

$$s ::= r^{\omega} \mid (r \cdot s) \mid (s + s) \qquad (\omega\text{-regular expr.})$$
$$r ::= r^{\star} \mid (r \cdot s) \mid (r + r) \mid p \mid \epsilon \qquad (\text{regular expr.})$$

Thus, the operator r^{ω} is a variant of the operator $r^{\star}$ where, instead of concatenating words a finite number of times, an infinite number of concatenations are performed.

For all finite words $u \in \Sigma^{\star}$ and $v \in \Sigma^{+}$, we denote by uv^{ω} the infinite word $uvvv\ldots$, which means that u is followed by an infinite repetition of v. For all $\sigma \in \Sigma^{\omega}$ and $i \in \mathbb{N}$, we define:

$$\sigma[i\ldots] := \sigma(i)\sigma(i+1)\ldots$$
$$\sigma[\ldots i] := \ldots\sigma(i-1)\sigma(i)$$

In other words, $\sigma[i\ldots]$ (resp. $\sigma[\ldots i]$) is the infinite *suffix* (resp. *prefix*) of σ obtained by starting (resp. ending) at index i. For example, the word ab^{ω} is formally the function σ such that $\sigma(0) = a$ and $\sigma[1\ldots] = bbbbbb\ldots$, meaning that $\sigma(i) = b$ for all $i > 0$

Thus, a *finite language* in $\Sigma^{\star}$, is a finite set of words over Σ; an ω-language $\mathscr{L}$ is a finite set of infinite words, *i.e.*, $\mathscr{L} \subseteq \Sigma^{\omega}$. Since we are interested in ω-languages, we will use the term "language" to refer to ω-languages and specify when discussing finite word languages.

System representation

Kripke structures [11] allow the representation of the state space induced by a model. In this structure, each state is labelled with a set of propositions. Transitions indicate state changes and symbolize the model's evolution. Thus, a Kripke structure differs from an LTS only in the labeling of states with propositions and not in the transitions.

Definition 1.3 (Kripke Structure). A Kripke structure is defined as $K = \langle S, T, I, Act, AP, \mathrm{L}\rangle$ with:

- $\langle S, T, I, Act\rangle$ is an LTS with $Act = \emptyset$,
- AP is a set of atomic propositions,
- $\mathrm{L} : S \to 2^{AP}$ is the labeling function.

The function L associates a subset of AP with each state. The propositions described by the subset $\mathrm{L}(s)$ are considered satisfied in state $s \in S$, whereas those of $AP \setminus \mathrm{L}(s)$ are said to be unsatisfied in state s.

Example 1.3. Reconsider the previous Example **1.1** and let complete the $\mathcal{TS}_1$ with atomic propositions that represent its Kripke structure representation. Suppose the atomic propositions are $AP = \{placed, closed, started\}$ correspond respectively to the action of **placing** the meal in the microwave's plate, the door of the microwave is **closed** and the last means that the microwave **started** the heating process.

Figure 1.2 depicts the Kripke structure of the microwave-like system example. At the top or bottom of each state is a label that represents its valuation of the atomic

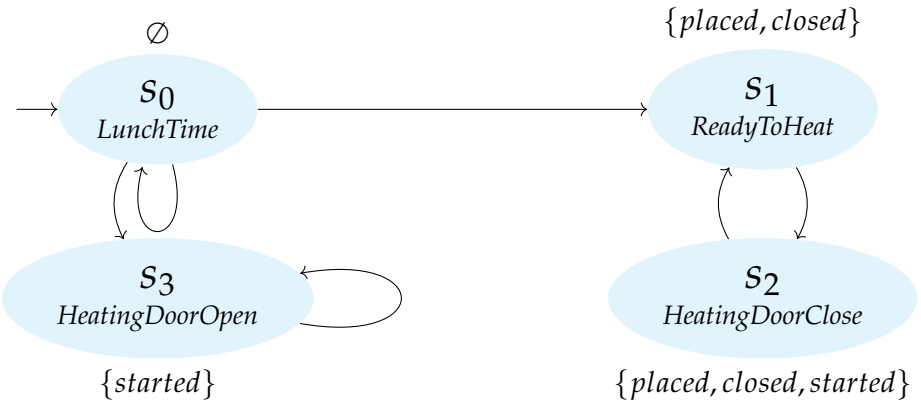


FIGURE 1.2: Kripke structure of microwave-like of Example **1.3** with $AP = \{placed, closed, started\}$

propositions. Thus, $L(s_0) = \emptyset$, signifying that in the initial state of this system, the microwave is not in use as there is no meal placed, and its door is open.

We may wonder if *the microwave is always launched with a closed door*. Some possible executions would be:

$$\rho := s_0\ s_1\ s_2\ s_1\ s_2 \ldots$$
$$\rho' := s_0\ s_0\ s_0\ s_3\ s_3 \ldots$$

Only ρ' results in an undesirable situation where the microwave is launched with the door open.

The verification of an undesirable situation in this system example has been briefly demonstrated. In the following sections, we will delve into the formalization of such properties using temporal logic, as opposed to natural language, in order to eliminate ambiguity and facilitate automated verification.

1.2 Linear Temporal Logic

The correctness of a system depends on checking whether its execution validate given properties. To verify formally if such properties are satisfied, they must be modeled formally, rather than in natural language, for two main reason: to avoid ambiguity, and to provide a structure that can be manipulated by an algorithm. Logic is the appropriate way to achieve this. Propositional logic is not sufficient to model interesting system properties since it lacks the ability to reason about behaviors and has no concept of *discrete* time. For example, propositional logic cannot express statements like *"every time a process wants to enter the critical section, it will get there eventually"*.This limitation is addressed by temporal logic; a logic that extends propositional logic with operators allowing reason about time.

In this thesis, we focus on Linear-time Temporal Logic (LTL) since it expresses many properties of interest and has been extensively studied [39–41]. For instance, Manna & Pnueli [39] established a full hierarchy of specifications that can be expressed using LTL. It has also been used to optimize the performance of certain model checking problems [42–44]. Since this thesis is oriented towards symbolic approaches

to bounded model checking using SAT procedures, some temporal logics, such as Computation Tree Logic (CTL or CTL*) [6], which provide more expressiveness than LTL, cannot be directly applied in a bounded context of model checking. The work presented in this thesis focuses exclusively on LTL logics. Nevertheless, it's noteworthy that McMillan [45] introduced a SAT-based *unbounded* CTL model checker, which opens the possibility of extending the scope of the work presented in this thesis to other logics beyond LTL.

1.2.1 Semantic

LTL logic was introduced by Amir Pnueli [4] to specify the behavior of systems over time. Based on propositional logic, LTL logic is defined over a set of atomic propositions AP, expressing infinite sequences. For any infinite word $\sigma \in \Sigma^\omega$, $\sigma \models \varphi$ denotes that *the word σ satisfies the formula φ*. This relationship is defined inductively as follows:

$$
\begin{array}{lcl}
\sigma \models \mathit{true} & & \\
\sigma \models p & \iff & p \in \sigma(0) \\
\sigma \models \varphi_1 \wedge \varphi_2 & \iff & \sigma \models \varphi_1 \wedge \sigma \models \varphi_2 \\
\sigma \models \neg\varphi & \iff & \sigma \not\models \varphi \\
\sigma \models \mathbf{X}\varphi & \iff & \sigma[1\ldots] \models \varphi \\
\sigma \models \varphi_1 \,\mathbf{U}\, \varphi_2 & \iff & \exists j,\ \sigma[j\ldots] \models \varphi_2, \text{and } \forall i \in \mathbb{N},\ 1 \leq i < j,\ \sigma(i) \models \varphi_1
\end{array}
$$

In addition, LTL encompasses four essential temporal operators: **F** (**F***inally something happens*), **G** (**G***lobally something holds*), **R** (***R**elease*) and **W** (***W**eak until*):

$$
\begin{array}{lcl}
\sigma \models \mathbf{F}\varphi & \iff & \exists j \geq 0,\ \sigma[j\ldots] \models \varphi \\
\sigma \models \mathbf{G}\varphi & \iff & \forall j \geq 0,\ \sigma[j\ldots] \models \varphi \\
\sigma \models \varphi_1 \,\mathbf{R}\, \varphi_2 & \iff & \sigma \models \neg(\neg\varphi_1 \,\mathbf{U}\, \neg\varphi_2) \\
\sigma \models \varphi_1 \,\mathbf{W}\, \varphi_2 & \iff & \sigma \models \varphi_2 \,\mathbf{R}\, (\varphi_1 \vee \varphi_2)
\end{array}
$$

The first operator **F** ensures that the system will "eventually" reach a state where the specified formula is true. The second operator **G** checks whether a property is "globally" true for all states in the execution. The third operator **R** ensures that a formula will continuously hold up to a certain point and must satisfy another formula until that point. Finally, the last operator **W** guarantees that a formula is satisfied until another formula is satisfied. For example, one can specify some atomic proposition: the variable a is equal to 42 ($"a = 42" \in AP$) that must hold at every point in time ($\mathbf{G}\ "a = 42"$). It means that the execution must always verify that $"a = 42"$ holds starting from the first time step. The atomic proposition $"a = 42"$ must eventually hold at some future point in time ($\mathbf{F}\ (a = 42)$). This translates to an execution where $"a = 42"$ will hold in future steps. Figure 1.3 illustrates the semantics of some basic LTL formulas.

Definition 1.4 (LTL languange). The language $\mathscr{L}_\varphi$ of an LTL formula φ over the set of atomic propositions AP is defined as:

$$\mathscr{L}_\varphi = \{v \in \Sigma^\omega \mid v \in [\![\varphi]\!]\}$$

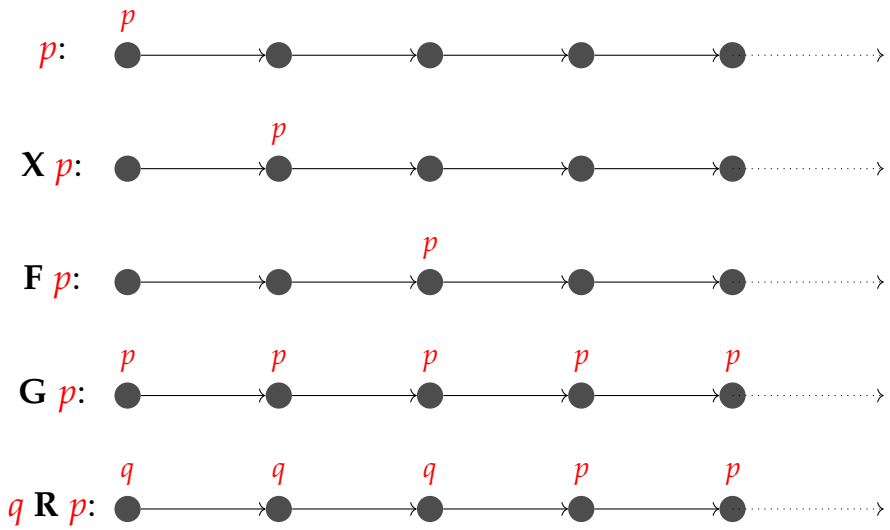


FIGURE 1.3: LTL semantic scheme. Each line represents an infinite word σ and each circle is σ's positions and the label on top of the i^{th} circle specifies the propositions $\sigma(i)$

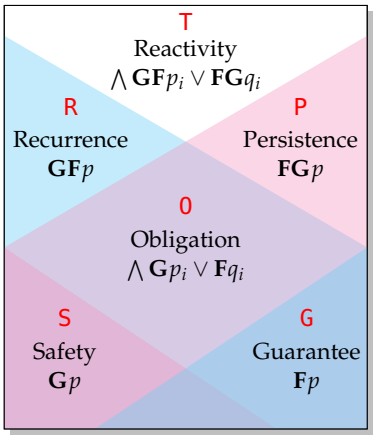


FIGURE 1.4: Hierarchy of Manna & Pnueli S ∪ G ⊆ O ⊆ R ∪ P ⊆ T

with $[\![\varphi]\!] := \{\sigma \in \Sigma^\omega \ : \ \sigma \models \varphi\}$ the set of words that satisfy the LTL formula φ on AP.

Thus, a system satisfies a property if and only if its language is included in that of the property, *i.e.*, if we denote K, the Kripke structure modeling the system, then:

$$\mathscr{L}_K \subseteq \mathscr{L}_\varphi$$

1.2.2 LTL properties classification

Lamport [46] classified the properties expressed by LTL into two classes: safety (*some bad thing never happens*) that uses the **G** operator and liveness (*some good thing eventually happens*) with the **F** operator. More precisely, a liveness property does not forbid any execution prefix, whereas a safety property forbids certain prefixes and allows the expression of *invariants* on the system. Based on the combination of the **F** and

G operators, Manna & Pnueli [39] further refined Lamport's classification into six categories:

- *Safety* (S): similar to the one described in the Lamport [46] classification. It consists of all infinite words σ such that *every* prefix of σ is in $[\![\varphi]\!]$.

$$[\![\mathbf{G}p]\!] = \underbrace{(p + (p \cdot q))^{\omega}}_{\text{occurrences of } p} \text{ with } p, q \in AP$$

- *Guarantee* (G): some good thing happens at least once in the future. It consists of all words σ such that *some* prefix of σ is in $[\![\varphi]\!]$.

$$[\![\mathbf{F}p]\!] = \underbrace{(\varnothing + q)^{\star}}_{\text{no occurrences of } p} \underbrace{(p + (p \cdot q))^{\omega}}_{\text{occurrences of } p} \text{ with } p, q \in AP$$

- *Obligation* (O): combines safety and guarantee properties. This enforces more restrictions on the sequences, leading to some good things.

$$[\![\mathbf{G}p \wedge \mathbf{F}q]\!] = [\![\mathbf{G}p]\!] \wedge [\![\mathbf{F}q]\!] = \underbrace{(p)^{\star}(p \cdot q)(p + (p \cdot q))^{\omega}}_{\text{occurrences of } p}$$

- *Persistence* (P): at some point, a good thing will happen and hold forever. It consists of all words σ such that *all but finitely many* prefixes of σ is in $[\![\varphi]\!]$.

$$[\![\mathbf{FG}p]\!] = \underbrace{(\varnothing + p + q)^{\star}}_{\text{possible occurrences of } p} \underbrace{(p + (p \cdot q))^{\omega}}_{\text{occurrences of } p} \text{ with } p, q \in AP$$

- *Recurrence* (R): some good things will appear infinitely often. It consists of all words σ such that *infinitely many* prefixes of σ is in $[\![\varphi]\!]$.

$$[\![\mathbf{GF}p]\!] = \underbrace{((\varnothing + q)^{\star}}_{\text{no occurrences of } p} \underbrace{(p + (p \cdot q)))^{\omega}}_{\text{occurrences of } p} \text{ with } p, q \in AP$$

- *Reactivity* (T): combines recurrence and persistence properties. This enforces more restrictions on the sequences between good things.

$$[\![\mathbf{FG}p \wedge \mathbf{GF}q]\!] = [\![\mathbf{FG}p]\!] \wedge [\![\mathbf{GF}q]\!] = \underbrace{(\varnothing + p + q)^{\star}}_{\text{possible occurrences of } p} \underbrace{(p \cdot q)(p + (p \cdot q))^{\omega}}_{\text{occurrences of } p}$$

Figure 1.4 illustrates the relationship between LTL formulas and the classes in the hierarchy proposed by Manna & Pnueli [39]. Each category of property has inherent characteristics that can be leveraged to optimize the performance of model checkers.

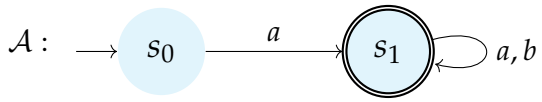


FIGURE 1.5: Büchi automaton $\mathcal{A}_1$ accepts the language $a(a+b)^\omega$.

1.3 Automata-based procedure for LTL verification

In this section, we will review the verification procedure of LTL specifications on Kripke structures. This procedure relies on the construction and manipulation of Büchi automata. The first subsections will introduce the Büchi structure and explain how specifications and systems are transformed into automata. The last subsection will detail the verification procedure [47], which requires these early transformations to determine if a system satisfies a given specification.

1.3.1 Büchi automata

A Büchi automaton [12, 13] is an LTS that represents a finite automaton operating on ω-words. A finite LTS accepts a finite word σ if it reaches an accepting state. The languages recognized by finite automata precisely correspond to the languages described by regular expressions. However, this concept doesn't apply in the context of an infinite word since there is no end to the word. Instead, a Büchi automaton accepts an infinite word σ expressed in ω-regular expressions if, while reading σ, it visits an accepting state infinitely often.

Definition 1.5 (Büchi Automata). A Büchi automaton is defined by a quintuplet $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$, where:

— Q is a finite set of states,

— Σ is a finite alphabet,

— $\delta : Q \times \Sigma \to 2^Q$ is the transition function,

— $I \subseteq Q$ is the set of initial states,

— $F \subseteq Q$ is the set of accepting states.

Since it operates on ω-words, a Büchi automaton is an ω-automaton. Unlike Kripke structures, the states in Büchi automata are not labelled; only the transitions are labelled ($Action \neq \emptyset$).

Example 1.4. Consider the Büchi automaton $\mathcal{A}_1$ shown in Figure 1.5.

- $Q = \{s_0, s_1\}$,
- $\Sigma = \{a, b\}$,
- $\delta(s_0, a) = \{s_1\}, \delta(s_1, a) = \{s_1\}, \delta(s_1, b) = \{s_1\}$,
- $I = \{s_0\}$,
- $F = \{s_1\}$.

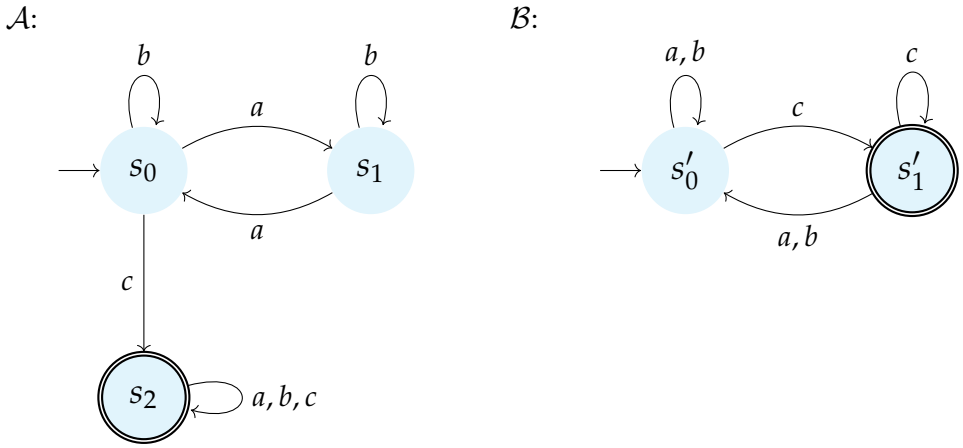


FIGURE 1.6: Buchi automata $\mathcal{A}$ and $\mathcal{B}$ of Example **1.6**

The automaton $\mathcal{A}_1$ begins in the initial state s_0 reads the letter a, then loops forever on the accepting state s_1 over an arbitrary letter. This automaton accepts words beginning with a.

Automata language

We write $s_i \xrightarrow{a} s_j$ to denote that $s_j \in \delta(s_i, a)$, and thus indicates that the automaton has a transition from s_i to s_j labelled by the letter a. An infinite word $\sigma \in \Sigma^\omega$ is accepted by a Büchi automaton $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$ if there exists an (infinite) sequence of states $s_0, s_1, \cdots \in Q$ such that:

- $s_0 \xrightarrow{\sigma(0)} s_1 \xrightarrow{\sigma(1)} \dots s_i \xrightarrow{\sigma(\dots)} \dots$,
- $s_0 \in I$,
- $s_i \in F$ for an infinity of $i \in \mathbb{N}$.

That is: an infinite word σ is accepted by $\mathcal{A}$ if it is possible to read its letters starting from an initial state then passing by a series of transitions that visit infinitely often accepting states.

Definition 1.6 (Büchi Automata Language). The Büchi automaton language $\mathcal{A}$, denoted by $\mathscr{L}_\mathcal{A}$, is the set of infinite words it accepts:

$$\mathscr{L}_\mathcal{A} := \{\sigma \in \Sigma^\omega \ : \ \sigma \text{ is accepted by } \mathcal{A}\}.$$

Example 1.5. Back to automaton $\mathcal{A}$ of previous example depicted in Figure 1.5. We say that $\mathcal{A}$ accepts the word $abab\dots$ since it visits infinitely often the state s_1 which is an accepting state ($s_1 \in F$). On the other hand, the word $ba\dots$ is not accepted by $\mathcal{A}$. So, the language of $\mathcal{A}$ is equivalent to:

$$\mathscr{L}_\mathcal{A} = a(a+b)^\omega$$

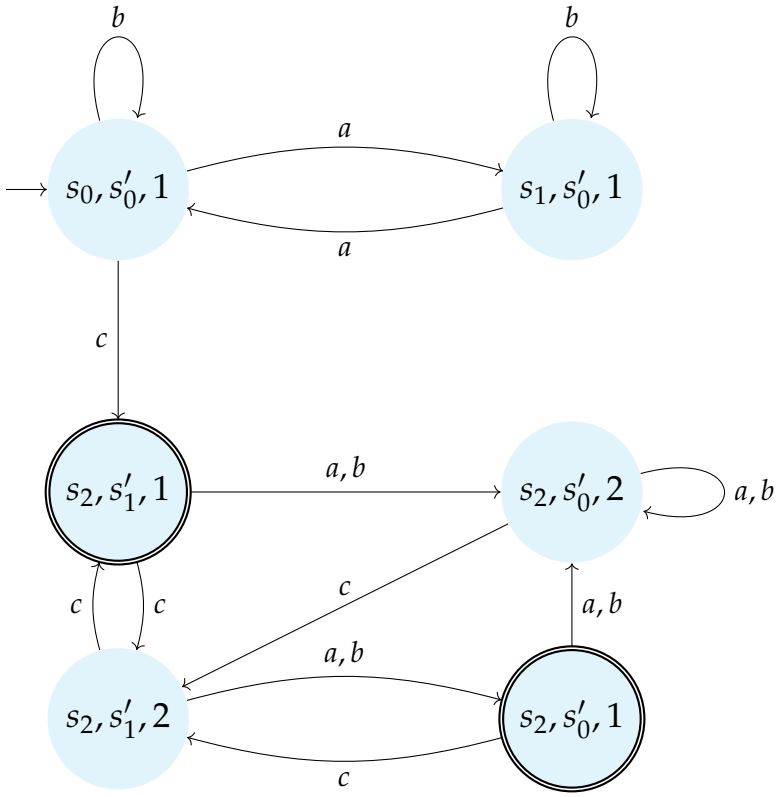


FIGURE 1.7: Resulting Büchi automton $\mathcal{C}$ of $\mathcal{A}$ and $\mathcal{B}$ intersection

Büchi automata intersection

Given two Büchi automata $\mathcal{A}$ and $\mathcal{B}$ defined on a common alphabet, we can think of each of these automata as data structures that symbolically represent specific languages. It is interesting to combine them to obtain a representation of the common words they accept. To achieve this combination, we construct an automaton that accepts the intersection of their languages. This results in a Büchi automaton, denoted as $\mathcal{A} \otimes \mathcal{B}$, such that $\mathscr{L}_{\mathcal{A}\otimes\mathcal{B}} = \mathscr{L}_{\mathcal{A}} \cap \mathscr{L}_{\mathcal{B}}$. This automaton will precisely accept the words that are accepted by both $\mathcal{A}$ and $\mathcal{B}$.

If the alphabets of these automata are not the same, one way to be able to apply the intersection procedure would be to add new "dead state". If we suppose that Σ_1 is the alphabet for $\mathcal{A}$ and Σ_2 for $\mathcal{B}$ with $\Sigma_1 \neq \Sigma_2$, $\forall x \in \Sigma_2 \setminus \Sigma_1$, add a dead state s_d to $\mathcal{A}$ with transitions $s_i \xrightarrow{x} s_d$, from every state s_i of $\mathcal{A}$ labelled with x to the dead state and a transition to itself $s_d \xrightarrow{\Sigma_1 \cup \{x\}} s_d$, labelled with all letters in $\Sigma_1 \cup \{x\}$. Similarly, apply the same procedure on $\mathcal{B}$. In this way, we have provided a common alphabet $\Sigma_1 \cup \Sigma_2$ that enables to reason about the intersection of their languages.

Definition 1.7 (Büchi Automata Intersection). Consider two Büchi automata $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}} \rangle$, and $\mathcal{B} = \langle Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}} \rangle$. The automaton intersection $\mathcal{C}$ of $\mathcal{A} \otimes \mathcal{B}$, $\mathcal{C} := \langle Q, \Sigma, \delta, I, F \rangle$ is defined as follows:

- States. $Q := Q_{\mathcal{A}} \times Q_{\mathcal{B}} \times \{1, 2\}$,

- Initial states. $I := I_{\mathcal{A}} \times I_{\mathcal{B}} \times \{1\}$,

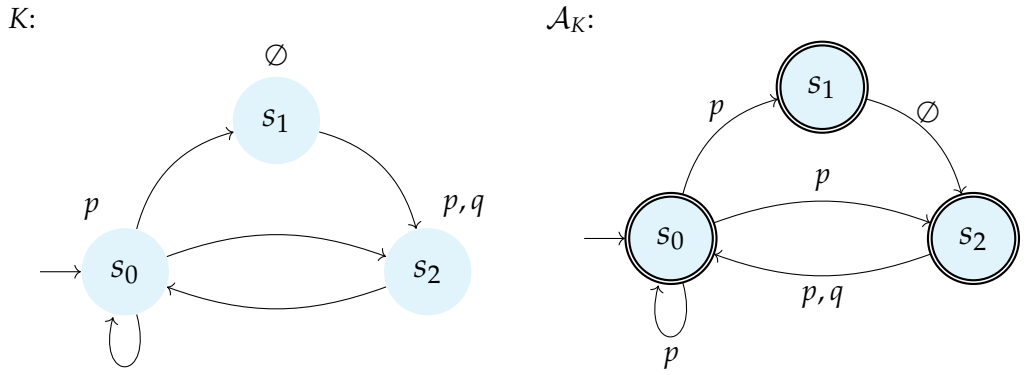


FIGURE 1.8: Kripke structure to Büchi automoaton

- Accepting states. $F := F_{\mathcal{A}} \times Q_{\mathcal{B}} \times \{1\}$.

The construction of $\mathcal{C}$ involves simultaneously traversing both automata and retaining only those pairs of states reachable in both automata. Two markers, 1 and 2, are added to $\mathcal{C}$ to indicate which automaton is being evaluated between $\mathcal{A}$ and $\mathcal{B}$, respectively.
Therefore, the transition function δ is defined as follows: for $a \in \Sigma$, if $s_{\mathcal{A}} \xrightarrow{a} s'_{\mathcal{A}}$ in $\mathcal{A}$ and $s_{\mathcal{B}} \xrightarrow{a} s'_{\mathcal{B}}$ in $\mathcal{B}$, then we add the transition for $n, m \in \{1, 2\}$ as follows:

$$(s_{\mathcal{A}}, s_{\mathcal{B}}, n) \xrightarrow{a} (s'_{\mathcal{A}}, s'_{\mathcal{B}}, m) \text{ where } m := \begin{cases} 2 & \text{if } n = 1 \text{ and } s_{\mathcal{A}} \in F_{\mathcal{A}} \\ 1 & \text{if } n = 2 \text{ and } s'_{\mathcal{B}} \in F_{\mathcal{B}} \\ n & \text{otherwise.} \end{cases}$$

Example 1.6. Consider the Büchi automata $\mathcal{A}$ and $\mathcal{B}$ of Figure 1.6.
The automaton $\mathcal{A}$ accepts words that contain at least one c and an even number of a before the first occurrence of c. $\mathcal{B}$ automaton accepts words containing an infinite number of occurrences of c. The resulting automaton $\mathcal{A} \otimes \mathcal{B}$ is drawn in Figure 1.7. It accepts words containing an infinite number of c and an even number of a before the first occurrence of c.

1.3.2 From LTL to Büchi automata

For an alphabet $\Sigma = 2^{AP}$, where AP is a set of atomic propositions, any LTL formula φ over AP, $[\![\varphi]\!]$ is a subset of Σ^ω. This means that $[\![\varphi]\!]$ corresponds to a language of infinite words over the alphabet Σ. Consequently, the LTL formula can be translated into a Büchi automaton[1]. It is important to emphasize that the language associated with an LTL formula is defined recursively in terms of repetition, concatenation, union, intersection, and complementation operations:

$$\begin{aligned} [\![\varphi \wedge \psi]\!] &= [\![\varphi]\!] \cup [\![\psi]\!] & [\![\mathbf{X}\varphi]\!] &= \Sigma\, [\![\varphi]\!] \\ [\![\varphi \vee \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!] & [\![\mathbf{F}X\varphi]\!] &= \Sigma^\star\, [\![\varphi]\!] \\ [\![\neg\varphi]\!] &= \overline{[\![\varphi]\!]} & [\![\mathbf{G}\varphi]\!] &= \overline{\Sigma^\star\, \overline{[\![\varphi]\!]}} \\ [\![\mathbf{true}]\!] &= \Sigma^\omega & [\![p]\!] &= \bigcup_{\{p\} \subseteq A \subseteq AP} A\, \Sigma^\omega \end{aligned}$$

[1]The reverse is not necessarily true

$$[\![\varphi \cup \psi]\!] = \bigcup_{j \in \mathbb{N}} \left[\left(\bigcap_{0 \leq i < j} \Sigma^i [\![\varphi]\!] \right) \cap \Sigma^j [\![\psi]\!] \right]$$

This way, we can construct a Büchi automaton recursively. Still, the complementation of a Büchi automaton is a task that is far from being simple and relies on relatively complex constructions. It is established in [48, 49], that for any Büchi automaton $\mathcal{A}$ with n states, there exists a Büchi automaton $\mathcal{B}$ such that $\mathscr{L}_{\mathcal{B}} = \overline{\mathscr{L}_{\mathcal{A}}}$, and $\mathcal{B}$ has $2^{O(n \log n)}$ states.

Numerous approaches have been developed to generate automata with a reduced number of states or to minimize the size of already constructed automata. Modern tools such as SPOT[2] [50] and OWL[3] [51] offer dedicated workflows for these tasks.

1.3.3 Kripke structures to Büchi automata

To apply verification using an automata-based procedure, both the LTL formula and the model must be represented as automata. Converting the system, which is characterized by a Kripke structure, into a Büchi automaton is a straightforward process. This is because a Kripke structure already exhibits similarities with an automaton.

Let $K_1 = \langle S, T, I, Act, AP, \mathrm{L} \rangle$ denotes a Kripke structure. We associate to K_1 the Büchi automaton $\mathcal{A}_{K_1}$. This automaton is derived from K_1 by transforming all its states into accepting ones and labeling each transition $T(s_i, s_j)$ with $\mathrm{L}(s_i)$ (as illustrated in Figure 1.8).

Although all states of the $\mathcal{A}_{K_1}$ automaton are marked as accepting, it does not necessarily accept all words. Indeed, the outgoing transitions from a state are all labelled with the same letter p. In particular, as we can see from the Figure 1.8, the empty word $\emptyset^\omega$ is not recognized by $\mathcal{A}_{K_1}$ ($\emptyset^\omega \notin \mathscr{L}_{\mathcal{A}_{K_1}}$).

1.3.4 Automata verification of LTL procedure

To verify whether a Kripke structure K satisfies a given LTL formula φ represented using automata-based techniques [47], it is sufficient to:

1. Build the Büchi automaton $\mathcal{A}_{\neg\varphi}$ representing the negated specification φ,
2. Build the Büchi automaton $\mathcal{A}_M$ representing the model M, and
3. Check if the intersection of the languages is empty, $\mathscr{L}_{\mathcal{A}_M} \cap \mathscr{L}_{\mathcal{A}_{\neg\varphi}} = \emptyset$.

Recall that in prior subsection 1.3.1, we have seen an algorithm capable of computing the intersection of two Büchi automata. Thus, the verification of whether $M \models \varphi$ reduces to determining whether their intersection produces an empty set, *i.e.*, $\mathscr{L}_{\mathcal{A}_M} \cap \mathscr{L}_{\mathcal{A}_{\neg\varphi}} = \emptyset$. This problem, commonly referred to as the **emptiness check**, allows for testing whether the language of an automaton is empty.

Most explicit emptiness check algorithms rely on a Depth First Search (DFS) exploration of the automaton. These algorithms can be categorized into two families:

[2] https://spot.lre.epita.fr/
[3] https://github.com/odoo/owl

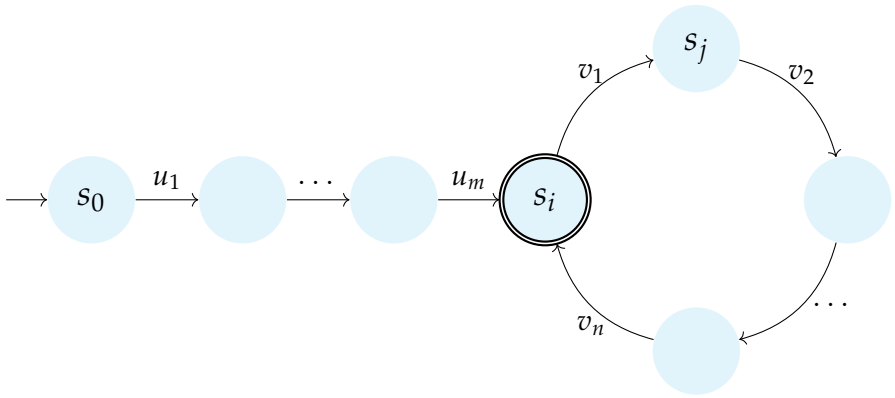


FIGURE 1.9: Cycle example where $|u| = m \geq 0$ and $|v| = n > 0$

Nested Depth First Search algorithms [52], which use a second DFS to detect an accepting cycle, and algorithms [53] based on the enumeration of Strongly Connected Components (SCC) to identify SCCs that contain accepting cycles.

In both families of algorithms, the verification procedure is reduced to the problem of detecting an *accepting cycle*. When the emptiness check detects an accepting cycle or an accepting strongly connected component, it can return a counterexample, meaning an execution that invalidates the property.

Cycle

A Büchi automaton is said to be non-empty if it accepts an infinite word, *i.e.*, if it contains a finite prefix followed by an accepting cycle.

First, a necessary and sufficient condition is stated to determine if the language of a Büchi automaton is empty. Let $\mathcal{C} = \langle Q, \Sigma, \delta, I, F \rangle$ be a Büchi automaton. A cycle of $\mathcal{C}$ is a sequence of the form:

$$s_0 \xrightarrow{\sigma(0)} s_i \xrightarrow{\sigma(1)} s_j \xrightarrow{\sigma(2)} \cdots \xrightarrow{\sigma(\dots)} s_i$$

where $s_0 \in I$, $s_i \in F$, $s_j \in Q$, and $\sigma \in \Sigma^\omega$. Figure 1.9 illustrates such a cycle.
The existence of a cycle with an accepting state, characterizes the emptiness of a Büchi automaton:

Proposition 1.1. *Let $\mathcal{C}$ be a Büchi automaton. We have $\mathscr{L}_\mathcal{C} \neq \emptyset$ if and only if $\mathcal{C}$ has a cycle.*

In the rest of the manuscript, we will also refer to a cycle as the term *loop*.

1.4 Bounded model-checking

Bounded Model Checking (BMC) [19, 21] refers to a model checking approach where the verification of the property is performed using a bounded traversal, *i.e.*, a traversal of symbolic representation of the state-space that is bounded by some integer k. Such an approach does not require storing state-space, and hence is found to be more scalable and useful [18, 22]. It is now an industrial practice to simply run BMC for a certain amount of time and gradually increase the bound k, looking for *witnesses* in longer and longer traces.

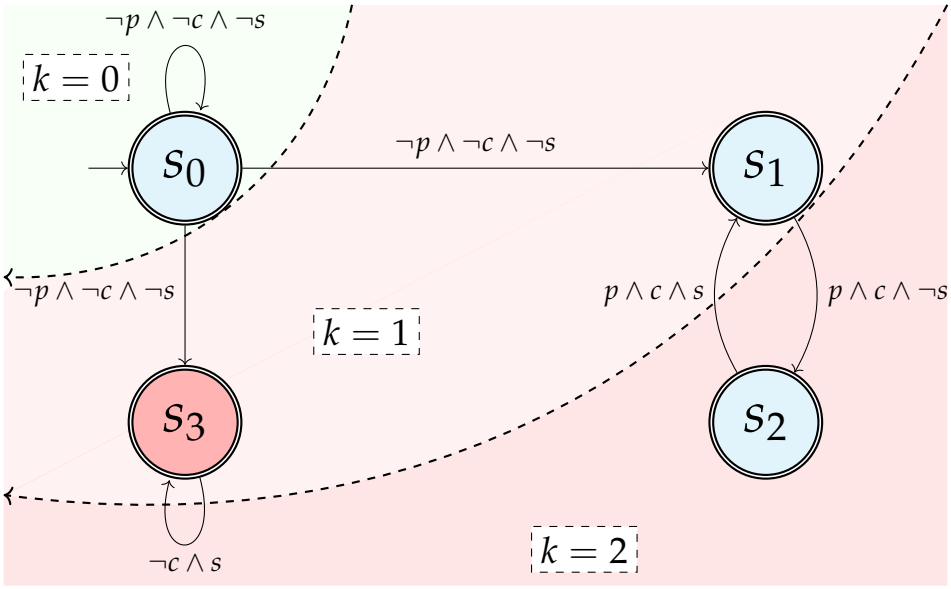


FIGURE 1.10: Büchi automaton of microwave-like system

There are several disadvantages to BMC, however. While the method may be extendable, the method is generally not complete, meaning one cannot be guaranteed a true or false determination for every specification. This is because the length of the propositional formula subject to satisfiability solving grows with each time step, and this greatly inhibits the ability to find long witnesses or counterexamples and certainly inhibits the ability to check all possible paths through a machine. However, even with these disadvantages, the advantages of the method make it a valuable complement to existing verification techniques. It is able to find bugs and sometimes determine correctness in situations where other techniques fail completely.

A crucial observation is that, though the prefix of a path is finite (up to length k), it still might represent an infinite path if there is a cycle from the last state of the prefix to any of the previous states as shown in Figure 1.9. If there is no such loop, then the prefix does not say anything about the infinite behavior of the path beyond step k. For instance, only a prefix with a loop can represent a witness for $\mathbf{G}p$. Even if $p \in \Sigma$ holds along all the states from s_0 to s_k, but there is no loop from s_k to a previous state, we cannot conclude that we have found a witness for $\mathbf{G}p$, since p might not hold at s_{k+1}.

The notion of *k-loops* to define the bounded semantics of model checking is derived from the following definitions [19]:

Definition 1.8 (*(k,l)*-loop). For $l \leq k$, a path σ is called a (k,l)-loop if a transition exists between $\sigma(k)$ and $\sigma(l)$ and $\sigma = uv^{\omega}$ with $u = \sigma[\ldots l-1]$ and $v = \sigma[l \ldots k]$.

Definition 1.9 (*k*-loop). We call a path σ a k-loop if there exists $k \geq l \geq 0$ for which σ is a (k,l)-loop.

Here, only finite prefixes of a path are considered. In particular, only the first $k+1$ states $(s_0, \ldots, s_k)$ of a path are used to determine the validity of a formula along that path. If a path is a k-loop, all the information about this (infinite) path is contained in the prefix of length k.

Definition 1.10 (Bounded Semantics for a Loop). Let $k \geq 0$ and path σ be a k-loop. Then an LTL formula φ is valid along the path σ with bound k ($\sigma \models_k \varphi$) iff $\sigma \models \varphi$.

Now let's describe how the model checking problem with model M and property φ can be reduced to a BMC problem ($M \models_k \varphi$). The basis for this reduction lies on the following two lemmas [19]:

Lemma 1.1. Let φ be an LTL formula and σ a path, then $\sigma \models_k \varphi \Rightarrow \sigma \models \varphi$.
Lemma 1.2. Let φ be an LTL formula and M a Kripke structure. If $M \models \varphi$ then there exists $k \geq 0$ with $M \models_k \varphi$.

Based on lemmas 1.1 and 1.2, it states the following theorem [19]:

Theorem 1.1. *Let φ be an LTL formula and M be a Kripke structure. Then $M \models \varphi$ iff there exists $k \geq 0$ s.t. $M \models_k \varphi$.*

Example 1.7 Let's revisit the microwave example from Example **1.3** and its Büchi representation in Figure 1.10. We have renamed the atomic propositions $AP = \{placed, closed, started\}$ to $AP = \{p, c, s\}$, respectively, for clarity. We want to ensure that $\varphi = \mathbf{G}(s \rightarrow c)$ is always satisfied. This seeks to find an execution where *it is possible to launch the microwave with an opened door*. We will check if there exist an execution on the model that verify the negation of φ: $\mathbf{F}(s \wedge \neg c)$.
As can be seen in Figure 1.10, at the initial bound $k = 0$, the specification is not satisfied. This does not provide information about the validity of the property, so we continue by unrolling the transition relation. Therefore, at $k = 1$, there exists a path starting from the initial state s_0 leading to s_3 that violates the property. The procedure can stop at this bound, resulting in the execution $\rho = s_0 s_1$ that loops infinitely often at state s_3, violating the property.

BMC is primarily solved by symbolic model checking procedures, such as BDD [10] and Boolean SAT solving. This thesis focuses solely on resolution through SAT procedures, detailed in the following subsections. We first define the concept of Boolean satisfiability, then discusses the reduction of BMC to a Boolean formula.

1.4.1 Boolean SATisfiability (SAT)

The satisfiability problem (SAT) is the canonical *NP-complete* problem [54], aiming to determine whether a given formula is satisfiable or unsatisfiable. Despite its theoretical complexity, it has found widespread practical application, particularly in the field of formal verification.

Propositional logic is a subset of logic without quantifiers, where variables can have two possible values: *true* ($\top$) or $false$ ($\bot$). It enables the formulation of logical deductions using operators such as negation ($\neg$), disjunction ($\vee$), and conjunction ($\wedge$). Other operators like implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), and exclusive disjunction (xor) can be expressed in terms of these three first operators. For instance, the formula $a \Leftrightarrow b$ is equivalent to $(\neg a \vee b) \wedge (a \vee \neg b)$.
The operators have a predefined order of precedence, listed in decreasing priority: negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), and exclusive disjunction (xor).

x	y	$\neg x$	$x \wedge y$	$x \implies y$
0	0	1	0	1
0	1	1	0	1
1	0	0	0	0
1	1	0	1	1

TABLE 1.1: Truth table of operators $\neg$, $\wedge$, and $\implies$

To evaluate a formula, we assign each variable a value from the set $\mathbb{B} = \{true, false\} = \{\top, \bot\} = \{1, 0\}$. Here, the truth values signify 0 for $false$ and 1 for $true$. The value of the entire formula is then obtained by replacing the variables by their assigned values and apply the *truth table* rules illustrated in Table 1.1 for some operators ($\neg$, $\wedge$, and $\implies$).

Definition 1.11 (Literal). A *literal* l can take the form of a propositional variable (x) or its negation ($\neg x$). For a given variable x, a positive literal is represented by x, while a negative one is denoted by $\neg x$.

The value that is given to different variables within a formula is referred to as an *assignment*. An assignment, also denoted by the symbol α and can be defined as follows:

$$\alpha : \mathcal{V} \Rightarrow \mathbb{B}$$

where $\mathcal{V}$ represents the set of variable in the given formula.

Depending on whether all the variables of the formula receive values through an assignment, α is said to be *total* (or *complete*), when all elements of $\mathcal{V}$ have corresponding values in α; otherwise, it is considered *partial*. By abuse of notation, an assignment is often represented by the set of its true literals. For instance, $\alpha = \{x_1, \neg x_4\}$ signifies that x_1 is set to $true$ and x_4 is set to $false$.

Definition 1.12 (Satisfiability). A formula is said to be *satisfiable* if there exists at least one assignment α of its variables for which the formula evaluates to $true$. This assignement is then called a *model*. Conversely, a formula is said to be *unsatisfiable* if no assignment α can make the formula evaluate to $true$. In this case, it is said to have a *counter-model*.

In Boolean logic, there are particular characteristic forms of formulas known as *normal forms*. To introduce some of these, we must first introduce the concepts of *cube* and *clause*:

Definition 1.13 (Cube). A *cube* is a finite conjunction of n literals, represented equivalently by

$$\bigwedge_{i=1}^{n} l_i$$

Definition 1.14 (Clause). A *clause* ω is a finite disjunction of n literals, represented equivalently by:

$$\omega = \bigvee_{i=1}^{n} l_i$$

A clause with a single literal is called a *unit clause*. A binary, ternary, or n-ary clause is a clause that contains two, three or $n \in \mathbb{N}^\star$ literals, respectively.

Definition 1.15 (CNF). A conjunctive normal form (CNF) formula $\mathcal{F}$ is a finite conjunction of clauses:

$$\mathcal{F} = \bigwedge_{i=1}^{n} \omega_i$$

For a given formula $\mathcal{F}$ and an assignment α, a clause $\omega \in \mathcal{F}$ is satisfied when it contains at least one literal that, according to α, evaluates to *true*.

Example 1.8. Given the propositional formula $\mathcal{F} = \overbrace{(a \vee b \vee \neg c)}^{\omega_1} \wedge \overbrace{(\neg a)}^{\omega_2} \wedge \overbrace{(\neg d \vee \neg b)}^{\omega_3}$, and assignement $\alpha = \{\neg a, b, \neg c, d\}$. The clause ω_1, ω_2 are satisfied, but not ω_3.

It is possible to transform any propositional formula into a logically equivalent normal form. The conjunctive normal form (CNF) is the preferred input format for state-of-the-art SAT solvers. The transformation of any propositional formula into CNF can be achieved in polynomial time [55, 56]. Therefore, the definition of the Boolean Satisfiability problem (SAT) is as follow:

Definition 1.16 (Boolean SATisfiability). The Boolean satisfiability (SAT) problem is the problem of determining the satisfiability of a given formula in CNF.

1.4.2 SAT-based Bounded Model-Checking

The introduction of BMC by Biere et al. [15] in 1999, along with the increasing efficiency of SAT solvers (GRASP [29] and CHAFF [30]), paved the way for SAT-based model checking and enabled systems with millions of variables to be studied. The advantage of SAT-based approaches over BDD-based ones [10] lies in their reduced need for hand manipulation. Additionally, SAT tools are capable of finding paths (models) of minimal length, which helps the user understanding the generated model.

The SAT-based BMC approach constructs a propositional formula, represented in CNF, that encapsulates both the system M and the negated specification $\neg\varphi$, both unrolled up to length k. This propositional formula $(M \otimes \neg\varphi)$ is deemed satisfiable if and only if there exists a violation of the property within the first k steps. Otherwise, it is unsatisfiable, meaning that the property is verified up to length k.

In the previous section we defined the semantics for *bounded* model checking. We now show how to reduce BMC to propositional satisfiability. This reduction enables us to use efficient propositional SAT procedures to perform model checking.

Definition 1.17 (BMC Propositional Formula). Given a Kripke structure $M = \langle S, T, I, Act, AP, \mathrm{L}\rangle$, an LTL formula φ and a bound k, we construct a propositional formula $[\![M, \varphi]\!]_k$. Let $s_0, \ldots, s_k$ be a finite sequence of states of the system M on a path σ. Each s_i represents a state at time step i and consists of an assignment of truth values to the set of state variables. The formula $[\![M, \varphi]\!]_k$ encodes constraints on $s_0, \ldots, s_k$ such that it is satisfiable if and only if σ is a witness for φ of length k. The

definition of $[\![M, \varphi]\!]_k$ can be synthesized as follows:

$$[\![M, \varphi]\!]_k = \underbrace{\overbrace{I(s_0)}^{\text{Initial states}} \wedge \bigwedge_{i=0}^{k-1} \overbrace{T(s_i, s_{i+1})}^{\text{Transitions relation}}}_{\text{Model}} \wedge \underbrace{[\![\neg\varphi]\!]_k}_{\text{Property}} \tag{1.1}$$

where I defines the initial states, T represents the transition relation of the model M and $[\![\varphi]\!]$ encodes the property. More specifically, this formula can be broken down into two main components:

1. $[\![M]\!]_k$: is a propositional formula that imposes constrains on states s_0 to s_k to form a valid path, starting from an initial state. It is defined as:

$$[\![M]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

2. L_k: a propositional formula that evaluates to *true* only if the path σ contains a loop (cycle). The loop condition is defined by ${}_lL_k$, which is set to *true* if and only if there's a transition from state s_k to state s_l with $k \geq l \geq 0$. By definition, ${}_lL_k$ is equivalent to the existence of a back loop $T(s_k, s_l)$. Thus, L_k is defined through ${}_lL_k$ as:

$$L_k = \bigvee_{l=0}^{k} {}_lL_k$$

It combines all possible cycle of distinct lengths, beginning from the final state s_k.

The **General formula** [19] can be written as follows: for a given LTL formula φ, a Kripke structure M and a bound $k \geq 0$:

$$[\![M, \varphi]\!]_k := [\![M]\!]_k \wedge \left(\underbrace{(\neg L_k \wedge [\![\neg\varphi]\!]_k)}_{\text{no loop}} \vee \underbrace{\bigvee_{l=0}^{k} ({}_lL_k \wedge {}_l[\![\neg\varphi]\!]_k)}_{\text{all possible } (k,l)\text{-loops}} \right) \tag{1.2}$$

The left hand side of the disjunction represents the case where there is no back loop and the property is invalid. This corresponds to a path without a loop. The right hand side encompasses all possible ending points l of a loop, translating a (k, l)-loop with the corresponding ${}_lL_k$ loop condition. The presence of a loop is conjoined with the validity of the property from step s_k to step s_l, noted by ${}_l[\![\varphi]\!]_k$.

Thus, to solve this formula 1.2, a translation step into CNF form is necessary for its verification through a SAT procedure. Several techniques exist for converting propositional formulas into CNFs, with the most well-known being the Tseitin [56] transformation, followed by the compact conversion that minimizes the number of produced clauses [55].

In addition to the above, there are alternative approaches involving SAT solvers in

BMC context, often referred to as *Induction-based methods*, which are used in both industry and academic tools. These methods fall into three categories: (i) k-induction [57], (ii) interpolation (ITP) [58, 59], and (iii) more recent approaches like IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness [60]) and PDR (Property Directed Reachability [61]). They are all based on the principle of inductive proof but differ in the inductive invariant they build. The k-induction is an extension of proof by induction, where the inductive invariant is a path of length k that satisfies the property. ITP computes an over-approximation of the reachable states using interpolants [36], which are themselves calculated from proofs of unsatisfiability returned by a SAT solver during BMC calls. It is worth noting that k-induction or ITP techniques require unfolding the transition relation of the system. IC3, on the other hand, incrementally refines a sequence of sets of states step by step and locally, meaning with queries to a solver that contain only a single transition relation.

1.5 Conclusion

This chapter has introduced necessary definitions to facilitate the understanding of this manuscript. It has been demonstrated that a system can be represented through a Kripke structure with a structure similar to that of a LTS, or through a more compact representation as a Boolean SAT formula.

In this manuscript, we primarily focus on the verification process through its symbolic representation. Only Chapter 6 deals with the automaton-based representation of the verification process.

Chapter 2

SATisfiability solving

Contents

This chapter aims to explore the mechanics of SAT solvers through the most well-known algorithm Conflict Driven Clause Learning (CDCL), firstly in a sequential context and then transition towards parallel SAT procedures.

2.1 Sequential SAT solving

In this thesis, we exclusively deal with complete SAT solvers that can, in a finite time, determine a solution or prove the unsatisfiability of a CNF formula. In contrast, incomplete approaches [62–64] are generally unable to prove the unsatisfiability of a formula. These approaches often involve a non-systematic search of the solution space for a given amount of time. Consequently, since such approaches do not guarantee a complete exploration of the formula's search space, regardless of the amount of time and memory allocated, it is impossible to assert that there is no solution.
When we talk about complete SAT solvers, we generally refer to CDCL algorithm, with significant improvements that considerably enhance its efficiency. Initially introduced by Marques-Silva and Sakallah [29] and later improved by Moskewicz et al. [30], the CDCL algorithm incorporates the concept of learning into the previous DPLL algorithm [65][1], allowing it to learn from conflicts (past errors) to avoid similar conflicts in the future.
Here, we present the essential concepts of the CDCL algorithm, which are fondamental for the understanding this thesis. For a complete details of this algorithm, you can refer to [66].

We begin this section by explaining how the CDCL algorithm works (Section 2.1.1), with a particular focus on some of its crucial components. Each of these components

[1]The fundamental concepts of DPLL have been kept in modern CDCL algorithms

employs generic heuristics that have proven effective in various problem domains. In each component description, we will review the different heuristics and metrics used in sequential SAT solving during the most critical steps of CDCL. External approaches to the CDCL algorithm that aim to simplify the initial formula or add new clauses before/during solving will be discussed in subsection 2.1.2.

2.1.1 CDCL Algorithm

Algorithm 1 CDCL

Require: F : *CNF formula*
Ensure: $\top$ if satisfiable, $\bot$ otherwise
1: $\alpha \leftarrow \emptyset$
2: $decisionLevel \leftarrow 0$
3: **forever**
4: $(F', \alpha') \leftarrow$ UNITPROPAGATION(F, α)
5: $\alpha \leftarrow \alpha \cup \alpha'$
6: **if** $F' = \emptyset$ **then**
7: **return** $\top$
8: **else if** $\{\} \in F'$ **then**
9: **if** $decisionLevel = 0$ **then**
10: **return** $\bot$
11: $\omega \leftarrow$ CONFLICTANALYSIS(F, α)
12: $F \leftarrow F \cup \{\omega\}$
13: $decisionLevel \leftarrow$ BACKJUMPANDRESTART$(decisionLevel, \omega, \dots)$
14: $\alpha \leftarrow \{l \in \alpha | \text{LEVEL}(l) \leq decisionLevel\}$
15: **else**
16: $\alpha \leftarrow \alpha \cup \{$DECISIONVARIABLE$(\dots)\}$
17: $decisionLevel \leftarrow decisionLevel + 1$

SAT solvers are often referred to as highly optimized *black-boxes*, performing the Boolean SATisfiability problem. This black-box approach is one of the reasons for SAT's success. Most solvers take as input a formula in CNF and output a variable assignment if the formula is satisfiable, without any knowledge of the meaning of the variables and encoded clauses. They can also produce a proof of unsatisfiability [67]. An early approach to solving the SAT problem, DP [68], involves eliminating variables one by one, using the solver rule. The elimination step is repeated until the empty clause is found (UNSATisfiable), or until the formula becomes empty (SATisfiable). However, due to the combinatorial explosion memory problem, this technique was soon abandoned in favor of backtracking (DPLL [65]). DPLL backtracking is the process of being able to return to the previous decision level. Nevertheless, variable elimination remains an important preprocessing step in all modern solvers. The elimination of a variable simply consists in performing all possible resolutions on this variable, then removing from the formula all clauses containing this variable, and finally adding all resolution clauses. Since the introduction of the clause learning technique [29] and its refinement [30], the vast majority of SAT problem-solving algorithms for real-world problems are Conflict-Driven Clause Learning (CDCL) procedures (Algorithm 1), also known as *modern* solvers.

Mechanisms. The CDCL algorithm performs a backtrack search; selecting at each node of the search tree, a decision literal which is set to a Boolean value (line 16 of Algorithm 1). This assignment is followed by an inference step that deduces and

propagates (line 4) some forced unit literal assignments (procedure called *Boolean Propagation Procedure*). This branching process is repeated until finding a model (line 6) or reaching a conflict (line 9). In the first case, the formula is answered to be satisfiable, and the model is reported, whereas in the second case, a **learnt clause** is generated by resolution (line 11), following a bottom-up traversal of the *implication graph* [29] (a procedure called *conflict-analysis*). If a conflict occurs without a decision having been made, the formula is considered unsatisfiable (line 9).

CDCL procedure combines several components (DECISIONVARIABLE, CONFLICT-ANALYSIS, ...) amenable to optimization. These components are based on generic heuristics that have demonstrated their effectiveness in various problem domains.

Unit Propagation

Unit propagation (also referred to Boolean Constraint Propagation procedure), used as an optimization component, assigns variables that logically have only one option due to unit clauses. A clause is considered unit under a partial assignment when this assignment yields in every literal in the clause unsatisfied, except for a single unassigned literal. The algorithm simply makes its next guess in a way that ensures the literal will be *true*, thus making the clause *true*.

One noteworthy feature of having a CNF formula is that, to satisfy the formula, every clause must include at least one *true* literal. Consequently, the propagation step entails traversing the set of formula clauses (a lazy traversal approach [30]), searching for clauses that contain one unassigned literal while all other literals are assigned to $false$.

Example 2.1. We take the formula $\mathcal{F} = (\neg a \vee \neg b) \wedge (a \vee c) \wedge (b \vee c) \wedge (\neg c \vee b) \wedge (a)$. The *unit* clause (a) is automatically assigned to *true*. Removing all clauses where the variable a appears, we obtain $\mathcal{F} = (\neg b) \wedge (b \vee c) \wedge (\neg c \vee d)$.

Unit propagation involves the repetitive application of this procedure until the clause database no longer contains unit clauses, ultimately reaching a fixed point.

Decision variable

The decision phase consists in selecting a variable from the set of unassigned variables in the formula, assigning it a value (either *true* or $false$), and subsequently adding the variable to the current assignment stack. When no more variables remain unassigned (*i.e.*, all variables have been assigned), the formula is satisfiable. In this case, the solver returns the current assignment, which represents a model for the formula.

The selection of the next variable to assign is undeniably the most critical criterion for SAT solvers [69]. Regrettably, it is also a computationally challenging task (NP-hard) to choose a variable that leads to a satisfying assignment, just as it is to find such an assignment in the first place [70]. Nonetheless, variable selection during the search significantly impacts the number of steps performed by CDCL and, consequently, execution time [71]. To overcome this difficulty, several heuristics have been developed to estimate the compatibility between a model and the as-yet-unassigned variables based on a given partial interpretation. To be effective, these heuristics

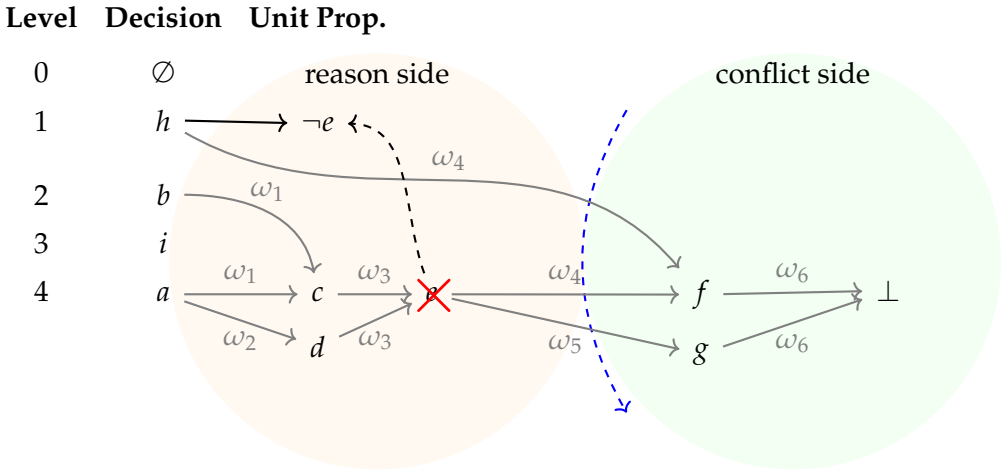


FIGURE 2.1: Resolution graph

must minimize the size of the search tree constructed by the CDCL as much as possible. Nevertheless, a heuristic that significantly reduces the search tree size but consumes too much computational time may be less efficient than a less time-consuming heuristic that generates more nodes. Therefore, a good heuristic is a balanced compromise between the computation time allocated to it and the "quality" of its choices. Here, we present some well-known decision heuristics used by modern SAT solvers.

VSIDS. The Variable State Independent Decaying Sum heuristic introduced by Zhang et al. [31] in the CHAFF solver [30], maintains a score for each variable. Variables with higher scores are given preference when making decisions. After learning a clause, the scores associated with its variables (referred to as "touched" variables) are incremented. More specifically, these touched variables are those that appear in resolutions during conflict-analysis. The list of scores is sorted based on the descending order of the scores. The decision procedure selects the next decision variable with the highest score from the last sorting operation.

CHB & LRB. The Conflict History-Based [72] and Learning Rate Branching heuristic [73], both based on the concept of Exponential Recency Weighted Average (ERWA) [74], share similarities with VSIDS in that they favors variables that have been involved in recent conflicts. CHB rewards variables that have been recently assigned by decision or propagation. These rewards are higher when a conflict is detected. The LRB heuristic extends CHB by exploiting locality and introduces the *learning rate* (LR) of the variables. LR is defined as the capacity of variables to generate learnt clauses. Thus, LRB decision process aims to select the variable that maximizes this LR.

Clause learning from conflict-analysis

The purpose of learning new clauses is to prevent the recurrence of the same failure throughout the search process. These learnt clauses are derived by analyzing the set of literals responsible for a conflict during the conflict-analysis step.

When CDCL encounters a conflict, it examines the decisions it has made and all the assignments it was forced to deduce through unit propgation, which ultimately

resulted in the conflict. To determine this subset of the assignment responsible for the conflict, solvers construct an *implication graph* representing the current state of the proof system. It is a directed acyclic graph where vertices represent assignments, and edges denote the reasons behind these assignments. It is continually updated every time a variable is assigned (via decision or propagation) or unassigned (due to backjumping or restarts). By analyzing this graph, CDCL can learn a clause that is potentially more informative than just knowing that the current partial assignment was problematic (by negating the current partial assignment). This way, CDCL can avoid repeating the same mistake multiple times and skip over large chunks of bad partial assignments DPLL will get stuck in. Figure 2.1 illustrates this implication graph.

Example 2.2. Consider a formula $\mathcal{F}$ composed of the variables a, b, c, d, e, f, g, h and i and the following constraints:

$$\mathcal{F} = \underbrace{(a \vee \neg b \vee c)}_{\omega_1} \wedge \underbrace{(\neg a \vee d)}_{\omega_2} \wedge \underbrace{(\neg c \vee \neg d \vee e)}_{\omega_3} \wedge \underbrace{(\neg h \vee \neg e \vee f)}_{\omega_4} \wedge \underbrace{(\neg e \vee g)}_{\omega_5} \wedge \underbrace{(\neg f \vee \neg g)}_{\omega_6}.$$

Given this formula $\mathcal{F}$, the SAT solver decides on the variables in the following order, as depicted in Figure 2.1: h is first to be decided (level 1), b next (level 2), i at level 3, and a at level 4. We can observe from the figure that several unit propagation have been applied by these decisions.

The conflict-analysis procedure analyzes this graph to identify the reason of the conflict. To do so, a search of a **Unique implication point (UIP)** is performed. A UIP at the last decision level of the implication graph is a variable that exists on every path from the decision to the conflict. It's worth noting that there can be multiple UIPs for a given decision level. First Unique Implication Point (FUIP) [31, 75]) or 1-UIP is a method for generating learnt clauses from an implication graph. It is considered to be the most effective strategy since it provides the smallest set of assignment that is responsible for the contradiction. The UIP procedure divides the implication graph in two sides; the *reason side* (highlighted in orange on the example graph 2.1), which contains decision variables that are responsible of the contradiction and the *conflict side* (in green), that contains the conflict itself. A UIP always resides in the reason side. Once the reason side of a conflict is established, a conflict clause is produced. To build this clause, it is sufficient to negate the literals that have an ongoing edge to the cut that contains the UIP. In the above example, the resulting learnt clause would be $(\neg h \vee \neg e)$. Since the information in this clause is redundant regarding the original formula, it can be added without any restrictions. All these learnt clauses are stored in a clause database.

Clause deletion policy

Conflict analysis and learning play a crucial role in improving the efficiency of modern SAT solvers. However, the expansion of the learnt clause database can significantly slow down unit propagation. This is because a larger number of learnt clauses requires more unit propagation during resolution (CDCL solvers can learn more than 5000 clauses per second [23]).

In order to avoid such a slowdown, modern SAT solvers must efficiently manage the learnt clause database. Given these challenges, it has become essential to selectively remove or temporarily discard some learnt clauses. However, determining which

clauses to delete or discard is a complex task. It requires assessing the "quality" of the clauses using a heuristic that ranks them based on their "usefulness" within a given search subspace. Moreover, deciding when to discard certain learnt clauses also relies on heuristics. So, the issue here is to find the best trade-off between what is considered to be a relevant information and how much of this information must be kept. Below, we describe some state-of-the-art heuristics used in the world's top solvers[2] for addressing these challenges:

Size bounded learning [76]. This approach protects learnt clauses that are sized less than a a predefined threshold. The majority of modern SAT solvers consistently protect unit and binary clauses. This is because these clauses can be managed more easily and their preservation effectively constrains the search space, thereby enhancing unit propagation.

Activity bounded learning [26]. This approach discards learnt clauses when they are no longer deemed relevant according to the activity metric. The concept of activity is the same as in VSIDS [30] and is thus dynamic. It is a positive integer associated with a specific learnt clause, initially set to 0. The activity value is incremented each time the learnt clause is utilized in any way during conflict-analysis and propagation. Thus, this metric provides a general estimation of how much the clause is used. Consequently, as the number of conflicts increases, the solver is less inclined to remove learnt clauses.

Literal block distance (LBD) [23]. LBD is a positive integer, that is used as a learnt clause quality metric in almost all competitive sequential CDCL-like SAT-solvers and parallel sharing strategies (see next Section 2.2). The LBD of a clause is defined as the number of different decision levels at which variables within the clause have been assigned. Hence, the LBD of a clause can change overtime, and it can be (re)computed whenever the clause is fully assigned. If $\text{LBD}(\omega) = n$, then the clause ω spans on n propagation blocks, where each block has been propagated within the same decision level.

Example 2.3. The learnt clause of Example **2.2**, obtained during conflict analysis of the implication graph $(\omega = (\neg h \vee \neg e))$ has $\text{LBD}(\omega) = 2$, as variable h was decided at level 1 and e at level 4.

The concept behind this heuristic is that a decision often leads to a large number of propagation blocks. Thus, adding dependencies between independent blocks can be a way to reduce the number of decisions. This can be achieved by introducing the strongest possible constraints (learnt clauses) between these blocks. More precisely, the LBD value of a learnt clause is equivalent to the number of blocks of literals that are propagated. Therefore, lower LBD scores are considered better. Indeed, it has been proven that *Glue Clauses* [23], clauses with LBD scores of 2, are the most important type of learnt clauses. The solver GLUCOSE [23] keeps all glue clauses throughout the resolution process. The learnt clause reduction function eliminates half of the total learnt clauses based on their LBD values. This reduction function is triggered after a specified number accumulated learnt clauses since the beginning of the search. The choice of when to reduce the learnt clause database is dynamically adjusted based on

[2] As per the results of the SAT competitions (`https://satcompetition.github.io/2023/results.html`)

data collected during the search to adapt to the specific instance [77].

Restart policy

In the DPLL algorithm, when a conflict is detected, a backtrack to the previous decision level is performed. In its enhanced version, CDCL, conflict analysis enables the computation of backjumps, which can be longer than simple backtracks and represent potentially better checkpoints to continue the search for a solution. Specifically, it must backjump to the highest level that allows the learnt clause to be asserted. However, if the search fails for a certain amount of time (evaluated in number of backtracks), then it is considered unlikely for the search to succeed in a reasonable time. Indeed, Gomes et al. [78] have shown experimentally that running the same approach on the same problem but with different initial choices leads to totally heterogeneous resolution times. These experiments allowed to identify a phenomenon of *heavy tail*. One way to escape from this zone is to restart the search while keeping useful information (*e.g.* learnt clauses, variable's scores, ...). Modern CDCL solver restart their search from time to time in order to avoid long tail problems, while keeping useful information such as learnt clauses. Below, we present a non-exhaustive list of well-known restart heuristics:

Luby. The default restart strategy of MINISAT (version 2.1 [79]) has become a standard. This strategy is based on the Luby sequence [80]: $\{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, \dots\}$. In this solver, the intervals between restarts (in terms of conflicts) are determined by the numbers in the Luby sequence, multiplied by 100. This strategy is considered static because it is predetermined and does not change based on the information obtained during the search.

LBD. The idea behind the restart strategy in solvers like MAPLECOMSPS and GLUCOSE is as follows: to generate a maximum number of good clauses (based on their LBD values), restarts can occur when the latest learnt clauses produced have excessively high LBD values. To implement this, the authors of [81] compare the current average LBD values of the last n conflicts with the overall average of all LBD values from the beginning of the search. If the current average is significantly higher than the overall average, a restart is initiated.

Machine Learning Restart. In a recent work, an ML-based restart policy [82] has been introduced to trigger a restart whenever an unfavorable prediction regarding the expected quality of newly created learnt clauses emerges. The researchers observed a correlation between the LBDs of successive learnt clauses. To leverage this correlation, they employed a machine learning mechanism to predict the LBD of the next clause. This prediction is based on the previous three LBDs and their pairwise products. A restart is initiated if the prediction indicates that the LBD of the next learnt clause is below a dynamic threshold determined by the solver's history on the specific input formula.

2.1.2 (In/Pre)-processing phase optimization

Preprocessing has become a crucial component of the SAT solving workflow in recent years. A significant milestone in the history of preprocessing is the technique of *bounded variable elimination (BVE)*, first implemented in the SATELITE preprocessor and later integrated into MINISAT 2.0 [83].

Preprocessing in SAT solving occurs between the encoding phase and the solving phase. However, it's not straightforward to consider this preprocessing phase as a separate procedure. Indeed, it can be seen as an automated re-encoding of a formula, making it part of the encoding phase. Alternatively, it can be viewed as a form of reasoning itself, which would place it within the solving phase. The latter perspective has gained prominence with the emergence of *inprocessing* techniques [84, 85]. In inprocessing, preprocessing techniques are applied intermittently during solving. The following points provide an overview of existing preprocessing and inprocessing techniques.

Simplifying the problem's formula

In order to optimize resolution time, a SAT formula can be preprocessed before solving it. This preprocessing is carried out using a preprocessing engine. One of the preprocessing method is to simplify the formula by reducing the number of unnecessary variables and clauses. Various techniques are implemented in preprocessing engines such as SATELITE [86], and NIVER [87]. Here are some of these techniques:

BVE. Bounded Variable Elimination is the elimination of variables through distribution. It's a highly effective method for preprocessing CNF formulas when applied to variables that do not increase the total number of clauses when eliminated. The key idea is to avoid the combinatorial explosion of the method. BVE preserves satisfiability, meaning the original formula and the simplified version are equisatisfiable. The algorithm performs resolution between clauses containing a literal x (denoted by X^+) and its negation $\neg x$ (denoted by X^-). Let C be the set of resulting clauses. If the number of clauses in C is less than or equal to the number of clauses in $X^+ \cup X^-$, then the algorithm eliminates x by replacing $X^+ \cup X^-$ by C. This procedure has swiftly evolved into an indispensable preprocessing step, with the majority of modern solvers incorporating it as a standard practice. This has also inspired the work in [88] to develop its complementary technique: Bounded Variable Addition (BVA). Instead of exchanging variables for clauses, BVA substitutes clauses for variables, while retaining the same heuristic principle of substitution to reduce the size of the CNF formula.

Subsumption & self-subsumption. Subsumption [86] is applied when some clauses share a particular pattern. A clause ω_1 is said to *subsume* another clause ω_2 if $\omega_1 \subseteq \omega_2$, meaning that ω_1 is more specific than ω_2 so it can be discarded from the CNF formula. Self-subsumption is a special case where one clause almost subsumes another except for one literal. For instance, if $\omega_1 = (\beta_1 \vee x)$ and $\omega_2 = (\beta_2 \vee \neg x)$ where β_1 is a clause subsumed by the clause β_2, then resolving on x will produce a clause $\omega_1' = (\beta_1)$ that subsumes ω_1. Thus, ω_1 is strengthened through self-subsumption using ω_2. It has been proved in [86] that combining self-subsumption with BVE improves preprocessing significantly.

Vivification. Rather than incorporating additional variables or clauses which can lead to a harder formula resolution than initial, the authors of [89] led to a new preprocessor called REVIVAL. It aims to strengthen (*vivify*) redundant clauses from the original formula without introducing additional variables or clauses. To this end, they applied a limited check of redundancy on each clause of the

CNF formula in order to derive or to approximate through unit propagation, one of its minimally redundant subclauses.

These preprocessing methods can also be applied during the solving process (inprocessing). For example, some approaches in [90] continue to perform BVE during the solving phase if it doesn't increase the size of the assertive clause, allowing the removal of redundant literals in learnt clauses immediately after their creation. Others have used subsume techniques to discover subsumed clauses during conflict analysis [91, 92]. Furthermore, authors of [93] have introduced an inprocessing techniques applying *Learnt Clauses Minimization (LCM)* that eliminate redundant literals in learnt clauses by proceeding unit propagation on clause literals. Such a procedure has a significant cost and is applied only at certain restarts and on specific clauses.

Adding relevant clauses to the formula

From the structural composition of the problem, it is possible to learn relevant clauses that will be added to speed-up the resolution. Whether during the preprocessing or inprocessing phases, several well-established studies propose harnessing the concept of community structure [24] or symmetrie breaking [27] to achieve this.

Community structures. In this approach [24, 94], the CNF formula at hand is represented as a graph. The shape of this graph is then analyzed to extract community structure. Roughly speaking, variables within the same community exhibit denser interconnections compared to variables in different communities. The concept of community structure is utilized as a preprocessing step. This involves identifying the communities and subsequently launching a solver on subparts of the formula for a limited time. These parts are created by apportioning the different clauses regarding the communities. In [95] Valuable clauses can be learnt from these preliminary searches and added to the initial formula. This approach notably enhances solver performance in many cases, especially for satisfiable formulas. Given that, this graph-based structuring yields superior performance in industrial problems [94]. The authors of [96] propose a methodology based on the community structure of the formula to categorize industrial instances from random/crafted ones. This categorization is then used to parameterize the SAT solving process. Furthermore, in [97], this concept is leveraged to introduce a new metric aimed at identifying during the solving high-quality learnt clauses and a parallel clause-sharing policy based on a combination of (low) LBD and community number (the community number of a clause ω quantifies the number of different communities of a formula that the variables in ω span).

Symmetries. The underlying idea in leveraging symmetries is to generalize learnt clauses based on an understanding of the repetitive structure inherent in SAT-encoded formulas. Intuitively, if a learnt clause solely depends on clauses that are inherently replicated at different time points, it is possible to replicate the learnt clause at these other time points as well. The clause duplication method is thus rooted in the symmetrical structure of formulas. SAT problems often exhibit symmetries [27], and not accounting for them forces solvers to needlessly explore isomorphic segments of the search space. Indeed, symmetries can facilitate the learning of interesting clauses that classical learning approaches may fail to capture [28, 98–100]. Static *symmetry breaking* [28, 101] identifies

equivalent subspaces (asymmetric constraints) within the search space, allowing the solver to cutoff certain subspaces by incorporating these constraints into the initial formula before resolution. These clauses can also be introduced during the search as a form of inprocessing, referred to as *dynamic symmetry breaking* [25]. Many of these techniques are based on learning symmetric counterparts of previously learnt clauses [99, 100]. A combination of static and dynamic symmetry breaking has been implemented in COSYSEL [102].

2.2 Parallel SAT solving

Parallel procedures have become a prominent axis for enhancing SAT solver efficiency due to the emergence of multi-core machines. In a general sense, parallelism involves executing multiple actions simultaneously. Today, most computers possess parallelism to varying degrees through various techniques, depending on the number of instructions executed simultaneously.
We observe three main components on the classical architecture of a parallel SAT solver: the **parallel strategy** specifies the split of the workload among threads, the **sharing strategy** handles the exchange of relevant information, and the **solver strategy** determines the SAT algorithm to use for given thread(s). The primary objective of all existing parallel SAT solvers is to find the best trade-off between these three components. For instance, using a sharing strategy that exchanges too much information may degrade the overall performances by flooding the underlying solvers. Conversely, no sharing at all might not help solve more problems.

In this section, we will explore the two main strategies for parallel SAT solving: the « portfolio » approach (Section 2.2.1) and the « divide and conquer » approach (Section 2.2.2). We will discuss how these strategies are employed through CDCL-based SAT procedures and the conditions for sharing information (Section 2.2.3).

2.2.1 Portfolio (competition-based)

Portfolio approaches [103] consist in deploying multiple solvers to compete on the same problem, with the winner being the first to provide an answer (illustrated in Figure 2.2). At that point, the search by the other solvers is stopped. Each solver consumes its own CPU time. The problem-solving time is then determined by the real-time of the solver that finds the solution. The first parallel portfolio solver was MANYSAT [103]. This approach is highly effective on industrial instances, as it has won numerous competitions in this category. In fact, several solvers based on this concept have gained the first place in the SAT competitions, such as: P-MCOMSPS [34] in 2021 and PARKISSAT-RS [35] 2022.
This strategy aims to increase the probability of finding a solution using the diversification and intensification principles [104].

Diversification

As we can observe, the justification for using portfolio-like approaches is that the performance of the CDCL algorithm is greatly influenced by numerous parameters, especially those used in decision or restart heuristics. Indeed, at this stage, there are numerous possible heuristics, and none of them dominates all others on all instances.

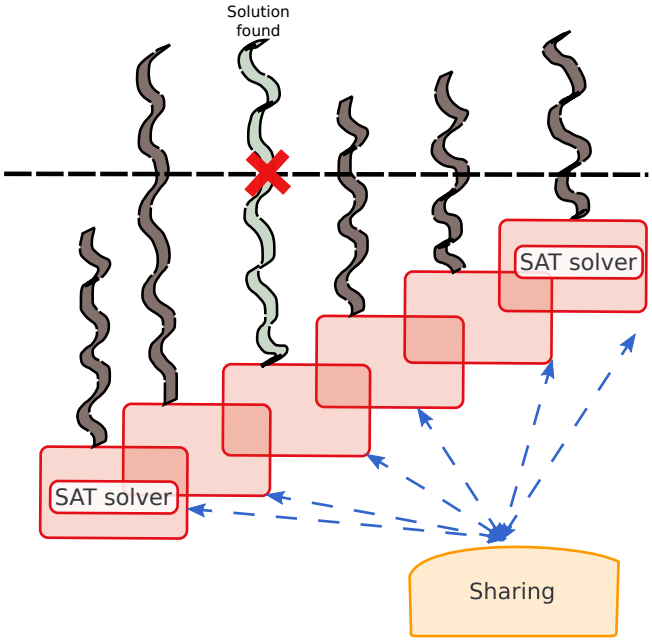


FIGURE 2.2: Portfolio based approach

In order to avoid too much redundant work, it is desirable for different solvers not to explore the search space in the same way. To prevent this, it is necessary to choose heterogeneous strategies; this is diversification [104]. This diversification is achieved through some parameters known to be highly sensitive in the CDCL algorithm, such as decision variables, restart policies, learning schemes, the used random seed, and more. For example, in MANYSAT [103], a fixed number of workers (threads) are used, each with differences in their restart strategies, decision heuristics, and learnt clause schemes. Others achieve diversification through *block branching* [105]. Each worker focuses on a particular subset of variables. For each worker, the decision ranking score of the variables (*e.g.*, VSIDS) it is in charge of is periodically adjusted. This strategy forces workers to choose decision variables within their own subset.

Intensification

Nevertheless, the use of completely orthogonal approaches can reduce the relevance of shared clauses. When a solver learns a new clause, it is within a certain relevant search space. When this clause is shared with another solver, that solver may be in a different part of the search space, so the clause is of no interest to it (already satisfied, for example). Thus, it is sometimes useful to direct multiple CDCL solvers towards the same search space to take advantage of shared learnt clauses; this is intensification [104].

To intensify the search, a master transmits to a slave a set of literals that appeared in conflict analysis during its search (obtained from 1-UIP) and the set of learnt clauses during the search. The goal is for the slave to consider the research conducted by the master around the same conflicts and ensure that it does not duplicate the work of the master by adding the received learnt clauses. To achieve this, when a slave receives this set of literals, it increases their scores (*e.g.*, VSIDS) to have a higher chance of choosing them as decision literals. It's worth noting that conflict analysis in MANYSAT is extended to include clauses satisfied in implication graphs. This

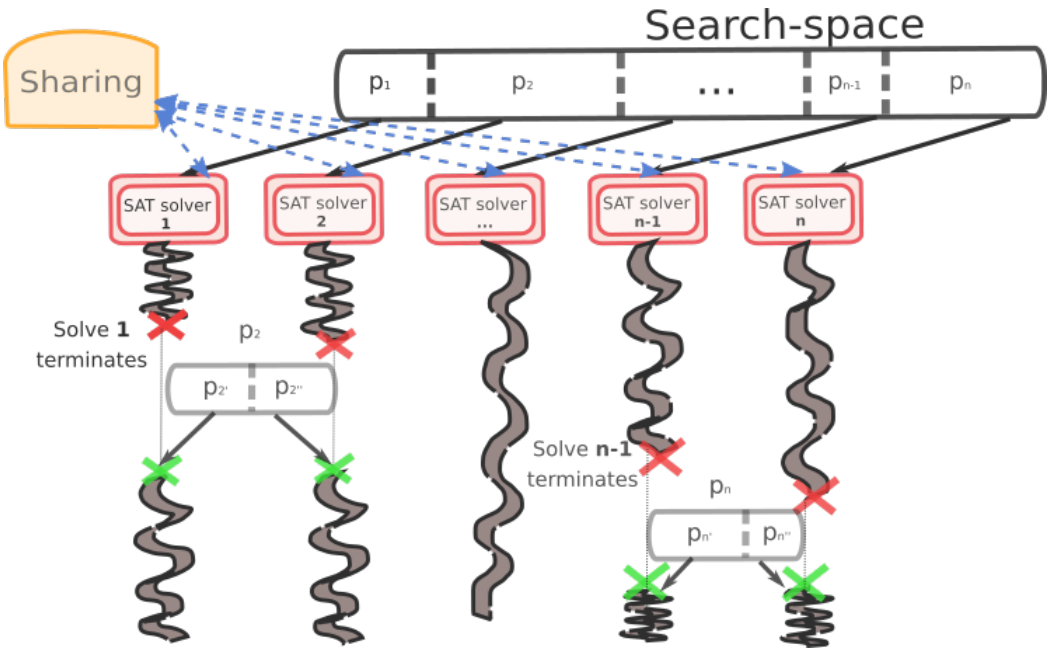


FIGURE 2.3: Dynamic Divide-and-Conquer approach

latter improvement is not common and introduces a new way to reduce the size of assertive clauses. Finally, it should be noted that when the master restarts, the slaves also restart and enter a new intensification phase.

Diversification and intensification are two orthogonal axes. The challenge, therefore, is to find the right "distance" between different processing units, that is, to strike the right balance between intensification and diversification.

2.2.2 Divide-and-Conquer (cooperation-based)

The Divide and Conquer approaches (cooperation-based) involve recursively dividing the search tree into multiple subproblems (Figure 2.2.2). Several approaches are proposed in the literature, with one of the most commonly used being the *guiding path* [106]. It divides the search tree in the form of a path to guide solvers to disjoint subspaces. However, this method has a drawback on the computational loads generated by the subspaces being unbalanced. Indeed, dividing a search tree into two parts can result in two completely different subspaces in terms of resolution time: one very easy to solve (in a few seconds) and the other much more difficult, potentially remaining undetermined. The decomposition into subproblems can be done in two different ways:

Static decomposition. The first approach, known as *static*, decomposes the search space into subspaces before resolution. It is employed in the CUBEANDCONQUER solver [107]. In this method, a large number of subspaces is generated to reduce their complexities and solve them more quickly in parallel. However, this division into subproblems can be time-consuming, even if it is limited to a certain number of subproblems. On some problems, the time spent on this division can be longer than their resolution.

Dynamic decomposition. The second approach is called *dynamic* since it divides the

search tree during resolution (Figure 2.3). When a solver is idle and there are no subspaces p_i left to solve, a load balancing policy must be implemented. The most well-known is *work stealing*: when a solver is idle, it "steals" a portion of a subspace from another solver by dividing it in two. For instance, in the Figure 2.3, the first solver terminates its search on the subspace p_1, so it steals a portion from the second solver subspace p_2 leading to $p_{2'}$ being solved by the first and $p_{2''}$ by the second solver. However, when all dynamically created subspaces become too easy, the parallel solver then performs a very large number of subspace splitting, slowing down the search [108].

Unlike the competitive method, these two Divide and Conquer methods must wait for all solvers to finish their searches to prove that an instance is unsatisfiable. Indeed, it is necessary to prove that all subproblems are unsatisfiable to demonstrate that the initial instance is unsatisfiable. However, to prove that an instance is satisfiable, it is sufficient to show that one of the subspaces is satisfiable.

2.2.3 Sharing strategies

In the aforementioned parallel approaches, solvers can dynamically share information. This exchange warrants a particular focus: if a solver shares its knowledge (consisting of its learnt clauses), then this information will allow the other solvers to avoid recomputing the same information (*i.e.*, descending into parts of the search tree that have already been proven to be unsatisfiable). Thus, exchanging learnt clauses is helpful in increasing the performance of the global system. However, sharing *all* learnt clauses can have a negative impact on the overall behavior. Indeed, a massive exchange can either flood the solver or redirect it towards an *irrelevant* part of the search space. Therefore, existing approaches [103, 104, 109, 110] raise the questions of "what are the relevant clauses to share?" and "between which workers?".

To answer the second question, in almost almost all parallel SAT solvers, clauses are shared between all workers. However, a more complex solution is to allow each workerto choose its emitters [111]. In this approach, each worker selects the ones that are allowed to send it clauses. The workers sending the most relevant clauses (according to some metric) are invited to continue. If a worker has a bad score, it is then removed from the emitters and another one takes its place.

When it comes to identifying the relevant clauses to share, the literature outlines two main strategies for efficiently filtering the learnt clauses:

Static threshold. Many solvers rely on the standard measures defined for sequential solvers (*i.e.*, activity [26], clause size [76] or the LBD value [23]). Only clauses with values below a given threshold for these measures are shared. One straightforward way to determine this threshold is to define it as a constant. For example, clauses up to size 8 are shared in MANYSAT [103], while recent strategies using parallel MAPLECOMSPS engines [34] share clauses with LBD values of up to 4 or LBD$\leq$2 for parallel PARKISSAT-RS portfolio [35]. In the SYRUP parallel solver [112], when a worker learns a clause, it waits for the clause to be used at least once before sending it to the others. The idea behind this is to send only clauses that appear to be useful because already used locally.

Dynamic threshold Other approaches adapt the above thresholds during the search. This allows fine control of the flow of learnt clauses during the solving time. Based on the *additive increase multiplicative decrease (AIMD)* algorithm used in TCP to avoid congestion, authors of [113] proposed dynamically adjusting the size of exchanged clauses between pairs of workers. In HORDESAT [114] sharing is limited in each round to a certain number of literals (*i.e.*, sum of clause sizes). If too few clauses are shared, the threshold of the measure limit is increased.

2.3 Conclusion

In this chapter, we have explored how sequential SAT solver operates, primarily focusing on the CDCL algorithm composed of heuristics (BVE, VSIDS, LBD, ...) that have demonstrated their effectiveness in various problem domains. It is evident that the impressive performance of modern sequential solvers arises from the combination of multiple techniques, making the associated sequential program highly complex. However, there are still unresolved problems that require considerable time to solve. Thanks to the power of multi-core machines, parallelism paradigms have opened up new possibilities for improving SAT solving. We have reviewed in this chapter, a (non exhaustive) of the techniques used in the literature to parallelize SAT solving, providing a comprehensive understanding of this thesis. It has allowed us to glimpse the advantages of parallel SAT solving as well as the challenges that emerged such as clause sharing strategies, fair distribution of search space among workers, *etc*. The next chapter is dedicated to the studies conducted in the context of BMC resolved via SAT procedures on both sequential and parallel settings.

Chapter 3

SAT-Based BMC - Positioning, Analysis and Benchmarking

Contents

Early chapters have shown that propositional formula and thus SAT solving are left with no information about the problem at hand.

While other external approaches to the CDCL algorithm (In/Pre-processing 2.1.2) aim to simplify the initial formula or add new clauses before/during the solving, it's worth noting that while these approaches study the structure of the formula, they rarely exploit the specific problem type. Most existing works in this area focus on developing generic approaches to fine-tune SAT-based CDCL solving.

The first two sections (Sections 3.1, and 3.2) of this chapter provide a state-of-the-art overview of approaches that exploit the characteristics of BMC problems. Following this, an in-depth analysis of the characteristics of the BMC instance will be conducted in Section 3.3. Some of these features will be studied to varying degrees in the remainder of the document in order to propose SAT optimization heuristics for solving BMC problems. To conclude, a description of the benchmark used for conducting experiments in this thesis is presented in Section 3.4.

3.1 State-of-the-art SAT-based BMC

After reviewing the fundamental concept and heuristics employed by CDCL-like SAT solvers, it becomes evident that some of the proposed heuristics can be adapted to the BMC context. Considering the specific structure of BMC, this can lead to optimizations of the SAT solver and accelerates its performance. The remainder of this section provides an overview of state-of-the-art BMC-based heuristics from decision ordering passing by learnt clause metric and symmetry detection in the BMC propositional formula.

3.1.1 Decision heuristics

Most heuristics in the literature leverage the incremental nature of BMC. SAT techniques aim to exploit the commonality between different unrolling depths instances and reuse previously learnt conflict clauses to prune the current search tree. For instance, Wang et al. [115] propose an algorithm to exploit the correlation among a sequence of successive bounds. They do this by predicting and successively refining a partial variable ordering. This variable ordering is computed based on the analysis of all previous unsatisfiable instances. It's then combined with the SAT solver's existing decision heuristic (*e.g.*, VSIDS), to determine the final variable decision ordering. For each previous unsatisfiable instance, they identify all the variables appearing in its unsatisfiable core. Before solving the next SAT formula of the next unrolling depth $k+1$, variables from all the previous unsatisfiable cores are combined to determine a partial linear variable ordering. This is called "partial" because only a subset of the variables may appear, and each variable is assigned a score. When solving the current instance, variables with higher scores are given higher priorities in the decision-making process. Yin et al. [116] propose an algorithm to analyze the transition system model, and then utilize the structure information hidden in the model to dynamically refine the decision ordering of variables in SAT solving. The basic idea is to guide the SAT solving search process based on the structure of the transition system: a transition variable can be set *true* only if its preceding transition has already been taken. If several variables have the same priority, it relies on the default heuristic like VSIDS to make decisions.

Nonetheless, other studies do not take into consideration the incremental nature of BMC and thus focus solely on a single unrolling depth. For instance, Strichman's [117] main idea is to exploit the variable of the original propositional formula, *i.e.*, system's variables without auxiliary variables used to convert the BMC problem in CNF. Indeed, in the Davis-Putnam decision procedure, the variables are decided in a specific static order. This static order is determined following either a forward or a backward Breadth - First Search (BFS) of the k-unfolding of the variable dependency graph. The search starts from the set of variables encoding the evaluated property. Roughly speaking, the intuition behind this approach is that it's possible to assign variables belonging to different time frame without the SAT solver realizing that these assignments are already contradicting each other. It might be the case, for example, that all variables between steps 2 and 10 are already assigned, and as are all variables between steps 12 and 15. To the SAT solver, the assignments are still consistent until it tries to assign the variables with the index 11, revealing that there is no transition between the currently assigned index 10 and index 12 states. Thus, many of the previous assignments have been tried in vain.

Authors of [118] have implemented, in the ISAT solver [119], the static order in [117] providing a BMC-BACKWARD and BMC-FORWARD ordering of the variables. These heuristics favor variables depending on the smallest (resp. the highest) unrolling iteration. They also provide other ordering schemes such as DOMINANT-FIRST (resp. DOMINANT-LAST) which starts (resp. ends) by deciding on the non-auxiliary variables (*i.e.*, the original variables that encode the BMC problem before conversion to CNF). The BOOLEAN-FIRST heuristic gives priority to Boolean variables originating from the BMC problem before conversion. As Boolean can only have the value 0 or 1 they can only be decided once. Deciding them first is assumed to cutoff large portions of the search space at an early stage of the algorithm. They proposed others optimization, but the first effective one they observed was a combination of multiple heuristics, including the BOOLEAN-FIRST and SHRUNK-INTERVAL-FIRST which decides first on variables whose their definition intervals (integer and real) have decreased (shrinked) the most since the beginning of the solving process.

It's worth noting that the majority of studied BMC problems involve a single property at a time. Chen et al. [120] present efficient decision ordering techniques that can improve the overall verification time of a cluster of similar properties. The method exploits the assignments of previously generated execution and incorporates it in the decision ordering heuristic for current evaluating property. Similar properties generally have a large intersection on both corresponding CNF clauses and counterexample assignments. Because of the significant overlap in the counterexample assignments, the result of previously checked properties can be used as a learning tool for unchecked properties.

3.1.2 Learnt clause metric

Here, we can see that there is a multitude of diverse research on decision ordering optimizations. This highlights the heightened sensitivity of CDCL algorithms to the selection of branching variables. Furthermore, it's important to note that learnt clauses are also of great significance as they allow for the identification of unnecessary subpaths to be revisited. However, to the best of our knowledge, very few studies have been conducted in providing a specific measure to asses the quality of the learnt clauses to protect when dealing with BMC problem resolution. It's worth mentioning that the sole study approaching this topic is the one found in [121]. The research conducted by [121] investigate the origin of each variable with respect to its unrolling depth. Their study reveals that the LBD [23] measure, widely used in nearly all modern SAT solvers to identify good learnt clauses, is correlated to the unrolling steps measure they have defined. This measure represents the difference between the maximum depth (*max*) and the minimum depth (*min*) of the variables within a clause, denoted by the (MAX-MIN)-DEPTH measure. They attempted several strategies to harness the information from variable unrolling iterations. However, the proposed heuristics did not yield positive results. These attempts included enforcing variable elimination based on their unrolling iterations, adjusting variable scores, and protecting clauses using a metric other than LBD. In the latter approach, they experimented replacing the strategy for protecting clauses from deletion, which primarily relies on the LBD score of clauses, with the (MAX-MIN)-DEPTH measure. Nevertheless, they observed a slight decline in performance. The given explanation for these results is that in the solver they employed for experimentation, GLUCOSE [23], the LBD score is used not only in assessing clauses but also in triggering restarts. Consequently, they suggest that it might be necessary to adapt the restart

heuristic using this new metric to achieve performance levels that are, at the very least, comparable.

3.1.3 (In/Pre)processing

All the preprocessing techniques discussed earlier (Section 2.1.2, Chapter 2) can be applied to a BMC problem with a fixed time bound k. However, many of these techniques cannot be directly used in the incremental version of BMC. The reason for this limitation lies in the fact that when variables are removed during the preprocessing phase for the initial bound k, their reintroduction in a later step (when increasing k) of the algorithm may yield to an incomplete and incorrect variable elimination process. For example, latch variables can be eliminated from the transition relation $T(s_0, s_1)$ when solving the problem unrolled to $k = 1$. However, this can create inconsistencies when adding $T(s_1, s_2)$ and solving the problem with $k = 2$, as the variables connecting $T(s_0, s_1)$ to $T(s_1, s_2)$ are no longer properly represented.

The authors of SATELITE [86] attempted, on a small benchmark of BMC problems, to preprocess the entire formula each time they reran the instance after unwinding the depth by one. However, multiplying the calls to the preprocessing engine can be time-consuming when dealing with problems involving large iteration steps. To leverage this issue, the authors of [122] came up with the idea of invoking the preprocessing engine only once. They introduced the concept of *Don't Touch* variables, which restricts the variables and clauses that the preprocessor is allowed to eliminate. They demonstrated that these variables can be easily calculated and are relatively few in number.

For the inprocessing approaches lots of studies have been conducted, especially on the use of the symmetrical feature of a BMC formula. Indeed, due to the successive unrolling of the transition relation, the formulas generated by BMC contain a symmetrical part:

$$I(s_0) \wedge \underbrace{T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdot \wedge T(s_{k-1}, s_k)}_{\text{symmetric part}} \wedge [\![\neg\varphi]\!]_k$$

In other words, if we ignore the parts related to initial states and the property, the formula is perfectly symmetrical. This means that if the CDCL solver deduces a new clause by performing resolutions only on the symmetrical part, then this new clause can be duplicated in other time frame simply by renaming the variables.

Strichman's approach [117] aims to encourage the learning of identical clauses at different time steps by forcing variable assignments to generate conflicts that result in the desired clause learning. Each of these clauses can be viewed as a constraint on the state-space, which, on the one hand preserves the formula's satisfiability, and on the other hand, it prunes the search tree.

Example 3.1. Consider the current variable assignments for a formula $\mathcal{F}$: $x_2 = \top, y_6 = \bot$ and $z_{10} = \bot$ where the variable's index corresponds to the time step it belongs to. This leads to a conflict, which is added to $\mathcal{F}$: $\mu = (\neg x_2 \vee y_6 \vee z_{10})$. According to [117], the assignment $x_3 = \top, y_7 = \bot$ and $z_{11} = \bot$ will also result in a conflict, leading in the *replicated clause* $\mu = (\neg x_3 \vee y_7 \vee z_{11})$ being added to F.

This reasoning only works if the formula is entirely symmetrical (excluding $I(s_0)$ and $[\![\neg\varphi]\!]_k$). When considering the entire BMC formula, which is no longer completely symmetrical, the approach needs adjustment. One way to do this, as suggested by Strichman [117] is to simulate an assignment μ for every potentially duplicable clause and check if it leads to a conflict. The additional computational overhead of adding and simulating the replicated clauses was found to be small, leading to an acceleration in the search, although not dramatically so.

Continuing the research on clause duplication, other alternatives to simulating potentially duplicable clauses were proposed. The approaches in [123, 124] involved marking clauses as duplicable (those that do not depend on $I(s_0)$ or $[\![\neg\varphi]\!]_k$). This marking allows for determining, during a conflict, if all the clauses used in the conflict analysis are marked as duplicable; if so, then the new clause is also considered duplicable.

In a similar context, Yin et al. [125] proposed to use additional variables, referred to as *activation literals*, to characterize duplicable clauses. To achieve this, they suggested adding activation literals to all transition relations. Thus, through the conflict analysis process, the parts of the formula from which the new learnt clause is derived can be identified simply by examining the variables in that clause. If a learnt clause contains one of the activation literals for initial states or the property, then the clause is considered non-duplicable. Otherwise, the clause can be duplicated based on the same methodology as [123, 124]. Their approach is primarily dedicated to incremental BMC resolution but remains applicable to a fixed bound as well.

Other works, such as [126], propose a method called Symmetry reduction in BMC (SBMC). This approach is used as a preprocessing procedure with the aim of reducing the problem's size by generating a set of representative transitions. These transition sequences consist of sequences of representative transitions that represent their equivalence classes. As a result, during model checking, the number of sequences dealt with is limited to these representative transition sequences, which are significantly smaller in number than the total number of transition sequences in the initial model.

Elsewhere, community structures have enabled the characterization of many industrial problems [94], including BMC. In [121], various experiments were conducted, revealing a relationship between the time steps of variables in BMC instances and the communities found in its CNF encoding. This demonstrates that the community structure has allowed the identification of a strong dependency between variables from different time steps. They observed that when the value k is sufficiently large, the total number of communities approximates this value. There is a clear pattern, suggesting that the correlation may be very strong. Notably, they observed that communities computed by the clustering algorithm contain variables from different unrolling depths, rather than simply aggregating all variables of the same depth. This suggests that communities are spread over successive time steps, revealing a non-trivial structure existing in the CNF encoding, as also pointed out in [127].

3.2 Parallel SAT-based BMC

Most parallel methods employed for BMC resolution leverage two key factors: the incremental nature of BMC and its structure into time frames, allowing for parallel

formula decomposition and the concurrent resolution of multiple distinct unrolling depths. Numerous studies have been conducted, and this section aims to summarize these findings.

3.2.1 Portfolio-based

Generally when we talk about portfolio parallel architecture on BMC instances, we generally refer to the execution of multiple BMC problem with different unrolling depths. Like in [22, 128], who proposed distributing successive bound values k to a network of workstations. As soon as a satisfiable assignment is found, the search is terminated. If a task finishes without finding a satisfiable assignment, the next task is assigned to the idle workstation until there are no tasks left. A generic framework, called TARMO, was developed for the same purpose [129], which parallelizes BMC for shared memory environments and clusters of workstations. The framework includes a generic architecture for a shared clause database that enables easy clause sharing between SAT solver threads solving different instances with distinct bounds k.

3.2.2 Decomposition-based

On parallelizing the decomposition of the BMC formula, several approaches have been studied, including the one proposed by [130] that extended the concept from a non-specific solving problem (generic) [131] to the domain of BMC. This method involves distributed-SAT solving across a network of workstations using a Master/-Client model, wherein each Client workstation holds an exclusive partition of the SAT problem. While authors of [131] aimed to address the scalability issue by partitioning the clauses disjointedly, the variables appearing in these clauses are not disjoint. In this setup, diagnosis is performed by the Master, and each Client executes a local backtrack when requested by the Master. However, when a Client completes a unit propagation on its set of clauses, it needs to broadcast the newly implied variables to all other processors, including the Master and other Clients. The authors observed that over 90% of messages are broadcast messages. In contrast, the work in [130] optimized communication within the context of BMC by introducing a structural partitioning approach. This strategy allocates at each processor, a distinct set of consecutive BMC time frames. As a result, when a Client workstation completes unit propagation on its assigned clauses, it only broadcasts the newly implied variables to specific Clients. This approach capitalizes on the knowledge of the SAT-problem partition topology possessed by each Client. This allows for effective communication between Clients and ensures that receiving Clients never need to process a message not intended for them.

Still, many researches have exploited the structure of a BMC formula and have proposed finer partitioning methods, mostly used in the incremental SAT-based BMC [132] framework. We can cite the work of [133] who proposed a disjunctive decomposition of BMC instances into simpler and independent subproblems based on *tunnels*, which are sets of control paths. Each subproblem is then simplified using slicing, data path simplification, and tunnel-specific control flow constraints before being solved independently. At each increment of the unrolling depth k, they recompute the number of tunnels needed to partition the problem. The approach was tested within an SMT-based BMC framework.

The authors of [134, 135] have used the symmetry of the formula to decompose it into partition as a Master/Slave scheme. The master collects the non-symmetric parts (initial state and property) and sends a different partial assignment to each slave. Besides, the unrolling of the transition relation is postponed and performed locally by each individual slave (partition). The first slave is in charge of the transition $T(s_0, s_1)$, the second slave with $T(s_1, s_2)$ and so forth. The similarity between partitions allows for sharing and replication of conflict clauses between partitions in a natural way; if a conflict occurs in the partition in charge of $T(s_2, s_3)$, it can be added to the partition $T(s_3, s_4)$ by just changing the indexes of the conflict's clause variables. When all slaves terminate, the master increases the bound limit k to $k+1$ and shares these conflicting clauses among the slaves.

3.3 Analysis of SAT-based BMC formula

As we have seen in the previous sections, a profound study of the specificities of the problem at hand leads to adaptations of the CDCL algorithm's components, resulting in a faster and more efficient resolution.

In this section, we summarize the various features of BMC problems that piqued our interest during this thesis. Specifically, the first feature provides additional information about the variables in the SAT formula, specifying the unrolling depth associated with each variable within the BMC problem. The second piece of information determines beforehand, the category of the evaluated LTL specification from the hierarchy of Manna & Pnueli [39]. Last but not least, in light of all this, we introduce an additional metric used to characterize the learnt clauses during solving. This new metric is based on the meaning of the variable's clause in the BMC problem.

In the rest of this section, we give the intuition behind extracting these information, illustrating it with an example.

3.3.1 A running example

Let's consider an example of a bit-like counter system with four bits: x, y, z, and w. A representation of this system is displayed in Figure 3.3.1 in an SMV language [1]. This program is divided into several parts:

(1). It starts by enumerating the *system variables* and their definition intervals. The provided program deals with Boolean variables only.

(2). The initialization part sets the initial values of the system variables.

Variables

x : boolean; *y : boolean;*
w : boolean; *z : boolean;*

Initialization

$x := FALSE;$ $y := FALSE;$
$w := TRUE;$ $z := TRUE;$

Transitions

$x := !(x = y)$
$y := !y$
$w := \begin{cases} !w & \text{if } x \neq y \\ w & \text{otherwise} \end{cases}$
$z := \begin{cases} z & \text{if } w \neq z \\ !z & \text{otherwise} \end{cases}$

LTLSPEC

$\mathbf{G}(!x \mid !y)$

FIGURE 3.1: SMV program of bit counter example

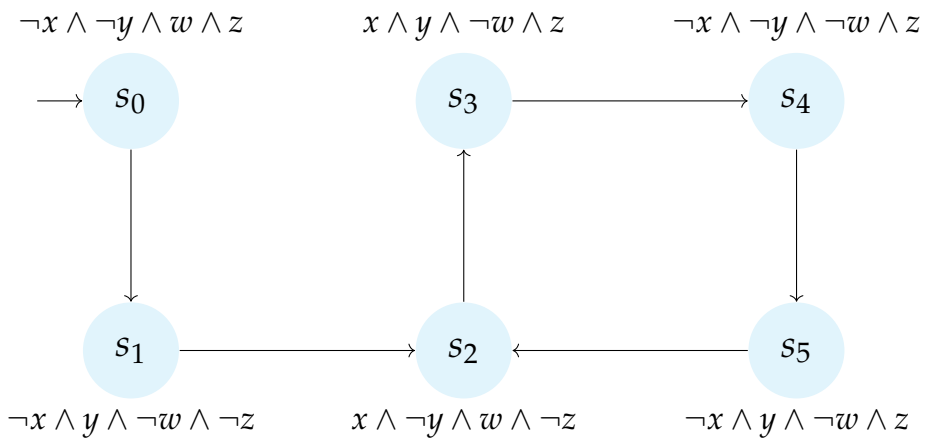


FIGURE 3.2: Kripke structure of bit-like counter example

(3). The different states each variable can take during the execution, are encoded as constraints. These constraints set the next value that each variable will take.

(4). The last section encodes the LTL property φ that will be checked on this model[1]. In this case, it expresses the condition where x and y must not hold at the same time at any stage in the system. The negation of $\mathbf{G}(!x \mid !y)$ would be a liveness property[2] of the form: $\mathbf{F}(x \,\&\, y)$, indicating that at some point in the future, both x and y will be simultaneously *true*.

We can naturally encode this model into its Kripke structure representation as depicted in Figure 3.2, where $M = \langle S, T, I, Act, AP, L \rangle$ with:

- $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$,
- $T = \{(s_i, s_{i+1}) | 0 \leq i \leq 4\} \cup \{(s_5, s_4)\}$,
- $I = \{s_0\}$,
- $Act = \emptyset$,
- $AP = \{x, y, w, z\}$,
- L $= \{s_0 : \neg x \wedge \neg y \wedge w \wedge z, s_1 : \neg x \wedge y \wedge \neg w \wedge \neg z, \dots\}$.

The complete representation of this system is thus reduced to 6 states. In the bounded semantics, we limit the model representation K up to specific depth k. For instance, at $k = 0$, s_0 serves as the initial state that represents the state of the variables during the initialization phase of the program. Here, x and y are set to $false$, while w and z are set to $true$. The property is still verified at this moment.
With this initial assignment, the program can progress to only one possible situation, represented by the state s_1. At this stage, the system is unrolled up to bound $k = 1$, where we observe no violation of φ: x and y are not $true$ simultaneously in either s_0 or s_1 also. Therefore, we cannot draw any conclusions about the validity of the property. One can apply automata-based verification procedures to evaluate the validity

[1] It is possible to specify multiple properties that will be checked one by one.
[2] specifically a *guarantee* property according to Manna & Pnueli's hierarchy

$$\underbrace{(\boxed{x_0 \wedge y_0} \wedge \boxed{w_0 \wedge z_0})}_{I(s_0)} \bigwedge \tag{3.1}$$

$$\underbrace{\left(\boxed{x_1 = (x_0 \neq y_0)} \wedge \boxed{y_1 = \neg y_0} \wedge \boxed{z_1 = (w_0 \neq z_0)} \wedge \boxed{w_1 = y_0}\right)}_{T(s_0,s_1)} \bigwedge \tag{3.2}$$

$$\underbrace{\left(\boxed{x_2 = (x_1 \neq y_1)} \wedge \boxed{y2 = \neg y_1} \wedge \boxed{z_2 = w_1} \wedge \boxed{w_2 = (x_1 \neq y_1)}\right)}_{T(s_1,s_2)} \bigwedge \tag{3.3}$$

$$\underbrace{\left(\begin{array}{l} \boxed{x_0 \wedge y_0} \vee \\ \boxed{x_1 \wedge y_1} \vee \\ \boxed{x_2 \wedge y_2} \end{array}\right)}_{[\![\neg\varphi]\!]_2} \tag{3.4}$$

FIGURE 3.3: Bit-like counter example propositional formula unrolled up to bound $k = 2$

of property φ in this system K. By closely examining the automaton in Figure 3.2, we can deduce that at $k = 3$, the sequence $s_0s_1s_2s_3$ is sufficient to invalidate property φ.

In a SAT-based verification, we need to encode the model and the property into a propositional formula. By specifying the unrolling depth value k to 2, the resulting propositional formula is displayed in Figure 3.3. The set of variables $\{x_i, y_i, w_i, z_i\}$ are decision variables, where each of them represents the state of the corresponding system variables x, y, w, and z, respectively, at step $i = 0, \ldots, k$. The encoding uses the simplified SAT formula 1.1 for simplifying the reading (this explains why cycle constraints do not appear in the encoding). Mainly, the constraints encode:

- **Initial.** Subformula 3.1 translates the initial states $I(s_0)$. Since there is only one initial state, this results in only one constraint.

- **Transition.** Subformulas 3.2 and 3.3 refer to the transition relation unwound down to depth 1 and 2, respectively. The first subformula encodes the transition $T(s_0, s_1)$ ($s_0 \rightarrow s_1$) whereas the second encodes the transition $T(s_1, s_2)$ ($s_1 \rightarrow s_2$).

- **Property.** Formula 3.4 describes the negation of the property φ. Each of these constraints expresses the same condition: x and y should be *true* simultaneously, on at least one step $i = 0, 1, 2$.

Formula 3.3 is then encoded in CNF to be evaluated by the SAT solver. Initially, with $k = 2$, the property is not invalidated. Therefore, the SAT solver will not find any possible assignment of variables x_i, y_i, w_i, z_i that invalidates the property. Consequently, it will deduce that the formula is UNSATISFIABLE.

3.3.2 Observations from propositional formula

We can observe quite clearly that the constraints highlighted in blue (*e.g.*, $x_0 \wedge y_0$), involve variables used to encode the property only, *i.e.* x and y. We refer to these variables as *property variables*. Where other constraints highlighted in blue, such as $x_1 = (x_0 \neq y_0)$, use *property variables* but the difference with blue constraints is that it will bring in extra variables derived from the conversion of some logical operators ($\neq$, $\Longrightarrow$, ...) to allow their conversion to a CNF. Therefore, the constraints in yellow include *auxiliary variables* not originating from the original problem.

In contrast, the green-highlighted constraints, such as $z_2 = w_1$, define the state of the problem variables, excluding those implicated in the encoding of the property, meaning the constraints that involve only w and z. Similar to the yellow constraints, the brown-highlighted constraint $z_1 = (w_0 \neq z_0)$ also introduces new auxiliary variables (due to $\neq$ operator).

Based on these observations, we can deduce a partitioning of the SAT formula variables into two disjoint subsets: $\mathcal{M}'$ and $\mathcal{J}'$, where $\mathcal{M}'$ is the set of variables of the original problem, corresponding to x, y, z, and w, while $\mathcal{J}'$ is a set of auxiliary variables used to finalize the conversion into a CNF formula. For instance logical operators ($\leftrightarrow$, $\Longrightarrow$, $\neq$, ...) will rely on auxiliary variables for their representation. Since the standard CNF conversion is obtained by the distributive properties of $\wedge$ and $\vee$; this results in an exponential increase in the size of the formula. A common technique to reduce the size of clauses is to introduce extra variables to represent the truth value of subformulas [55, 56]. Usually, these extra variables represent more than 80% of the total number of variables in the CNF formula.

Example 3.1. Consider the formula $\mathcal{F} = (A \vee \neg B) \wedge \neg(C \vee D)$ where A, B, C and D are subformulas already in CNF. $\mathcal{F}$ may be converted more succinctly by the introduction of a variable $x_{C \vee D}$ that represents the subformula $(C \vee D)$. Thus, $\mathcal{F} = ((A \vee B) \wedge \neg x_{C \vee D}) \wedge (x_{C \vee D} \leftrightarrow (C \vee D))$. After converting each equivalence into CNF, we obtain $\mathcal{F} = (A \vee \neg B) \wedge x_{C \vee D} \wedge (C \vee D \vee \neg x_{C \vee D}) \wedge (\neg C \vee x_{C \vee D}) \wedge (\neg D \vee x_{C \vee D})$. We can state that the previous formulation of $\mathcal{F}$ is equisatisfiable with the new formula that introduces the variable $x_{C \vee D}$. The transformation clearly preserves satisfiability [55, 56].

The encoding of the property φ involves variables from $\mathcal{M}'$ (x and y). It may include some auxiliary variables already present in $\mathcal{J}'$ due to CNF transformation but can also introduces new auxiliary variables denoted by the set $\mathcal{D}$.

Let $\mathcal{M}'_\varphi$ denotes the set of variables from $\mathcal{M}'$ involved in φ and $\mathcal{J}'_\varphi$ denotes the set of variables from $\mathcal{J}'$ involved in φ. These definitions lead to the definition of three disjoint sets:

- $\mathcal{P} = \mathcal{J}'_\varphi \cup \mathcal{M}'_\varphi \cup \mathcal{D}$, the set of the variables used to encode the property (meaning, x and y),
- $\mathcal{M} = \mathcal{M}' \backslash \mathcal{M}'_\varphi$, the set of variables that encodes the model excluding the property variables (*i.e.* w and z only), and

- $\mathcal{J} = \mathcal{J}' \backslash \mathcal{J}'_{\varphi}$, the set of auxiliary variables not involved in the encoding of the specification.

3.3.3 BMC features

It turns out when switching from a system representation (SMV [1], VERILOG [2], AIGER [38], ...) to a propositional formula, valuable information may be lost. Among them the inability to differentiate the variables related to the property from those generated by the CNF conversion algorithm. Additionally, having insights into the unrolling depth to which each variable belongs, or having indications within the SAT solving process regarding the nature of the property (*e.g.*, safety, guarantee, *etc.*), can be beneficial. Unfortunately, all of these details remain hidden from SAT procedures.

The starting point of our investigation is to extract some features of BMC problems. As a first insight, one can observe that SAT-based BMC variables can be trivially divided according to the step each variable belongs to. In our example, there are three possible partitions: variables associated with the initial state ($\{x_0, y_0, w_0, z_0\}$), those belonging to step 1 ($\{x_1, y_1, w_1, z_1\}$), and finally, those of step $k = 2$. Many studies on SAT-based BMC have pointed out this characterization and proposed optimizations particularly in the decision variable ordering heuristic (see Section 3.1).

Secondly, even though the majority of works on SAT-based BMC solving, presented subsequently, focus on the verification of safety properties, it would be interesting to apply the syntactic characterization of Manna & Pnueli [39]. This could be done in order to compute the corresponding class in the hierarchy, given that the LTL formula is known *a priori*. This allows to propose a parametrization of the SAT solving tailored to each category of the hierarchy.

Clause classification metric C_X

Finally, since we can categorize the CNF variables into three disjoint sets: variables encoding the model $\mathcal{M}$, those for the property $\mathcal{P}$ and auxiliary variables $\mathcal{J}$ used for the conversion, it would be interesting to integrate this information within the learnt clauses. Typically, a learnt clause spans on variables that belong to both the model and the LTL property. Therefore, we suggest here a sharper classification based on the composition of clause variables.

In the remainder of this section, we introduce a classification of variables, and consequently, a classification of clauses, based on the underlying BMC problem. The approach outlined here can be applied to any problem, provided that the relevant information is captured during the conversion of the original problem into a propositional SAT formula. For instance, one can consider the graph coloring problem, used for register allocation. Basically this process assigns one variable per register. This is achieved by translating an interference graph into a set of constraints. During this translation, the information about "critical" variables is lost, *i.e.* which node is in conflict with most of others. One could use this information to split the set of variables into multiple classes. Thus, a clause could be characterized by summing the degree of each of its variables.

For a set of clauses $\mathcal{F}$, $C_X = \{\omega \in \mathcal{F} \,|\, \forall v \in V(\omega), v \in X\}$ denotes the categories of clauses, where X can be either $\mathcal{P}$ (the property), $\mathcal{M}$ (the model), $\mathcal{J}$ (the auxiliary

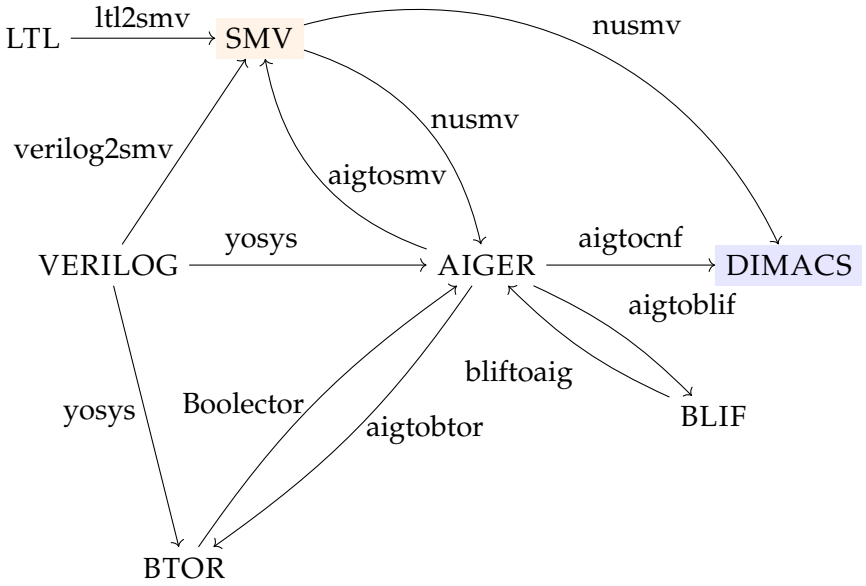


FIGURE 3.4: System format mapping

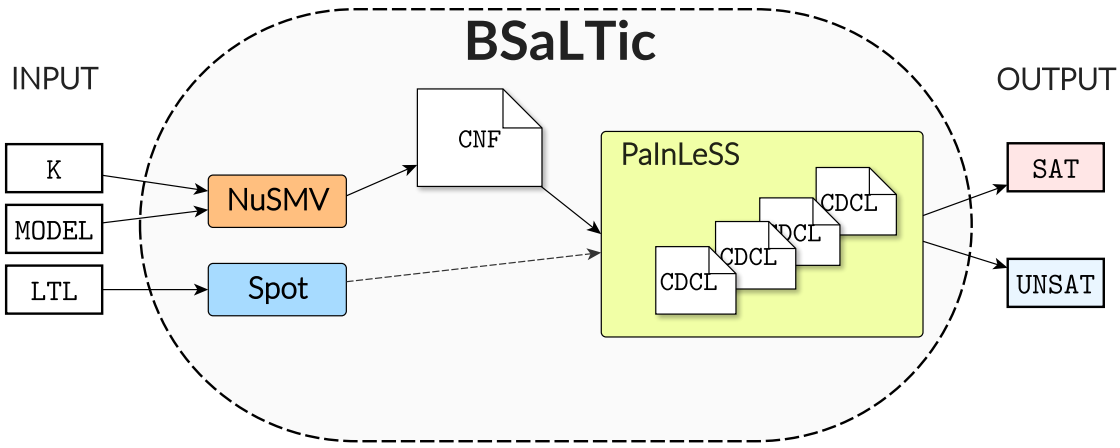


FIGURE 3.5: The dashed box marks the BSALTIC framework

variables of the model), $\mathcal{PJ}$ (property and auxiliary variables), $\mathcal{PM}$ (property and model variables), $\mathcal{MJ}$ (model and auxiliary variables) or $\mathcal{PMJ}$ (property, model and auxiliary variables).

As a result, the colored boxes of the formula 3.3 highlight clauses according to this classification: red box defines $C_{\mathcal{PMJ}}$ clauses covering variables from the three sets $\mathcal{P}$, $\mathcal{M}$, and $\mathcal{J}$. The brown boxes encompass $C_{\mathcal{MJ}}$ clauses containing model variables ($\{w, z\} \in \mathcal{M}$) and auxiliary variables, *etc*. Consequently, we can study the utility of each of the above classes of clauses during the solving process. Further analyses are conducted and presented in the subsequent Chapter 4.

3.4 Benchmarking

A wide range of openly available LTL model checking problems come under the formats SMV [1], VERILOG [2], AIGER [38], *etc*. For the purpose of this thesis, the SMV language appeared to be the most suitable for facilitating the extraction of the aforementioned features.

The graph in Figure 3.4 provides an overview of the possible translations between these languages into SMV, and consequently into a propositional formula in CNF (DIMACS file format).

During this thesis, we combined several tools that led to the development of a framework called BSALTIC[3], tailored to our needs. Figure 3.5 displays the architecture of the framework, which involves NUSMV [136], SPOT [50], and PAINLESS [137] frameworks. BSALTIC takes three parameters as input, which are required for any BMC problem:

1. the modeling system M in SMV language,
2. the LTL specification φ, and
3. the bound $k \geq 0$.

SPOT is used to identify, syntactically, the class of LTL formula it belongs to according to the hierarchy of Manna & Pnueli. Additionally, it was also used to compute the automata representation of the problem (see Chapter 6). On the other hand, NUSMV [136] provides direct access to conversion procedures such as the Tseitin transformation [56] or the compact conversion of Sheridan [55]. Consequently, the produced DIMACS file serves as an encoding of the BMC problem into a CNF formula. These two pieces of information are then processed by the PAINLESS interface, which acts as an aggregator of solvers [137]. Our modified versions of the CDCL solvers, such as MAPLECOMSPS solver [32] or KISSA-MAB [33], are integrated into this interface. This interface also facilitates the integration of parallel approaches.

Our dataset comprises SMV hardware programs (each with its respective LTL properties) collected from various benchmarks, including the HWMC Competition (2017[4] and 2020[5]), hardware verification problems [136], the BEEM database [138], and the RERS Challenge[6]. Some LTL properties have been generated using Spot [50] on various bound $k = \{10, 20, 40, 60, \ldots, 1000\}$.

3.5 Summary & Discussion

We have observed a rich state-of-the-art, indicating significant interest in BMC and its impact on the SAT solving. We have also taken a global view of the studies conducted in the context of parallel SAT solving, noticing that most of these studies take into account the specific structure of the BMC formula, leading to a multitude of decomposition strategies that effectively enhance the solving process.

[3]For a description of our setup, detailed results and code, see `https://akheireddine.github.io/`

[4]http://fmv.jku.at/hwmcc17/

[5]http://fmv.jku.at/hwmcc20/

[6]RERS models translated using NUSMV: `https://tinyurl.com/29a4jcme`

Through a bit-counter example, we have listed interesting features of BMC that can allow fine-tuning of some components of the CDCL algorithm.
The main focus of this thesis revolves around the learning mechanism, whether in adapting the clause deletion heuristic for BMC solving or generating more meaningful learnt clauses to share with the SAT solver. All of this is carried out in both sequential and parallel contexts.

For the remainder of this thesis, the next chapter presents our first contribution, which leverages the clause classification presented above to propose an adaptation of the clause deletion and sharing policies with the aim of protecting or share interesting learnt clauses.

Chapter 4

Tuning the learnt clause databases

Contents

The concept of relevant information in SAT procedures is quite unclear. Many existing techniques manage learnt clause databases using generic metrics (*e.g.*, LBD [23], Activity [26], *etc.*) that help characterize potentially high-quality learnt clauses. However, in our study, which specifically focuses on solving BMC problems, we have not come across any research that provides a meaningful measure that allow the characterization of relevant learnt clauses.

This work resulted in a publication at the CP'2021 conference, and its extension was later published in the CONSTRAINT journal. Our primary focus is on the "learnt clause" features with the intuition that: while the generic LBD metric [23], which has been used by the best SAT solvers to date, efficiently characterizes learnt clauses for most problem instances, it can be sharply adjusted with structural information in the context of BMC. The goal is to design a new way of detecting the relevance of a set of clauses characterized by their LBD value and, in addition, their belonging to a class of clauses C_X, a feature that we have introduced in Chapter 3.

To achieve this, we introduce the concept of a SELECTOR, which is a vector that specifies the appropriate LBD value for each category of clauses C_X. This allows to tailor the clause deletion policy based on the clause's category. For example, a selector might indicate that learnt clauses of type $C_{\mathcal{P}}$ should be protected up to LBD ≤ 8, $C_{\mathcal{J}}$ with LBD ≤ 4, and so on.

In particular, our contributions in this chapter are as follows:

- We propose new heuristics for computing selectors that preserve interesting learnt clauses from deletion during sequential SAT solving.

- We employ these selectors to facilitate the exchange of information among different CDCL workers in the context of parallel SAT solving, utilizing a portfolio-based strategy.

- We demonstrate the applicability of this study to any CDCL-based SAT solver by conducting experiments with our approaches on the two leading SAT solvers: MAPLECOMSPS[32] and KISSAT-MAB[33].

We begin by investigating the impact of clause classification C_X in Section 4.1. Based on this study, we develop two heuristics, H_S and H_{LP}, to compute suitable selectors for identifying valuable learnt clauses within the SAT solver (Section 4.2). We then experimented these heuristics in the context of sequential SAT solving using the well-known solvers MAPLECOMSPS and KISSAT-MAB (Section 4.3). Subsequently, we explore the application of these selectors in parallel settings. Section 4.4 presents our sharing strategy for exchanging relevant learnt clauses among CDCL workers in a parallel portfolio-based strategy. Finally, Section 4.5 presents the experimental results for the parallel setting, as well as its combination with sequential ones.

4.1 Analysis of clause classification feature

A clause is considered useful when it is involved in either the *conflict-analysis* procedure or *unit propagation*. In both cases, the clause is deemed interesting, as it either contributes to the generation of a conflict clause or aids in propagating a unit clause.

To assess the impact of clause classification, we measured the usage rate of the learnt clauses during the *conflict-analysis* procedure[1]. To do this, through our BSALTIC[3] tool, we conducted analyses on MAPLECOMSPS and KISSAT-MAB, the winners of the SAT competition in the main track 2016 and 2021, respectively. Unlike MINISAT [79] or GLUCOSE [81] solvers, these solvers do not have a single clause database; instead, it is divided into three distinct databases (*core*, *tier-2* and *local*), and the criterion used for detecting new learnt clauses to store is based on their LBD value:

core. A permanent database that stores clauses with LBD$\leq$3 in MAPLECOMSPS and LBD$\leq$2 in KISSAT-MAB. Clauses in this database are considered highly important and are never deleted. It mainly contains unit clauses and glue clauses (LBD$=$ 2).

tier-2. A temporary database containing potentially useful information. It stores learnt clauses with LBD$\leq$ 6. Clauses in this database can be promoted to the **core** if their LBD decreases to LBD$\leq$ 3, or they can be moved to the third database if they are not used during *conflict-analysis* or *unit propagation*.

local. Also a temporary database that holds remaining clauses with LBD$>$ 6. These clauses may be permanently deleted if they are not used during the last conflicts, or they can be upgraded to the **tier-2** database if they satisfy the LBD bound constraint of **tier-2**.

[1]Unit propagation analysis was omitted since we observed similar information to the *conflict-analysis*.

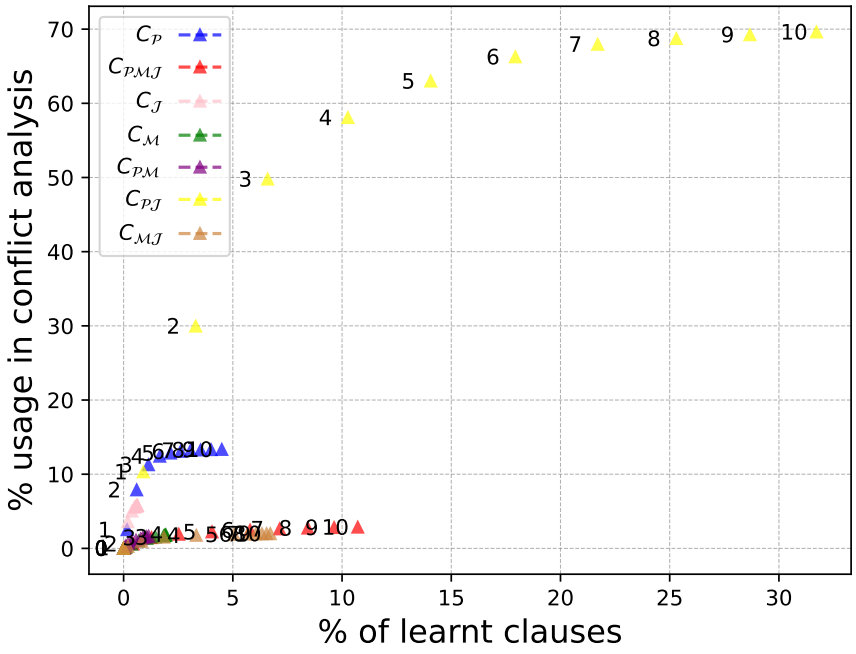


FIGURE 4.1: Measures on the *training benchmark* with MAPLECOMSPS solver, showing learnt clauses usage in *conflict-analysis* phase. Each class of clauses is colored and annotated by its LBD value.

The difference between MAPLECOMSPS and KISSAT-MAB lies in the optimization heuristics and their memory footprint improvements. KISSAT-MAB has demonstrated its efficiency on satisfiable problems, while MAPLECOMSPS remains competitive on unsatisfiable ones.

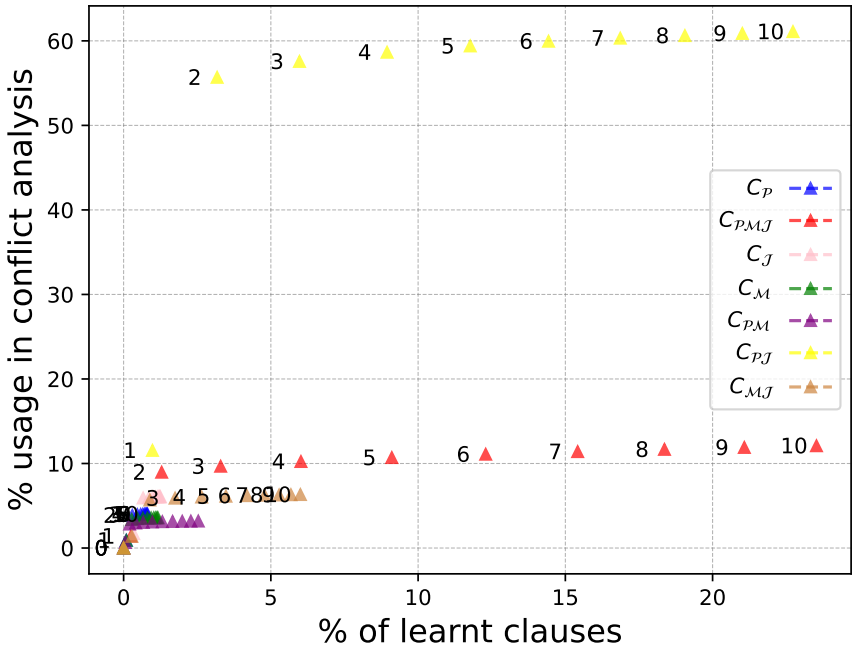


FIGURE 4.2: Measures on the *training benchmark* with KISSAT-MAB solver, showing learnt clauses usage in *conflict-analysis* phase. Each class of clauses is colored and annotated by its LBD value.

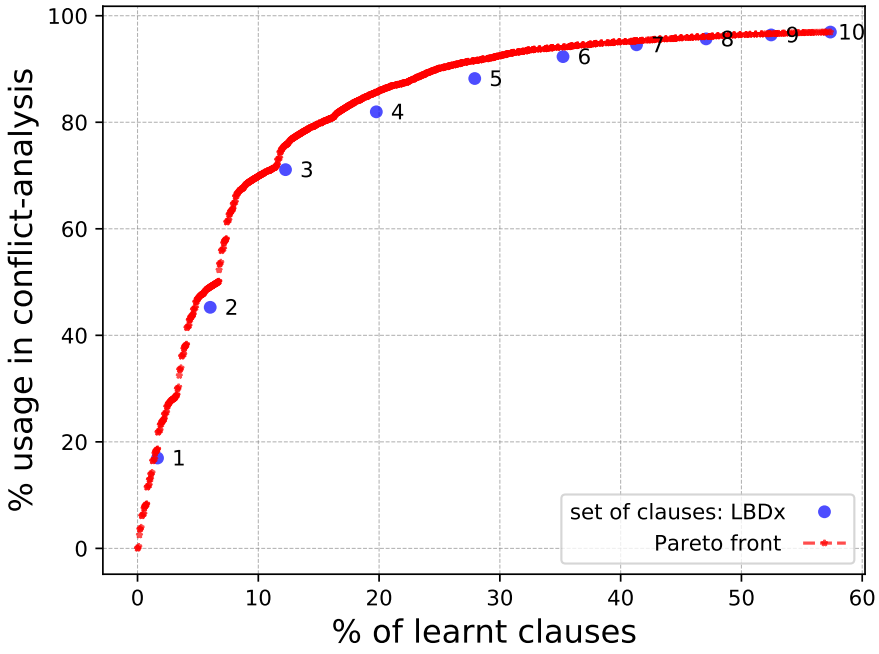


FIGURE 4.3: Measures of learnt clauses usage with MAPLECOMSPS solver, during *conflict-analysis* phase. Blue dots denote LBD while red points depict the Pareto front of H_{LP} strategy.

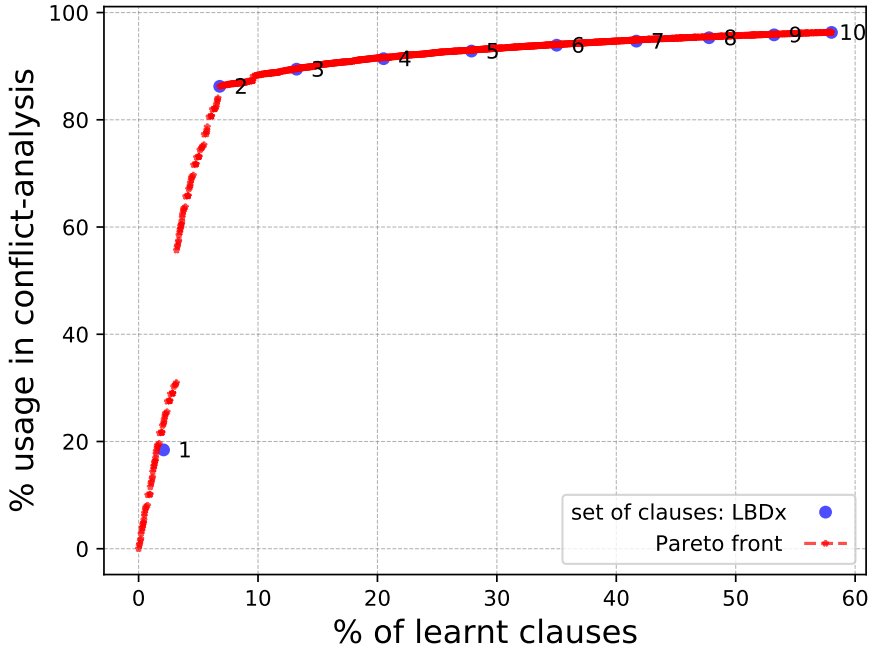


FIGURE 4.4: Measures of learnt clauses usage with KISSAT-MAB solver, during *conflict-analysis* phase. Blue dots denote LBD while red points depict the Pareto front of H_{LP} strategy.

Measures

The experiments were conducted on a benchmark consisting of 620 BMC instances. The distribution of properties within the benchmark is as follows: 50% of Safety problems, 27% Guarantee, 3% Persistence, and 10% Recurrence, based on the Hierarchy of Manna & Pnueli [39]. Additionally, 10% of the instances were generated using Spot [50], including 1% Obligation and 9% Reactivity properties. Trivial instances that were solved in less than 1 second using the MAPLECOMSPS solver were omitted from the benchmark.

The analysis was performed on 23% of the entire benchmark, referred to as the *training benchmark*. For each instance, information related to each learnt clause used in the *conflict-analysis* procedure was logged, including the LBD value and the corresponding class C_X of the clauses. The results are presented in Figure 4.1 for MAPLECOMSPS and Figure 4.2 when using KISSAT-MAB solver.

The *x-axis* of the figures represents the cumulative mean percentage of learnt clauses for the *training benchmark*, while the *y-axis* corresponds to the cumulative mean usage percentage of these clauses during *conflict-analysis* phase. Each point on the graph represents the percentage of learnt clauses of a certain LBD (from 1 to 10) for a particular class of clauses. For example, in Figure 4.1, the yellow triangle with the left annotation "4" indicates that 10% of learnt clauses of class $\mathcal{PJ}$ have an LBD$\leq$4 and were used, in average, 58% of the solving time.

In both MAPLECOMSPS and KISSAT-MAB results, we observe that the $C_{\mathcal{PJ}}$ clauses, which implies the property, has a significant usage rate (around 60%) with a total coverage of approximately 20%. This suggests that these clauses are good candidates for being considered as relevant information in the solving process.

To further investigate the impact of LBD metric on the relevance of the gathered information, we can refer to Figure 4.3 and Figure 4.4. These figures display the same data as the previous ones, but without clause classification. Here, we observe that the default strategy for characterizing relevant information stored in the **core** database on MAPLECOMSPS (LBD$\leq$3), covers 70% of usage for only 12% of the learnt clauses. Similarly, the **core** database selection strategy in KISSAT-MAB (LBD$\leq$2) covers 86% of usage with 7% of the involved learnt clauses. Notably, more than 70% of the relevant information in both solvers is attributed to clauses belonging to the **core** database. However, this observation is somewhat biased since the **core** clauses are never deleted, making them naturally more frequently used than clauses in the **tier-2** and **local** databases.
Interestingly, however, we can observe that, on both solver's figures (Figure 4.1 and Figure 4.2), more than half of the relevant information came from one category of clause: $C_{\mathcal{PJ}}$ clauses.

These findings reinforce the idea that the performance of the SAT solver is heavily influenced by a specific class of clauses. The fine-grained classification C_X reveals that clauses implying the property are more pertinent and crucial for the solving process, while auxiliary clauses are primarily there to connect with the model.

Solver	Heur.	$C_{\mathcal{J}}$	$C_{\mathcal{M}}$	$C_{\mathcal{MJ}}$	$C_{\mathcal{PM}}$	$C_{\mathcal{P}}$	$C_{\mathcal{PMJ}}$	$C_{\mathcal{PJ}}$	Orig. Core
MAPLECOMSPS	H_S	3	3	3	3	**5**	3	3	3
	H_{LP}	**10**	3	3	3	**4**	3	**5**	
KISSAT-MAB	H_S	2	2	2	2	**4**	2	2	2
	H_{LP}	**3**	**3**	2	**3**	**3**	2	**3**	

TABLE 4.1: Selectors computed using H_S or H_{LP} on MAPLECOMSPS and KISSAT-MAB training information.

4.2 Heuristics to identify interesting clauses

Based on the previous studies, we present our strategies for enhancing the resolution of SAT-based BMC problem. Our proposal involves identifying and protecting from deletion new sets of clauses that hold high relevance during the solving process. To achieve this, we introduce two new heuristics H_S and H_{LP}, designed to compute a selector that will effectively highlight the clauses of interest.

4.2.1 Non-automated procedure (H_S)

The heuristic H_S is derived from a simple observation made from the previous figures (Figure 4.1 and Figure 4.2). The idea behind this heuristic is to encourage the solver to prioritize clauses that involve the LTL property variables ($\mathcal{P}$ variables), such as clauses from the $C_{\mathcal{PJ}}$ class, as they appear significantly more often than $C_{\mathcal{P}}$ clauses. Given that model checking procedures heavily depend on the structure of the LTL property, protecting additional clauses that implicate the property in SAT learning can only be of benefit.

To implement this idea, we expand the **core** database of the solver with a subset of clauses from $C_{\mathcal{PJ}}$. Specifically, we select all clauses from $C_{\mathcal{PJ}}$ that have an LBD$\leq$5 for MAPLECOMSPS and LBD$\leq$4 for KISSAT-MAB, resulting in the selectors shown in the first line of each solver in Table 4.1. Table 4.1 summarizes the LBD values for each class of clauses, highlighting in bold the values that have changed compared to the base selector (last column) for each solver. The base selector represents a single LBD value because it does not take into account the clause classification metric.

The LBD thresholds were chosen based on the inflection points observed in the curves of the above figures. These inflection points indicate that beyond a certain LBD value, no further relevant information is captured. Hence, selecting clauses with LBD values below the threshold ensures that the focus is on the most pertinent and useful clauses while avoiding the inclusion of less useful ones.

4.2.2 Semi-automated procedure (H_{LP})

This approach aims to mathematically predict the usefulness of each learnt clause. It involves determining the appropriate selector by optimizing the overall usage of learnt clauses while minimizing their quantity. This optimization is achieved through solving a linear programming system, which yields multiple solutions. The procedure is termed semi-automated since it requires an expert inputs to select the most suitable solution among the possible solutions.

The linear program is formulated with the an objective that combines the two aforementioned criteria: maximizing the usage with a minimum number of involved learnt clauses. The constraints of the linear programming system restrict the search space to selecting at most one LBD value per category of clauses. The input information for these constraints is extracted from the analysis results presented in Figure 4.3 and Figure 4.4.
To describe the linear system, we introduce the following notations:

- u_i^X: the percentage of learnt clauses (*x-axis*) with LBD≤i of class X.
- v_i^X: the percentage usage of learnt clauses (*y-axis*) with LBD≤i of class X.
- x_i^X: a Boolean variable representing the decision variable of the linear system. It takes the value 1 if the LBD≤i is chosen for the class of clauses X, 0 if not.
- $\mathcal{S}$=$\{\mathcal{P}, \mathcal{M}, \mathcal{J}, \mathcal{PM}, \mathcal{PJ}, \mathcal{MJ}, \mathcal{PMJ}\}$ denotes the set of classes.

Hence, our modeling of the optimization problem is as follows:

$$\textbf{maximize } \mathbf{f}_\mu = (\mu - 1) \overbrace{\sum_{i=1}^{10} \sum_{X \in \mathcal{S}} u_i^X \, x_i^X}^{\mathbf{O}_1} + \mu \overbrace{\sum_{i=1}^{10} \sum_{X \in \mathcal{S}} v_i^X \, x_i^X}^{\mathbf{O}_2}$$

$$\textbf{subject to} \begin{cases} \sum_{i=1}^{10} x_i^j \leq 1 & \forall X \in \mathcal{S} \qquad \textit{//At most one LBD value per class} \\ x_i^X \in \{0,1\} & \forall i \in [\![1; 10]\!], \forall X \in \mathcal{S} \end{cases}$$

The function $\mathbf{f}_\mu$ is the aggregation function, defined as a weighted sum and parameterized with a value μ ($0 \leq \mu \leq 1$). It combines two criteria: $\mathbf{O}_1$, representing the number of learnt clauses to be minimized (switched into maximization in the objective f_μ, by reversing the sum's sign), and $\mathbf{O}_2$, the total usage percentage of learnt clauses to be maximized. It is noteworthy that other aggregation functions, such as Ordered Weighted Average or Choquet integral, could also be used for this purpose.

The bi-objective optimization problem is transformed into a single maximization problem that can be solved with different values for parameter μ. In fact, the larger the value of μ, the more importance is given to searching for a solution that optimizes criterion O_2, and vice-versa. By solving this system with different values of μ with the data collected from the *training benchmark* analysis, we can draw a *Pareto front* representing a set of optimal solutions where no other solution exists that improves one criterion without degrading another and there are no other solutions that are better in both criteria. The Pareto front of the data collected from MAPLECOMSPS experiments (resp. KISSAT-MAB) is highlighted with red points in Figure 4.3 (resp. Figure 4.4). Each red point on the front corresponds to a specific selector (*i.e.*, solution) derived from the optimization procedure with a specific μ value. However, it's important to note that the resulting Pareto front may not be the optimal front due to the variation of μ with a fixed increment of 0.001. Thus, there may be better solutions that we didn't generate. Nevertheless, since our increment value is quite small, the set of solutions obtained tend to an (almost) optimal front solutions.

Solver	UNSAT	SAT	TOTAL	PAR-1	PAR-2	CTI
MAPLECOMSPS	255	113	368	523h00	943h00	**91h50**
MAPLECOMSPS-LBD≤4	254	114	368	522h00	942h00	**91h50**
MAPLECOMSPS-H_S	253	116	369	524h40	943h00	94h50
MAPLECOMSPS-H_{LP}	**258**	**117**	**375**	**521h30**	**929h50**	93h30
KISSAT-MAB	310	**144**	454	384h20	661h00	86h20
KISSAT-MAB-LBD≤3	302	142	444	397h00	690h20	92h30
KISSAT-MAB-H_S	311	142	453	386h10	664h30	84h20
KISSAT-MAB-H_{LP}	**319**	142	**461**	**379h40**	**644h40**	**80h40**

TABLE 4.2: Comparison between state-of-the-art MAPLECOMSPS and KISSAT-MAB solvers and H_S and H_{LP} heuristics. MAPLECOMSPS-LBD≤4 (resp. KISSAT-MAB-LBD≤3) uses a strategy where learnt clauses with LBD≤4 (resp. LBD≤3) are considered as relevant.

Our analysis reveals that the red points on both Figure 4.3 and Figure 4.4, which represent our proposed approach, dominate the blue points (representing the standard LBD-based approach). This suggests that our filter, which combines the LBD metric and the classification of clauses, can significantly improve the solver's performance by choosing one of these points as a basis for detecting new relevant information.

To determine the most promising point on the Pareto front, we focus on the inflection point of the curves. For MAPLECOMSPS, the most promising point is located between the blue points tagged with 3 and 4, which correspond to LBD≤3 and LBD≤4, respectively. This promising point covers 81.8% of usage for a total of 16.5% involved learnt clauses. The selector associated with this point characterizes the clauses with properties shown in the second line of Table 4.1: it fixes an LBD≤3 for all classes except $C_{\mathcal{P}}$, $C_{\mathcal{J}}$, and $C_{\mathcal{PJ}}$, which have LBD≤4, LBD≤10, and LBD≤5, respectively. For KISSAT-MAB, the most promising point is located between the blue points tagged with 2 and 3, which correspond to LBD≤2 and LBD≤3, respectively. This promising point covers 87.3% of the *conflict-analysis* for a total of 9.57% of learnt clauses. The selector associated with this point is shown in the last line of Table 4.1. It has LBD≤3 for $C_{\mathcal{M}}$, $C_{\mathcal{J}}$, $C_{\mathcal{PM}}$, and $C_{\mathcal{PJ}}$, and LBD≤2 for the remaining classes.

These results confirm the importance of clauses with LBD≤3 (for MAPLECOMSPS) and LBD≤2 (for KISSAT-MAB) as identified in the standard approach but also reveal new interesting ones.

4.3 Experimental Evaluation of BMC-based Selectors

To assess the relevance of our heuristics on the learnt clause deletion component of CDCL-like solvers, we conducted experiments on 620 BMC instances (detailed in Section 4.1). Each instance was executed on an Intel Xeon@2.40GHz machine with 12 processors and 64 GB of memory, with a time limit of 6000 seconds, using the BSALTIC tool.

Table 4.2 summarizes these results, comparing the basic strategies of MAPLECOMSPS and KISSAT-MAB solvers with their versions that integrate the H_S and H_{LP}

heuristics. The table displays the number of UNSAT (unsatisfiable) and SAT (satisfiable) solved instances, the total number of solved instances, the cumulative time (PAR-1) and PAR-2 metric[2]. The last column gives the Cumulated Execution Time of the Intersection (CTI) of solved instances by all solvers. It consists of 357 instances for MAPLECOMSPS experimentation and 433 for KISSAT-MAB. The heuristics H_S and H_{LP} do not include pre-processing time, which corresponds to 68 hours in MAPLECOMSPS experiments and 47 hours in KISSAT-MAB. The computation time of the Pareto front from resolving the linear systems $\mathbf{f}_\mu$ of the H_{LP} heuristic does not take more than one second.

From the first line of the table, we can observe that the state-of-the-art MAPLECOMSPS solves 368 instances with a PAR-2 metric of 943 hours. Increasing the **core** database to protect learnt clauses with LBD$\leq$4 (MAPLECOMSPS-LBD$\leq$4) yields similar results. This indicates that while some problems were not overwhelmed by the additional clauses (with LBD=4), simply increasing the number of relevant clauses based solely on LBD does not significantly improve performance.

The next two lines display the results using the H_S and H_{LP} approaches, respectively. Both outperform the state-of-the-art approach: MAPLECOMSPS-H_{LP} shows a notable improvement with a gain of 13 hours in the PAR-2 metric, solving 4 UNSAT and 2 SAT instances more. MAPLECOMSPS-H_S gave a PAR-2 metric similar to the original strategy but manages to solve 1 additional SAT instance. However, the PAR-1 and CTI times of these modified solvers are slightly slower than the original solver, with at most 1 hour 40 minutes more on the PAR-1 and 3 hours on the CTI metric. These variations are left to the user consideration, whether it is the number of solved instances to prioritize or the limited time to solve them.

The remain segment displays the experimentation results on the KISSAT-MAB engine. The state-of-the-art solver solves 454 instances with a PAR-2 of 661 hours. However, augmenting the core database with LBD$\leq$3 (KISSAT-MAB-LBD$\leq$3) deteriorates the solver's performance: 10 instances less with a PAR-2 (resp. CTI) of 29 hours (resp. 6 hours) slower. The last two lines present its modified versions. KISSAT-MAB-H_S solves 1 UNSAT instance more but 2 instances less than the state-of-the-art, whereas KISSAT-MAB-H_{LP} solves 9 UNSAT instances more but 2 SAT instances less and manages to decrease the PAR-2 time to 3 hours.

In conclusion, both solvers perform better with the H_{LP} heuristic. KISSAT-MAB seems more suitable when solving BMC problems, especially UNSAT instances, known to be challenging. The two heuristics highlight the importance of the information captured by $C_{\mathcal{PJ}}$: when performing the model-checking approach, a synchronous product between the Kripke structure of the model and the automaton of the (negated) property is executed. Thus, forcing the SAT procedure to consider property clauses will eliminate invalid paths in the property automaton, leading to a smaller synchronized product. Additionally, H_{LP} captures other important information such as: $C_{\mathcal{J}}$ (and $C_{\mathcal{M}}$ when considering KISSAT-MAB) configuration. $C_{\mathcal{J}}$ is composed of auxiliary variables used for the conversion of the problem into a CNF formula [55], making the connection between the property and the model. Consequently, they also help to compute information related to the synchronous product.

[2]PAR-k is a measure used in SAT competitions that penalizes the run-time, counting each timeout as k times the running time cutoff.

4.4 BMC-based Sharing strategy

The majority of works on parallelizing SAT-based BMC applies to different unrolling depths that spawn multiple solvers in parallel. These employ portfolio-based schemes where each worker solves the same BMC problem but with a different bound k [22, 128, 129]. However, to the best of our knowledge, no work has attempted to tune the clause exchange policy between workers for the sake of BMC. Some works [134, 135] have considered the symmetry structure of the formula to decompose it into partitions to parallelize the resolution procedure, where each worker handles a portion of the formula. This can result in exchanged clauses that could be useful, as each worker focuses on a part (one unrolling depth), and the learnt clauses produced might be duplicated due to the symmetry of the formula. Instead of embarking on complex procedures that involve partitioning the formula or dealing with multiple unrolling depths k, we turned our attention to the filtering of shared clauses between CDCL solvers that work on the same bound k. We propose the first BMC-based portfolios that use specialized sharing strategies for exchanging clauses.

In Chapter 3, we identified seven classes of clauses and utilized them in the context of sequential SAT solving to develop two new heuristics designed for BMC (Section 4.2). These heuristics aim to protect relevant clauses from deletion. In this section, we suggest applying a similar approach in the parallel SAT solving context. Through a portfolio of concurrent workers, we propose modifying the metric used for limiting the exchanged clauses between workers by incorporating the clause classification measure. More precisely, the filter that characterizes the clauses of interest to exchange is the selector from prior studies.
Currently, procedures in parallel SAT solving employ generic metrics to identify clauses to share, with the most common one being the LBD value. In this strategy, only learnt clauses with a fixed LBD threshold are shared between solvers, without distinguishing between classes C_X. However, as observed in previous analyses (Section 4.1), some classes of clauses are more valuable for the solving process than others. To leverage the importance of these relevant classes and significantly increase their sharing, we propose adjusting their corresponding LBD threshold.

We introduce the SH_{LP} sharing strategy that exports clauses identified by H_{LP} selector, w.r.t, the used solver (Table 4.1). For example, $C_{\mathcal{J}}$ clauses with an LBD$\leq$10 are allowed to be shared among other MAPLECOMSPS workers.

4.5 Parallel Experiments

SAT strat.	Sharing strat.	UNSAT	SAT	TOTAL	PAR-1	PAR-2	CTI (472)
MAPLECOMSPS	HORDESAT	357	**120**	477	342h50	590h35	107h20
	SH_{LP}	**362**	**120**	**482**	**339h20**	**578h50**	**106h50**
MAPLECOMSPS-H_{LP}	HORDESAT	361	119	480	343h05	585h50	107h00
	SH_{LP}	**362**	**120**	**482**	341h20	580h40	108h00

TABLE 4.3: Comparison between state-of-the-art MAPLECOMSPS Portfolio and our modified portfolio MAPLECOMSPS-H_{LP} for various sharing approaches.

In this section, we evaluate the impact of the BMC-based sharing heuristics in a parallel portfolio setting. All experiments were conducted on the same benchmark as

the sequential evaluation, using an Intel Xeon@2.40GHz machine with 12 processors, 64 GB of memory, and a time limit of 6000 seconds. The portfolios consist of 10 CDCL solvers (workers).

Table 4.3 presents the results of different portfolios: MAPLECOMSPS and MAPLECOMSPS-H_{LP} based portfolios. The latter uses our modified version of MAPLECOMSPS, which incorporates the H_{LP} selector described in Section 4.2. For each portfolio, workers vary in their diversification strategy [104].

The table displays the different sharing strategies: SH_{LP} and HORDESAT [114], which dynamically increases the LBD threshold based on certain constraints. This strategy is employed by the best parallel solver of the parallel track in the SAT competition 2021. For each strategy, the number of solved UNSAT and SAT instances, the PAR-1, PAR-2, and CTI metrics are displayed.
We divide the analysis of the results into two parts:

Choice of the Sharing strategy

The first observation pertains to the sharing strategy; comparing the HORDESAT strategy to our proposed SH_{LP} strategy:

MAPLECOMSPS. When employing the BMC-based sharing strategy (SH_{LP}) in a MAPLECOMSPS portfolio, it outperforms the state-of-the-art sharing strategy HORDESAT. Specifically, SH_{LP}solves 5 more UNSAT instances and reduces the running time by 12 hours (PAR-2) and 30 minutes for the CTI.

MAPLECOMSPS-H_{LP}: Similarly, when using the SH_{LP} sharing strategy in a MAPLECOMSPS-H_{LP} portfolio, it outperforms the HORDESAT strategy on both SAT and UNSAT instances. The SH_{LP} strategy solves 2 more instances in total, with a PAR-2 time reduced by 5 hours.

These results indicate that using a BMC-based sharing strategy, tailored to the specific characteristics of the BMC problem, yields better performance compared to generic strategies. The BMC-based sharing strategy, SH_{LP}, effectively identifies and shares relevant clauses, leading to improved solving efficiency in the parallel portfolio setting.

Figure 4.5 provides a detailed view of the comparison between the state-of-the-art portfolio and our best approach. The scatter-plot illustrates the solving time (in seconds) for each solved instance. Red dots represent UNSAT instances and blue ones for SAT instances. The *y-axis* represents the solving time for our best portfolio, while the *x-axis* represents the solving time for the state-of-the-art portfolio.

The scatter-plot clearly shows that a MAPLECOMSPS-based portfolio using the SH_{LP} sharing strategy performs significantly faster than the generic portfolio, especially on UNSAT problems. The majority of the red dots (representing UNSAT instances) are located below the diagonal line, indicating that SH_{LP} strategy leads to faster solving times for these instances compared to the state-of-the-art strategy.

This visual comparison provides further evidence that the BMC-based sharing strategy, SH_{LP}, significantly improves the performance of the parallel portfolio.

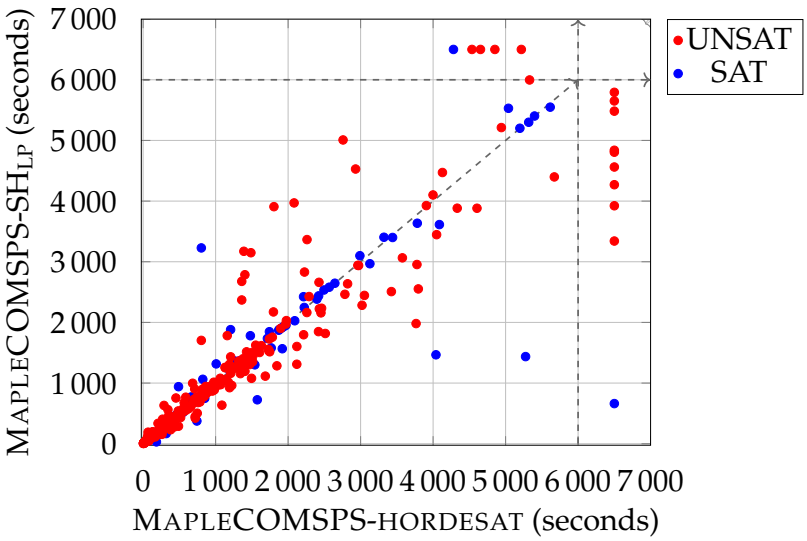


FIGURE 4.5: Scatter-plot comparing state-of-the-art portfolio (MAPLECOMSPS using HORDESAT-strategy) to our best one (MAPLECOMSPS with SH$_{LP}$-strategy)

Choice of the SAT solver

The second observation is related to the choice of the SAT solver depending on whether it integrates the H$_{LP}$ heuristic:

HORDESAT Sharing. Among the tested configurations, the MAPLECOMSPS-H$_{LP}$ portfolio stands out as the most suitable solver with 480 solved instances. This is an improvement over the generic MAPLECOMSPS portfolio, which solves 3 instances less. This result highlights the effectiveness of using our tuned solver for the HORDESAT strategy, leading to better performance in terms of the number of solved instances.

SH$_{LP}$ Sharing. In this case, both engines (MAPLECOMSPS and MAPLECOMSPS-H$_{LP}$) achieve similar results in terms of the number of solved instances, with 362 UNSAT and 120 SAT instances solved. However, the original MAPLECOMSPS-based portfolio (without tuning) stands out by achieving a reduction in running time (PAR-1, PAR-2, and CTI) by at least 2 hours. This finding suggests that the information captured by the tuned SAT solver may not contribute significantly to improving the sharing strategy's performance. Thus, focusing on tuning the sharing strategy may be sufficient when using a MAPLECOMSPS solver.

We conclude that the combination of a BMC-based SAT solver and a tuned sharing strategy for the MAPLECOMSPS portfolio is the best configuration, which performs better overall on both SAT and UNSAT instances.

PARKISSAT-RS experiments

Based on the above results, we go further and conduct the experiments displayed in Table 4.4. These are realized using a parallel portfolio PARKISSAT-RS, the winner of the the Parallel SAT Competition 2022 that uses Kissat-MAB as a back-end engine, and we compared it when using our tuned Kissat-MAB-H$_{LP}$ solver, namely

SAT strat.	Sharing strat.	UNSAT	SAT	TOTAL	PAR-1	PAR-2	CTI (514)
PARKISSAT-RS	LBD≤2	368	148	516	248h00	429h20	81h30
	SH_{LP}	372	149	521	246h40	416h20	79h25
PARKISSAT-RS-H_{LP}	LBD≤2	373	149	522	248h40	415h40	80h10
	SH_{LP}	**375**	**150**	**525**	**244h10**	**405h40**	**77h20**

TABLE 4.4: Comparison between state-of-the-art PARKISSAT-RS Portfolio and our tuned portfolio PARKISSAT-RS-H_{LP} for various sharing approaches.

PARKISSAT-RS-H_{LP}. PARKISSAT-RS uses the PAINLESS framework [137] to implement the clause sharing method, and each thread shares clauses with LBD≤2. Diversification is also used. The experiments conducted using the parallel portfolio PARKISSAT-RS, which uses KISSAT-MAB as a back-end engine, provide further insights into the effectiveness of the SH_{LP} sharing strategy:

The results in Table 4.4 show that the state-of-the-art PARKISSAT-RS portfolio, which uses static sharing with LBD≤2, solves 516 instances and achieves a PAR-2 metric of 429 hours. On the other hand, our best portfolio PARKISSAT-RS-H_{LP} with SH_{LP} sharing solves 9 instances more with a faster PAR-2 of 23 hours. This finding indicates that the PARKISSAT-RS-H_{LP} portfolio, with the SH_{LP} sharing strategy, is the most suitable configuration for parallelism when using the KISSAT-MAB solver.

The detailed analysis of the experiments focusing on fixed SAT solvers reveals the following insights:

PARKISSAT-RS. The best configuration in this case employs the SH_{LP} sharing strategy, which solves 5 instances more and reduces the PAR-2 time by 13 hours, as well as the CTI time by 2 hours.

PARKISSAT-RS-H_{LP}. When tuning both the SAT solver and the sharing mechanism, the performance is even better compared to sharing only clauses of LBD≤2. The PARKISSAT-RS-H_{LP} portfolio solves 3 instances more and reduces the computation time on the PAR-2 metric by 4 hours.

The experiments support the conclusion that BMC-based sharing strategies indeed enhance the solving performance in terms of both the total number of solved instances and the computation time. When combined with a tuned SAT solver, the effectiveness of these sharing strategies is further amplified, resulting in better overall performance and efficiency.

The scatter-plots in Figure 4.6 provides a visual comparison between PARKISSAT-RS with LBD≤2 sharing to our best one (PARKISSAT-RS-H_{LP} with SH_{LP}). Based on these comparisons and the detailed analysis, the following observations on the sharing approach can be made:

LBD≤2 Sharing. The tuned portfolio, PARKISSAT-RS-H_{LP}, outperforms the original portfolio, PARKISSAT-RS, when both use the LBD≤2 sharing strategy. PARKISSAT-RS-H_{LP} solves 5 more UNSAT instances and 1 more SAT instance compared to PARKISSAT-RS. Additionally, it reduces the computation time (PAR-2) by 13 hours and PAR-1 and the CTI time by 1 hour.

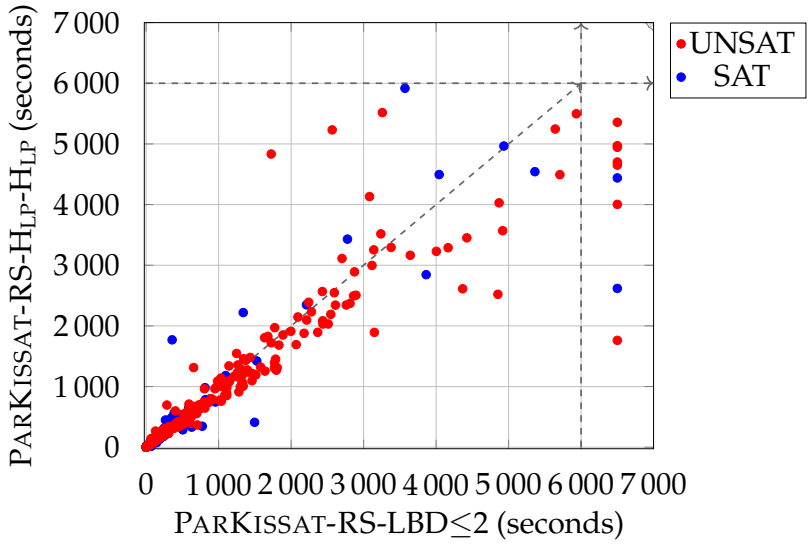


FIGURE 4.6: Scatter-plot comparing state-of-the-art portfolio (PARKISSAT-RS that shares clauses of LBD≤2 only) to our best one (PARKISSAT-RS-H_{LP} with H_{LP}-strategy)

SH_{LP} Sharing. PARKISSAT-RS-H_{LP} remains the best performer. It solves 4 more UNSAT instances and reduces the computation time (PAR-2) by 10 hours compared to the state-of-the-art PARKISSAT-RS.

In conclusion, the observations derived from Tables 4.3, Table 4.4, Figure 4.5 and Figure 4.6 confirm that incorporating components specifically tuned for BMC (sharing heuristics, relevant clauses heuristics) has a significant impact on the performance and efficiency of parallel SAT solvers. Utilizing BMC-based sharing strategies and relevant clauses heuristics can lead to a substantial improvement in the number of solved instances and computation time, making them valuable additions to parallel SAT solving portfolios.

4.6 Global conclusion

Our journey towards building new heuristics for SAT procedures started with the observation that the relevant information used by SAT solvers can be refined. We proposed a generic methodology to classify learnt clauses and we applied it to the special case of BMC using two of the best SAT solvers: MAPLECOMSPS and KISSAT-MAB. These learnt clauses have been classified according to their meaning in the original problem which helped us to suggest two heuristics (H_S and H_{LP}) based on the information carried by the LTL property. The two heuristics improve the state-of-the-art approach, with the particularity of having a structural reasoning behind H_S heuristics. In the other hand, the procedure used to build H_{LP} relies on a mathematical reasoning.

We extended this idea to the context of parallel BMC solving by offering a new sharing strategy, namely SH_{LP}. This focuses on tuning two, out of the three, components of a parallel SAT solving, *i.e.*, the sharing strategy and the SAT strategy.

Our ongoing work aims to refine the proposed classification by exploiting the specification of the property (using the Hierarchy of Manna & Pnueli).

Chapter 5

Decomposition-based BMC

Contents

Some of the state-of-the-art SAT-based BMC techniques presented in Chapter 3 have demonstrated promising results, that have helped to guide the SAT procedure towards interesting search spaces, thereby reducing the solving time. One such optimization involves decomposing the propositional formula 1.1 into multiple subformulas. We explore a clause learning framework that leverages Craig interpolation [36]: Starting from the SAT formula representing the BMC problem to be solved, the idea here is to decompose the formula into parts. Then, it is a matter of enriching the problem with information that are not explicitly encoded. These information are derived from the analysis of the relationships between the different parts. This is accomplished through the exploitation of interpolants.
In pursuit of our objectives, we revisit the concept presented in [37], which outlines a reconciliation scheme employing interpolation. Their primary aim was to address the challenges of solving exceptionally large formulas in distributed environments. We have adapted this reconciliation scheme to suit our purposes. Firstly, we introduce a novel decomposition method tailored specifically for BMC problem-solving. Secondly, we harness the interpolation mechanism as a means to generate learnt clauses, thereby accelerating SAT solving. This enables us to leverage the knowledge acquired from BMC-based decomposition in two distinct dimensions:

- **Interpolants as a Preprocessing Engine**: Our approach harnesses the introduction of interpolants before the solving, as preprocessing procedure, within a sequential context. This novel utilization of interpolants enhances the efficiency of the SAT solving process by introducing valuable clauses derived from interpolation.

- **Interpolants in a Parallel Environment**: To further leverage the wealth of information provided by interpolants, we extend the application of interpolants into parallel environments by sharing these interpolants to guide other CDCL solvers towards promising search spaces. This collaborative approach among solvers improves their collective reasoning capabilities and ultimately leads to more efficient problem-solving.

Our contribution includes the development of a decomposition-based BMC solver (BMC-D), a versatile component that functions as a clause generator using interpolation techniques in both sequential and parallel environments. BMC-D leverages the structural properties of the BMC formula by partitioning it into multiple segments, enabling the generation of highly relevant interpolants.
Considering that interpolation computation are generally time consuming, the interpolants obtained from the random decomposition approach introduced in [37] (LZY-D) and our BMC-based decomposition (BMC-D) can be used during preprocessing to improve sequential solving efficiency. Additionally, one can share these interpolants within a portfolio of CDCL solvers to enhance effective communication and collaboration among them.

The chapter is structured as follows: We start by reviewing the reconciliation procedure of [37]. This procedure employs communication mechanisms among these subformulas based on Craig interpolation mechanisms [36]. This is discussed in Section 5.1. Section 5.2 recalls the random decomposition presented by the authors [37] and investigates the adaptation of this approach within the context of BMC in subsection 5.2.2. Section 5.3 and 5.4 study the effectiveness of interpolation-based learning on both sequential an parallel settings, respectively.

5.1 An Interpolant-based decision procedure

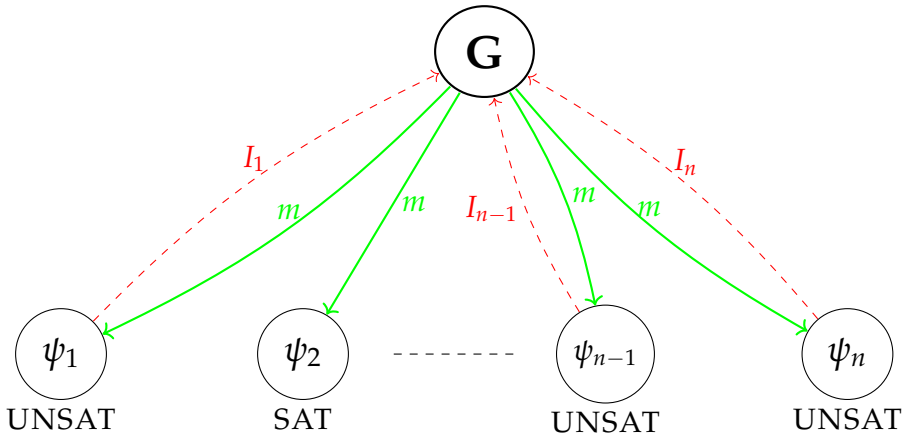


FIGURE 5.1: Reconciliation scheme

The section outlines the framework proposed in [37], describing an interpolation-based decision procedure for SAT formulas. We start by defining the notion of interpolation, then we delve into the reconciliation framework.

5.1.1 Craig Interpolation

The Craig interpolation theorem [36] provides a powerful tool for analyzing the relationship between two formulas A and B in the context of satisfiability. It guarantees the existence of an interpolant when $A \wedge B \implies \bot$, allowing us to extract additional information about the logical structure of A and B.

Given an unsatisfiable conjunction of formulas, specifically $A \wedge B$, an interpolant, denoted as I, is a formula that adheres to the following properties:

- $A \implies I$: This implies that if A holds true, then I must also be true.
- $B \wedge I \implies \bot$: The conjunction of B and I is unsatisfiable, indicating that there is no assignment of variables that simultaneously satisfies both B and I.
- I is defined over the common language of A and B: I is constructed using variables that appear in both A and B, ensuring that it captures the relevant information shared by both formulas.

The interpolant I provides an over-approximation of formula A while still conflicting with formula B. It can be thought of as a logical abstraction of A that captures the essential features needed to demonstrate the conflict with B.

While the Craig interpolation theorem guarantees the existence of an interpolant, it does not provide an algorithm for finding it. However, there are known algorithms for generating interpolants for various logics. One common approach is to derive an interpolant for $A \wedge B$ from a *proof of unsatisfiability* of the conjunction. By analyzing the proof structure, it is possible to extract the necessary information to construct the interpolant. Importantly, these algorithms have the advantage of operating in linear time relative to the size of the proof, making them efficient for practical use.

In the context of SAT procedures, McMillan's interpolation [58] has been widely employed. It has been shown to be competitive with SAT solving algorithms and can provide effective interpolants for Model Checking problems. To gain a deeper understanding of McMillan's interpolation and other interpolation systems, a complete study can be found in [139].

It is worth noting that interpolants are generally not in clausal form. If an interpolant is to be retained as part of the problem (as learnt clauses, for example), it may require conversion. Simply expanding the interpolant into CNF can lead to exponential growth in its size. The Tseitin transformation [56] provides an alternative method that introduces new variables while maintaining a linear increase in size.

5.1.2 Reconciliation algorithm

Apart from the usage of interpolation mechanisms in IC3 [60, 140], PDR [61], some incremental SAT-based BMC [141–143], to the best of our knowledge, no one has explored their usage on one single BMC instance. Used either during a preprocessing phase where interpolants are injected to the original problem or shared among classical CDCL solvers in a parallel setting. The most closely work related to ours is [143].

This preliminary research [143] proposes to extract information from an interpolant-based model-checking engine during the solving process (in-processing). They manage to derive an over-approximation of fixed time frames with the aim of early detecting invalid variable assignments at a specific time frame. This initial study can provide additional insights when integrated with our ongoing work, where preprocessing and in-processing interpolation procedures are simultaneously applied. However, it's worth noting that the effectiveness of this approach for various types of specifications is not known, as their study was exclusively applied to invariant properties, while our approach is applied for any type of LTL property.

Hamadi et al. [37] used an interpolation-based technique when treating formulas that are too large to be handled by a single computing unit. To achieve this, they propose decomposing the formulas into partitions that can be solved by individual computing units. Once each partition is solved, the partial results are combined using Craig interpolation [36] (McMillan [58] or HKP [144] interpolation computation techniques), to obtain the final solution. Craig interpolation is chosen for its effectiveness in reconciling partial results and its ability to enable arbitrary splitting of formulas without restrictions on the nature or size of the cut. The paper demonstrated the effectiveness of this approach in handling large formulas by comparing its efficiency to state-of-the-art CDCL solver techniques. However the proposed approach did not use any structural information of the problem at hand.

The main idea is to compute partial solutions for different parts of the formula at hand and then calculate a global solution using interpolation mechanisms. Indeed, they implemented the reconciliation schema depicted in Figure 5.1, where each partition of the formula is reconciled through the variables they share.

Let's denote by ψ_i for $i = 1, \ldots, n$, the subformulas of the studied formula $\mathcal{F}$. These subformulas are solved by individual SAT solver units. However, when the partitions happen to share variables (*i.e.*, $\mathcal{V}(\psi_i) \cap \mathcal{V}(\psi_j) \neq \emptyset$, for some $j \neq i$), the resolutions of the different partitions must be reconciled and synthesized into a feasible global solution.

To do so, another solver is added to the schema. Named as the manager G, it is responsible for finding a global solution accepted by each partition ψ_i. It reconciles the solutions returned by each partition into a feasible global solution that satisfies the entire formula $\mathcal{F}$. The reconciliation procedure is build using the following lemma:

Lemma 1. Let $\mathcal{F} = \psi_1 \wedge \cdots \wedge \psi_n$ and let m be a model for G that covers the shared variables of the decomposition, $\mathcal{V}(G) = \bigcup_{i,j=1}^{n} \big(\mathcal{V}(\psi_i) \cap \mathcal{V}(\psi_j)\big)$. Let $1 \leq i \leq n$, if I is an interpolant for $\neg(\psi_i \wedge m)$, then $\mathcal{F} \Longrightarrow I$.

The resulting interpolant I (red arrows in Figure 5.1) from $\neg(\psi_i \wedge m)$ is incorporated into G eliminating the model m (green arrows in Figure 5.1) in future iterations.

Using the above lemma, authors of [37] present a reconciliation algorithm, for solving SAT problems (in a distributed manner) using any decomposition method. The algorithm takes as input a CNF formula $\mathcal{F}$ and a number of partitions n.

The entire procedure has been implemented in a framework called DESAT[1]. It integrates various interpolation algorithms (ex. McMillan [58], HKP [144], *etc.*) and several decomposition methods, including the *lazy decomposition* that will be detailed in Section 5.2.1. DESAT uses MINISAT1.14P [83] as a core engine. Compared to newer versions of MiniSat, this older version has the ability to provide a *proof of unsatisfiability*.

Discussion. As previously mentioned, Hamadi et al.'s approach was originally designed as a comprehensive decision procedure for satisfiability problems. However, it heavily relies on computationally intensive methods, with interpolation being the primary one. This characteristic diminishes its practical feasibility for direct application. Nevertheless, it is entirely conceivable to adapt and repurpose this approach for other purposes.

One potential utility lies in using it as a preprocessing tool that furnishes insights about the problem at hand, thereby assisting classical SAT solvers in achieving more efficient resolutions.

Hence, the idea we will explore here is to leverage the interpolants generated by a (potentially partial) execution of this approach as a set of auxiliary information that can be incorporated into the original problem. This additional information can enhance the resolution process. We will delve into this concept further in both sequential and parallel contexts in the remainder of the paper.

5.2 Decomposition-based strategies

To formally explore the idea mentioned earlier, we begin by examining how the formula is decomposed. We first study the initial decomposition proposed in [37], and then we detail our new splitting method tailored to the BMC problem.

Throughout this work, we maintain using the DESAT framework (cited in Section 5.1) for all the presented experiments since it has already in place the interpolation algorithms and the reconciliation mechanism. The integration of modern CDCL SAT solver, like KISSAT-MAB [33], is in our perspective.

5.2.1 Lazy Decomposition (LZY-D)

The process of finding sparsely connected partitions in a formula and eliminating connections to make the partitions independent is not a straightforward operation. Hamadi et al. [37] propose a computationally-free decomposition (LZY-D), known as *lazy decomposition*:

Definition Lazy Decomposition (LZY-D). Let $\mathcal{F}$ be in conjunctive normal form of q clauses, *i.e.*, $\mathcal{F} = F_1 \wedge \cdots \wedge F_q$. A lazy decomposition of $\mathcal{F}$ into n partitions is an equivalent set of formulas $\psi_1, \ldots, \psi_n$, where each ψ_i is equivalent to some conjunction of clauses from $\mathcal{F}$. In other words, there exist integers a and b (with $a < b < n$) such that $\psi_i = F_a \wedge \cdots \wedge F_b$.

[1] https://www.winterstiger.at/christoph/

The lazy decomposition approach (LZY-D) does not explicitly enforce independence among the partitions. Instead, it divides the clauses of the problem into a number of equally sized partitions. The clauses are ordered as they appear in the input file, and each partition ψ_i is assigned the clauses numbered from $i \cdot \lfloor \frac{q}{n} \rfloor$ to $(i+1) \cdot \lfloor \frac{q}{n} \rfloor$.

This decomposition allows for the separate processing and solving of each partition, without explicitly ensuring their independence. Consequently, the resulting subformulas $\psi_1, \ldots, \psi_n$ can be solved using parallel computing devices by applying the reconciliation procedure discussed in Section 5.1.

5.2.2 BMC Decomposition (BMC-D)

Cutting the set of clauses randomly remains a generic approach which do not utilize the knowledge of the problem's structure. Indeed, consider the specific structure of the BMC problem, characterized by a finite state system. As the system operates in discrete states, each state can be represented by a set of variables and its corresponding constraints in the SAT formula. It becomes evident that isolating each state encoding from the SAT formula is a straightforward task. By leveraging this inherent structure, we can partition the SAT-based BMC formula into subformulas based on the $k+1$ states of the system (k steps + initial state). This finer decomposition will allows us to create (relatively) more independent and smaller subproblems. Moreover, it enables the extraction of relevant information about the system's behavior to easily split potential error paths through the generation of precise interpolants.

In light of this observation, we propose a decomposition-based BMC (BMC-D) approach that takes advantage of the problem's structure.

Recall the encoding of BMC problem into propositional formula 1.1, unrolled up to bound k:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \bigwedge_{j=0}^{k} [\![\varphi]\!]_j$$

where $I(s_0)$ defines the initial states, $T(s_i, s_{i+1})$ for $i = 0 \ldots k$ are the transition relations between two successive states s_i and s_{i+1} and $[\![\varphi]\!]_k$ encodes the **negated** property up to bound k.

The partitioning approach we propose is based on system states, where each partition ψ_i (for $i = 1 \ldots n$) is assigned a subset of adjacent states of equal size $\rho = \lfloor \frac{k+1}{n} \rfloor$. Formally:

$$\psi_i = \bigwedge_{j=(i-1)\cdot\rho}^{i\cdot\rho-2} T(s_j, s_{j+1}) \bigwedge_{j=(i-1)\cdot\rho}^{i\cdot\rho-1} [\![\varphi]\!]_j \tag{5.1}$$

Each partition encompasses a segment of the transition unrolling T as well as the constraints encoding a portion of the property φ. The partitions on both ends, ψ_1 and ψ_n, contain more information than the internal partitions which follow the above formula 5.1:

– ψ_1 includes constraint encoding the initial states $I(s_0)$.

– ψ_n includes the remaining transition constraints.

Example 5.1. Consider a partitioning with $n = 4$ for a BMC problem that has been unrolled up to bound $k = 20$ that verifies an invariant property (*e.g.*, "**G**p" for any $p \in AP$). Each partition groups $\rho = \lfloor \frac{21}{4} \rfloor = 5$ frames. $\mathcal{V}(\psi_1)$ contains variables of steps s_0 to s_4, implying that the constraints assigned to:

- ψ_1 are $I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_3, s_4) \wedge [\![\varphi]\!]_0 \wedge \cdots \wedge [\![\varphi]\!]_4$,
- ψ_2 encloses variables of state s_5 to state s_9, *i.e.* $T(s_5, s_6) \wedge \cdots \wedge T(s_8, s_9) \wedge [\![\varphi]\!]_5 \wedge \cdots \wedge [\![\varphi]\!]_9$,
- ψ_3 are $T(s_9, s_{10}) \wedge \cdots \wedge T(s_{14}, s_{15}) \wedge [\![\varphi]\!]_{10} \wedge \cdots \wedge [\![\varphi]\!]_{14}$, and,
- ψ_4 with $T(s_{15}, s_{16}) \wedge \cdots \wedge T(s_{19}, s_{20}) \wedge [\![\varphi]\!]_{15} \wedge \cdots \wedge [\![\varphi]\!]_{20}$.

Two successive partitions ψ_i and ψ_{i+1} are connected through the last and first state of the partition, respectively. In the example above, partitions ψ_1 and ψ_2 are connected through the transition $T(s_4, s_5)$ where state s_4 is included in the first partition and s_5 on the second. Thus, this transition involves variables from both partitions. Due to the ambiguity of including these constraints in either ψ_i or ψ_{i+1}, it is reasonable for G to integrate them into its clause database. This ensures consistent decisions across different partitions and prevents conflicting decisions regarding the shared variables.

This last observation leads to a modification, where G is initialized with a subset of the problem's clauses corresponding to the transitions linking the n partitions together. This initialization also encompasses a segment of the property's constraints. This adaptation proves particularly relevant for certain types of formulas where their interpretation involves adjacent time steps. For instance, consider the specification "**G X** p" for any $p \in AP$. Its propositional formula encoding implies variables at time steps i and $i+1$, which correspond to states s_i and s_{i+1} of the system's automaton. This implies that G incorporates the following constraints of the problem:

$$G = \bigwedge_{i=1}^{N} T(s_{i \cdot t-1}, s_{i \cdot t}) \wedge \bigwedge_{j=1}^{n-1} \bigwedge_{j<l}^{n} {}_j[\![\varphi]\!]_l \text{ with } N = \begin{cases} n & \text{if } n \textbf{ mod } k = 0 \\ n-1 & \text{otherwise} \end{cases}$$

where ${}_j[\![\varphi]\!]_l$ represents the property's constraints that entail the j-th and l-th partitions, meaning that a state from ψ_j is linked to a state in ψ_l in the property formula. Hence, G will be in charge of deciding on the following shared variables among all partitions:

$$\mathcal{V}(G) = \bigcup_{i=1}^{n-1} \underbrace{\mathcal{V}(\psi_i) \cap \mathcal{V}(\psi_{i+1})}_{\substack{\text{common variables} \\ \text{between two partitions}}} \bigcup \underbrace{\mathcal{V}(G)}_{\substack{\text{variables that} \\ \text{compose } G\text{'s clauses}}}$$

When solving G, the process involves the assigning of values to a subset of the whole

part. n	5		10		20		30		40		50	
	#S	P	#S	P	#S	P	#S	P	#S	P	#S	P
BMC-D	15	622h	28	586h	44	541h	35	561h	21	604h	49	534h
LZY-D	4	654h	6	649h	5	653h	9	642h	10	638h	9	642h

TABLE 5.1: Comparison of LZY-D and BMC-D decomposition approaches for different partition sizes n

formula's variables ($\mathcal{V}(G) \subset \mathcal{V}(F)$). This approach narrows down the focus to a limited set of variables, thereby decreasing the communication overhead with the partitions.

It's worth noting that G has a partial view of the problem, incorporating the transitions between successive partitions. Each partition can be seen as a representation of a portion of the paths connecting the initial state s_0 (contained in ψ_1) to the final state s_k (in ψ_n). Consequently, the generated models m are constructed in a way that aligns the partitions in order to identify a complete path that violates the property. Where, in contrast, the manager G of the LZY-D strategy starts with no initial constraints. This decomposition is conducted randomly, distributing constraints encoding a transition or property constraints at a fixed depth unevenly among partitions. The following section will allow a comparison of these two decomposition methods.

5.2.3 Comparing LZY-D and BMC-D

This first experiment aims to compare the aforementioned decomposition approaches when employed as complete solving procedures for BMC problems. This will shed light on the quality of the interpolants generated by each approach.
It is important to reiterate that the primary goal of our work is to use the interpolation learning mechanism differently than Hamadi et al. [37] approach (see Section 5.3 and Section 5.4).

Benchmark setup. Our BMC benchmark comprises SMV [1] programs. These programs, along with their respective LTL properties, have been sourced from a diverse range of benchmarks, including the HWMC Competition (2017[2] and 2020[3]), hardware verification problems [136], the BEEM database [138], and the RERS Challenge[4].
Additionally, certain LTL properties have been generated using Spot [50] to ensure that each category of the Manna & Pnueli hierarchy [39] is represented. We utilized various bounds k for each BMC problem, with k ranging in $\{60, 80, ..., 800, 1000\}$. We excluded trivial instances that executed in less than 1 second on the MINISAT1.14P solver [83].

Table 5.1 presents the results of the approaches on 200 randomly selected BMC problems from the aforementioned benchmark. The partition sizes used were $n = 5, 10, 20, 30, 40, 50$, and a time limit of 6000 seconds was applied to both LZY-D and

[2]http://fmv.jku.at/hwmcc17/
[3]http://fmv.jku.at/hwmcc20/
[4]RERS models translated into NuSMV: https://tinyurl.com/29a4jcme

part. n	**5**		**10**		**20**		**30**		**40**		**50**	
	#S	P	#S	P	#S	P	#S	P	#S	P	#S	P
BMC-D-ITP	**111**	**328h**	**110**	**333h**	**111**	**329h**	**111**	329h	**109**	**331h**	**110**	**329h**
LZY-D-ITP	109	335h	107	338h	105	341h	110	**327h**	107	338h	107	338h
original inst.	107								337h			

TABLE 5.2: Impact of interpolants' clauses on the solving

BMC-D approaches. We restricted the evaluation to these 6 partition sizes, aligning with the choices made in the original paper's experiments [37], employing the same interpolation algorithm (McMillan [58]). The table highlights the number of solved instances (#S) and the PAR-2 time (P).

We observe that BMC-D outperforms LZY-D significantly, especially when dealing with larger partition sizes ($n = 50$). BMC-D successfully solves 49 instances, leading to a noteworthy reduction of 108 hours in PAR-2 time compared to LZY-D. These results seem to imply that the improvement is attributed to concentrating the majority of clauses in partition G, resulting in empty partitions within φ_i as n approaches the bound k of the considered problem. This brings us back to the scenario of a standard (flat) resolution. However, our concrete observations invalidate this hypothesis, revealing that the φ_i partitions do indeed encompass a fair portion of the problem's clauses. On the contrary, the BMC-D strategy helps to separate independent subspaces providing better performances than LZY-D strategy.

Due to interpolation computation, neither of the two approaches managed to surpass the performance of a classical solver (MINISAT1.14P). This is in contradiction with the reported results in [37]. Actually, LZY-D fails to outperform MINISAT1.14P within a verification benchmark context. One potential explanation is that the benchmark used by the authors is composed of fully *symmetrical problems*, whereas the BMC benchmark contains relatively fewer symmetries than expected: the conversion of BMC problems into CNF format disrupts symmetries, largely due to the introduction of extra variables during the encoding.

In light of these results, we draw two conclusions: (1) the clauses produced by the interpolants appear to provide valuable insights, and (2) the current approach is hindered by the computational complexity of interpolation. This prompts the question: **how can we leverage these interpolants in an optimal solving process?** Our suggestions for addressing this question are discussed in the following two sections.

5.3 Interpolation-based Offline Learning

It is intriguing to thoroughly assess the relevance and quality of information generated by interpolation when compared to that naturally acquired by a state-of-the-art SAT solver during its learning process. Our intuition suggests that clauses derived from interpolants could be highly beneficial in aiding a SAT solver, potentially leading to reduced solving times.

To validate our intuitions and hypotheses, we conducted an experiment employing

part. n	**5**	**10**	**20**	**30**	**40**	**50**	**Avg aug.**
BMC-D	1.14 %	1.24 %	1.53 %	1.63 %	1.48 %	1.48 %	1.41 %
LZY-D	4.13 %	2.68 %	2.03 %	1.16 %	0.69 %	0.79 %	1.91 %

TABLE 5.3: Average rate of interpolants size

LZY-D and BMC-D as preprocessing steps for a classical SAT solver. All experiments were conducted within the context of the BMC problem benchmark introduced in Section 5.2.3. The primary objective here was to evaluate the information's value provided by interpolants in contrast to the information gathered by a conventional SAT solver, all within the same time constraints.

To be more specific, each of the two algorithms was run for a specified time period, with a particular partition size, on a BMC instance. The interpolants generated during this period were converted into clauses and added to the initial instance. We refer to this process as "offline learning". The augmented instance was then solved by a CDCL-like SAT solver.

In this context, we randomly selected a set of 200 BMC instances from the benchmark setup described in Section 5.2.3. Each instance underwent an enrichment process involving the incorporation of interpolation clauses generated through offline learning over a period of 600 seconds. These instances were solved by MINISAT1.14P within a timeout of 6000 seconds.

For reference, the original instances (without additional clauses) were also solved by MINISAT1.14P within a timeout of 6600 seconds to accommodate the additional time required for offline learning.

Table 5.2 highlights the obtained results. **BMC-D-ITP** (resp. **LZY-D-ITP**) indicates the line where instances are augmented with BMC-D (resp. LZY-D) interpolants. The line labeled as **original inst** refers to the original instances without any additional clauses. The rest of the reported information is the same as in Table 5.1.

Both decomposition approaches exhibit improved solving times and succeed in resolving additional instances that could not be tackled without the offline learning. Notably, interpolants obtained through BMC-D decomposition have enhanced the solving. They showcase superior performance with a partition size of $n = 20$, solving 4 instances more and reducing the PAR-2 time by up to 8 hours compared to solving the original problem. LZY-D decomposition yields optimal outcomes with $n = 30$, resolving 3 instances more while achieving a PAR-2 time that is 9 hours shorter than solving the original instances.

These results substantiate our initial intuition regarding the significance of information acquired through the interpolation process. Furthermore, it becomes evident that the interpolants obtained from the structural decomposition method BMC-D prove to be more valuable compared to those derived from LZY-D.

Indeed, Table 5.3 illustrates the average percentage of the number of additional clauses added to the original problems, that were learnt by the LZY-D and BMC-D strategies during the offline learning phase, and across various partitioning sizes n.

The last column displays the average percentage increase across all partition sizes.

The BMC-D decomposition consistently generates a stable and equivalent set of clauses across all partitioning sizes. The increase in the total number of clauses remains limited, reaching a maximum of 1.63 % of additional clauses, with an average augmentation of 1.41 %. Regardless of the chosen partition size, on the contrary, the LZY-D approach tends to produce a larger number of clauses, including up to 4.13 % of interpolation clauses with an average augmentation of 1.91 %. We observe a decrease in the number of generated interpolants relative to the partition size.

These two trends can be explained as follows: the decomposition-based BMC strategy allowed us to generate a relatively consistent amount of information within the 600-second time frame. This consistency arises from two related aspects: (a) the distribution of shared variables between two partitions is homogeneous, with the exception of the first and last partitions, each containing more or less information than the others (ψ_1 contains the initial state $I(s_0)$ and ψ_n the remaining states if any). These shared variables are designed to connect the partitions, thereby identifying a complete path that violates the property; (b) The partial assignment m, generated by G, consistently produces conflicts, *i.e.*, interpolants, regardless of the partition size. For instance, when using a partitioning scheme with $n = 5$ (resp. $n = 50$), BMC-D generates an average of 4.40 (resp. 34.06) interpolants per *round*, where a *round* signifies when the procedure has traversed all partitions over the current model m.

In contrast, the random partitioning approach LZY-D generates fewer interpolants per *round*, with an average of 1.81 interpolants for $n = 5$ and 2.86 for a partition size of $n = 50$. We observed that the distribution of shared variables is less homogeneous between partitions. This non-homogeneity arises because the partitioning is random, leading to some partitions sharing many more variables than others. Thus, due to this randomness in the shared variables, it becomes challenging to produce many conflicts regardless of the given assignment m. Additionally, the manager G of the LZY-D approach starts with no constraints in its database, which can result in the generation of assignments m that do not differ significantly. Consequently, it becomes more challenging for G to find a model m that violates a majority of the partitions, leading to a reduced number of interpolants.

Based on these measurements, this analysis clearly underscores the competitive and efficient nature of a decomposition approach that takes into consideration the structural aspects of the BMC problem, in contrast to a randomized decomposition strategy. Hence, it is entirely conceivable to utilize interpolation-based learning as a preprocessing phase to enhance performance in a sequential setting.

5.4 Interpolation-based Learning in Parallel Solving

As demonstrated earlier, interpolation clauses have a positive impact on the overall resolution time for BMC problems (refer to Table 5.2). This finding underscores the potential advantages of integrating our concept within a parallel computing context.

One of the most effective strategies in parallel SAT solving is the "portfolio" approach. In essence, a portfolio consists of a set of sequential SAT solvers that run in parallel and compete to solve a problem. These core engine solvers vary in several ways, including the algorithms they employ and their initialization parameters.

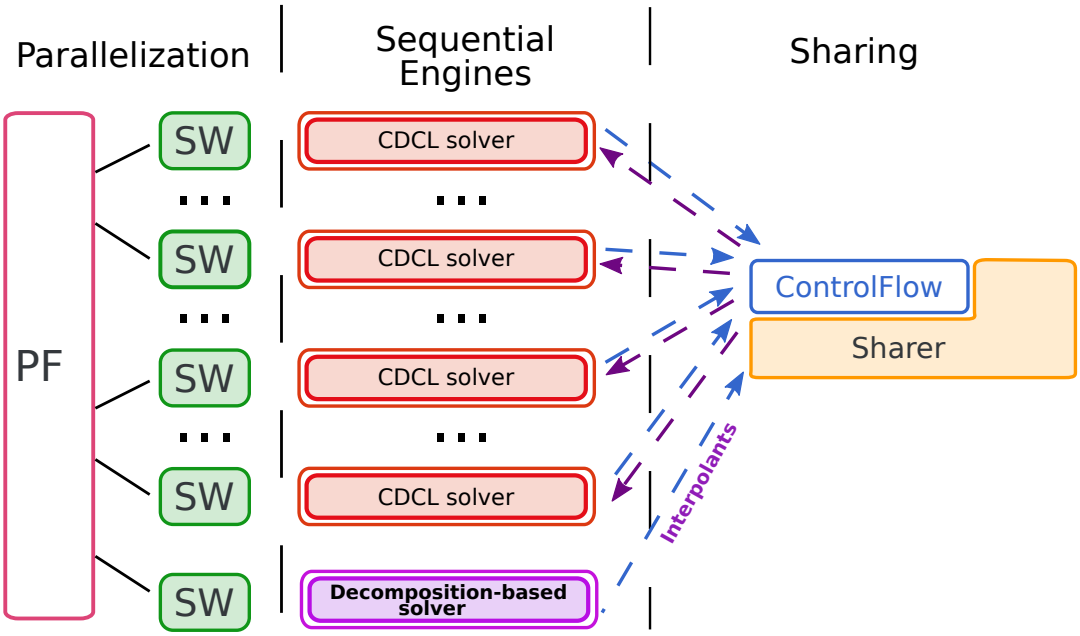


FIGURE 5.2: Portfolio of solvers with sharing scheme using the framework PAINLESS

Moreover, they can exchange information to expedite problem-solving and avoid repeating the same mistakes. This aspect forms the basis for the integration of the approaches discussed thus far.

Indeed, we seamlessly incorporate our decomposition-based solver into a portfolio solver (utilizing the PAINLESS framework [137]), alongside multiple sequential CDCL engines, as illustrated in Figure 5.2. Three main components arise when treating parallel SAT solvers:

- *sequential engines.* it can be any CDCL state-of-the art solver,
- *parallelization.* is represented by a tree-structure of arbitrarily depth. The internal nodes of the tree represent parallelization strategies, and leaves are core engines (SW), and
- *sharing.* is in charge of receiving and exporting the set of clauses provided by the sequential engines during the resolution process. Its behaviour is reduced to a loop of sleeping and exchange phases.

In this integration, the interpolation-derived clauses are shared among the CDCL solvers. This aims to enhance the knowledge base of CDCL solvers and support them throughout the solving process. In this framework, the decomposition-based solver does not import information from other solvers; instead, it exclusively provides its interpolants to them. It functions as a "black box", serving as a specialized clause generator designed for BMC problems.

For the sake of simplicity, the exchange phase of the *sharing* component is to share clauses with a limited LBD[5] value [23]. Specifically, CDCL solvers export learnt

[5]LBD is a positive integer used as a learnt clause quality metric in nearly all competitive sequential CDCL-like SAT solvers and parallel sharing strategies.

Portfolio	part. n	SAT	UNSAT	Total sol.	PAR-2
P-BMC-D	5	**186**	115	301	356h05
	50	**186**	**119**	**305**	**345h40**
P-LZY-D	5	184	113	297	371h09
	50	185	113	298	367h20
P-MINISAT	-	185	113	298	362h57

TABLE 5.4: Performance comparison between different Portfolios

clauses identified by an LBD ≤ 4, a threshold that has been empirically proven to be effective in recent portfolios and has demonstrated competitive performance in parallel SAT competitions[6].

Upon receiving the interpolants from the n partitions, the manager G calculates their corresponding LBD values and shares only those with LBD ≤ 4, following a similar approach as used for sharing conflicting learnt clauses.

To encourage the solvers to explore diverse search subspaces, it is essential to introduce some variation in the solver's parameters, such as the initial phase of the variables. By ensuring that each solver runs with a different initialization phase, they are more likely to make distinct decisions, leading to exploration of distinct search subspaces. This diversification approach will be applied to all the portfolios evaluated in the subsequent analysis.

Experimental evaluation

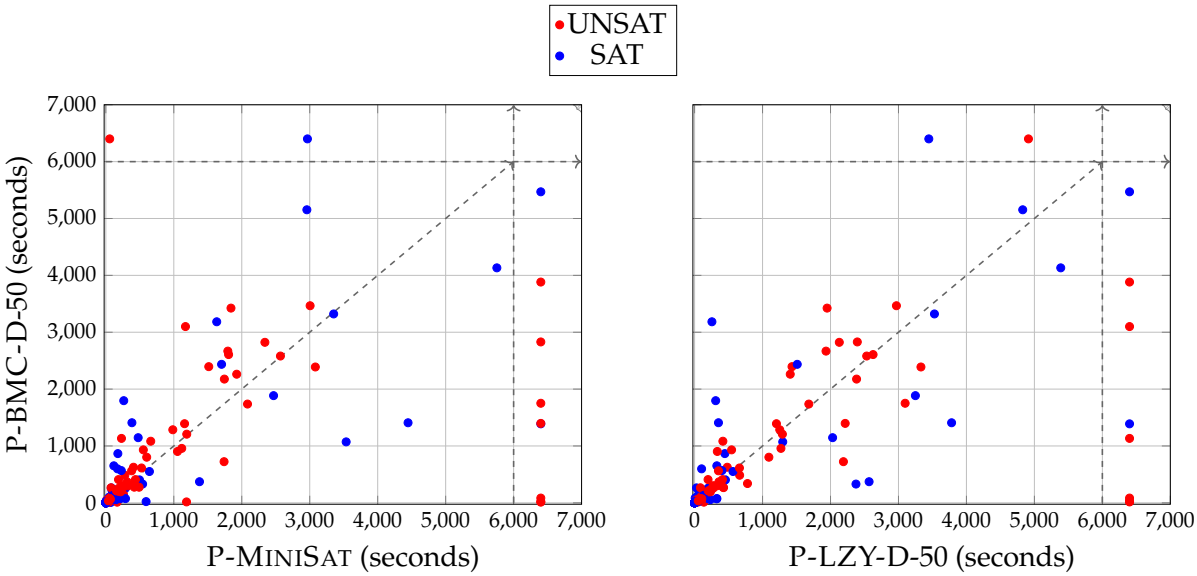
The experiments were conducted on the same benchmark described in Section 5.2.3, which consists of 400 randomly selected BMC instances. Each instance had a time limit of 6000 seconds for execution. The portfolio setups comprised 10 threads, and the solvers used in these portfolio configurations were as follows:

- **P-MINISAT :** This portfolio exclusively employed the state-of-the-art MINISAT1.14P solver.
- **P-BMC-D :** In this configuration, one MINISAT1.14P solver was replaced with a decomposition-based solver using BMC-D decomposition.
- **P-LZY-D :** Similar to P-BMC-D , this portfolio incorporated LZY-D decomposition instead.

Table 5.4 presents the results for both smaller ($n = 5$) and larger ($n = 50$) partition sizes. The remaining configurations yielded outcomes similar to those with $n = 5$. The table provides information on the number of solved SAT and UNSAT instances, along with the total instances. Additionally, it indicates the PAR-2 metrics[7].

[6] https://satcompetition.github.io/2023/

[7] PAR-k is a measure used in SAT competitions that penalizes the average run-time, counting each timeout as k times the running time cutoff



(A) Scatter-plot comparing P-MINISAT to our best approach P-BMC-D with $n = 50$

(B) Scatter-plot comparing P-LZY-D to our best approach P-BMC-D with $n = 50$

FIGURE 5.3: Runtime comparison between Portfolios

Unsurprisingly, P-BMC-D outperforms the state-of-the-art P-MINISAT by solving 6 more UNSAT instances and 1 additional SAT instance, all within a remarkable 17 hours reduction in PAR-2 solving time. Furthermore, P-BMC-D exhibits a clear advantage over the randomized approach P-LZY-D for both partitioning sizes ($n = 5, 50$), solving at least 7 more instances and achieving a PAR-2 time reduction of at least 21 hours.

A visual representation of this comparison is illustrated in the scatter plots shown in Figures 5.3a and 5.3b. Figure 5.3a depicts a comparison of the running times in seconds for UNSAT and SAT instances between P-BMC-D with $n = 50$ and P-MINISAT . Similarly, Figure 5.3b provides the same comparison of P-BMC-D with P-LZY-D on the same partition size $n = 50$. These figures demonstrate significant enhancements achieved by P-BMC-D in solving known challenging UNSAT problems, while maintaining a competitive edge in solving SAT instances.

Given these results, we sought to examine the relationship between the partitioning size and the unrolling depth k of the BMC problem. To do this, we conducted an analysis in which we categorized the entire benchmark of 400 BMC problems based on the value the number of frames within a single partition ψ_i, noted ρ. This categorization was performed for various partition sizes, namely $n = 5, 10, 20, 30, 40, 50$.

Table 5.5 provides an overview of the additional instances solved (+) or lost (-) by P-BMC-D in comparison to the P-LZY-D and P-MINISAT portfolios, indicated in the first and second rows, respectively. We categorized the BMC problems into three groups based on the number of frames ρ contained within a partition ψ_i. The first column ([1,7]) includes instances where each partition contains at least one frame and at most 7 frames ($1 \leq \rho \leq 7$). The next column corresponds to instances where ρ falls within the range of 8 to 30. Finally, the last column groups the remaining values of ρ up to 200. This upper limit is dictated by the fact that the evaluated instances

Portfolio \ num. steps ρ	[1 , 7]	[8 , 30]	]30, 200]
P-LZY-D	0	+12	-1
P-MINISAT	+3	+10	-1

TABLE 5.5: Number of solved instances of P-BMC-D versus P-MINISAT and P-LZY-D for different frame sizes ρ

have lengths k varying from 10 to 1000 steps. Thus, for the smallest partition size ($n = 5$), we have $\rho = \frac{1000}{n} = 200$ frames.

An interesting observation is that clustering a large number of frames within a single partition ($30 < \rho \leq 200$) negatively impacts the performance of P-BMC-D . This is evident from Table 5.5, where P-BMC-D failed to solve one instance compared to the other two portfolios.

The most significant improvement is observed when $\rho \in [8, 30]$, where P-BMC-D solved 10 and 12 additional problems compared to P-MINISAT and P-LZY-D , respectively.

Furthermore, grouping a small number of frames within a single partition ([1,7]) only marginally enhances the performance of P-BMC-D . It results in solving 3 additional instances compared to P-MINISAT, while showing no increase in solved instances compared to P-LZY-D .

Based on the above analysis, it becomes evident that utilizing interpolation-based clause learning through a BMC-based partitioning approach, which balances the inclusion of a reasonable number of frames within each partition (between 8 and 30), yields the most favorable outcomes in terms of solving efficiency. This suggests that the granularity of partitioning n and the total number of frames ρ within each partition play a key role in enhancing the computation of relevant interpolants.

5.5 Conclusion

In this work, we embarked on a journey to enhance the efficiency of SAT-based BMC solvers by harnessing the interpolation mechanism as a mean to generate learnt clauses. We drew inspiration from the work of Hamadi et al. [37], who introduced a partitioning-based technique (reconciliation algorithm) utilizing interpolation theorems for generating relevant clauses. Their "lazy decomposition" approach (LZY-D) focused on partitioning without considering the problem's structure.

We introduced a novel decomposition technique, BMC-D, which capitalizes on the inherent structure of the BMC problem. This approach involves partitioning the formula based on system states, with each partition encapsulating a subset of adjacent states. We showcased the effectiveness of incorporating interpolation clauses into the solving process. This integration can occur during the preprocessing phase, within a limited time frame, or as part of clause exchange in a portfolio of classical CDCL solvers. In both of these scenarios, the use of interpolation-based clauses

stemming from the structural decomposition (BMC-D) consistently demonstrated superior solving efficiency compared to state-of-the-art methods, for both UNSAT and SAT instances.

Our ongoing work will extend this concept to more recent solvers, such as KISSAT-MAB [33], which has the potential to provide robust *unsatisfiable proofs* and consequently more informative interpolants. It is also plausible to consider using more effective interpolation algorithms such as those implemented in the PERIPLO framework [145], in the context of software verification.

We also intend to use the partitioning size of the problem as a diversification parameter by running multiple BMC-D solvers within a parallel strategy. Each BMC-D engine will employ a different partition size. Additionally, conducting a more in-depth analysis of the type of property being evaluated would be valuable to develop a partitioning method customized for each specification, following the hierarchy of Manna & Pnueli [39]

Chapter 6

Programmatic SAT for BMC

Contents

As seen in the previous chapters, the quality of learnt clauses stands as a critical factor that affects the performance of CDCL solvers.

This chapter introduces an alternative approach to incorporate structural information during the SAT solving without necessitating a deep dive into the SAT solver's code. This concept is known as *Programmatic SAT*. Programmatically, it presents a simplified means of interacting with the solver to guide it during the resolution process. This interaction occurs through an external entity (a *black-box*), which is typically specialized in domain-specific knowledge related to the problem at hand. In our context, this entity is tailored for model checking problems.

Our primary contribution entails introducing a novel approach for harnessing the automata representation of model checking to extract information lost during the encoding of the original BMC problem into a propositional formula. These pieces of information, when transformed into a set of clauses, can effectively steer the SAT solver within significant search subspaces. Essentially, we can generate information that the SAT solver might have overlooked. This is achieved through the new external component (the black-box). More precisely, when the SAT solver invokes it, providing a current partial assignment α, the black-box extracts some facts hidden from the solver. These facts are derived from the Büchi automaton representing the synchronized product between α and the evaluated property φ. The extracted information are then encoded into a set of clauses that will be injected into the SAT solver.

The drawback of automata-based procedures, however, is the time and memory consumption that invoking the black-box can entail. This is primarily due to the computational overhead of building the synchronized product, which can become

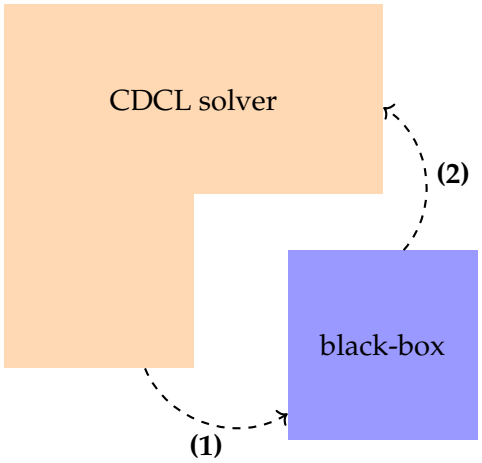


FIGURE 6.1: Programmatic SAT scheme

substantial as the problem's complexity increases. To mitigate this, we attempted to control the flow of calls to the black-box. But, still, this requires more refined parameterization, which will be the subject of future investigation. In fact, so far, no experimentation has yielded encouraging results. Therefore, this chapter presents the theory behind combining both explicit and symbolic model checking procedures to harness the strengths of both worlds in solving BMC problems.

In the subsequent sections of this chapter, we begin by providing a brief definition to summarize the concept of Programmatic SAT solving and some related works employing this concept along with presenting our motivation for using this technique in the context of BMC in Section 6.1. We then introduce the theoretical aspect we build inside the black-box (Section 6.2) and communication control flow between these two entities (Section 6.3). Finally, we conclude with an overall and short-term perspectives for this work.

6.1 Literature and motivations

6.1.1 State-of-the-art

The concept of a programmatic SAT solver was first introduced in LYNX [146] used to learn facts about potential solutions to guide the solver's search that standard solvers cannot acquire. It is an extension of DPLL(T) or nowadays known as SMT (SAT modulo Theory) [147]. Concretely, a programmatic SAT solver is a standard SAT procedure with an extra piece of code incorporated on the solving phase of the SAT solver tested periodically. This piece of code encodes conditions with respect to the domain-specific requirements of the evaluated instance. These conditions specify the requirements that the SAT solver's solution must meet, whether by learning new clauses, deciding on the next variable to branch on, *etc*. In essence, this extends any of the fundamental components of the CDCL solver to guide it toward more effective search spaces. Since our focus in this thesis is on learning new clauses, the current partial assignment computed by the SAT solver is provided to the additional code for evaluation. If it violates a condition from the provided code, then a conflict clause or multiple conflicting clauses are generated encoding this fact. The conflict clauses are then added to the SAT solver's database of learnt clauses, with the aim of increasing the efficiency of the remainder of the search. These clauses generation

have proven their usefulness on many applications such as on instances derived from RNA folding problems [146], Cryptography [148], Williamson matrices [149], *etc*. This is due to the fact that the SAT solver could not learn such facts since it has no knowledge of the domain of the problem at hand.

This extra expressiveness of the domain specific knowledge is also a feature of SMT solvers [147]. The different, however, is that SMT solvers deal with a specific theory of first-order logic, requiring more overhead to use. This is because SMT solvers require prior knowledge of how a SAT solver functions since they apply modifications directly within the internal code of the SAT solver.

To the best of our knowledge, no one has studied this paradigm in light of the BMC problem and model checking in general. As discussed in Chapter 3, the state-of-the-art on SAT-based BMC indicates that the enrichment of the SAT solver with structural information and the adaptation of these heuristics to the types of problems being handled are developed directly inside the solver, such as symmetry procedures, decision heuristics, and so on. The only study we have found that can come close to the programmatic context of SAT-based BMC is the work by [150], implemented in a framework called DIVER. Their aim is to generate relevant information from the original problem. To achieve this, they use a BDD structure that randomly selects a node representing a subspace of the search space. This selection is repeated after a certain number of decisions. The call to the BDD produces potentially interesting clauses to further guide the solver. The authors categorized the clauses according to their importance (we have employed a similar methodology discussed in Section 6.3) and tested several configurations by excluding certain categories and clause sizes to avoid overloading the solver. It should be noted that the work of [150] was used in the incremental context of BMC (which remains applicable for a fixed bound), something we have not experimented in this present work but which will be the subject of future work.
The difference with our approach is that instead of using a BDD structure, we construct the automaton representation of the property and reason about the resulting automaton from the synchronized product with the current assignment α transmitted by the SAT solver.

6.1.2 Usage in a BMC context

As previously mentioned, programmatic SAT procedures allows non-expert users to *easily* introduce domain-specific code into CDCL SAT solvers [29, 30, 33, 79], thus enabling users to guide the behavior of the solver. Our objective is to enhance the SAT solver with structural information from BMC. The black-box interface (Figure 6.1) will be capable of sending such information to the solver. The user-provided black-box will examine the partial assignment α, generated by the solver during its search, and will respond by dynamically adding clauses to the solver. As a result, the user-provided code can finely tune and guide the solver's search. The callback interface (black-box) acts as a bridge between two realms. It allows non-experts in model checking to utilize a SAT-oriented BMC solver, and simultaneously, model checking experts can enhance this black-box with their expertise.

For this interaction to occur, the SAT solver sends the current partial assignment to the black-box. The black-box then determines whether the given partial assignment

could lead to a violation (or not) of the property at hand, resulting in an *accepting* or *rejecting* path within the model.

Accepting-path. This means that the finite path encoded from the partial assignment could be extended to an infinite path (an execution) that violates the property. Consequently, the black-box generates new constraints to communicate to the solver, with the aim of completing the provided assignment, effectively creating a counter-example **(1)**.

Rejecting-path. This indicates that the finite path encoded from the partial assignment cannot be extended to an infinite path that violates the property. Consequently, there is no need from the SAT solver to further explore this decision subtree, and should be cutoff **(2)**.

The following section will provide details on how to detect both **(1)** and **(2)** situations.

Remark: Throughout the remainder of this chapter, the term "path" will refer to an infinite path and thus an infinite execution, meaning an infinite sequence of states that starts from an initial state. We might occasionally interchangeably use both terms, "path" and "execution", which have the same meaning. We will specifically mention "finite path" when necessary.

6.2 Inside the Black-box

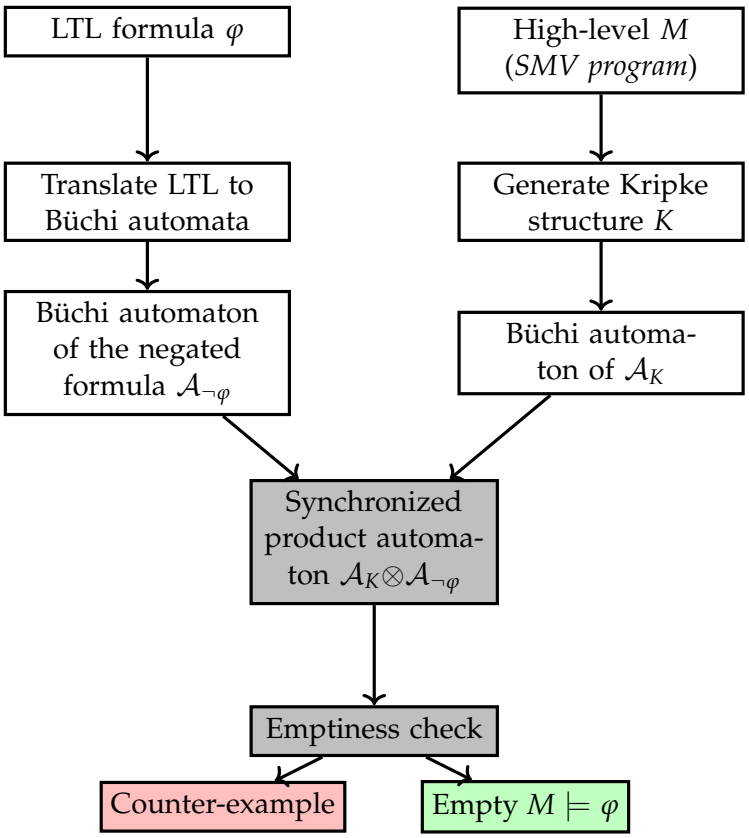


FIGURE 6.2: Automata-based approach for model checking

Figure 6.2 recalls and summarizes the broad lines of the automata-based model checking procedure. At the top of the figure, the model and the LTL formula are

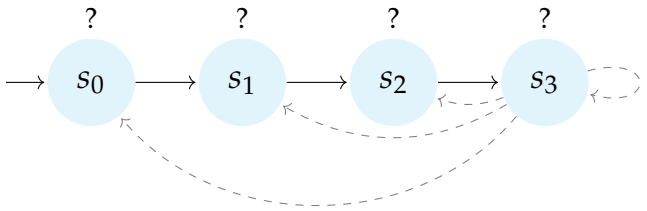


FIGURE 6.3: Kripke structure K of a segment of the model M

the two elements provided by user who aims to determine whether the model M satisfies the LTL formula φ. The result, at the bottom of Figure 6.2, can either be the assurance that the model verifies the formula ($M \models \varphi$) or the detection of a counter-example, which is an execution of the model that invalidates the formula.

The high-level representation of the system M initially transformed into a Kripke structure K and subsequently into a Büchi automaton $\mathcal{A}_K$, whose language represents the set of feasible executions of the model. In parallel, the negated LTL formula φ is also translated into a Büchi automaton $\mathcal{A}_{\neg\varphi}$, representing the set of executions that invalidate the formula (or that validate the negation of the formula).

As we introduced in Chapter 1, to determine whether a model M satisfies a formula φ, it is enough to examine if the language associated with the automaton $\mathcal{A}_K$ is contained within the language of $\mathcal{A}_{\neg\varphi}$ ($\mathscr{L}_{\mathcal{A}_K} \cap \mathscr{L}_{\mathcal{A}_{\neg\varphi}} \neq \emptyset$). This procedure, known as the emptiness check, is performed by traversing the synchronized product $\mathcal{A}_K \otimes \mathcal{A}_{\neg\varphi}$. It characterizes the set of model executions that refute the property.

Thus, we have defined the behavior of the black-box for BMC, which will apply the operations of an explicit model checker. In this strategy, the automaton for the LTL property, as well as a segment of the model, are constructed to facilitate reasoning about the resulting synchronized product automaton.

In the following subsection, we will delve into the inner workings of the black-box, explaining how it generates and deduces information from the synchronized product. The insights derived from this process are critical for guiding the solver towards relevant search spaces.

6.2.1 Extracting Model executions

Through an example, we will illustrate the first part of the BMC-based black-box mechanism. This part shows how model and property automata are constructed. The second part, detailed in the next subsection, will provide more insight into how the automata are manipulated to deduce relevant information.

In the context of programmatic SAT for solving BMC problems, we have restricted the provided assignment to pertain to specific variables of the CNF formula. More precisely, it involves variables responsible for encoding the property (*i.e.*, a set of variables $\mathcal{P}$ in the CNF formula that excludes *auxiliary* variables, see Chapter 3). This restriction allows us to analyze a segment of the model, preventing the need for a full representation of the model and thus avoiding memory-related issues. Hence, the Kripke structure representation of the model M is a limited subset of the alphabet Σ

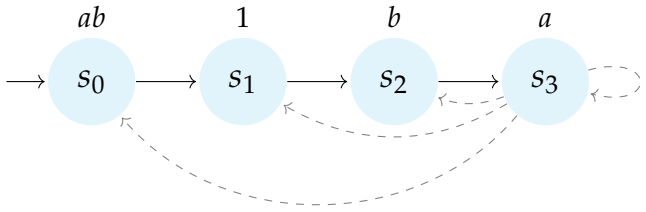


FIGURE 6.4: Kripke structure K with a labeling function

and, consequently, the set of atomic propositions AP involved in describing the LTL property.

We recall that each variable in the SAT formula translates the system variable domain value[1]. For the sake of simplicity and without lose of generality, for the rest of this chapter, we assume that all atomic propositions encoding the property use Boolean variables. Thus, if we need to check a specification $\mathbf{G}"p"$, this means that the atomic proposition $"p"$ represents a Boolean variable of the system p that must be *true* throughout model executions. Therefore, its representation in CNF simply corresponds to a Boolean variable x_p^j that is set to *true* when the atomic proposition $"p"$ is *true* at time step $0 \leq j \leq k$.

Given a BMC problem unrolled up to bound $k = 3$, involving 4 atomic propositions $AP = \{a, b, c, d\}$, and their representation in the SAT formula is composed of the following Boolean variables: $\{x_a^j, x_b^j, x_c^j, x_d^j\}$ for $j = 0, \ldots, 3$. Suppose that $\{a, b\}$ are the only atomic propositions involved in the LTL. We illustrate the Kripke structure $K = \langle S, T, I, Act, AP, \mathrm{L} \rangle$ of the model in Figure 6.3, where the labeling function L is not *yet* defined. Multiple executions can be constructed from K; to be precise, there are $k + 2$ executions:

- $\rho_0 = s_0 s_1 s_2 s_3$
- $\rho_1 = s_0 s_1 s_2 \underbrace{s_3 s_3 s_3 \ldots}_{\text{inf. repetition of } s_3}$
- $\rho_2 = s_0 s_1 \underbrace{s_2 s_3 s_2 s_3 s_2 s_3 \ldots}_{\text{inf. repetition of } s_2 s_3}$
- $\rho_3 = s_0 \underbrace{s_1 s_2 s_3 s_1 s_2 s_3 s_1 s_2 s_3 \ldots}_{\text{inf. repetition of } s_1 s_2 s_3}$
- $\rho_4 = \underbrace{s_0 s_1 s_2 s_3 s_0 s_1 s_2 s_3 s_0 s_1 s_2 s_3 \ldots}_{\text{inf. repetition of } s_0 s_1 s_2 s_3}$

where ρ_0 represent a finite execution and ρ_1, ρ_2, ρ_3 and ρ_4 are infinite executions with a cycle of length of 1, 2, 3, and 4, respectively.

The labeling function L ($\mathrm{L} : S \rightarrow 2^{AP}$), with S a finite set of states) of K defines, at each time step of the system, the state of variables describing the (negation) of the property. It is defined through the partial assignment α and is updated each time the SAT solver sends a query (a new α) to the black-box. Thus, from the set of variables

[1] If we take a model that defines a 32-bit variable denoted by a, its encoding in the SAT formula is represented by 32 Boolean variables at each k steps ($x^j_{"a=i"}$ for $i = 1, \ldots, 32$ and $0 \leq j \leq k$). An atomic proposition *"a = 12"* is translated into a Boolean variable $x^j_{"a=12"}$, that is set to *true* if the atomic proposition *"a = 12"* holds at time step $0 \leq j \leq k$)

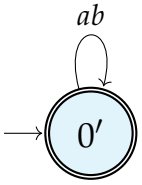


FIGURE 6.5: Büchi automaton of $\mathbf{G}(a \wedge b)$

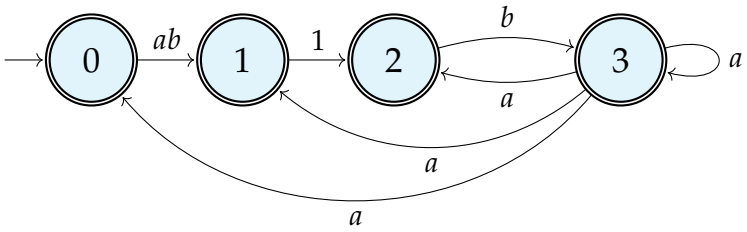


FIGURE 6.6: Büchi automaton $\mathcal{A}_K$

encoding the property in the CNF formula, we redefine the labeling function L of K. Nonetheless, not all state propositions are defined due to the CDCL solver has not yet decided on all the variables in $\mathcal{P}$. Some state propositions are thus undefined.

Suppose the given partial assignment α is as follows:

$$\alpha = \{x_a^0, x_b^0, x_b^2, x_a^3\}$$

We can observe that at step 1, neither x_a^1, nor x_b^1 were assigned by the SAT solver. To address this situation and to have a complete labeling function, we over-approximated the missing state's labels by setting them the full set of labels: 2^{AP} (marked with 1 in the automata illustrations). We define the function L of K according to α as follows:

$$-\mathrm{L}(s_0) = \{\{a, b\}\} \qquad -\mathrm{L}(s_1) = \underbrace{2^{AP}}_{\text{all propositions}}$$

$$-\mathrm{L}(s_2) = \{\{b\}\} \qquad -\mathrm{L}(s_3) = \{\{a\}\}$$

This completes the representation of K shown in Figure 6.4.

Let's now consider that the LTL specification the model should satisfy is: $\varphi = \mathbf{F}(\neg a \vee \neg b)$. The translation of this negated property into an automaton is drawn in Figure 6.5, involving only the variables a and b. The other variables, c and d, are

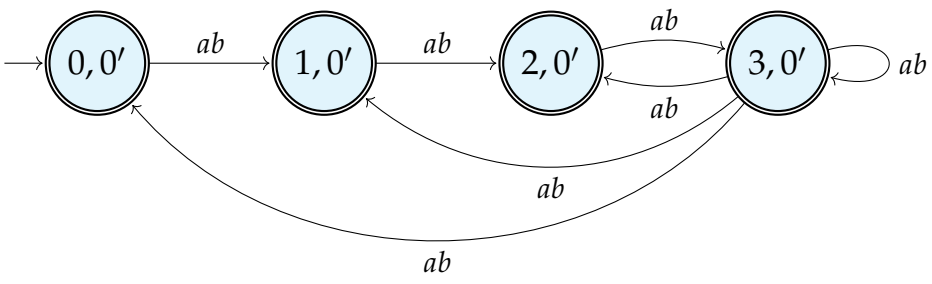


FIGURE 6.7: Synchronized Product automaton $\mathcal{A}_S$ of $\mathcal{A}_K \otimes \mathcal{A}_{\neg\varphi}$

discarded since they are not used in φ. The construction of the automaton for the property $\mathcal{A}_{\neg\varphi}$ is built only once by the black-box.

The black-box will construct the model structure K from Figure 6.4 into a Büchi automaton $\mathcal{A}_K$ displayed in Figure 6.6.

From $\mathcal{A}_{\neg\varphi}$ and $\mathcal{A}_K$, the black-box initiates the language intersection procedure (detailed in Section 1.3.4 of Chapter 1). Figure 6.7 showcases the synchronized product automaton $\mathcal{A}_S$ from $\mathcal{A}_K$ and $\mathcal{A}_{\neg\varphi}$ intersection. We can already deduce from the observation of $\mathcal{A}_S$ 6.7 that φ is invalidated because $\mathcal{A}_S$ is not empty: there exist only executions where a and b hold. This means that the specification φ is never verified since there exist no execution that results in a and b becoming $false$.

In summary, the BMC-based black-box mechanism determines the validity of the property φ through the model executions, which have been constructed using the partial assignment α provided by the SAT solver.

6.2.2 Learnt constraints from the Synchronized product automaton

The obtained automaton $\mathcal{A}_S$ from the synchronized product ($\mathcal{A}_K \otimes \mathcal{A}_{\neg\varphi}$) allow us to analyze the paths that might lead to a violation of the property and discard the exploration of unnecessary subspaces for the SAT solver. This is done by pruning uninteresting search subtrees through the addition of new clauses.

By adopting a concept similar to emptiness check algorithms [53], which are based on the enumeration of Strongly Connected Components (SCC) to identify SCCs that contain accepting cycles, we can determine the interesting parts of the $\mathcal{A}_S$ automaton. More precisely, this approach will help identify whether paths of the model can lead to an accepting or rejecting path only if it passes through SCCs.

Definition 6.1 (Strongly connected graph). *A graph $G = (V, E)$ is connected if, for every pair of distinct vertices $(u, v) \in V^2$, there exists a path from u to v or a path from v to u. A graph is strongly connected if, for every pair of vertices $(u, v) \in V^2$, there exists a path from u to v.*

Definition 6.2 (Strongly connected component). *An SCC is a maximal set of states V such that there is a finite path between any two distinct states of V.*

- ***Accepting:*** *An SCC is said to be **accepting** iff it contains an accepting cycle.*
- ***Non-Accepting:*** *There are two kinds of non-accepting SCCs. If an SCC can only reach other non-accepting SCCs, it is **useless** and may be removed from the automaton without changing its language. This simplification is traditionally performed right after the translation into an automaton. If the non-accepting SCC can reach an accepting one, it is **transient**. For the sake of simplicity, we assume that that useless SCCs have been removed, i.e., all non-accepting SCCs are transient.*

Figure 6.8 shows an example of a Büchi automaton with three accepting cycles. The dashed boxes highlight the SCCs of the automaton. Red boxes C_1 and C_2 are transient SCCs that can reach an accepting SCC (in green box). C_1 can reach the accepting SCC C_3, where C_2 can reach C_3 and C_4.

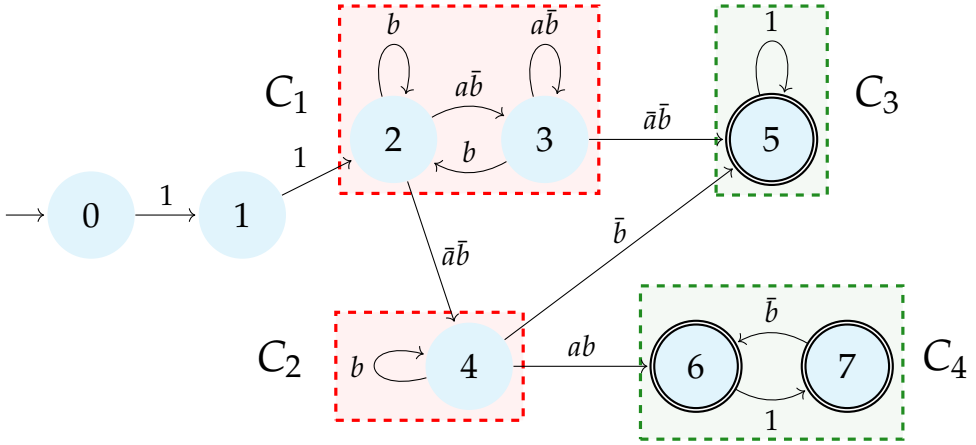
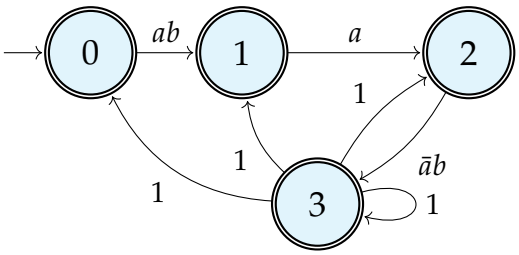
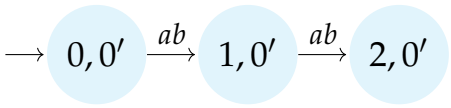


FIGURE 6.8: SCCs of a Büchi automaton



(A) $\mathcal{A}_{K_1}$ automaton



(B) Synchronized product between $\mathcal{A}_{K_1}$ and $\mathcal{A}_{\neg\varphi}$

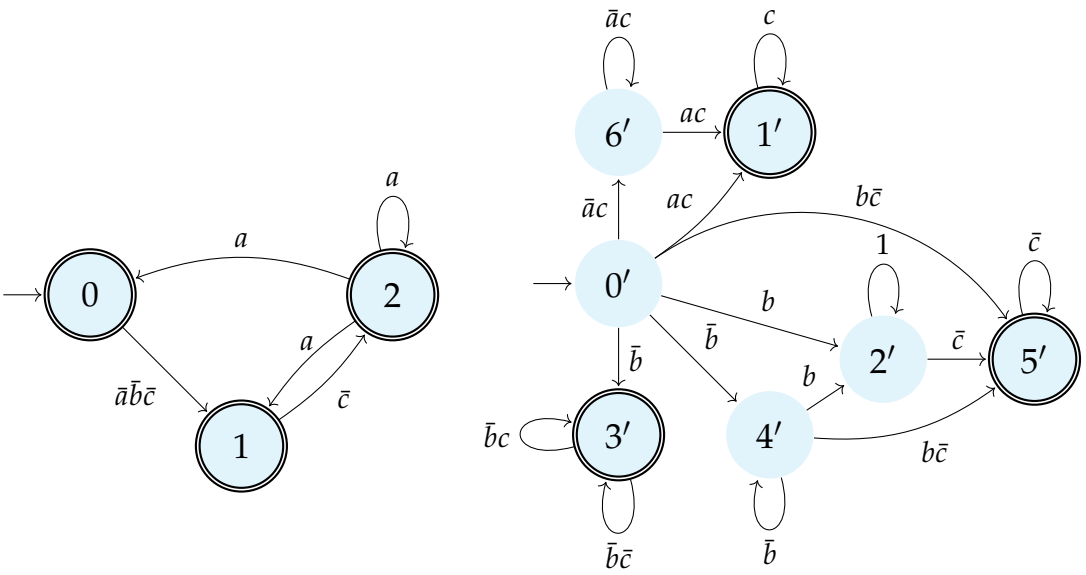
FIGURE 6.9: Büchi automata of Example **6.1**

Based on the notion of SCC, we can deduce two types of information depending on whether there exists an accepting-paths in the synchronized product between $\mathcal{A}_K$ of the model structure and $\mathcal{A}_{\neg\varphi}$.

The existence of an SCC proves that there is an infinite path that can lead to the violation of the property φ, *i.e.*, there exists a path containing an accepting cycle. However, when there is no accepting SCC, this result in K not invalidating the property. The intersection of the two automata languages is empty.

1. **Empty intersection.** this implies the non-existence of any accepting SCC. This signifies that the partial assignment α provided by the SAT solver can not be extended to any assignment capable of violating the property φ. Continuing along this set of decisions would be pointless, as they will not result in any contradiction. It is possible to halt the solver from exploring the subspace prefixed by the decisions α. This can be achieved by constructing a clause representing the negation of α. Consequently, communicating this information to the CDCL solver, will result in the elimination of this irrelevant path.

   Example 6.1. Let's revisit the same specification $\varphi = \mathbf{F}(\neg a \vee \neg b)$, as depicted in Figure 6.5. It has been verified up to $k = 3$. Let $\alpha = \{x_a^0, x_b^0, x_a^1, \neg x_a^2, x_b^2\}$. We initialize the labeling function L of the model Kripke structure K_1 following α. We depict the set of executions of the model in Figure 6.9a. The resulting synchronized product automaton contains no accepting SCC (Figure 6.9b). As such, the constraint that effectively removes this unuseful path would be a



(A) Büchi automaton $\mathcal{A}_{K_2}$

(B) Büchi automaton $\mathcal{A}_{\neg\varphi_2}$ with $\neg\varphi_2 = (\mathbf{FG}\neg c \wedge \mathbf{F}b) \vee \mathbf{G}(\neg b \wedge \mathbf{F}c) \vee (\mathbf{G}c\ \mathbf{U}\ (a \wedge \mathbf{G}c))$

FIGURE 6.10: Büchi automata of Example **6.2**

clause $C_{\neg\alpha}$, representing the negation of the assignment α:

$$C_{\neg\alpha} = (\neg x_a^0 \vee \neg x_b^0 \vee \neg x_a^1 \vee x_a^2 \vee \neg x_b^2).$$

2. **Transition constraints.** The automaton $\mathcal{A}_S$ is non-empty if there exists an SCC containing an accepting cycle. This implies the existence of at least one accepting SCC leading to a witness violating φ. As a result, it becomes possible to trace one or multiple paths leading to an accepting cycle under the provided partial assignment α. When α contains gaps, meaning there are some unassigned variables related to the property, we can infer these missing assignments from the synchronized product automaton. This information is forwarded to the SAT solver as a set of constraints, denoted as f_i, each of which dictates the values the unassigned variables in the property should assume at time step i. All of these constraints are formulated under the assumption of α. Therefore, we can formulate the the constraints generated for this purpose:

$$C_\alpha \implies f_i \qquad \text{for } i = 0, \ldots, k \tag{6.1}$$

with the clause C_α is constructed from partial assignment α.

Example 6.2. Consider the formula $\varphi_2 = (\neg a\ W\ \mathbf{F}\neg c) \wedge (\mathbf{GF}c \vee \mathbf{G}\neg b) \wedge \mathbf{F}(b \vee \mathbf{G}\neg c)$ for a system unrolled up to $k = 2$, as depicted in Figure 6.10b. Let the partial assignment provided by the SAT solver be:

$$\alpha = \{\neg x_a^0, \neg x_b^0, \neg x_c^0, \neg x_c^1, x_a^2\}$$

After defining the labeling function L of K_2, Figure 6.10a displays executions

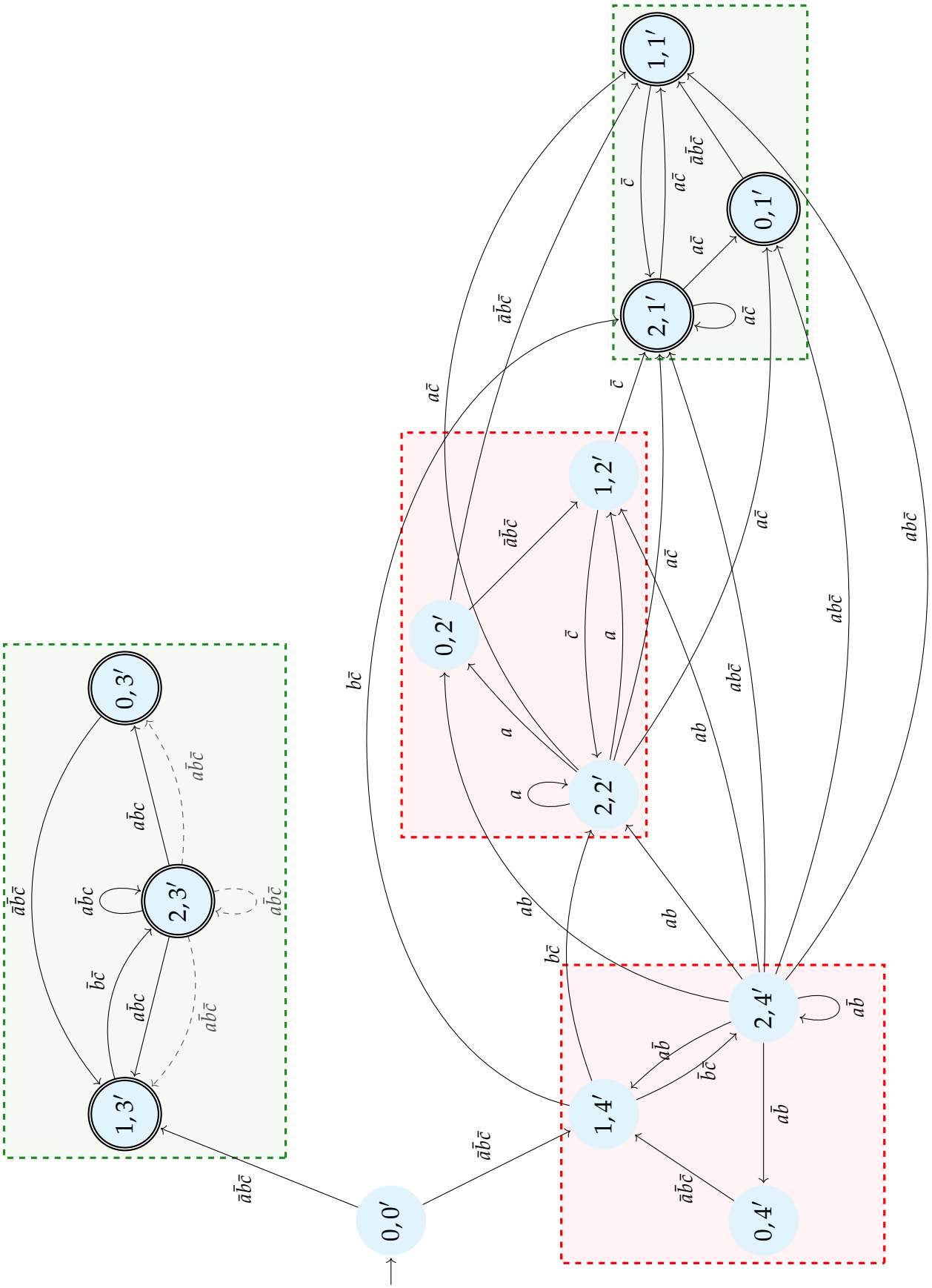


FIGURE 6.11: Synchronized product $\mathcal{A}_{S_3}$ between $\mathcal{A}_{\Lambda_3}$ and $\mathcal{A}_{\neg\varphi}$ of Example **6.4**

of the system's automaton $\mathcal{A}_{K_2}$. The computed synchronized product $\mathcal{A}_{S_2}$ depicted in Figure 6.11 contains thirteen states, two transient SCCs, and two accepting SCCs. **Note that dashed edges colored in gray are not part of the accepting paths transitions**.

There are multiple paths of length $k+1$ (k + initial state) leading to an accepting SCC:

$$
\begin{aligned}
&- (0,0') \xrightarrow{\bar{a}\bar{b}\bar{c}} (1,3') \xrightarrow{\bar{b}\bar{c}} (2,3') \xrightarrow{a\bar{b}c} (2,3') \\
&- (0,0') \xrightarrow{\bar{a}\bar{b}\bar{c}} (1,3') \xrightarrow{\bar{b}\bar{c}} (2,3') \xrightarrow{a\bar{b}c} (0,3') \\
&- (0,0') \xrightarrow{\bar{a}\bar{b}\bar{c}} (1,4') \xrightarrow{b\bar{c}} (2,1') \xrightarrow{a\bar{c}} (2,1') \\
&- (0,0') \xrightarrow{\bar{a}\bar{b}\bar{c}} (1,4') \xrightarrow{b\bar{c}} (2,1') \xrightarrow{a\bar{c}} (1,1') \\
&- (0,0') \xrightarrow{\bar{a}\bar{b}\bar{c}} (1,4') \xrightarrow{\bar{b}\bar{c}} (2,1') \xrightarrow{a\bar{c}} (0,1') \\
&- (0,0') \xrightarrow{\bar{a}\bar{b}\bar{c}} (1,4') \xrightarrow{\bar{b}\bar{c}} (2,4') \xrightarrow{ab\bar{c}} (1,1') \\
&- (0,0') \xrightarrow{\bar{a}\bar{b}\bar{c}} (1,4') \xrightarrow{\bar{b}\bar{c}} (2,4') \xrightarrow{ab\bar{c}} (0,1') \\
&- (0,0') \xrightarrow{\bar{a}\bar{b}\bar{c}} (1,4') \xrightarrow{\bar{b}\bar{c}} (2,4') \xrightarrow{ab\bar{c}} (2,1')
\end{aligned}
$$

From these paths, we can deduce a global constraint for each transition of the automaton $\mathcal{A}_{K_2}$:

(T1) The first transition from the initial state 0 to state 1 is already complete, so no additional information can be deduced from $\mathcal{A}_{S_2}$.

(T2) The second transition between states 1 and 2 results in:

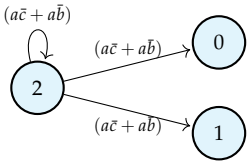
$$\bar{b}\bar{c} + \bar{b}\bar{c} + b\bar{c} + b\bar{c} + \bar{b}\bar{c} + \bar{b}\bar{c} + \bar{b}\bar{c} + \bar{b}\bar{c} \equiv \bar{c}$$

We can't deduce any extra information that we already have: $\bar{c}$.

(T3) The last transition between states 2 and 0 yields the following formula:

$$a\bar{b}c + a\bar{b}c + a\bar{c} + a\bar{c} + a\bar{c} + ab\bar{c} + ab\bar{b} + ab\bar{c} \equiv a\bar{c} + a\bar{b}$$

We deduce information specifying that for the transition from state 2 to other states of $\mathcal{A}_{K_2}$, it is required that:



The resulting formula to be sent to the solver is:

$$C_\alpha \implies (a \wedge \neg c) \vee (a \wedge \neg b)$$

To summarize, the operations performed by the black-box are as follows:

1. Creation of the Büchi automaton $\mathcal{A}_K$ associated with the partial assignment α sent by the SAT solver.

2. Computation of the synchronized product automaton $\mathcal{A}_S$ between $\mathcal{A}_K$ and $\mathcal{A}_{\neg\varphi}$.

3. Extraction of new information from $\mathcal{A}_S$: empty intersection, transition constraints.

This theoretical concept seeks to combine the two realms of model checking (explicit and symbolic), with the goal of uncovering structural information that is hidden from SAT procedures. All of this can be accomplished with minimal modifications to the SAT solver's code. The operations involving automata are carried out using SPOT framework [50]. This tool specializes in ω-automata and provides optimized algorithms, including synchronized product computation.

6.3 Interaction between Black-box and SAT solver

The internal processes of the black-box entail computationally intensive operations, which include generating automata from the provided partial solution α and performing synchronized product calculations. These operations impose significant demands on computational time and memory resources. Therefore, it is crucial to make well-informed decisions about when to initiate these processes. Moreover, flooding the solver with external clauses, can have counterproductive effects. Therefore, it becomes essential to control the volume of exchanged information.

In this section, we will thoroughly examine the communication process between these two entities and the strategies for managing the flow of calls and shared information. Algorithm 2 offers a succinct representation of the code lines integrated into the CDCL SAT solver. Importantly, the invocation of the black-box (CALLBLACKBOX) occurs only when the solver fails to detect a conflict (when unit variables are propagated without detecting any conflict).

(CDCL) → (Black-box): This involves a straightforward exchange initiated by the SAT solver towards the black-box (indicated by arrow **(1)** in Figure 6.1). As introduced earlier, this exchange entails sending a partial assignment α of the problem's variables. Importantly, this assignment is limited to variables within the property, spanning from the initial to the final time step k. Within the context of automata-based model checking, this partial solution α translates into set of finite paths leading to multiple infinite paths (executions). The black-box aims to evaluate the intersection of this set of executions with the language of the property's automaton.
However, calling upon the black-box can be redundant or unproductive, given the computational overhead associated with its use. To streamline the communication flow in this direction, it is prudent to restrict calls to the black-box (as indicated in line 9 of Algorithm 2) only when there have been substantial changes in the assignment α compared to the previous successful call (α'). For instance, when this threshold is set at 5%: it signifies that for the SAT solver to

Algorithm 2 CDCL[BMC] - *search* method

```
Require: F : CNF formula
learntCls: [ ]
α = { }, α' = { }
forever
    (F', α') ← UnitPropagation(F, α)
    if conflict then
        ...
    else
        ...
        if |α' ∩ α| ≥ X then
            learntCls ← callBlackbox(α)
            α' ← α
            for c ∈ learntCls do
                if Satisfied(c) then
                    continue
                else if Falsified(c) or Assertive(c) or LBD(c) ≤ Y then
                    addClause(c)
        ...
    ...
end for
```

engage the black-box, there must be a 5% change in property decision variables compared to the last successful call.

(Black-box) → (CDCL): The second exchange (arrow tagged by **(2)** in Figure 6.1) involves the black-box, which, upon receiving a partial solution α, constructs automata that recognize this assignment applying the procedure detailed in Section 6.2. The synchronization procedure generates constraints, which can be categorized based on their utility for the solver. Specifically, within the CDCL algorithm, when importing an external clause, that clause can be in one of four states: (i) all literals of the clause are $false$, (ii) The clause contains only one unassigned literal, (iii) the clause is already satisfied, or (iv) any other type of clauses that do not fall into the first three categories:

(i) Falsified clause. All variables in the clause are assigned to $false$. This type of clause is produced by the black-box when the intersection of the two automata languages is empty, leading to the creation of a clause that represents the negation of the provided assignment α ($C_{\neg\alpha}$). It has a significant impact on the solver as it systematically cuts off the current search subtree. Therefore, it is crucial and must be retained by the solver.

(ii) Assertive clause. The clause contains only one unassigned literal or one that has been assigned in the opposite polarity through the propagation process. Two situations arise here: (1) The variable corresponding to the last literal in the clause has not yet been assigned. The propagation process then forces the assignment of the remaining literal to satisfy the clause. (2) The variable corresponding to the last literal in the clause has already been assigned in the opposite polarity. This leads to a conflict, indicating a contradiction in the variable assignment. The solver must

backtrack and change this assignment. This is useful for the solver as it forces the satisfiability of the clause by propagating the implicated variable.

(iii) Satisfied clause. This clause is already satisfied, meaning it contains a literal that is *true* in the current assignment. These clauses are not of immediate interest during the resolution process. While it is possible to retain them and hope they become useful (falsified or assertive) at a later point in the resolution, in our study, we completely ignored them to avoid overburdening the solver given the substantial number of clauses received (line 13 in Algorithm 2).

(iv) Others. These are clauses that do not fall into any of the previous categories at the time of import. They contain many undefined variables. For these clauses, their level of importance is challenging to determine, and they can be numerous. We chose to limit the storage of these clauses based on their corresponding LBD values. As shown in Algorithm 2, the last condition in line 15 states that any clause that is neither $false$ nor assertive can be added to the solver's learnt clause database if its LBD is less than or equal to a fixed threshold.

Therefore, these restrictions on both sides ensure that the black-box is called at more critical moments, and the resulting information will not overwhelm the solver.

6.4 Discussion and future works

This study introduces a new way to leverage the structure of BMC instances. This idea allows combining two worlds: explicit and symbolic model checking, resulting in an intertwined integration of these two techniques with the objective of producing a robust summation of their orthogonal strengths.

As of now, we haven't achieved fruitful results, and there are several reasons for this. Firstly, it's primarily due to the parameters that regulate the number of calls and the quantity of exchanged clauses. Excessive communication with the black-box inevitably leads to longer times for automaton creation, intersection computation, and synchronized product automaton traversal. Thus, when the given partial assignment α contains few gaps, it implies that most of the property's variables are already assigned. In such cases, there is no need to invoke the black-box. Additionally, this also affects the size of the clauses exported by the black-box. The constraints defined in Formula 6.1 become larger as α becomes more complete. Similarly, the formulas f_i derived from synchronized product computations are not in clausal form (CNF). This means that if the formula f_i is large and complex, it will result in a substantial number of clauses.

Another factor that might influence the results is the quality of the information extracted from the synchronized product. Further investigation is required to determine whether more concise and informative data can be deduced.

The future directions of this work are being pursued. We are considering a shift to a parallel context in which the black-box communicates with multiple CDCL workers.

These workers will share their assignments α with the black-box. The black-box will process each received α from the workers and share the learnt clauses with all CDCL workers in the portfolio. This approach not only prevents the SAT solvers from being blocked but also ensures that the information learnt from the assignment α of one solver is shared with all workers. This prevents the other workers from making the same mistake.

Chapter 7

Conclusion

The literature has shown significant interest in improving bounded model checking using SAT procedures. Researchers have achieved this by adapting internal solver heuristics and by parallelizing problem resolution, breaking it down into specific subparts. All of these contributions have significantly accelerated the solving process. This heightened interest has driven the development of approaches that consider specific problem characteristics, such as the characteristics of propositional formula variables and formula symmetry, aspects that SAT procedures typically are unaware of.

This thesis contributes a tiny building block to this area. It focuses on extracting high-level information that characterizes bounded model checking problems with the goal of identifying and/or constructing relevant clauses, enabling the efficient removal of unnecessary subspaces by injecting these clauses into the SAT solver. The thesis proposes various orthogonal techniques for this purpose to improve the performance of SAT procedures in evaluating BMC instances, both in sequential and parallel contexts. Three primary axes of exploration were pursued:

Tune clause deletion policy. We observed that integrating additional information from the new classification clause metric, allowed us to propose an approach for detecting relevant learnt clauses based on this new metric. During our study, we observed that certain classes of learnt clauses were significantly more valuable to keep in memory than others due to their usefulness in the solving process. Therefore, we introduced two heuristics as H_S and H_{LP}, which rely on the LBD metric. These heuristics aid in determining the appropriate LBD value for each type of learnt clause, prioritizing the protection of useful categories to be kept in the clause database throughout the resolution process. Across the two leading modern SAT solvers, MAPLECOMSPS and KISSAT-MAB, the experiments demonstrated a significant improvement. Based on these results, we proposed using this classification method to identify relevant clauses for sharing among multiple workers in a portfolio-based environment. The experimental part of this research has reinforced the importance of considering clause composition variables. As a result, the methodology for classifying clauses can be applied to any CDCL solver and is adaptable to problems that can partition the variables that constitute the problem. All of this work is integrated into our BSALTIC framework.

Exploiter BMC formula structure. By leveraging the unique characteristics of BMC instances, we introduced a partitioning scheme that divides the formula into

several independent subparts. These subparts are structured in a way that they encapsulate successive states together. To reconcile them, we integrated Hamadi et al.'s reconciliation scheme [37], which involves communication between the subparts using Craig interpolation mechanisms. The end result is a global solution validated across all partitions. Our contribution involves proposing a BMC-based decomposition that isolates segments of the model's path. We can then reason about them independently, leading to the generation of more expressive interpolants compared to partitioning methods that don't consider the original problem's structure. We presented two approaches to leverage these interpolants: (1) by generating and injecting them into the original problem during the preprocessing phase of the SAT solver (this was experimented with in a sequential context), and (2) by introducing a solver into a portfolio-based approach that operates this decomposition and shares the interpolants returned by the partitions throughout the resolution process with other classical CDCL solvers that lack this partition view. In both cases, these interpolants, stemming from a partitioning approach that considers the problem's structure, significantly improved solver performance.

Combine explicit & symbolic model checking. A preliminary investigation was conducted to combine two worlds of model checking: explicit system representation through automata-based techniques and symbolic representation via Boolean SAT formula resolution. Here, the high-level knowledge from explicit model checking was integrated into SAT resolution. The concept of Programmatic SAT solving was employed to streamline the implementation in order to incorporate structural information into SAT problems without necessitating a deep dive into the SAT solver's code. To achieve this, we introduce an external component within the CDCL algorithm, referred to as the black-box. When invoked by the SAT solver and provided with the current variable assignments, this black-box operates in a manner akin to emptiness check algorithms between that compute the synchronized product automaton between the execution provided by the given assignment, which represents the model, and the evaluated property. This relies on the enumeration of Strongly Connected Components (SCCs). The objective is to identify SCCs that contain accepting cycles, indicating the presence of a path that violates the property. Consequently, the black-box extracts information in the form of learnt clauses, which are then injected to the SAT engine. Nevertheless, the experimental results yielded unpromising outcomes, and these findings are discussed in greater detail in the corresponding Chapter 6, along with potential directions for future work.

These three axes, which are completely independent, have each contributed in their own way to identifying or generating relevant learnt clauses. These works can be merged together to create a solver that is nearly specialized in solving BMC instances.

7.1 Short-term Perspectives

In addition to the theoretical work introduced in Chapter 6, which will be the subject of further investigation, we have simultaneously explored other interesting directions that extend the work presented in Chapter 4. The first idea involves considering the category of the evaluated LTL property according to the Hierarchy of Manna & Pnueli. The second idea is to apply the concept of clause classification to characterize relevant learnt clauses in the incremental version of BMC using incremental SAT solvers.

7.1.1 LTL-based tuning

One aspect that our previous research did not extensively investigate is the consideration of LTL properties category within the hierarchy defined by Manna & Pnueli. In this work, we dive deeper into tailoring the verification process to each class of the hierarchy. Our current focus is on investigating the impact of the clause classification C_X metric on each type of LTL property. This entails categorizing BMC problems based on the specific type of property under consideration. Indeed, we observed differences in the usage of the class of clauses when the analysis is performed on a partitioned benchmark according to the type of property being checked.

For more detailed insights, Section A.1 in Appendix A provides additional information about these analyses. We propose a new heuristic based on solving a linear program that, instead of producing multiple optimal solutions, changes the system's objective to find a single optimal solution. The linear system seeks to maximize the overall usage frequency of the classes C_X.

Up to this point, the experimental results using MAPLECOMSPS solver have not outperformed a generic approach that does not account for property classes (H_{LP} heuristic). This may be attributed to the proposed heuristic, which involves multiple parameters.

Ongoing work in this area focuses on refining the frequency heuristic to be applied in both sequential and parallel approaches, and extending this concept to recent CDCL solvers such as KISSAT-MAB [33].

7.1.2 Tuning learnt clauses in Incremental SAT-based BMC

This preliminary study is a result of a collaboration with Guillaume Carrières[1], focusing on incremental SAT-based BMC. Our goal was to investigate the impact of clause classification C_X and its role in identifying relevant learnt clauses within the context of incremental SAT-based BMC. As a part of this study, we extended the concept presented in Chapter 4 in an incremental manner. More precisely, we adapt the H_{LP} heuristic to compute a selector for identifying learnt clauses that require protection. This selector is refined dynamically, during the solving between two unrolling iterations.

Our purpose is to explore the potential of leveraging BMC's gradual nature to speed up the overall verification process. Incremental resolution provides us with the flexibility to adjust the selector from one iteration to the next, based on prior learning.

[1] An engineering student from EPITA

We investigated heuristics designed to enhance incremental SAT-based BMC solving by incorporating external information derived from the original model checking problem, particularly through the clause classification measure. A comprehensive description of our approach can be found in Section A.2 in Appendix A. The results indicate a slight increase in the number of solved instances. While these are promising findings, they also suggest that further exploration of using external BMC information is warranted. Additionally, we demonstrated that (MAX-MIN)-DEPTH of [121] could be considered as a substitute for LBD, albeit with occasional modest performance improvements. The nature of this measure makes it a valuable candidate for fine-tuning BMC solving, and further investigations into this metric, along with a broader comparison with modern incremental BMC approaches such as IC3 [60] and PDR [61] techniques, should be considered.

7.2 Long-term Perspectives

With the emergence of more sophisticated procedures, such as IC3, which leverage SAT solving techniques to approximate the search space at each time frame, it would be interesting to consider replacing the SAT solver with a solver more tailored to BMC, in line with one of the ideas presented in this manuscript.

Additionally, an intriguing aspect that hasn't yet surfaced from BMC features is whether the synchronicity of the system (synchronous or asynchronous models) being evaluated has different characteristics. Much like our attempt to exploit property categories, as presented in the short-term perspectives, segmenting the benchmark according to the synchronicity of the model could reveal additional insights or, at the very least, allow for parameter adjustments (i.e., the selector) based on the model's synchronicity.

In a parallelism context, this thesis has focused on BMC problem resolution through solver portfolios. It would be interesting to explore the second form of parallelism, which is the Divide and Conquer approach. More specifically, by employing the guiding path method [106], we can dynamically partition the search tree during resolution, employing the *work-stealing* concept. Given the structure of the BMC formula, the decomposition here would be intuitive and could, for instance, depend on the system's state variables.

Appendix A

Implementation details of ongoing works

A.1 LTL-based tuning of learnt clauses databases

The hierarchy of Manna & Pnueli [39] has been used to fine-tune explicit model-checkers [42–44]. These studies suggested various strategies, including a decomposition of the input automaton or propose optimizations for specific classes of the hierarchy. To the best of our knowledge, our idea is the first covering all the classes of this hierarchy within the context of SAT-based BMC. Most existing literature gives attention to safety or guarantee properties [57, 124] only, such as IC3 [140] or PDR [61] procedures. While there exist some rigorous methods to convert liveness properties into safety properties [151] for application in IC3/PDR approaches, our work delves deeper into tailoring the verification procedure for each class of this hierarchy. We investigate the impact of the clause classification metric on each type of property. This involves categorizing BMC problems based on the type of property under study.

Indeed, we observed differences in the analysis of the usage rate of the class of clauses[1], when we partition the benchmark based on the type of property being checked. Figures A.1 display these differences in the analysis when applied to *Recurrence* and *Reactivity* properties, respectively.

From these findings, we aim to identify the most suitable selector for each class of property. Instead of using the H_{LP} heuristic, we introduce a novel automated heuristic called "$H_{\mathcal{F}}$. This heuristic is based on a linear programming system that leverages insights gained from the preliminary study to compute a selector specific to each LTL category. This introduces a way of finding interesting selectors based on the usage frequency of the clause class C_X, precisely:

$$f = \frac{\%\text{usage of class } C_X}{\%\text{generated learnt class } C_X}$$

A.1.1 Optimization

The idea is to derive a selector that does not overpass a given threshold on the number of generated learnt clauses, while ensuring a higher usage frequency. To achieve

[1] Similar study as in Section 4.1 of Chapter 4

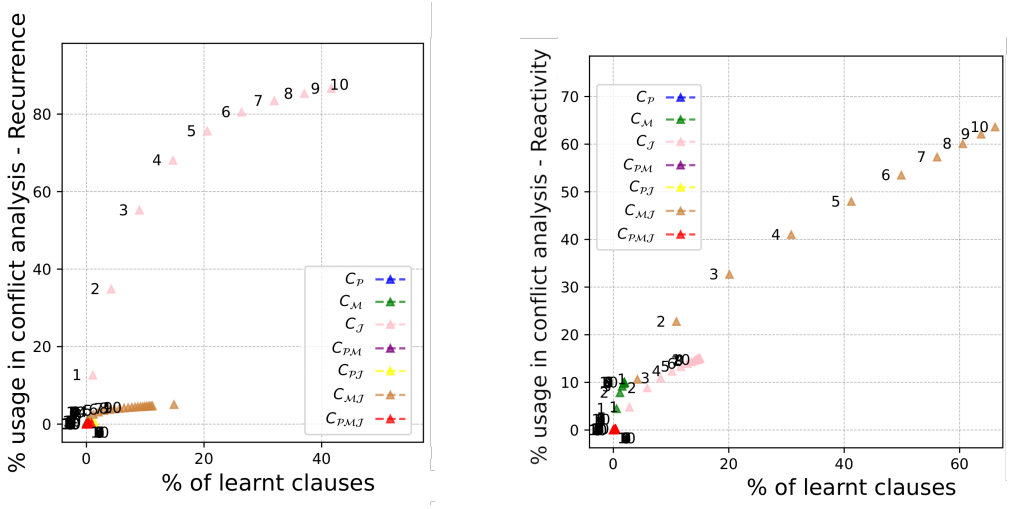


FIGURE A.1: Measures on the *training benchmark* with MAPLECOMSPS solver, showing learnt clauses usage in *conflict-analysis* phase. Each class of clauses is colored and annotated by its LBD value.

this, we formulate a linear program $\mathbf{M}(b,\epsilon)$, that seeks to maximize the total usage frequency while adhering to three major constraints:

(1) A constraint to ensure that the total number of learnt clauses does not exceed a specified threshold (LBD$\leq b$) with a precision of $\epsilon > 0$. For example, when using the MAPLECOMSPS engine, the program would concentrate on a number of learnt clauses, at most equal to $b = 3$, meaning, those of LBD$\leq$3, that construct the **core** database.

(2) A constraint which ensures that the total usage rate is, at worst, equal to that corresponding to LBD$\leq b$.

(3) Constraints that reflect the importance of each class of clauses. Indeed, from the Figures A.1, we can observe that some classes are more relevant than others ($C_{\mathcal{MJ}}$). This must have an influence on the resolution of the linear system. To account for this, we constraint the linear program to prioritize the protection of the more relevant classes of clauses by requiring a solution that provide them to have an LBD$\geq b$. To do so, we associate a probability to each class C_X, according to their relevance from prior study (Figures A.1). If the probability for a class is above a specified threshold, we require the program to generate a solution where the LBD score of that class is at least equal to b (LBD$\geq b$).

The formal description of the linear system needs the introduction of the following notations:

- $\mathcal{S}$=$\{\mathcal{P}, \mathcal{M}, \mathcal{J}, \mathcal{PM}, \mathcal{PJ}, \mathcal{MJ}, \mathcal{PMJ}\}$: denotes the set of classes.
- x_i^X: a Boolean variable representing the decision variable of the linear system. It takes the value 1 if the LBD$\leq$i is chosen for the class of clauses X, 0 if not.
- L_i^X: percentage of generated learnt clauses (*x-axis*) with LBD$\leq i$ and $X \in \mathcal{S}$.
- U_i^X: usage rate of class $X \in \mathcal{S}$ (*y-axis*) with an LBD$\leq i$.

- b: LBD value of reference used in the constraints.

- ϵ: precision gap.

Hence, our optimization problem, $\mathbf{M}(b,\epsilon)$, is as follows:

$$\mathbf{M}(b,\epsilon) = \quad \textbf{maximize} \sum_{X\in\mathcal{S}} \mathbf{f}_X = \sum_{i\geq 1}\sum_{X\in\mathcal{S}} \frac{\overbrace{x_i^X U_i^X}^{\%\text{ usage}}}{\underbrace{x_i^X L_i^X}_{\%\text{ learnt}}}$$

$$\text{s.t.} \begin{cases} \sum\limits_{i\geq 1}\sum\limits_{X\in\mathcal{S}} x_i^X L_i^X & \leq (\sum\limits_{i=1}^{b}\sum\limits_{X\in\mathcal{S}} L_i^X) + \epsilon & & \textbf{(1)} \\ \sum\limits_{X\in\mathcal{S}}\sum\limits_{i\geq 1} x_i^X U_i^X & \geq \sum\limits_{i=1}^{b}\sum\limits_{X\in\mathcal{S}} U_i^X & & \textbf{(2)} \\ \sum\limits_{i=1}^{\text{b - 1}} x_i^X & = 0 & \forall X\in\mathcal{S} & \textbf{(3)} \\ \sum\limits_{i\geq 1} x_i^X & = 1 & \forall X\in\mathcal{S} & \\ x_i^X \in \{0,1\} & \forall X\in\mathcal{S}, & \forall i, 0<i\leq 22 & \end{cases}$$

A.1.2 Discussion & perspectives

The experimental results have not yet outperformed the prior $\mathrm{H_{LP}}$ heuristic that does not account for property classes. This may be attributed to the proposed $\mathrm{H}_{\mathcal{F}}$ heuristic, which involves multiple parameters. We did not yet evaluated prior heuristic $\mathrm{H_{LP}}$ into a partitioned benchmark.

Ongoing work in this area focuses on refining the frequency heuristic to be applied in both sequential and parallel approaches, and extending this concept to recent CDCL solvers such as KISSAT-MAB [33].

A.2 Tuning learnt clauses in Incremental SAT-based BMC

The principle of an incremental solver, and more precisely an incremental CDCL solver [152, 153], applies when a problem requires checking the satisfiability of several similar formulas, i.e., containing a subset of common clauses. Rather than checking the satisfiability of each of the formulas one at a time, the idea is to take advantage of the learnt clauses that concern the part common to these formulas during successive calls. This particularity adapts naturally well to incremental BMC, by producing a multitude of similar SAT-based BMC instances with different bound k. Learnt clauses from previous bounds can be of benefit for solving the following bounds. However, not all the specificity of BMC is used though, where most CDCL-based SAT solvers use heuristics to perform the solving.

Our purpose is to investigate the possibility of exploiting BMC's gradual nature for speeding up the overall verification time. We will show how it is possible to exploit information gathered while solving a k-instance, for solving faster the consecutive instance. To do so, we will extend the idea of finding good-quality learnt clauses as presented in the previous Chapter 4 within an incremental manner. More precisely,

incremental resolution gives us the ability to adjust the selector from one iteration to another, based on what has been learnt previously.

Strichman [123] was among the first to observe that in BMC some clauses are known to survive through all instances in the sequence. A formula passed by BMC to the SAT solver contains clauses that describe the transition relation of the model unrolled a number of times. These clauses are not discarded when the length of the counterexample is increased, from k to $k+1$ for instance. Hence, a conflict clause that depends only on them can be forwarded. Its first application was originally done within the SATIRE [153] framework. Incremental SAT for BMC was also used to tune variable ordering heuristic [115, 154]. They enhanced the default variable order of the SAT solver with structural information from BMC without tampering with the internal details of the solver.

During this section, we will introduce the heuristics for building a selector in the incremental context. We'll also review the use of clause (MAX-MIN)-DEPTH, a metric presented in [121], to replace the generic LBD metric. We close the section with some preliminary experiments.

A.2.1 Identify relevant information dynamically

Static selectors to protect relevant learnt clauses don't fully exploit the potential of the incremental procedure. Similarly to variable ordering heuristics [115, 154], we can apply a dynamic approach that will recalculate the selector at the end of each iteration, thereby predicting an adequate patterns of learnt clauses that may be useful for the upcoming iterations. It allows for that, a better adaptability to the problem from a step to another.

Therefore, we have built a new dynamic procedure called $\mathrm{H_{LPD}}$ that is entirely based on the $\mathrm{H_{LP}}$ heuristic. Indeed, the objective remains the same: at the end of each iteration, we seek to maximize the usage rate of clauses during the *conflict-analysis* phase, whilst limiting their quantity. Let's recall the linear program formula:

$$\textbf{maximize } f_\mu = \mu \mathbf{O}_1 + (1-\mu)\mathbf{O}_2$$

$\mathrm{H_{LP}}$ lists all feasible solutions by varying the weight value $\mu \in [0,1]$. This parameter μ defines how much importance we give to the usefulness of clauses compared to their number. This means that the only parameter that changes between incremental and non-incremental is μ value. $\mathrm{H_{LP}}$ utilizes a static value of $\mu \in [0,1]$ to generate a solution, i.e. a selector. Only one of the multitude of solutions is chosen in the light of our observation. In the incremental scheme, however, the procedure has to be fully autonomous, so the suitable selector is automatically elected. The following points outline how to update μ value during the incremental solving:

1. **Initialization**: Firstly, we arbitrarily initialized μ to 0.5, providing equal importance to both objectives $\mathbf{O}_1$ and $\mathbf{O}_2$.

2. **μ-value**: Secondly, to determine when it's worth prioritizing the usage rate criteria (increasing μ) over learnt clauses generation (decreasing μ) and inversely, we've based our measurements on the number of unit propagation per the

number of performed conflicts. We define the Performance value as follow:

$$\textbf{\textit{Perf}}_k = \frac{\#(\text{unit propagation})_k}{\#(\text{conflict literals})_k}$$

So the more propagation occur the better, meaning that learnt clauses we are protecting from the deletion are used and contributes in the solving.

3. μ**-refinement**: Lastly, we need to update μ value whenever we move to the next bound, so the resulting selector will be more appropriate. μ is increased whenever the average performance of previous bounds is inferior to the current average performance. Conversely, μ is then decreased. In this way, using the Performance value will allow to converge to the ideal μ.

The selector retrieved by H_{LPD} will protect potential relevant clauses using a soften protection and act around the purge of the temporary databases of MAPLECOMSPS solver (**tier-2** and **local**) rather than an aggressive protection (**core**), where they are never deleted. Indeed, in this incremental context, the learnt clauses stored in the permanent database since the first iteration are never removed. By adding them to the temporary databases, we allow the solver to potentially discard the additional clauses obtained from the selector, thereby preventing an overload of the solver.

Although we did tried some variations of H_{LPD}. Here are two of them:

Use previous iterations (H_{LPD}-R). When solving a specific bound k, learnt clause usage can vary greatly that sometimes the resulting selector provides extreme variation in comparison to the previous iteration $k-1$. To solve this problem, instead of limiting the computation of selector on one iteration's collected information, we can reuse information of all previous solving. We use for that, the weighted average across iterations of LBD value. Thus, recent steps have a heavier weight since they are more relevant than earlier ones. We arbitrarily set the weight to $1 - \frac{1}{2^k}$.

The advantage in considering previous steps is an increase in stability, with less sensibility to individual variations in the learnt clauses usage.

Use of (MAX-MIN)-DEPTH measure (H_{LPD}-D). As described in [121], the (MAX-MIN)-DEPTH metric has been shown to be correlated to the LBD metric albeit with much more information. We were interested on computing selectors using the (MAX-MIN)-DEPTH measure in replacement of the LBD.

Example: a BMC problem unrolled up to bound $k = 3$ with set variables $\{x_i, y_i, z_i\}$ for $i \in \{0,1,2,3\}$ encoding the step i. We have the set of clauses $\mathcal{F} = (x_0 \vee \neg x_1 \vee y_0) \wedge (x_1 \vee z_1) \wedge (x_1 \vee \neg y_2 \vee \neg z_0)$. The (MAX-MIN)-DEPTH value for each clause is respectively $1, 0$, and 3.

To this end, we can redefine the selector as a configuration where each clause of class has its corresponding (MAX-MIN)-DEPTH value.

Heuristic	# Solved instances
H_{LPD}-**D**	**232**
H_{LPD}-**RD**	230
MAPLECOMSPS	229
H_{LPD}-**R**	229
H_{LPD}	228

TABLE A.1: Number of solved instances by each heuristic

Heuristic	# Solved bounds
H_{LPD}-**R**	**54147**
H_{LPD}	53810
MAPLECOMSPS	53784
H_{LPD}-**RD**	53730
H_{LPD}-**D**	53724

TABLE A.2: Number of solved bounds by each heuristic

A.2.2 Preliminary Experiments

We experiment the H_{LPD} heuristic and its two modified versions: H_{LPD}-**R** and H_{LPD}-**D**, plus the combination of both H_{LPD}-**RD**. We compared them to state-of-the-art incremental MapleCOMSPS on a benchmark of 754 SMV instances, with a mix of SAT, UNSAT and UNKNOWN problems. For each instance, an incremental BMC resolution using each configuration has been executed with a time limit of 6000 seconds and a bound limit of 10000, although this bound limit has never been reached.

Table A.1 displays the number of solved instances while Table A.2 shows the total sum of bounds solved by each configuration (in order to distinguish performance on potential UNSAT problems).

First observation is that from both tables, our heuristics perform slightly better than state-of-the-art with 3 SAT instances more by H_{LPD}-**D** and 363 more iterations by H_{LPD}-**R** approach than MAPLECOMSPS. Meaning that, H_{LPD}-**D** seems to perform generally well on SAT problems where H_{LPD}-**R** performs well on the long-term when solving potentially UNSAT instances.

Though none of the approaches shows up, it is noticeable that using previous iterations seems to bring better results on both criteria - in terms of solved instances and reached depths.

A.2.3 Discussion & perspectives

This section presents an overview of incremental SAT-based BMC optimization. We explored numerous different heuristics designed to improve the MapleCOMSPS incremental SAT solver on BMC problems using external information coming from the original model checking problem, notably through the classification measure. These heuristics span on multiple aspects of the solver such as clause storage or variable ordering that we did not present here. The result is a small increase of solved instances with our best performing heuristics. These are promising results, and show that further work on using external BMC information should be considered. We have also shown that (MAX-MIN)-DEPTH could be used as a replacement for LBD albeit with an occasional modest performance's improvement. However, the nature of this measure still makes it a good candidate for tuning the BMC solving, and further investigation on this metric could be considered.

Bibliography

[1] Kenneth L. McMillan. "The SMV System". In: *Symbolic Model Checking*. Boston, MA: Springer US, 1993, pp. 61–85. ISBN: 978-1-4615-3190-6.

[2] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. USA: Prentice-Hall, Inc., 1996. ISBN: 0134516753.

[3] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. First. Addison-Wesley Professional, 2003. ISBN: 0321228626.

[4] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[5] Kristin Y. Rozier. "Survey: Linear Temporal Logic Symbolic Model Checking". In: *Comput. Sci. Rev.* 5.2 (May 2011), pp. 163–203. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2010.06.002. URL: https://doi.org/10.1016/j.cosrev.2010.06.002.

[6] Edmund M. Clarke and E. Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Logics of Programs*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71. ISBN: 978-3-540-39047-3.

[7] Edmund Clarke, E. Emerson, and Joseph Sifakis. "Model checking". In: *Communications of the ACM* 52 (Nov. 2009). DOI: 10.1145/1592761.1592781.

[8] Gerard J. Holzmann. "Explicit-State Model Checking". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 153–171. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_5. URL: https://doi.org/10.1007/978-3-319-10575-8_5.

[9] E. Clarke et al. "Symbolic model checking". In: *Computer Aided Verification*. Ed. by Rajeev Alur and Thomas A. Henzinger. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 419–422. ISBN: 978-3-540-68599-9.

[10] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Comput.* 35.8 (Aug. 1986), pp. 677–691. ISSN: 0018-9340. DOI: 10.1109/TC.1986.1676819. URL: https://doi.org/10.1109/TC.1986.1676819.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN: 026202649X.

[12] J. Richard Büchi. "On a Decision Method in Restricted Second Order Arithmetic". In: *The Collected Works of J. Richard Büchi*. Ed. by Saunders Mac Lane and Dirk Siefkes. New York, NY: Springer New York, 1990, pp. 425–435. ISBN: 978-1-4613-8928-6. DOI: 10.1007/978-1-4613-8928-6_23. URL: https://doi.org/10.1007/978-1-4613-8928-6_23.

[13] M.Y. Vardi and P. Wolper. "Reasoning about Infinite Computations". In: *Information and Computation* 115.1 (1994), pp. 1–37. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1092.

[14] Edmund M. Clarke et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35746-6.

[15] Armin Biere et al. “Symbolic Model Checking without BDDs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-49059-3.

[16] Jerry R. Burch et al. “Symbolic model checking for sequential circuit verification”. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 13.4 (1994), pp. 401–424. DOI: 10.1109/43.275352. URL: TCAD94.pdf.

[17] Dirk Beyer and Andreas Stahlbauer. “BDD-Based Software Model Checking with CPAchecker”. In: *Mathematical and Engineering Methods in Computer Science*. Ed. by Antonín Kučera et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–11. ISBN: 978-3-642-36046-6.

[18] Malay K. Ganai. “SAT-Based Scalable Formal Verification Solutions”. In: *Series on Integrated Circuits and Systems, Springer-Verlag New York*. 2007.

[19] Armin Biere et al. *Bounded Model Checking*. Dec. 2003. DOI: 10.1016/S0065-2458(03)58003-2.

[20] Armin Biere and Daniel Kröning. “SAT-Based Model Checking”. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 277–303. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_10. URL: https://doi.org/10.1007/978-3-319-10575-8_10.

[21] Edmund Clarke et al. “Bounded Model Checking Using Satisfiability Solving”. In: *Form. Methods Syst. Des.* 19.1 (July 2001), pp. 7–34. ISSN: 0925-9856. DOI: 10.1023/A:1011276507260. URL: https://doi.org/10.1023/A:1011276507260.

[22] Emmanuel Zarpas. “Simple Yet Efficient Improvements of SAT Based Bounded Model Checking”. In: *Formal Methods in Computer-Aided Design*. Ed. by Alan J. Hu and Andrew K. Martin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 174–185. ISBN: 978-3-540-30494-4.

[23] Laurent Simon and Gilles Audemard. “Predicting Learnt Clauses Quality in Modern SAT Solver”. In: *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*. Pasadena, United States, July 2009. URL: https://hal.inria.fr/inria-00433805.

[24] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. “The Community Structure of SAT Formulas”. In: *Theory and Applications of Satisfiability Testing – SAT 2012*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 410–423. ISBN: 978-3-642-31612-8.

[25] Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. “Symmetric Explanation Learning: Effective Dynamic Symmetry Handling for SAT”. In: Aug. 2017. ISBN: 978-3-319-66262-6. DOI: 10.1007/978-3-319-66263-3_6.

[26] Matthew L. Ginsberg and David A. McAllester. “GSAT and Dynamic Backtracking.” In: *PPCP*. Ed. by Alan Borning. Vol. 874. Lecture Notes in Computer Science. Springer, 1994, pp. 243–265. ISBN: 3-540-58601-6. URL: http://dblp.uni-trier.de/db/conf/ppcp/ppcp94-lncs.html#GinsbergM94.

[27] James M. Crawford et al. "Symmetry-Breaking Predicates for Search Problems". In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*. KR'96. Cambridge, Massachusetts, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 148–159. ISBN: 1558604219.

[28] F.A. Aloul, K.A. Sakallah, and I.L. Markov. "Efficient symmetry breaking for Boolean satisfiability". In: *IEEE Transactions on Computers* 55.5 (2006), pp. 549–558. DOI: 10.1109/TC.2006.75.

[29] 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*. ICCAD '96. San Jose, California, USA: IEEE Computer Society, 1997, pp. 220–227. ISBN: 0818675977.

[30] Matthew W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver." In: *DAC*. ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2. URL: http://dblp.uni-trier.de/db/conf/dac/dac2001.html#MoskewiczMZZM01.

[31] Lintao Zhang et al. "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver". In: *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '01. San Jose, California: IEEE Press, 2001, pp. 279–285. ISBN: 0780372492.

[32] Jia Hui Liang et al. "Maple-comsps, maplecomsps lrb, maplecomsps chb". In: *Proceedings of SAT Competition* 2016 (2016).

[33] Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. "Un bandit manchot pour combiner CHB et VSIDS". In: *Actes des 16èmes Journées Francophones de Programmation par Contraintes (JFPC)*. Nice, France, June 2021. URL: https://hal-amu.archives-ouvertes.fr/hal-03270931.

[34] Vallade Vincent et al. "New Concurrent and Distributed Painless solvers: P-MCOMSPS P-MCOMSPS-COM P-MCOMSPS-MPI and P-MCOMSPS-COM-MPI". In: 2021.

[35] Cai S. Zhang X. Chen Z. "ParKissat: Random Shuffle Based and Pre-processing Extended Parallel Solvers with Clause Sharing". In: 2022, p. 51.

[36] Burton Dreben. "William Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. The journal of symbolic logic, vol. 22 (1957), pp. 250–268. - William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. The journal of symbolic logic, vol. 22 (1957), pp. 269–285." In: *Journal of Symbolic Logic* 24.3 (1959), pp. 243–244. DOI: 10.2307/2963831.

[37] Youssef Hamadi, Joao Marques-Silva, and Christoph Wintersteiger. "Lazy Decomposition for Distributed Decision Procedures". In: *Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'11)*. Vol. 72. Nov. 2011, pp. 43–54.

[38] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 And Beyond*. Tech. rep. 11/2. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, 2011.

[39] Z. Manna and A. Pnueli. "A hierarchy of temporal properties (invited paper, 1989)". In: *PODC '90*. 1990.

[40] Bowen Alpern and Fred B. Schneider. "Recognizing Safety and Liveness". In: *Distrib. Comput.* 2.3 (Sept. 1987), pp. 117–126. ISSN: 0178-2770. DOI: 10.1007/BF01782772.

[41] Doron Peled and Klaus Havelund. "Refining the Safety–Liveness Classification of Temporal Properties According to Monitorability". In: *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard*

Steffen on the Occasion of His 60th Birthday. Cham: Springer International Publishing, 2019, pp. 218–234. ISBN: 978-3-030-22348-9.

[42] Etienne Renault et al. "Strength-Based Decomposition of the Property Büchi Automaton for Faster Model Checking". In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 580–593.

[43] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. "Directed explicit-state model checking in the validation of communication protocols". In: *International Journal on Software Tools for Technology Transfer* 5.2–3 (2004), pp. 247–267.

[44] Sami Evangelista et al. "Improved Multi-Core Nested Depth-First Search". In: *Proceedings of the 10th international conference on Automated technology for verification and analysis (ATVA'12)*. Vol. 7561. Lecture Notes in Computer Science. Springer-Verlag, 2012, pp. 269–283.

[45] Ken L. McMillan. "Applying SAT Methods in Unbounded Symbolic Model Checking". In: *Computer Aided Verification*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 250–264. ISBN: 978-3-540-45657-5.

[46] L. Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Trans. Softw. Eng.* 3.2 (Mar. 1977), pp. 125–143. ISSN: 0098-5589. DOI: 10.1109/TSE.1977.229904. URL: https://doi.org/10.1109/TSE.1977.229904.

[47] M.Y. Vardi and P. Wolper. "An automata-theoretic approach to automatic program verification". In: *In Proceedings of the 1st Symposium on Logic in Computer Science*. Cambridge, Massachusetts, USA, 1986, pp. 322–331.

[48] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. "The complementation problem for Büchi automata with applications to temporal logic". In: *Theoretical Computer Science* 49.2 (1987), pp. 217–237. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90008-9. URL: https://www.sciencedirect.com/science/article/pii/0304397587900089.

[49] Moshe Y. Vardi. "The Büchi Complementation Saga". In: *Proceedings of the 24th Annual Conference on Theoretical Aspects of Computer Science*. STACS'07. Aachen, Germany: Springer-Verlag, 2007, pp. 12–22. ISBN: 9783540709176.

[50] Alexandre Duret-Lutz et al. "Spot 2.0 — a framework for LTL and ω-automata manipulation". In: *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*. Vol. 9938. Lecture Notes in Computer Science. Springer, Oct. 2016, pp. 122–129.

[51] Jan Křetínský, Tobias Meggendorfer, and Salomon Sickert. "Owl: A Library for omega-Words, Automata, and LTL: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings". In: Sept. 2018, pp. 543–550. ISBN: 978-3-030-01089-8. DOI: 10.1007/978-3-030-01090-4_34.

[52] Costas Courcoubetis et al. "Memory-Efficient Algorithms for the Verification of Temporal Properties." In: *Formal Methods in System Design* 1.2/3 (1992), pp. 275–288. URL: http://dblp.uni-trier.de/db/journals/fmsd/fmsd1.html%5C#CourcoubetisVWY92.

[53] Jaco Geldenhuys and Antti Valmari. "Tarjan's Algorithm Makes On-the-Fly LTL Verification More Efficient". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2004, pp. 205–219. ISBN: 978-3-540-24730-2.

[54] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[55] Paul Jackson and Daniel Sheridan. "Clause Form Conversions for Boolean Circuits". In: *Theory and Applications of Satisfiability Testing*. Ed. by Holger H. Hoos and David G. Mitchell. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 183–198. ISBN: 978-3-540-31580-3.

[56] G. S. TSEITIN. "On the complexity of derivation in propositional calculus". In: *Structures in Constructive Mathematics and Mathematical Logic* (1968), pp. 115–125. URL: https://ci.nii.ac.jp/naid/10030021172/en/.

[57] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *Formal Methods in Computer-Aided Design*. Ed. by Warren A. Hunt and Steven D. Johnson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 127–144. ISBN: 978-3-540-40922-9.

[58] K. L. McMillan. "Interpolation and SAT-Based Model Checking". In: *Computer Aided Verification*. Ed. by Warren A. Hunt and Fabio Somenzi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–13. ISBN: 978-3-540-45069-6.

[59] Kenneth L. McMillan. "Interpolants and Symbolic Model Checking". In: *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*. 2007, pp. 89–90. DOI: 10.1007/978-3-540-69738-1_6. URL: https://doi.org/10.1007/978-3-540-69738-1%5C_6.

[60] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87. ISBN: 978-3-642-18275-4.

[61] Niklas Een, Alan Mishchenko, and Robert Brayton. "Efficient implementation of property directed reachability". In: *2011 Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 125–134.

[62] Jin-Kao Hao and Raphaël Dorne. "An Empirical Comparison of Two Evolutionary Methods for Satisfiability Problems". In: *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, June 27-29, 1994*. IEEE, 1994, pp. 451–455. DOI: 10.1109/ICEC.1994.349908. URL: https://doi.org/10.1109/ICEC.1994.349908.

[63] Shaowei Cai, Chuan Luo, and Kaile Su. "CCAnr: A Configuration Checking Based Local Search Solver for Non-random Satisfiability". In: *Theory and Applications of Satisfiability Testing – SAT 2015*. Ed. by Marijn Heule and Sean Weaver. Cham: Springer International Publishing, 2015, pp. 1–8. ISBN: 978-3-319-24318-4.

[64] Pierre Hansen, Nenad Mladenovic, and Dionisio Perez-Britos. "Variable Neighborhood Decomposition Search". In: *Journal of Heuristics* 7 (July 2001), pp. 335–350. DOI: 10.1023/A:1011336210885.

[65] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-Proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397.

ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557.

[66] Nicolas Szczepanski. "Thesis: SAT en paralléle". In: (2017). URL: http://www.cril.univ-artois.fr/~szczepanski/res/SATenParalleleSzczepanskiNic pdf.

[67] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. "DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs". In: *Theory and Applications of Satisfiability Testing – SAT 2014*. Ed. by Carsten Sinz and Uwe Egly. Cham: Springer International Publishing, 2014, pp. 422–429. ISBN: 978-3-319-09284-3.

[68] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: https://doi.org/10.1145/321033.321034.

[69] João Marques-Silva. "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms". In: *Progress in Artificial Intelligence*. Ed. by Pedro Barahona and José J. Alferes. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 62–74. ISBN: 978-3-540-48159-1.

[70] Paolo Liberatore. "On the complexity of choosing the branching literal in DPLL". In: *Artificial Intelligence* 116.1 (2000), pp. 315–326. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(99)00097-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370299000971.

[71] Chu Min Li and Anbulagan Anbulagan. "Heuristics Based on Unit Propagation for Satisfiability Problems". In: *Proceedings of the 15th International Joint Conference on Artifical Intelligence - Volume 1*. IJCAI'97. Nagoya, Japan: Morgan Kaufmann Publishers Inc., 1997, pp. 366–371.

[72] Jia Hui Liang et al. "Exponential Recency Weighted Average Branching Heuristic for SAT Solvers". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, 2016, pp. 3434–3440.

[73] Jia Hui Liang et al. "Learning Rate Based Branching Heuristic for SAT Solvers". In: *Theory and Applications of Satisfiability Testing – SAT 2016*. Ed. by Nadia Creignou and Daniel Le Berre. Cham: Springer International Publishing, 2016, pp. 123–140. ISBN: 978-3-319-40970-2.

[74] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA, 1998. URL: https://www.worldcat.org/oclc/37293240.

[75] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. "Towards a Better Understanding of the Functionality of a Conflict-Driven SAT Solver". In: *Theory and Applications of Satisfiability Testing – SAT 2007*. Ed. by João Marques-Silva and Karem A. Sakallah. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 287–293. ISBN: 978-3-540-72788-0.

[76] Daniel Frost and Rina Dechter. "Dead-End Driven Learning". In: *Proceedings of the National Conference on Artificial Intelligence* 1 (Aug. 2000).

[77] Gilles Audemard and Laurent Simon. "Extreme Cases in SAT Problems". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 87–103. DOI: 10.1007/978-3-319-40970-2_7. URL: https://doi.org/10.1007/978-3-319-40970-2%5C_7.

[78] Carla P. Gomes et al. "Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems". In: *J. Autom. Reason.* 24.1/2 (2000), pp. 67–100. DOI: 10.1023/A:1006314320276. URL: https://doi.org/10.1023/A:1006314320276.

[79] Niklas Eén and Niklas Sörensson. "Minisat 2.1 and minisat++ 1.0 sat race 2008 editions". In: 2008.

[80] Michael Luby, Alistair Sinclair, and David Zuckerman. "Optimal speedup of Las Vegas algorithms". In: *Information Processing Letters* 47 (Apr. 1997), pp. 173–180. DOI: 10.1016/0020-0190(93)90029-9.

[81] Gilles Audemard and Laurent Simon. "GLUCOSE: a solver that predicts learnt clauses quality". In: (Jan. 2009).

[82] Jia Hui Liang et al. "Machine Learning-Based Restart Policy for CDCL SAT Solvers". In: *Theory and Applications of Satisfiability Testing – SAT 2018*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Cham: Springer International Publishing, 2018, pp. 94–110. ISBN: 978-3-319-94144-8.

[83] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *International Conference on Theory and Applications of Satisfiability Testing*. 2003.

[84] Matti Järvisalo, Marijn Heule, and Armin Biere. "Inprocessing Rules". In: *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR 2012)*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Vol. 7364. Lecture Notes in Computer Science. Springer, 2012, pp. 355–370.

[85] Katalin Fazekas, Armin Biere, and Christoph Scholl. "Incremental Inprocessing in SAT Solving". In: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, 2019, pp. 136–154.

[86] Niklas Eén and Armin Biere. "Effective Preprocessing in SAT Through Variable and Clause Elimination". In: *Theory and Applications of Satisfiability Testing*. Ed. by Fahiem Bacchus and Toby Walsh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 61–75. ISBN: 978-3-540-31679-4.

[87] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. "NiVER: Non-increasing Variable Elimination Resolution for Preprocessing SAT Instances". In: *Theory and Applications of Satisfiability Testing*. Ed. by Holger H. Hoos and David G. Mitchell. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 276–291. ISBN: 978-3-540-31580-3.

[88] Norbert Manthey, Marijn Heule, and Armin Biere. "Automated Reencoding of Boolean Formulas". In: Jan. 2013, pp. 102–117. ISBN: 9783642396106. DOI: 10.1007/978-3-642-39611-3_14.

[89] Cedric Piette, Youssef Hamadi, and Lakhdar Sais. "Vivifying Propositional Clausal Formulae". In: Jan. 2008, pp. 525–529. DOI: 10.3233/978-1-58603-891-5-525.

[90] Niklas Sörensson and Armin Biere. "Minimizing Learned Clauses". In: *Theory and Applications of Satisfiability Testing - SAT 2009*. Ed. by Oliver Kullmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 237–243. ISBN: 978-3-642-02777-2.

[91] Hyojung Han and Fabio Somenzi. "On-the-Fly Clause Improvement". In: *Theory and Applications of Satisfiability Testing - SAT 2009*. Ed. by Oliver Kullmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 209–222. ISBN: 978-3-642-02777-2.

[92] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. "Learning for Dynamic subsumption". In: *CoRR* abs/0904.0029 (2009). arXiv: 0904.0029. URL: http://arxiv.org/abs/0904.0029.

[93] Mao Luo et al. "An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. IJCAI'17. Melbourne, Australia: AAAI Press, 2017, pp. 703–711. ISBN: 9780999241103.

[94] Carlos Ansótegui et al. *Community Structure in Industrial SAT Instances*. 2019. arXiv: 1606.03329 [cs.AI].

[95] Carlos Ansótegui et al. "Using Community Structure to Detect Relevant Learnt Clauses". In: *Theory and Applications of Satisfiability Testing – SAT 2015*. Ed. by Marijn Heule and Sean Weaver. Cham: Springer International Publishing, 2015, pp. 238–254. ISBN: 978-3-319-24318-4.

[96] Chunxiao Li et al. "On the Hierarchical Community Structure of Practical SAT Formulas". In: *CoRR* abs/2103.14992 (2021). arXiv: 2103.14992. URL: https://arxiv.org/abs/2103.14992.

[97] Vincent Vallade et al. "Community and LBD-Based Clause Sharing Policy for Parallel SAT Solving". In: June 2020, pp. 11–27. ISBN: 978-3-030-51824-0. DOI: 10.1007/978-3-030-51825-7_2.

[98] Hakan Metin et al. "CDCLSym: Introducing Effective Symmetry Breaking in SAT Solving". In: *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18)*. Vol. 10805. Lecture Notes in Computer Science. Thessaloniki, Greece: Springer, Apr. 2018, pp. 99–114.

[99] Belaid Benhamou et al. "Enhancing Clause Learning by Symmetry in SAT Solvers". In: vol. 1. Nov. 2010, pp. 329–335. DOI: 10.1109/ICTAI.2010.55.

[100] Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. "Symmetric Explanation Learning: Effective Dynamic Symmetry Handling for SAT". In: Aug. 2017. ISBN: 978-3-319-66262-6. DOI: 10.1007/978-3-319-66263-3_6.

[101] Jo Devriendt et al. "Improved Static Symmetry Breaking for SAT". In: July 2016, pp. 104–122. ISBN: 978-3-319-40969-6. DOI: 10.1007/978-3-319-40970-2_8.

[102] S. Saouli et al. "CosySEL: Improving SAT Solving Using Local Symmetries". In: *24th International Conference on Verification, Model Checking, and Abstract Interpretation*. Vol. 13881. Springer, Jan. 2023, pp. 252–266. DOI: 10.1007/978-3-031-24950-1_12. URL: https://doi.org/10.1007/978-3-031-24950-1%5C_12.

[103] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. "ManySAT: a Parallel SAT Solver." In: *J. Satisf. Boolean Model. Comput.* 6.4 (2009), pp. 245–262. URL: http://dblp.uni-trier.de/db/journals/jsat/jsat6.html#HamadiJS09.

[104] Long Guo et al. "Diversification and Intensification in Parallel SAT Solving". In: *Principles and Practice of Constraint Programming – CP 2010*. Ed. by David Cohen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 252–265. ISBN: 978-3-642-15396-9.

[105] Tomohiro Sonobe and Mary Inaba. "Portfolio with Block Branching for Parallel SAT Solvers". In: *Revised Selected Papers of the 7th International Conference on Learning and Intelligent Optimization - Volume 7997*. LION 7. Catania, Italy: Springer-Verlag, 2013, pp. 247–252. ISBN: 9783642449727. DOI: 10.1007/978-3-642-44973-4_25. URL: https://doi.org/10.1007/978-3-642-44973-4_25.

[106] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. "PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems". In: *Journal of Symbolic Computation* 21 (Dec. 1996), pp. 543–560. DOI: 10.1006/jsco.1996.0030.

[107] Marijn J. H. Heule et al. "Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads". In: *Hardware and Software: Verification and Testing*. Ed. by Kerstin Eder, João Lourenço, and Onn Shehory. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 50–65. ISBN: 978-3-642-34188-5.

[108] Bernard Jurkowiak, Chu Min Li, and Gil Utard. "Parallelizing Satz Using Dynamic Workload Balancing". In: *Electronic Notes in Discrete Mathematics* 9 (2001). LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001), pp. 174–189. ISSN: 1571-0653. DOI: https://doi.org/10.1016/S1571-0653(04)00321-X. URL: https://www.sciencedirect.com/science/article/pii/S157106530400321X.

[109] Gilles Audemard et al. "Revisiting Clause Exchange in Parallel SAT Solving". In: *15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*. Vol. 7962. Lecture Notes in Computer Science (LNCS). Trento, Italy: Springer, 2012, pp. 200–213. URL: https://hal.archives-ouvertes.fr/hal-00865596.

[110] Michael Kaufmann et al. "SArTagnan - a parallel portfolio SAT solver with lockless physical clause sharing". In: *In Pragmatics of SAT*. 2011.

[111] Said Jabbour et al. "Cooperation control in Parallel SAT Solving: a Multi-armed Bandit Approach". In: *Workshop on Bayesian Optimization and Decision Making*. Lake Tahoe, United States, 2012. URL: https://hal.archives-ouvertes.fr/hal-00870946.

[112] Gilles Audemard and Laurent Simon. "Lazy Clause Exchange Policy for Parallel SAT Solvers". In: *Theory and Applications of Satisfiability Testing – SAT 2014*. Ed. by Carsten Sinz and Uwe Egly. Cham: Springer International Publishing, 2014, pp. 197–205. ISBN: 978-3-319-09284-3.

[113] Youssef Hamadi, Said Jabbour, and Jabbour Sais. "Control-Based Clause Sharing in Parallel SAT Solving". In: *Autonomous Search*. Ed. by Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 245–267. ISBN: 978-3-642-21434-9. DOI: 10.1007/978-3-642-21434-9_10. URL: https://doi.org/10.1007/978-3-642-21434-9_10.

[114] Tomas Balyo, Peter Sanders, and Carsten Sinz. *HordeSat: A Massively Parallel Portfolio SAT Solver*. 2015. arXiv: 1505.03340 [cs.LO].

[115] Chao Wang et al. "Refining the SAT Decision Ordering for Bounded Model Checking". In: *Proceedings of the 41st Annual Design Automation Conference*. DAC '04. San Diego, CA, USA: Association for Computing Machinery, 2004, pp. 535–538. ISBN: 1581138288. DOI: 10.1145/996566.996713. URL: https://doi.org/10.1145/996566.996713.

[116] Liangze Yin, Fei He, and Ming Gu. "Optimizing the SAT Decision Ordering of Bounded Model Checking by Structural Information". In: *Proceedings of the 2013 International Symposium on Theoretical Aspects of Software Engineering*. TASE '13. USA: IEEE Computer Society, 2013, pp. 23–26. ISBN: 9780769550534. DOI: 10.1109/TASE.2013.11. URL: https://doi.org/10.1109/TASE.2013.11.

[117] Ofer Shtrichman. "Tuning SAT Checkers for Bounded Model Checking". In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad

Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 480–494. ISBN: 978-3-540-45047-4.

[118] Linus Feiten. "Development and Analysis of Decision Heuristics for an Interval Constraint Solver Handling Non-linear Arithmetic". PhD thesis. Diplomarbeit, Albert-Ludwigs-Universität Freiburg im Breisgau, Germany, 2010 . . ., 2010.

[119] "The iSAT web page". In: URL: http://isat.gforge.avacs.org.

[120] Mingsong Chen, Xiaoke Qin, and Prabhat Mishra. "Efficient decision ordering techniques for SAT-based test generation". In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. 2010, pp. 490–495. DOI: 10.1109/DATE.2010.5457156.

[121] Guillaume Baud-Berthier, Jesús Giráldez-Cru, and Laurent Simon. "On the Community Structure of Bounded Model Checking SAT Problems". In: *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT'17)*. 2017, pp. 65–82.

[122] Stefan Kupferschmid et al. "Incremental preprocessing methods for use in BMC". In: *Formal Methods in System Design* 39 (Oct. 2011), pp. 185–204. DOI: 10.1007/s10703-011-0122-4.

[123] Ofer Shtrichman. "Pruning Techniques for the SAT-based Bounded Model Checking Problem". In: *LNCS* (Dec. 2002).

[124] Ofer Strichman. "Accelerating Bounded Model Checking of Safety Properties". In: *Formal Methods Syst. Des.* 24.1 (2004), pp. 5–24.

[125] Liangze Yin et al. "Clause Replication and Reuse in Incremental Temporal Induction". In: *2014 19th International Conference on Engineering of Complex Computer Systems*. 2014, pp. 108–115. DOI: 10.1109/ICECCS.2014.23.

[126] Brahim Nasraoui, Syrine Ayadi, and Riadh Robbana. "SBMC: Symmetric Bounded Model Checking". In: (July 2010). DOI: 10.14236/ewic/vecos2010.9.

[127] Xavier Gillard and Charles Pecheur. "On the community structure of SAT-BMC problems". In: *PhD Symposium at iFM'17 on Formal Methods: Algorithms, Tools and Applications (PhD-iFM'17)*. 2017.

[128] Erika Ábrahám et al. "Parallel SAT Solving in Bounded Model Checking". In: *Formal Methods: Applications and Technology*. Ed. by Luboš Brim et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 301–315. ISBN: 978-3-540-70952-7.

[129] Siert Wieringa, Matti Niemenmaa, and Keijo Heljanko. "Tarmo: A Framework for Parallelized Bounded Model Checking". In: *Proceedings 8th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2009, Eindhoven, The Netherlands, 4th November 2009*. Ed. by Lubos Brim and Jaco van de Pol. Vol. 14. EPTCS. 2009, pp. 62–76. DOI: 10.4204/EPTCS.14.5. URL: https://doi.org/10.4204/EPTCS.14.5.

[130] Malay Ganai et al. "Efficient distributed SAT and SAT-based distributed Bounded Model Checking". In: *International Journal on Software Tools for Technology Transfer* 8 (Aug. 2006), pp. 387–396. DOI: 10.1007/s10009-005-0203-z.

[131] Ying Zhao et al. "Accelerating Boolean Satisfiability through Application Specific Processing". In: *Proceedings of the 14th International Symposium on Systems Synthesis*. ISSS '01. Montréal, P.Q., Canada: Association for Computing Machinery, 2001, pp. 244–249. ISBN: 1581134185. DOI: 10.1145/500001.500059. URL: https://doi.org/10.1145/500001.500059.

[132] Siert Wieringa. "On Incremental Satisfiability and Bounded Model Checking". In: *CEUR Workshop Proceedings* 832 (Jan. 2011), pp. 13–21.

[133] Malay Ganai and Aarti Gupta. "Tunneling and Slicing: Towards Scalable BMC". In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: Association for Computing Machinery, 2008, pp. 137–142. ISBN: 9781605581156. DOI: 10.1145/1391469.1391507. URL: https://doi.org/10.1145/1391469.1391507.

[134] H. Barros et al. "Exploring Clause Symmetry in a Distributed Bounded Model Checking Algorithm". In: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. 2007, pp. 531–538. DOI: 10.1109/ECBS.2007.40.

[135] S. Campos et al. "Distributed BMC: A Depth-First Approach to Explore Clause Symmetry". In: *2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. 2009, pp. 89–94. DOI: 10.1109/ECBS.2009.26.

[136] A. Cimatti et al. "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking". In: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, July 2002.

[137] Ludovic Le Frioux et al. "PaInleSS: a Framework for Parallel SAT Solving". In: *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT'17)*. Vol. 10491. Lecture Notes in Computer Science. Springer, Cham, Aug. 2017, pp. 233–250.

[138] Radek Pelánek. "BEEM: Benchmarks for Explicit Model Checkers". In: *Model Checking Software*. Ed. by Dragan Bošnački and Stefan Edelkamp. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 263–267. ISBN: 978-3-540-73370-6.

[139] Vijay D'Silva. "Propositional Interpolation and Abstract Interpretation". In: *Programming Languages and Systems*. Ed. by Andrew D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 185–204. ISBN: 978-3-642-11957-6.

[140] Aaron R. Bradley. "Understanding IC3". In: *SAT*. Vol. 7317. Lecture Notes in Computer Science. Springer, 2012, pp. 1–14.

[141] Yakir Vizel, Arie Gurfinkel, and Sharad Malik. "Fast Interpolating BMC". In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 641–657. ISBN: 978-3-319-21690-4.

[142] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. "Interpolation-Based Function Summaries in Bounded Model Checking". In: *Hardware and Software: Verification and Testing*. Ed. by Kerstin Eder, João Lourenço, and Onn Shehory. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 160–175. ISBN: 978-3-642-34188-5.

[143] Gianpiero Cabodi et al. "Interpolation-Based Learning as a Mean to Speed-Up Bounded Model Checking (Short Paper)". In: *Software Engineering and Formal Methods*. Ed. by Alessandro Cimatti and Marjan Sirjani. Cham: Springer International Publishing, 2017, pp. 382–387. ISBN: 978-3-319-66197-1.

[144] Pavel Pudlák. "Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations". In: *The Journal of Symbolic Logic* 62.3 (1997), pp. 981–998. ISSN: 00224812. URL: http://www.jstor.org/stable/2275583 (visited on 07/17/2023).

[145] S.F. Rollini et al. "PeRIPLO: A Framework for Producing Efficient Interpolants for SAT-based Software Verification". In: *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*. Stellenbosch, South Africa, 2013.

[146] Vijay Ganesh et al. "Lynx: A Programmatic SAT Solver for the RNA-Folding Problem". In: *Theory and Applications of Satisfiability Testing – SAT 2012*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 143–156. ISBN: 978-3-642-31612-8.

[147] Clark Barrett et al. "Satisfiability modulo theories". English (US). In: *Handbook of Satisfiability*. 1st ed. Frontiers in Artificial Intelligence and Applications 1. IOS Press, 2009, pp. 825–885. ISBN: 9781586039295. DOI: 10.3233/978-1-58603-929-5-825.

[148] Saeed Nejati et al. "Algebraic Fault Attack on SHA Hash Functions Using Programmatic SAT Solvers". In: *Principles and Practice of Constraint Programming*. Ed. by John Hooker. Cham: Springer International Publishing, 2018, pp. 737–754. ISBN: 978-3-319-98334-9.

[149] Curtis Bright, Ilias Kotsireas, and Vijay Ganesh. "A SAT+CAS Method for Enumerating Williamson Matrices of Even Order". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (Apr. 2018). DOI: 10.1609/aaai.v32i1.12203. URL: https://ojs.aaai.org/index.php/AAAI/article/view/12203.

[150] A. Gupta et al. "Learning from BDDs in SAT-based bounded model checking". In: *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*. 2003, pp. 824–829. DOI: 10.1145/775832.776040.

[151] Armin Biere, Cyrille Artho, and Viktor Schuppan. "Liveness Checking as Safety Checking". In: *Electronic Notes in Theoretical Computer Science* 66 (Dec. 2002), pp. 160–177. DOI: 10.1016/S1571-0661(04)80410-9.

[152] Niklas Eén and Niklas Sörensson. "Temporal Induction by Incremental SAT Solving". In: *Electronic Notes in Theoretical Computer Science* 89.4 (2003). BMC'2003, First International Workshop on Bounded Model Checking, pp. 543–560. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(05)82542-3. URL: https://www.sciencedirect.com/science/article/pii/S1571066105825423.

[153] J. Whittemore, J. Kim, and K. Sakallah. "SATIRE: A new incremental satisfiability engine". In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 2001, pp. 542–545. DOI: 10.1145/378239.379019.

[154] Liang Zhang, M.R. Prasad, and M.S. Hsiao. "Incremental deductive & inductive reasoning for SAT-based bounded model checking". In: *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* 2004, pp. 502–509. DOI: 10.1109/ICCAD.2004.1382630.