Thèse de doctorat de
l'Université Pierre et Marie Curie

Spécialité Informatique

(École Doctorale Informatique, Télécommunications et Électronique de Paris – ED130)

Présentée par

M. Maximilien Colange

Pour obtenir le grade de

Docteur de l'Université Pierre et Marie Curie

Sujet de la thèse

Symmetry Reduction and Symbolic Data Structures for Model Checking of Distributed Systems

Soutenue le 10 décembre 2013
devant la commision d'examen formée de :

Ahmed Bouajjani	*Rapporteur*
François Vernadat	*Rapporteur*
Béatrice Bérard	*Examinatrice*
Monika Heiner	*Examinatrice*
Tommi Junttila	*Examinateur*
Fabrice Kordon	*Directeur de thèse*
Soheib Baarir	*Encadrant*
Yann Thierry-Mieg	*Encadrant*

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

First of all, I would like to thank the people who accepted to be part of my jury: Prs. Bouajjani and Vernadat, for their further kindness to be rapporteurs; Pr. Heiner and Dr. Junttila, who came all the way to Paris to listen to me; Pr. Bérard, with a further thank for her time and advices at various occasions during the three years of my Ph.D.

I also would like to thank Pr. Kordon, Dr. Baarir and Dr. Thierry-Mieg, supervisor and advisors, for their patience during these three years, their constant attention, their questions, ideas, talks, jokes, and for teaching me the hard life of a researcher. I will remember and follow their advices, even though they often thought I was not paying attention.

I would thank all the people I met at the LIP6: it has always been a pleasure to come (almost) every morning to work. With no particular order, and for various good reasons: Dr. Sassolas, who initially recruited me, Dr. Voron, Mr. Démoulins (one day, I will understand how Maven works. Hopefully.), Dr. Hong, Dr. Linard (thanks for the advices for Switzerland), Dr. Poitrenaud for his insights, Dr. Paviot-Adet, Dr. Zhang and Dr. Preud'homme, and my favorite co-jokers almost-Drs. Millet and Renault, Dr. Ben Maïssa and his cakes, almost-Dr. Ben Salem, Dr. Besse (yes, I have lungs). And the people I met around the LIP6: Dr. Petrucci, Dr. Haddad, Dr. André, Mr. Hulin-Hubard, almost-Dr. Barbot.

I would like to thank all my friends, most of them who also know the joy of the Ph.D. student: Luc and Damien, my flatmates for 3 years, Abdelkader, my Jussieu teammate, the ineffable Karim, Cédric and the rhum, Alexandre, Vincent, Pierrick, Clément, foxy Charles, Jules, Pascal, and the list goes on.

A big thank to my friends from Thiais and Choisy (and co.), whose happiness is a precious gift. A big thank to all my friends from Cachan. Also a big thank to my friends from the prépa: I finally did it, I will finally have a job!

I would especially thank my family, without whom I would not have overcome the obstacles of these three years: my mother Christine, my father Pascal, my sisters Amélie and Marion, Gilbert, Aaron and Hadrien. And the rest of my (big) family, for the all the good times we spend together, in Choisy, in Auvet, in La Clusaz.

And a final thank for her who endures me, and seems however to like it, my dear Charlotte.

Résumé

In summing up, I wish I had some kind of affirmative message to leave you with, I don't. Would you take two negative messages?

Woody Allen (1964)
Woody Allen

1 Introduction

Le développement des technologies de communication a accru l'usage de systèmes répartis dans de nombreux domaines. De tels systèmes sont constitués de plusieurs entités (comme des processus), qui évoluent en parallèle tout en communiquant. Les exemples les plus courants comprennent les chaînes de production, les applications logicielles en ligne, le contrôle de transports (avions, métros automatisés ...) ou la gestion de centrales électriques.

Ils concernent de plus en plus des domaines critiques: transports, chirurgie (*life critical*), militaire, aérospatiale (*mission critical*), sans oublier le commerce et l'industrie (*business critical*). La criticité de ces domaines nécessite une garantie de fonctionnement des systèmes impliqués. Cependant, leur taille et leur complexité croissantes rendent plus ardue leur analyse, et, partant, la vérification de leur fonctionnement.

Même de lourdes campagnes de tests ne peuvent être exhaustives. C'est là qu'interviennent les méthodes formelles, à même de garantir que tous les comportements d'un système ont été considérés.

Afin de raisonner correctement sur les spécifications et propriétés, elles doivent d'abord être exprimées dans des formalismes appropriés: langages de modélisations pour les spécifications, et logiques adéquates pour les propriétés. En choisissant les langages et le niveau d'abstraction, on peut se concentrer sur plusieurs types de propriétés. Elles sont exprimées au moyen de logiques sur les modèles, parmi lesquelles les plus communément utilisées sont les logiques temporelles telles que CTL* et ses fragments.

Les méthodes formelles peuvent être classées en trois catégories :

- L'analyse structurelle [Cou90, BHR84] permet d'obtenir des garanties comportementales, sans explorer les comportements eux-mêmes. Elle est souvent basée sur des méthodes algébriques. La classe des propriétés vérifiables dépend du formalisme choisi, et est souvent limitée à des propriétés génériques, comme des invariants. L'interprétation des résultats est réservée aux experts.

- Les méthodes basées sur la preuve automatique de théorèmes [AMO99] construisent une preuve formelle de la propriété sur le modèle. Ces méthodes ne sont que partiellement automatisées et requièrent un guidage humain.

- Les méthodes basées sur l'exploration de l'*espace d'états* des comportements du système pour vérifier les propriétés. Ces comportements sont présentés sous la forme d'un graphe, dont les noeuds sont les états du système et les arc les transitions entre eux. Ces méthodes sont aujourd'hui adaptées à la vérification automatique de propriétés, même pour des systèmes présentant un nombre infini de comportements ou d'états [Esp97]. Dans ce cas, le processus est décidable et largement automatisé. Malgré tout, nous nous concentrons ici sur les systèmes avec un nombre fini d'états, et un nombre potentiellement infini de comportements.

Le principal désavantage de ces méthodes exploratoires réside dans la taille excessive de l'espace d'états, qui peut être exponentiel par rapport à la taille de la description du système. Dans le cas des systèmes réparties, cette *explosion combinatoire* provient principalement de l'entrelacement dans l'exécution parallèle des composants du système, et de la taille des domaines des données manipulées par le système. L'explosion combinatoire conduit à une explosion de la complexité, tant en temps qu'en mémoire, de l'algorithme de vérification.

De nombreuses techniques ont été proposées pour pallier à cette explosion. Leur qualité dépend de leur pouvoir de réduction, des propriétés qu'elles préservent, et de leur capacité à se combiner avec d'autres techniques de réduction. Deux classes indépendantes peuvent être considérées: combattre la taille de l'espace d'états, et gérer cette taille.

Combattre. De telles méthodes visent à réduire la taille de l'espace d'états, en lui substituant un graph plus petit, sur lequel les propriétés à vérifier sont préservées. Selon le système et les propriétés considérés, différentes techniques peuvent être utilisées :

- *Vérification compositionnelle.* Comme un système réparti est fait de plusieurs composants, son graphe d'accessibilité peut être vu comme la composition des graphes de ses composants. La vérification compositionnelle [Val93, NM94, CP95] utilise les connexions entre les composants pour dériver les propriétés du système entier à partir des propriétés des sous-systèmes. La manière dont les composants interagissent n'est pas toujours assez forte pour en déduire des propriétés globales, et le type de propriétés pouvant être vérifiées dépend fortement de la structure du système.

- *Ordre partiel.* Les systèmes répartis présentent souvent des composant indépendants qui évoluent de manière asynchrone. En raison de la sémantique par entrelacements, plusieurs séquences de transitions, qui diffèrent uniquement par l'ordre d'apparitions des transitions, conduisent toutes au même état. Les techniques d'ordre partiel [WG93, VAM96] considèrent des classes de comportements qui partagent des propriétés communes, afin d'accélérer l'exploration de l'espace d'états. Selon les propriétés à vérifier, plusieurs variantes sont disponibles.

- *Symétries.* Les systèmes répartis sont conçus comme une combinaison de composants, parmi lesquels certains exhibent des comportements similaires. De tels composant sont dits symétriques, and la connaissance du comportement de l'un d'eux est souvent suffisant

pour inférer le comportement de l'ensemble. Plus formellement, les symétries du système définissent une relation d'équivalence sur ses états. Cette relation est ensuite utilisée pour produire un *espace d'états réduit*, où au moins un états par classe d'équivalence est conservé. Si l'on conserve exactement un représentant par classe, alors la réduction maximale est atteinte et l'espace d'états réduit est appelé *espace d'états quotient*. La définition des symétries garantit que la réduction préserve les propriétés compatibles avec les symétries utilisées [CEFJ96, CDFH90]. Cette réduction est souvent significativement (exponentiellement) plus petite que l'espace d'états original, ce qui facilite d'autant le processus de vérification.

Gérer. Ces méthodes visent à minimiser l'impact de la taille du graphe d'états sur les performances de l'algorithme de model-checking, en utilisant des structures de données compactes.

- Les méthodes à la volée (on the fly) [CVWY93] réduisent l'empreinte mémoire en stockant le minimum d'informations possible sur l'espace d'états en cours d'exploration. Ces méthodes ne peuvent pas éviter de recalculer des états ou des comportements déjà calculés, mais non stockés. Elles sont souvent efficaces lorsque l'on cherche à isoler un comportement précis (soit pour prouver l'existence d'un comportement désiré, soit comme contre-exemple d'une propriété globale), car elles autorisent un arrêt précoce. Malgré leur consommation mémoire réduite, elles présentent des temps de calculs prohibitifs. Elles sont donc souvent utilisés avec des mécanismes de cache, afin d'obtenir un bon équilibre entre le temps de calcul et l'utilisation de la mémoire.

- Des structures de données compactes ont été développées pour représenter efficacement de grands espaces d'états. Les plus connus sont certainement les diagrammes de décision (DD). Introduits par [Bry86] comme une représentation canonique de fonctions booléennes, leur efficacité à représenter de grands ensembles d'états dans le contexte du model-checking a été démontrée [BCM+92]. De nombreuses variantes ont été proposées, comme les Multi-Valued DD [SHMB90], les Edge-Valued DD [LS92] ou les Data DD (DDD) [CEPA+02].

Combattre et gérer. Les deux catégories sont théoriquement orthogonales, et peuvent donc être combinées afin de cumuler leurs gains respectifs. Cette association n'est pas pourtant pas évidente d'un point de vue pratique.

Ainsi, [CEFJ96] propose de combiner la réduction par symétries avec des diagrammes de décision, mais conclut à l'inapplicabilité de l'approche. La construction d'un espace d'états réduit s'effectue en conservant un seul représentant par classe d'équivalence. Ce représentant est choisi *via* une *fonction représentative*. Plus précisément, [CEFJ96] suggère un encodage d'une telle fonction en termes de DD, et prouve que la taille de cet encodage n'admet pas de borne polynomiale. Ce résultat est généralement interprété comme une preuve que l'approche combinée est vouée à l'échec.

Nous nous proposons d'analyser ce résultat, et les possibilités de le dépasser.

Tout d'abord, le calcul d'une fonction représentative est plus dur que de déterminer si deux états appartiennent à la même classe. Ce dernier problème est lui-même exigeant, puisqu'on ne lui connaît pas de solution en temps polynomial. Cette complexity, inhérente au calcul de la fonction représentative, se reflète dans la conclusion de [CEFJ96].

Cependant, cette conclusion n'est valable que pour l'encodage proposé. Il est basé sur l'encodage en DD "classique", qui associe à un vecteur d'entrée un vector de sortie. Il souffre d'un inconvénient majeur: chaque état rencontré est une entrée pour la fonction, and sera représenté dans l'encodage. Mais le but même de la réduction par symétries est de représenter uniquement les vecteurs de sortie (les représentants). Cet encodage n'apparaît donc pas bien adapté, ce qui peut contribuer au résultat négatif.

Objectif de la thèse

Cette thèse s'attache à examiner la possibilité d'une combinaison de la réduction par symétries et des diagrammes de décision. Ce travail est motivé d'une part par l'analyse ci-dessus du résultat de [CEFJ96], et d'autre part par les travaux ultérieurs menés sur les symétries et les diagrammes de décision.

Plusieurs travaux concernant le calcul efficace d'une fonction représentative ont été menés. [Jun03] est une étude très complète de ce problème. Il propose un algorithm, efficace en pratique, pour le calcul de cette fonction, bien que sa complexity dans le pire cas reste non-polynomiale. Des travaux similaires ont aussi été menés quant à l'exploitation des symétries dans les problèmes de satisfaction de contraintes [CGLR96].

Du côté des diagrammes de décision, [CEPA+02] présente une avancée en proposant d'encoder les opérations et fonctions sur les DD comme des entités distinctes, et plus comme des DD particuliers. Cette nouvelle approche permet de raisonner directement sur les opérations, les combiner, et d'utiliser leurs propriétés algébriques afin d'optimiser leur évaluation. Ce qui a donné naissance à un moteur de DD, `libddd` [MoV13], qui se distingue par exemple par des réécritures automatiques des opérations pour en minimiser le coût d'évaluation [HTMK08].

Nous avançons que ces travaux, conduits après [HTMK08], contiennent de nouvelles pistes pour la combinaison efficace de la réduction par symétries et des diagrammes de décision. De précédentes tentatives [HTMK08] ont également démontré la faisabilité de l'approche. Cependant, ces résultats manquent de généralité, car ils se limitent à des formalismes spécifiques et/ou à des types de symétries précis.

L'efficacité des DD repose sur leur representation compacte de grands ensembles. Afin de profiter de cette efficacité, ils nécessitent des algorithms dédiés; plus précisément, les algorithmes en DD doivent travailler sur des ensembles plutôt que sur des entrées individuelles. Quoique pratique, l'algorithme présenté par [Jun03] suppose une analyse d'un seul état en entrée. Sa généralisation à un ensemble d'états semble compromise.

Notre première tâche sera donc de concevoir un algorithme capable de calculer une fonction représentative qui accepte un ensemble d'états en entrée.

La seconde tâche sera de concevoir des opérations efficaces en vue d'implémenter cet algorithme, afin de profiter des optimisations automatiques fournies par `libddd`.

Cette thèse présente les trois contributions que nous proposons. La première est un algorithme original pour le calcul d'une fonction représentative dans le cadre des DD. Cet algorithme supporte l'usage de groupes de symétries arbitraires, et peut être implémenté efficacement sur des structures symboliques. Étant donné un ordre total sur les états, on choisit le plus petit état d'une orbite comme étant son représentant canonique. Au lieu de représenter directement la relation comme un ensemble de paires d'états, nous introduisons une fonction décroissante qui, à chaque état s, associe un état s' dans la même orbite tel que $s' < s$, si un tel état existe. En répétant l'application de cette fonction monotone au sein d'un point fixe, nous atteignons le même résultat que la relation d'orbite, sans jamais avoir à la calculer et à la représenter explicitement. Parce que cette fonction agit sur un ensemble d'états, elle évite de calculer le représentant de chaque états individuellement, et conduit donc à un algorithme général et adapté aux structures symboliques.

Le codage en termes de DD de cet algorithm repose sur l'échange efficace de deux variables dans un DD. Nous définissons ainsi de nouvelles opérations pour la manipulation des DD, et les instancions de manière à répondre à cet impératif. Ces opérations raffinent successivement des sous-ensembles d'états en vue d'évaluer par étape une expression syntaxique. La portée de ces nouvelles opérations va cependant plus loin que l'échange de deux valeurs. Nous illustrons leur large champ d'applications en montrant comment les utiliser pour encoder sur des DD une relation de transition décrite dans un langage de haut-niveau. Il est intéressant de noter que ces opérations améliore les performances de notre moteur symbolique, aussi bien avec que sans la réduction par symétries.

Nous présentons finalement la combinaison des deux techniques pour un formalisme particulier, les Symmetric Nets with Bags. Les groupes de symétries spécifiques qu'ils font apparaître permettent de simplifier plusieurs des problématiques exposées ci-dessus.

Les travaux présentés ici ont été déjà partiellement publiés. Ainsi

Please note that the work presented here was already partly published. Chapter 3 restates and extends [CKTMB12], especially regarding complexity considerations, and thorough details about the DD encoding. Similarly, chapter 4 is based on [CBKTM13], with more details about the instantiation of the defined operations for the evaluation of a transition relation and of symmetry-based operations. Finally, chapter 5 restates the work presented in [CBKTM11] and [CHKP12].

2 Réduction par symétries avec des diagrammes de décision

La première difficulté que nous rencontrons tient à la nature de l'algorithmique sur les diagrammes de décision. Afin de profiter de leur compacité à représenter des ensembles, les algorithmes symboliques doivent manipuler des ensembles autant que faire se peut. Malheureusement, les principaux algorithmes connus pour le calcul d'un représentant canonique sont basés sur une analyse poussée d'un état en entrée et du groupe de symétries. Ils sont donc difficilement portables à des ensembles d'états en entrée.

Nous proposons ici un nouvel algorithme pour le calcul de représentants canoniques, à même d'accepter un ensemble d'états à canoniser en entrée.

Nous choisissons le plus petit élément d'une orbite (selon l'ordre lexicographique) comme son représentant canonique. Si d'autres choix sont possibles, celui-ci a l'avantage de ne pas dépendre de l'ordre dans lequel on rencontre les états. Nous sommes alors confrontés au calcul du leader lexicographique sur des vecteurs. [Jun03] propose des algorithmes efficaces pour un tel calcul. Étant donné un état s en entrée, ces algorithmes cherche une symétrie g telle que $g.s = \min[s]_G$. Ils déterminent d'abord un petit nombre de candidats pour g, puis les applique à s pour sélectionner le bon. Ces algorithmes ont de bonnes performances en pratique, et peuvent être étendus au calcul d'une fonction représentative non canonique. Cette dernière approche donne aussi de bons résultats, sans construite l'espace d'états quotient. Cependant, ces algorithmes reposent sur une analyse poussée de l'état donné en entrée. Ils sont donc difficilement portables sur des diagrammes de décision. Ceux-ci représentant des ensembles d'états, les considérer un par un serait contre-productif.

[CEFJ96] tente d'implémenter une fonction représentative à l'aide de diagrammes de décision. Il s'appuie sur la représentation classique des opérations sur les DD, qui consiste à représenter une relation binaire sur les états (telle une relation de transition, ou une fonction représentative) par son graphe, qui est un ensemble de paires d'états. Le principal résultat de cette étude montre que la taille de l'encodage en DD d'une fonction représentative sous cette forme est exponentielle, quel que soit l'ordre des variables considéré. Elle conclut à l'impraticabilité de l'approche.

Nous pensons cependant que cette limitation peut être surmontée en utilisant des homomorphismes pour encoder la fonction représentative.

2.1 Un nouvel algorithme

Nous proposons un algorithme original pour le calcul des représentants canoniques des états d'un ensemble donné. À chaque itération, chaque état s qui n'est pas canonique est remplacé par un état s' de la même orbite, qui lui est inférieur lexicographiquement ($s' < s$).

Plus précisément, on fournit à l'algorithme un sous-ensemble H des symétries du système. À chaque itération, et pour chaque $h \in H$, on détermine l'ensemble des états s qui vérifient $s < h.s$[1]. Chacun de ces états s est alors remplacé par $h.s$. Le processus est répété jusqu'à stabilisation, c'est-à-dire qu'on ne trouve plus de tels états s, pour aucun des éléments de H.

Le principal intérêt de cet algorithme est de manipuler uniquement des ensembles d'états, ce qui autorise une implémentation en diagrammes de décision, sans contrevenir aux principes d'applicabilité des diagrammes de décision.

Le lecteur attentif notera que le choix de H est crucial à la correction de l'algorithme. L'algorithme reste néanmoins robuste : quel que soit le choix de H, les orbites sont respectées, et l'algorithme peut produire un (dans le meilleur des cas) ou plusieurs représentants canoniques. Ceci permet de construire des espaces d'états réduits. Notons en outre que le choix de H impacte également la performance de l'algorithme. Intuitivement, moins H compte d'éléments, plus il y

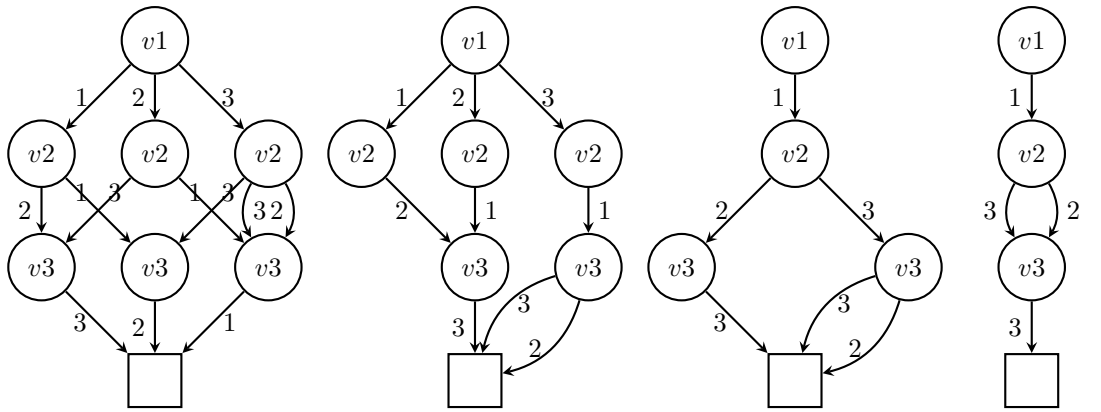
[1] Où . désigne l'action du groupe de symétries sur les états.

aura d'itérations (le nombre d'itérations est borné par la taille de la plus grande orbite, et donc par la taille de G), mais chaque itération sera également plus rapide (linéaire en la taille de H). À l'inverse, en prenant $H = G$, une unique itération conduit nécessairement aux représentants canoniques. La complexité de cette unique itération est plus élevée, le groupe G pouvait être de taille exponentielle.

Ces remarques sont congruentes avec les observations de [CEFJ96], qui indiquent une taille exponentielle pour la taille de cette fonction de canonisation.

La complexité d'application d'une permutation à un ensemble de permutations (représenté par un diagramme de décision), est lié à la taille du diagramme, et non à la taille de l'ensemble. La manipulation de grands ensemble n'est donc pas pénalisante. C'est en réalité sur ce point que les performances de notre algorithme vont à l'encontre de la conclusion de [CEFJ96]. Chaque opération peut être "factorisée" entre les différents états, ce qui améliore la complexité moyenne de l'algorithme, la complexité dans le pire des cas restant la même.

Exemple illustratif. Nous détaillons maintenant le calcul de notre algorithme sur un petit exemple illustratif. Considérons un système avec 3 variables $v1$, $v2$ et $v3$ qui sont toutes permutables. Le groupe de symétries considéré est donc $G = Sym(\{v1, v2, v3\})$. Choisissons $H = \{\tau_{1,2}, \tau_{2,3}\}$. Figure 1 montre les étapes intermédiaires de l'application de notre algorithme sur un ensemble contenant initialement toutes les permutations du vecteur $(1, 2, 3)$ ainsi que l'état $(3, 3, 1)$. Chaque étape correspond à l'application d'un élément $h \in H$ aux états que g réduit. Le premier pas applique $\tau_{2,3}$, ce qui met à jour l'ensemble des états considérés. Puis on applique $\tau_{1,2}$, réduisant encore le nombre d'états. Ces deux étapes sont répétées, et l'algorithme retourne finalement un ensemble avec deux états. Bien qu'elle ne soit pas représentée ici, une dernière itération est en réalité nécessaire afin de détecter la convergence.



(a) contient l'orbite de $(1, 2, 3)$ et l'état $(3, 3, 1)$. (b) après l'application de $\tau_{2,3}$ (c) après $\tau_{1,2}$ (d) après $\tau_{2,3}$

Figure 1: Avec $G = Sym(\{v1, v2, v3\})$, on obtient $H = \{\tau_{1,2}, \tau_{2,3}\}$. Notre algorithme appliqué à (a) produit successivement (b), (c), (d). (d) est l'ensemble des représentants canoniques $\{(1, 2, 3), (1, 3, 3)\}$.

Comme nous pouvons le voir à travers cet exemple, chaque étape de l'algorithme réduit simultanément plusieurs états. En un seul pas, chaque permutation réduit autant les états que possible, même dans des orbites différentes. Les états dans la même orbite sont peu à peu réduits en leur représentant canonique commun. En raison du partage des sous-structures, les états $(2, 1, 3)$ et $(2, 3, 1)$ en fig. 3.1a sont tous deux réduits sur $(2, 1, 3)$ en fig. 1b. $(2, 1, 3)$ n'est pas un représentant canonique, mais est plus petit que $(2, 3, 1)$. À ce point, les deux états fusionnent, ce qui permet de partage le reste du calcul. En général, chaque étape peut fusionner un nombre exponentiel d'états avec une complexité polynomiale. Ceci contraste avec les approches explicites qui canonisent tous ces états individuellement.

2.2 Choix de H

Comme nous l'avons vu, le résultat et les performances de l'algorithme dépendent du choix d'un sous-ensemble de symétries H. Afin d'être en mesure de garantir que l'on génère un espace d'états quotient, nous explicitons une condition pour que la fonction représentative calculée soit bien canonique.

Cette condition nous conduit à un certains nombres de considérations supplémentaires. En établissant un lien avec les prédicats de brisures de symétries utilisés notamment dans les problèmes de satisfaction de contraintes, nous établissons que la taille d'un tel H n'est en général pas bornée polynomialement.

Nous décrivons néanmoins de bons choix de H, de taille linéaire, pour les groupes de symétries les plus courants. Ces ensembles pour les groupes courants sont ensuite utilisés afin de valider expérimentalement notre approche.

2.3 Encodage en homomorphismes

Nous nous penchons ensuite sur le codage efficace de notre algorithme en homomorphismes. Au coeur de ce codage, pour $h \in H$, l'opération "si $h.s < s$, alors appliquer h à s, sinon laisser s tel quel". Afin de pouvoir encoder des H arbitraires, cette opération doit être codée pour n'importe quelle permutation.

En suivant les bonnes pratiques de construction des homomorphismes, nous proposons d'utiliser un nombre restreint d'homomorphismes élémentaires qui seront utilisés pour encoder cette opération pour toutes les permutations. Ceci permet de partager des éléments du calcul dans les caches entre différents h.

Nous nous basons pour cela sur la décomposition des permutations en produits de transpositions. Si h est une transposition, le test $h.s < s$ est une simple comparaison entre deux variables. De même, appliquer h revient à échanger deux variables. Nous exhibons donc un codage basé sur les transpositions de notre algorithme.

Afin d'améliorer les performances de notre algorithme sur les diagrammes de décision, nous étudions la possibilité pour le codage proposé de réaliser des calculs inutiles. De tels calculs encombrent en effet les caches, sans progresser vers le résultat. En utilisant certaines propriétés élémentaires sur les homomorphismes utilisés (notamment la commutation et la distributivité),

nous proposons un second encodage, plus efficace. Il s'agit d'éviter d'échanger deux variables ayant la même variable, en se basant sur les comparaisons de variables effectuées auparavant.

L'algorithme, et l'encodage que nous en proposons, sont ensuite validés expérimentalement en confrontant les performances de notre implémentation et celles d'un autre outil établi de réduction par symétries.

3 Nouvelles opérations pour les diagrammes de décision

Nous proposons ensuite de nouvelles opérations pour la manipulation des diagrammes de décision. Nous les appliquons en particulier à l'évaluation de la relation de transition d'un système distribué, ainsi qu'à l'évaluation des opérations liées aux transpositions évoquées ci-dessus.

L'approche symbolique initiale [BCM+92] encode une transition d'un système comme un diagramme de décision avec deux fois plus de variables : la relation de transition est une relation binaire sur les états, représentée par son graphe, c'est-à-dire l'ensemble des paires d'états liés par la relation. Cette approche monolithique correspond bien à la sémantique synchrone rencontrée dans le matériel. Cependant, comme elle doit considérer tous les états potentiels, elle conduit à des représentations trop lourdes dans de nombreux cas.

Ceci force à introduire de nouvelles stratégies [RAB+95], où un ensemble explicite de diagrammes de décision stocke des sous-ensembles disjoints du graphe de la relation de transition. Ce processus dit "amas de transition" permet de surmonter certaines des limites de l'approche monolithique.

Dans le cas de systèmes globalement asynchrones, localement synchrones (GALS), [CMS03] propose de construire l'amas de transitions selon la variable la plus haute (dans l'ordre des variables des DD) qui apparaît dans le support de la transition. La sémantique de tels systèmes est donnée comme un entrelacement asynchrone d'actions localement synchrones (comme les réseaux de Petri). Une telle structure pour l'amas permet d'utiliser la *saturation* pour optimiser l'évaluation du plus petit point fixe de l'amas : en se basant sur la sémantique d'entrelacement, le point fixe est d'abord calculé sur les parties les plus basses du DD avant de remonter à la racine. Ceci permet de ne pas surcharger les caches avec des résultats intermédiaires qui seront rapidement invalidés.

Un formalisme similaire est proposé par `LTSmin` [BvdPW10]. Un système y est composé de k variables d'état avec un domaine discret $\mathcal{D}$, and de transitions décrites principalement par leur support, composé de $k' \leq k$ variables. Pour construire l'espace d'états, `LTSmin` s'appuie sur des model-checkers tiers existant, qui fournissent une procédure appelée pour chaque valeur de support rencontrée dans l'espace d'états global. Grâce à cette projection, le nombre de ces appels est borné par $\mathcal{D}^{k'}$, et est en pratique limité aux états réellement rencontrés, et non plus aux états potentiels. Cet outil implémente également des techniques symboliques de l'état-de-l'art, comme la saturation, en utilisant l'encodage classique du graphe de la relation.

Cette approche est cependant rapidement mise en défaut quand le support croît : plus une transition impacte de variables, plus la complexité de son évaluation symbolique est grande. Si le model de haut niveau présente des manipulation de tableaux, des hypothèses pessimistes sur

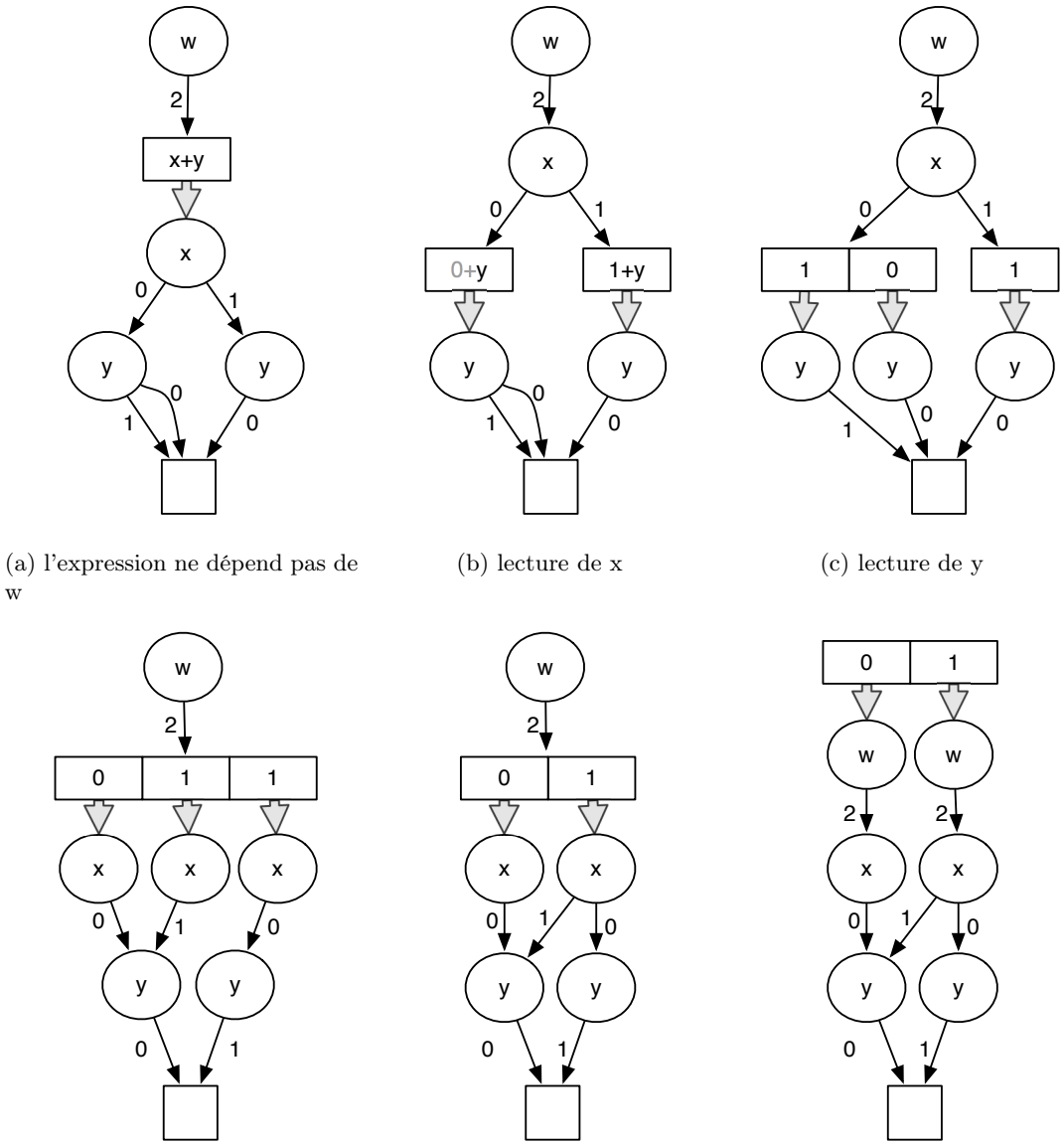
le support aboutissent à des supports incluant la plupart, si ce n'est l'intégralité, des variables d'état. Dans un tel cas extrême, le moteur explicite est sollicité au moins une fois pour chaque état rencontré, effaçant les éventuels gains résultant de l'usage de diagrammes de décision. De plus, l'insertion de chemins individuels dans un DD est susceptible de produire des effets de pic mémoire exponentiel. Les grands supports limitent également les possibilités d'utiliser la saturation, car les amas de transition se basent sur les supports des transitions.

Nous proposons donc de nouvelles opérations pour la manipulation de diagrammes de décision, qui surmontent ces difficultés, à travers l'algorithme *EquivSplit*. Étant donnée une expression syntaxique ϕ (une formule logique, par exemple) à évaluer sur un ensemble d'états, *EquivSplit* raffine successivement des sous-ensembles de l'ensemble initial, donné sous la forme d'un diagramme de décision. À chaque étape du raffinement, il évalue partiellement la formule ϕ, de manière à obtenir à la fin du calcul des sous-ensembles de l'ensemble initial, sur chacun desquels l'expression ϕ s'évalue en une seule valeur. L'évaluation partielle permet de réduire dynamiquement le support de l'expression au cours du calcul. La manipulation de tableaux consiste l'exemple courant de cette réduction dynamique des supports : l'evaluation partielle des sous-expressions décrivant les indices du tableau à accéder permet de s'affranchir des hypothèses pessimistes sur le support. Notre encodage entièrement symbolique des expressions évite toute étape explicite dans laquelle les états sont considérés individuellement.

3.1 Exemple

Un exemple des étapes de l'algorithme est présenté en fig. 2. On cherche à évaluer l'expression `x + y` sur un ensemble de trois états, qui comptent trois variables d'états `w`, `x` et `y`. Comme elle n'intervient pas dans le support de l'expression à évaluer, la variable `w` est sautée, sans besoin de la lire (fig. 2a). Les différentes valeurs possibles de `x` sont lues. Se forme alors un sous-ensemble de l'ensemble de départ pour chaque valeur possible de `x`. Chacun de ces ensembles est pointé par un rectangle. Les valeurs de `x` sont utilisées pour simplifier l'expression `x+y` en `y` (si `x` vaut 0) et `1 + y` (fig. 2b). De la même manière, on lit ensuite les valeurs de `y`, et l'expression est alors complètement évaluée (fig. 2c). Reste à reconstruire les sous-ensembles sur lesquels `x+y` a la même valeur. Les résultats de l'évaluation remontent alors au sommet du DD (fig. 2d). Au cours de la remontée, les sous-ensembles qui s'accordent sur la valeur de l'expression sont fusionnés (fig. 2e), afin d'avoir des ensembles maximaux à la fin de l'algorithme, qui partitionnent l'ensemble initial (fig. 2f).

Cet algorithme ne dépend pas de l'ordre dans lequel les variables sont codées (et lues) dans un diagramme de décision. À chaque pas de raffinement, l'algorithme s'assure que l'information lue dans le diagramme ne sera plus utile dans les étapes ultérieures. Cela se traduit par l'élimination de la dépendance de l'expression à la variable courante. Ceci peut nécessiter d'évaluer l'expression un peu plus avant, ce qui est réalisé par un mécanisme d'anticipation (*look-ahead*), qui va lire l'information nécessaire plus bas dans le diagramme de décision. Cette anticipation peu être vue comme un pré-raffinement, guidé par la structure de l'expression. L'indépendance de l'ordre de variables est une amélioration part rapport aux opérations habituellement utilisées sur les



(a) l'expression ne dépend pas de w

(b) lecture de x

(c) lecture de y

(d) remontée des résultats

(e) fusion des résultats intermédiaires

(f) remontée finale des résultats

Figure 2: Exemple de l'algorithme *EquivSplit*.

diagrammes de décision, et rend notre algorithme très flexible et simple d'utilisation.

3.2 Applications

Notre nouvelle opération a un champ d'application très vaste. Nous le démontrons en appliquant *EquivSplit* à l'évaluation sur des diagrammes de décision de la relation de transition d'un système

décrit dans un langage de haut niveau. Cela permet de faire un lien direct entre les spécifications de haut-niveau et la manipulation des structures symboliques, de bas-niveau, sans besoin de traduire les expressions de haut-niveau en concepts de bas-niveau. Pour démontrer cet effet, nous avons également conçu un langage intermédiaire pour exprimer des relations de transitions. Ressemblant au langage C, il agit comme un assembleur pour les diagrammes de décision, au sens où chaque operateur du langage correspond à une opération basique des diagrammes de décision.

L'instruction au coeur de ce langage est l'assignation d'une nouvelle valeur ϕ (qui dépend de la valeur des autres variables) à une variable ψ. L'algorithme associe deux *EquivSplit* : le premier va repérer la position de la variable à écrire ψ, éventuellement en résolvant des accès indicés dans des tableaux. L'autre résout l'expression de ϕ pour trouver la valeur à écrire dans ψ. Chacun des *EquivSplit* détermine une partition de l'ensemble d'états initial. Ces deux partitions sont intersectées, afin d'obtenir une partition plus fine sur laquelle les valeurs de ψ et ϕ sont cohérentes. L'affection de la constante ϕ à la variable ψ est faite sur chacun de ces sous-ensembles, qui sont finalement refusionnés pour obtenir le résultat final. L'imbrication des deux *EquivSplit*est de plus optimisée, de manière à partager des étapes (par exemple la lecture d'une variable qui apparaît dans les supports de ϕ et de ψ). Cette application représente une grande amélioration par rapport aux approches antérieures, où l'affectation décrite dans le langage de haut-niveau est typiquement traduite en des instructions de bas-niveau, par la disjonction de toutes les valeurs potentielles des expressions ϕ et ψ. L'évaluation de l'affectation est ici dynamique, ne dépend pas de l'ordre des variables. La réduction dynamique du support permet en outre d'activer la technique de saturation dynamiquement, dès que les conditions sont réunies.

Nous utilisons également *EquivSplit* pour encoder l'échange de deux variables, ce qui fournit un encodage direct et efficace des transpositions utilisées dans la section précédente. Ceci représente une amélioration supplémentaire par rapport aux optimisations déjà discutées dans ce cadre.

4 Application à un formalisme : les Symmetric Nets with Bags

Après ces deux principales contributions, nous mettons en application la seconde pour un formalisme particulier : les Symmetric Nets with Bags (SNB). Historiquement, ces travaux ont été réalisés avant ceux décrits ci-dessus. Les limites des méthodes développées sur les SNB ont néanmoins conduit à la formalisation de *EquivSplit* notamment, ce qui explique le lien décrit maintenant.

Les SNB sont une extension syntaxique (mais non sémantique) des Symmetric Nets (SN), eux-mêmes une variantes des réseaux de Petri colorés. Ils permettent de former des "super-jetons" contenant un bag (c'est-à-dire un multi-ensemble) de jetons. La définition des SNB impose de borner la taille de ces bags, de manière à conserver la même expressivité que les SN. S'ils sont aussi expressifs que les SN, les SNB autorisent des réseaux beaucoup plus compacts. La fig. 3

présente un exemple de SNB, et de son équivalent en SN.

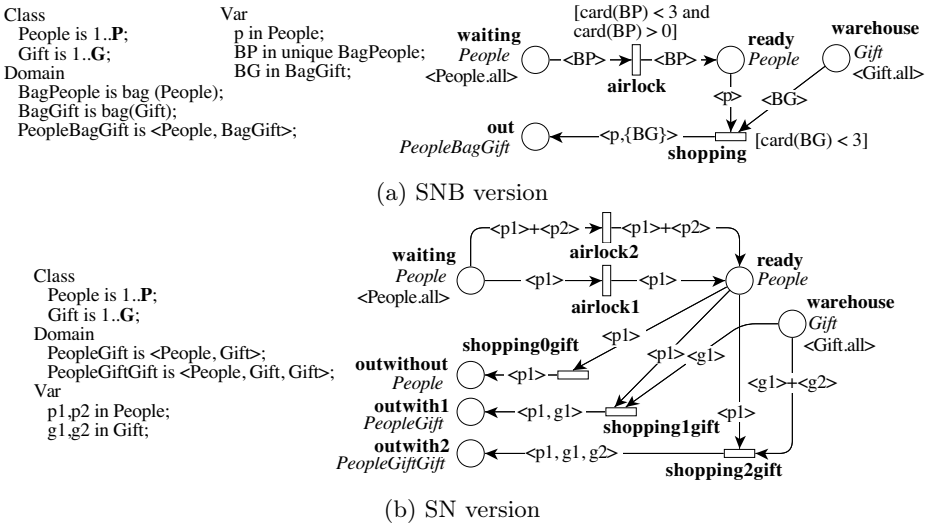


Figure 3: Exemple de SNB et de son équivalent SN

Comme leur nom le suggère, les SNB supportent la réduction par symétries. Les symétries sur les couleurs des jetons sont déterminées par la définition de classes de couleurs, et par les annotations des arcs et des transitions du réseau, qui peuvent induire des brisures de symétries.

Leurs caractéristiques structurelles (forme des marquages, forme des symétries, travaux historiques) conduisent à une représentation particulière des orbites, sous forme d'un *marquage symbolique*. Différant dans les détails des états représentatifs, ils constituent pourtant bien la forme canonique d'une orbite. Leur principale différence avec des états (ou en l'occurrence des marquages) tient à leur processus de canonisation légèrement simplifié. Il n'en faut pas moins les canoniser, selon un groupe de symétries déterminé.

Le codage des marquages symboliques interdit de recourir à l'algorithme de canonisation qui constitue notre première contribution. Cependant, en raison de la forme particulière des symétries dans les SNB, la procédure de canonisation se résume à un forme de tri. Pour l'accomplir, il convient de choisir quelle (orbite de) couleur(s) placer en première position. Ce choix s'effectue selon une opération très similaire à *EquivSplit*. Étant donnés un ensembles de marquages à canoniser, cette opération associe à chaque sous-ensemble de marquage les (orbites de) couleurs à permuter. Les marquages devant permuter les mêmes couleurs sont rassemblées, afin de partager le calcul de l'échange lui-même.

Nous évoquons également d'autres aspects propres aux SNB dans la canonisation, qu'ils seraient trop longs à décrire ici, mais dont certains sont également résolus grâce à des opérations calquées sur *EquivSplit*.

Une série de comparisons expérimentales face à un outil reconnu dédié au calcul des réductions par symétries sur les SN démontre la supériorité de notre approche sur les SN. Cette comparison est complétée par des tests effectués sur les SNB correspondants aux SN testés, qui font apparaître

un gain supplémentaire lorsque le calcul est effectué directement sur les SNB.

5 Conclusion

À travers deux contributions principales, cette thèse élargit les outils à disposition de la réduction par symétries d'une part, et des approches symboliques d'autre part. Elle fait une plongée dans la combinaison des deux approches, et en démontre la faisabilité, notamment grâce à un exemple d'application à un formalisme pré-existant. Les deux contributions restent pertinentes par elles-mêmes et peuvent également être utilisées indépendamment l'une de l'autre.

Introduction

Chapter 1

> Trois critères permettent de considérer une affirmation comme valide : la vérification par l'expérience directe, la déduction irréfutable, et le témoignage digne de confiance.[1]
>
> *Le moine et le philosophe* (1997)
> Jean-François Revel

1.1 Motivations

The development of communication technologies has increased the use of distributed systems in numerous domains. Such systems are made of several entities (*e.g.* processes), that evolve in parallel and are able to communicate. Common examples include flexible manufacturing systems, online applications, transportation control (planes, automated subways ...) or power plant management.

They concern more and more critical domains: transportation, surgery (*life critical*), military, aerospace (*mission critical*), but also trade and industry (*business critical*). The criticity of such domains requires a guarantee regarding the functionality of involved systems. However, their increasing size and complexity render their analysis harder, and therefore, reliability is more difficult to guarantee.

Heavy testing to detect undesirable behaviors cannot be exhaustive. Formal methods, that guarantee that all possible behaviors have been accounted for, can instead be used. Model-checking is one of the most promising formal approaches towards this end. It emerged in the 80's [CE81, QS82], and was recently recognized as a major approach of formal verification, through the ACM Turing Award attributed to Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis in 2007. It consists in formally modeling the system to be checked, then exploring exhaustively its behaviors, so as to ensure that the desired properties hold, or find a counter-example. These behaviors can be represented as a graph, the *state space* (or state graph), whose nodes are system's states and arcs are transitions between them. Behaviors of the system are paths in the state space. We restrict ourselves in this work to *finite systems*, that have a finite number of states, but possibly infinitely many behaviors.

We focus on *distributed asynchronous systems* in this work. Such systems particularly suffer from the *combinatorial explosion*: a great increase of their number of states and behaviors when they grow. Mainly due to the interleaving in the parallel execution of the system components,

[1] Three criteria allow to consider a statement as valid: verification through direct experience, irrefutable deduction, and trustworthy testimony.

the size of their state spaces is frequently at least exponential in the size of their description. The combinatorial explosion however greatly hinders the performance of model-checking algorithms.

Several methods have been proposed to deal with this limitation. We focus here on the use of *symmetries* and symbolic data structures, namely *decision diagrams*.

Symmetries. Distributed systems are often structured as a combination of components, among which several exhibit similar behaviors. Such components are said to be symmetrical, and knowing the behavior of one such component is often sufficient to infer the behavior of the combination. More formally, the symmetries of the system define an equivalence relation over its states. This relation can be used to produce a *reduced state space*, where at least one state per equivalence class is kept. If exactly one representative state per class is kept, then maximal reduction is achieved and the reduced state space is called *quotient state space*. The definition of symmetries guarantees that the reduced state space preserves properties, provided that they respect the symmetries [CDFH90, CEFJ96, ES96]. This reduction is usually significantly smaller than the original state space, thus easing greatly the verification process.

Decision Diagrams. Sophisticated data structures have been developed to efficiently represent large state spaces. The most famous ones are Decision Diagrams (DD). Introduced by [Bry86] as a canonical representation for boolean functions, they have then proved being successful at representing large sets of states very compactly and efficiently in the context of model checking [BCM+92]. They rely on unique representation of structures in memory (thanks to unique hash tables), and a memoization mechanism (a variant of caching) in order to reduce their memory footprint and to speed up computations. One should have this in mind when designing algorithms for decision diagrams, as a clever use of the memoization mechanisms can significantly increase performances. Since decision diagrams represent sets of states compactly, individualizing states appears to be counter-productive. Efficient algorithms should therefore work with sets to achieve efficiency. These two features can turn the implementation of the simplest algorithm on decision diagrams into a challenging problem.

Combining Symmetries and Decision Diagrams. Symmetry reduction and decision diagrams are theoretically two orthogonal methods that can be combined to stack their performance gains. This combination is however not straightforward in practice. A reduced state space is built by keeping only one state per equivalence class. This representative state is picked using a *representative function*. In a first attempt to combine symmetry reduction and decision diagrams, [CEFJ96] proposes a DD encoding of such a representative function. It however shows that the proposed encoding is not polynomially bounded, and concludes to the impracticality of the approach. Let us analyze this result and the possibilities to bypass it.

First of all, computing a representative function is harder than determining whether two states belong to the same equivalence class. This latter problem, known as the orbit problem, is itself quite challenging as it is not known to have a polynomial complexity: it is at least as hard as the graph isomorphism problem. This complexity, inherent to the computation of a representative function, is reflected in the conclusion of [CEFJ96].

Secondly, their conclusion holds only for the proposed encoding. It is based on the "classical" DD encoding, where an input vector is associated an output vector. This encoding suffers from one main drawback: each encountered state is an input for the representative function, and will be represented in this encoding. The very aim of the symmetry reduction is however to represent only the output states (the representative ones). This encoding does not seem to be fine-tuned, and that may contribute to the negative result.

1.2 Contributions

We present two contributions: one concerning the very combination of symmetry reduction and decision diagrams, the other dealing with new high-level operations for the manipulation of decision diagrams.

1.2.1 Combining Symmetries and Decision Diagrams

We investigate the possible combination of symmetry reduction and decision diagrams. This is motivated on the one hand by the above analysis of the result of [CEFJ96], and on the other hand by subsequent studies regarding symmetries and decision diagrams.

Several works have examined the efficient computation of a representative function for symmetry reductions. [Jun03] is a very complete study of this problem. It proposes an algorithm, efficient in practice, to compute a representative function, although its worst-case complexity remains not polynomial. Other similar works have been conducted for the efficient exploitation of symmetries in constraint satisfaction problems [CGLR96].

On the decision diagrams side, [CEPA+02] presents a breakthrough when it proposes to encode DD operations as distinct entities, not as particular DD. This new approach allows to reason directly on operations, combine them, and to use their algebraic properties to optimize their evaluation. This gave rise to a decision diagrams engine, `libddd` [MoV13], that for instance features automatic rewritings of operations to minimize the cost of their evaluations [HTMK08].

We claim that these works conducted after [CEFJ96] contain new leads for an efficient combination of symmetry reduction and decision diagrams. Previous attempts [EW03, TMIP04] also showed the feasibility of this approach. However, these results lack generality, as they are limited to specific formalisms and/or specific kind of symmetries.

The efficiency of decision diagrams relies on their compact representation of large sets. To keep this efficiency, they require dedicated algorithms; more specifically, algorithms on DD should work on sets rather than on individual elements in order to be efficient. As practical as it may be, the algorithm presented in [Jun03] relies on the analysis of a single input state. Its generalization to handle a symbolic set of states in input seems compromised.

We propose an original algorithm to compute a representative function with decision diagrams. It allows to work with arbitrary symmetry groups, and can be effectively implemented on top of symbolic data structures.

Given a total ordering on states, the smallest state in an equivalence class is chosen to be its canonical representative. Instead of directly representing the so-called orbit relation, that maps each state onto its canonical representative, we introduce a monotonic function that, given a state s, returns a state s' in the same class such that $s' < s$, if such an element s' exists. By repeatedly applying such a monotonic function in a fixpoint operation, we achieve the same effect as if we were using the symmetry relation, whithout ever having to explicitly compute and represent it.

This monotonic function is based on a subset H of the system symmetries. When applied to a state s, it chooses a symmetry $h \in H$, such that $h.s < s$, if such a symmetry exists. Note that this function operates over sets of states. More precisely, for each $h \in H$, it selects the states s such that $h.s < s$ and apply h to them. Thus, it avoids individual representative computations for each states, thus leading to a general and efficient algorithm to combine the use of symmetries with symbolic data structures.

Interestingly, this allows to always compute a reduced state space, whatever the chosen subset H. The degree of reduction is nevertheless impacted by the choice of H. This feature renders our algorithm quite flexible, as the choice of H allows to balance between the complexity of the

algorithm and the size of the obtained state space. We thus detail a condition on H for the obtained reduced state space to be as reduced as possible. We then show that the size of such a subset is not polynomially bounded in general, but we exhibit small subsets for commonly encountered groups.

We also discuss the implementation of this algorithm on decision diagrams, and detail some of the optimizations that can be applied. The biggest challenge here is to find a clever way of encoding the symmetries. We rely on the decomposition of permutations into products of transpositions. The optimization we propose relies on the fact that the symmetries are conditionally applied on states.

1.2.2 New Efficient Operations for Decision Diagrams

The transition relation of a system is a binary relation over its set of states S. The original symbolic approach [BCM+92] considers such a binary relation as a subset of the cartesian product $S \times S$, encodes it as a big decision diagrams, with twice as many variables. The second set of variables allows to associate to each variable its new value after the operation, for all potential states. This matches well the synchronous semantics of hardware. But since it considers the relation as a whole, and accounts for all potential states, this monolithic approach yields intractable representations in many cases.

This forces to introduce new strategies [RAB+95], where the binary relation is split into smaller chunks, represented with an explicitly managed set of DD. This process, called transition clustering, allows to overcome some of the limits of the monolithic approach.

For Globally Asynchronous Locally Synchronous (GALS) systems, [CMS03] proposes to design the clustering according to the top-most variable in transition supports. The semantics of such systems is given as an asynchronous interleaving of locally synchronous actions (e.g. Petri nets). Such a clustering allows *saturation* to optimize the evaluation of the least fixpoint of a set of conjuncts: based on the interleaving semantics of the conjuncts, the fixpoint is first computed on lower parts of the DD.

A similar formalism is proposed by `LTSmin` [BvdPW10]. A system is defined as consisting of k state variables with a discrete domain $\mathcal{D}$ and of transitions described primarily by their support composed of $k' \leq k$ variables. To compute the state space, `LTSmin` relies on existing third-party explicit model checkers that provide a computation procedure called for each encountered value of the support in the global state space. Thanks to this projection, the number of these calls is bounded by $\mathcal{D}^{k'}$ and in practice is limited to actually encountered states. This tool also implements state-of-the-art symbolic techniques, such as saturation, using classical encoding with two "before" and "after" variables per system state variable.

This approach is however severely challenged when the support grows: the more variables are impacted by a transition, the higher the complexity of evaluating it symbolically. If the high-level model features array manipulation, pessimistic assumptions on the supports end up with supports including most (if not all) state variables. In such an extreme case, the explicit engine is invoked at least once for each state, negating any possible gain from the use of DD. Additionally, such individual insertion of paths in a DD is liable to produce exponential memory peak effects. Large supports also severely limit the possibilities of saturation as clusters are based on the supports of the concurrent transitions.

We propose new operations for decision diagrams manipulation, that overcome these difficulties, through the algorithm *EquivSplit*. Given a syntactical expression ϕ (such as a logic formula) to be evaluated on states, *EquivSplit* successively refines subsets of the input set (given as a decision diagram). At each refinement step, it partially evaluates the formula ϕ, so as to obtain in the end subsets of the input set, on each of which ϕ evaluates to a single value. Par-

tial evaluation allows to dynamically reduce the support of the expression. Our fully symbolic encoding of the expressions avoids any explicit step where states are individually considered in the model-checking algorithm.

This algorithm does not depend on the order in which variables are coded (and read) on a decision diagram. At each refinement step, the algorithm ensures that the information read in the decision diagram will not be needed in the further refinement steps. This may require to evaluate the expression a bit further. This is done by a look-ahead mechanism that reads required information further in the decision diagram. This look-ahead can be seen as a pre-refinement guided by the structure of the expression. Independence from the variable ordering is an improvement with respect to commonly used operations on decision diagrams, and makes our algorithm very flexible and easy to use.

Our new operation shows to have a wide scope of applications. We first instantiate them to implement the evaluation on decision diagrams of a system transition relation, expressed in a high-level modeling language. This allows to directly bridge the gap between high-level specifications and the low-level manipulation of symbolic structures, without need to translate the high-level expressions into low-level concepts. As a demonstration of this, we have designed a rather high-level language to express transition relations, that resembles the C language, and acts as an assembler for decision diagrams – in the sense that each operator in this language corresponds to a basic operation on decision diagrams.

Interestingly, we also use *EquivSplit* to encode the swapping of two variables in decision diagrams. This gives a direct efficient encoding of the transpositions that we use to implement the previous symmetry reduction algorithm. It represents a further improvement beyond the optimizations discussed in the previous contribution.

1.3 Outline

We first give in Chapter 2 an overview of the model-checking process, we describe and define the fundamental concepts that will be used throughout this thesis, especially symmetries, symmetry reduction and decision diagrams.

Our first contribution is presented in Chapter 3. It described the algorithm, analyses the condition on the subset H to yield maximum reduction. It then details the implementation of the algorithm on decision diagrams, along with some optimizations.

Our second contribution is presented in Chapter 4. It introduces the concepts used in the expression evaluation algorithm, describes the said algorithm and proves its correctness. It then details its application to the evaluation of a transition relation (both forwards and backwards), and to symmetry evaluation, as a continuation of the previous chapter.

A more detailed application of the concepts described in Chapter 3 is presented in Chapter 5. It details how they can be used towards the model checking of a pre-existing formalism, the Symmetric Nets with Bags. It gives all useful definitions, and details the implementation of the mechanism in this framework, using the specificities of the formalism to simplify some of the encountered problems.

Note that every chapter, besides Chapter 2, features an assessment section, that compares the performance of our implementation against various state-of-the-art tools.

Please note that the work presented here was already partly published. Chapter 3 restates and extends [CKTMB12], especially regarding complexity considerations, and thorough details about the DD encoding. Similarly, Chapter 4 is based on [CBKTM13], with more details about the instantiation of the defined operations for the evaluation of a transition relation and of

symmetry-based operations. Finally, Chapter 5 restates the work presented in [CBKTM11] and [CHKP12].

Chapter 2

Overview and Definitions

To be, or not to be, that is the question.

Hamlet (1623)
William Shakespeare

This chapter gives an overview of the model checking process, and places our contribution in the big picture. Informal presentation is accompanied with formal definitions of main concepts.

Section 2.1 describes the automata-based model checking approach, and identifies the problems that motivate our work. Section 2.2 defines the symmetries and shows how they can be used towards more efficient model checking, then Decision Diagrams are introduced in Section 2.3. Other annex definitions are presented in Section 2.6. This chapter closes by the assumptions that will be used throughout this work in Section 2.4.

2.1 Overview of model checking

After mentioning some related works, this section presents the technique of automata-based model checking.

2.1.1 Related Work

Three categories may be distinguished to classify formal methods, although not exhaustively:

- Structural analysis [BHR84, Cou90] allows to obtain behavioral guarantees without exploring the behaviors themselves. Often based on algebraic methods, the class of checkable properties depends on the chosen formalism, and is often limited to generic properties, such as invariants. Interpretation of the results is often reserved to experts.

- Methods based on automatic theorem proving derive a set of logical formulae (such as Hoare triples [Hoa69]), whose truth implies the conformance of the system to the properties. The truth of the formulae is then evaluated using theorem provers [AMO99], SAT or SMT solvers [BCC+99, PBG05, AMP06]. These methods are not always fully automated (theorem provers), and their results must be interpreted by experts.

- Methods based on the *state space* exploration of the behaviors of the system to check properties. These behaviors may be represented as a graph, whose nodes are system's states and arcs are transitions between them. Alternatively, states of the systems are represented as the finite sequences of transitions that lead to them from an initial state. These methods are nowadays adapted to the automatic checking of properties, even (in

special cases) for systems with an infinite number of behaviors or an infinite number of states [Esp97, BJNT00]. The process is decidable and mostly automated. However, we are concerned here with systems with a finite number of states, with possibly infinitely many behaviors. This restriction stems from the fact that the decision diagrams we use can represent only finite sets of states. Provided that one is able to represent infinite sets of states with decision diagrams, our methods should work as well without this restriction.

We focus in this work on the state space-based methods, mainly because they provide the best degree of automation so far. They nevertheless lack of applicability, as their main drawback is the excessive size of the state space, that often is at least exponential with respect to the size of the system's description. In the case of distributed systems, this *combinatorial explosion* mainly stems from the interleaving in the parallel execution of the system's components and from the size of the data types handled by the system. The size of the state space of a distributed system is more or less as big as the cartesian product of the state spaces of its components. Miniaturization of electronic devices allows to design powerful yet small components. The development of communication technologies has increased the use of compositional and hierarchical design patterns. Nowadays, systems made from hundreds or thousands of components having themselves hundreds or thousands of states are not uncommon.

This combinatorial explosion leads to an explosion of the complexity, both in time and space, of the exploration algorithm. Nevertheless, the essence of distributed systems gives hints of how to manage it. Several techniques have been proposed to tackle this combinatorial explosion. Their quality depends on their power of reduction, the preserved properties, and on their ability to combine with other reduction methods. Two independent classes are to be considered: fighting the size of the state space and handling this size.

Fighting. Such methods aim to reduce the size of the state space, by substituting it by a smaller graph, on which the properties to be checked are preserved. Depending on the system and the properties to be checked, different techniques can be used, among which are the following:

- *Compositional verification.* As a distributed system is made of several components, its reachability graph can be seen as the composition of the reachability graphs of its components. Compositional verification [Val93, NM94, CP95] uses the connections between the components to derive properties of the whole system from properties of its components. The way components interact is not always strong enough to derive global properties from local ones, and the type of properties that can be verified using compositional verification thus depends on the systems.

- *Partial order.* Distributed systems often exhibit independent components that evolve asynchronously. Due to the interleaving semantics, several sequences of transitions, that differ only by the order in which transitions are fired, all lead to the same state. Partial order techniques [WG93, VAM96] take advantage of this and consider classes of behaviors that share common properties, thus speeding up the exploration of the state space. Depending on the kind of properties to be checked, several variants can also be used.

- *Symmetries.* Distributed systems are often structured as a combination of components, among which several exhibit similar structures, hence similar behaviors. Such components are said to be symmetrical, and knowing the behavior of one such component is often sufficient to infer the behavior of the combination. More formally, the symmetries of the system define an equivalence relation over its states. This relation can be used to produce a *reduced state space*, where at least one state per equivalence class is kept. If exactly

one representative state per class is kept, then maximal reduction is achieved and the reduced state space is called *quotient state space*. The definition of symmetries guarantees that the reduced state space preserves properties, provided that they respect the symmetries [CDFH90, CEFJ96, ES96]. This reduction is usually significantly (exponentially) smaller than the original state space, thus easing greatly the verification process.

Handling. Such methods aim to minimize the impact of the graph's size on the performance of model checking algorithms, especially by using concise data structures, like:

- On-the-fly methods [CVWY93, BCG95] reduce the memory footprint by storing as little information on the state space as possible. Such methods however cannot avoid to recompute already explored states and behaviors. They are often more efficient when a specific behavior is to be found (either to prove the existence of a desired behavior, or as a counter-example of a global property), as it may allow an early stop. Despite of their low memory consumption, their runtime can be prohibitive. They are therefore often used with caching mechanism, in order to achieve a good balance between computation time and memory usage.

- The state of a distributed system changes when one of its components' state changes. Therefore, a given state differs only by a small amount from its immediate neighbors (successors or predecessors). An efficient memory representation of such two neighbors should not store twice the common part. This motivates the search for efficient data structures that take advantage from the common information in states. The most famous are Decision Diagrams (DD). Introduced by [Bry86] as a canonical representation for boolean functions, they have then proved successful at representing large sets of states very compactly and efficiently, in particular in the context of model checking [BCM+92, FFK+01, SRB02, SH09]. Several variants have then be proposed, such as Multi-Valued DD [SHMB90], Edge-Valued DD [LS92] or Data DD (DDD) [CEPA+02] and Interval DD [ST98].

Combining Methods. Both categories are theoretically orthogonal and can be combined to stack their respective performance gains. For example, [Pel94] describes how to combine on-the-fly methods with partial-order reduction. The combination is however not always straightforward. As discussed in the introduction, [CEFJ96] studies the possibility of combining symmetry reduction with symbolic data structures, drawing a rather negative conclusion.

Methods from the same categories can also be combined together. Symmetry and partial-order reductions are thus often associated, as shown in [EJP97]. Similarly, symbolic data structures can also be used in conjunction with on-the-fly algorithms [BTY97].

2.1.2 Modeling Distributed Systems

A distributed system consists of several components with various possible synchronizations. A simple example is made of p processes evolving in parallel and that share a common ressource. We would like to reason about their use of this shared ressource, to verify properties such as "only one process at a time uses the shared ressource" (mutual exclusion), or "if a process asks for the shared ressource, it will eventually access it" (non-starvation). This example will be used throughout the chapter to exemplify the notions defined. It is voluntarily simple so that it can be exhaustively presented.

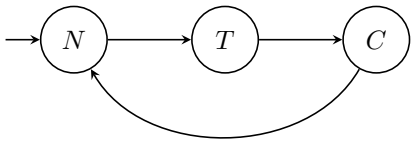
Several high-level formalisms may be used to model such a system, such as Petri nets [Pet62], communicating automata [BZ83] or process algebra [Hoa78]. They all give a description of the internal *state* of the system, and how this internal state can evolve.

For instance, Figure 2.1a describes a process as a finite automaton. It has three states: N when it is not in critical section, T when it tries to enter the critical section, and C when it is in the critical section. As described by the automaton, a process must go through state T before entering the critical section, and gets back immediately to N when it leaves the critical section. The parallel composition of two such processes ($p = 2$) is presented in Figure 2.1b. The state of the global system is the conjunction of the two processes' states: state $\langle C, N\rangle$ means that the first process is in the critical section, whereas the second one is in non-critical section.

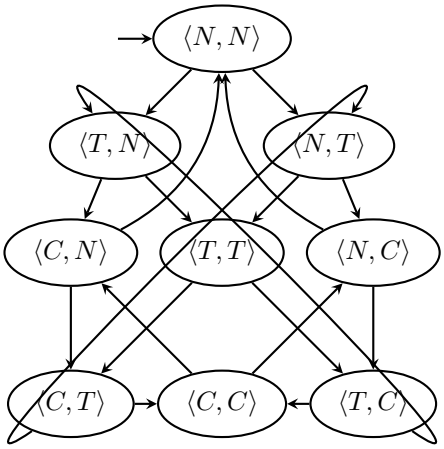
The choice of a formalism is often guided by its capability to express specific features of the system, but is not very relevant to model checking. For the model checking *stricto sensu*, only the *state space* of the system is required, although the structure of the system is often useful for optimization purposes. The state space is a graph whose nodes are the states of the system, and where two states are connected if and only if the system can evolve from the first one to the second one. Though not always necessary for model checking, we further require that this state space be finite.

Note that Figure 2.1a is the state space of a single process, whereas Figure 2.1b is the state space of the composition of two processes. This last state space has 9 states and 18 transitions.

Although not used in the example, it is common to label transitions, with action names for instance.



(a) the state space of a single process



(b) the state space of the parallel composition of two processes

Figure 2.1: Example of state space

We now formalise the notion of state space as a Transition System (TS).

Definition 2.1: Transition System
A TS is a tuple (S, Δ, S_0) such that:

- S is a set of states;
- $S_0 \subseteq S$ is the set of initial states;

- $\Delta \subseteq S \times S$ is a set of transitions.

A TS is said to be finite if its set of states is finite.

If $(s_1, s_2) \in \Delta$, we also note $s_1 \rightarrow s_2$. The set of successors of a state s is noted $\texttt{succ}(s) = \{s' | s \rightarrow s'\}$. `succ` is also called the successor function.

It occurs that the set of states of a system is infinite, but that the set of states *reachable* from a given initial state is finite. The state space of the system can thus be safely restricted to these reachable states. This restriction is called the *reachability graph* of the system. Unreachable states are not of interest because they will never occur, so that we assume from now on that the state space of the system is actually given by the nodes in its reachability graph, and we will use both terms indifferently.

A path in the state space is called a *behavior* of the system. Despite the finiteness of the state space, behaviors can be infinite, as the state space may have cycles. For instance, $\langle N, N\rangle \rightarrow \langle N, T\rangle \rightarrow \langle N, C\rangle \rightarrow \langle N, N\rangle$ is a behavior of our example. It is cyclic, and repeating it leads to an infinite behavior.

2.1.3 Expressing and Verifying Properties

Several kinds of properties can be checked on distributed systems. Some require the presence or absence of certain states, that can be directly checked by observing the state space. In our example, the mutual exclusion property "at most one process uses the shared ressource at a time" is expressed as the absence of the state $\langle C, C\rangle$.

Other properties specify the existence (or absence) of precise behaviors. Such properties can be expressed with temporal logics. These extend propositional logic with temporal operators, expressing temporal relationships between formulae. For instance, our example "every request to access the shared ressource will eventually be satisfied" can be expressed as $G(\phi \implies F\psi)$, where ϕ is the proposition "a process asks for the shared ressource", ψ is the proposition "the access to the shared ressource is granted", $\implies$ is the usual implication, F is read "finally" or "eventually", and G is read as "globally" or "always". This formula assumes linear time, and has to be checked against all possible behaviors of the system. This is done by translating the (negation of the) formula into a Büchi automaton, synchronizing it with the state space and checking whether the language recognized by the obtained Büchi automaton is empty [CES86]. This language is empty if and only if the property holds for all behaviors.

The method described above applies to the verification of LTL (Linear Time Logic) formulae. Similar algorithms, based on the translation of the formula into an appropriate automaton to be synchronized with the system, exist for various other logics.

To properly define a logic, one should first decide, by defining a set of *atomic propositions* AP, what information is accessible from a single state. The validity of atomic propositions is then given by a function $Q : S \mapsto 2^{AP}$ that associates to each state the set of atomic propositions that holds in it. These atomic propositions and validity function could be embedded in Definition 2.1 as a state-labeling function, but keeping it outside allows more flexibility. As atomic propositions reflect what is observable on the system, rather than an intrinsic property, different sets of atomic propositions may be used depending on the context. Furthermore, they will be of little concern in the rest of this work, so ruling them out of Definition 2.1 clarifies the discourse.

In our running example, typical atomic propositions would be of the form "process P_i is in state x" where $i \in \{1, 2\}$ and $x \in \{N, C, T\}$. It is however possible to define more complex atomic propositions, such as "at least/at most i processes are in state x", where $i \in \{0, 1, 2\}$ and $x \in \{N, T, C\}$.

Note that other temporal logics exist, with dedicated algorithms to check the corresponding formulae against a state space. Let use cite the expressive CTL*, and its well-known fragments LTL and CTL.

We do not develop any further about logics and dedicated verification algorithms, as this would go beyond the scope of this thesis. As a summary of this section, one should keep in mind that appropriate logics allow to express a wide range of properties about a system, and that verification algorithms are based on an exploration (be it partial or exhaustive) of the system's state space.

2.2 Symmetries in model checking

We present here the use of symmetries in model checking, by first defining what a symmetry is. We then show how to use symmetries to ease model checking, and finally present the problems that arise in this end.

2.2.1 Symmetries

In our example of p processes sharing a common ressource, one immediately notes that all processes have the same behavior. Asserting "whenever P_1 asks for the shared ressource, it will gain access to it" is exactly the same as asserting "whenever P_2 asks for the shared ressource, it will gain access to it", up to a change in process' name. In fact, besides the name change, any property that holds for one process also holds for all the other ones.

This similarity between the processes is a *symmetry*: up to a renaming, their behaviors are the same and can be superposed one on the other. Therefore the system does not change when two processes are swapped. This invariance then naturally leads to common properties for the p processes.

A symmetry can however be more complex: in a system with a ring topology, the system is invariant by rotating the ring. In fact, as in geometry, the key aspect of a symmetry is that it leaves the state space invariant.

Definition 2.2: Symmetry
Let $\mathcal{K} = (S, \Delta, S_0)$ be a TS. A symmetry σ is a bijection of S such that:

- $\sigma(S) \subseteq S$;
- $\sigma(S_0) = S_0$;
- σ is congruent with respect to Δ: $\forall s_1, s_2 \in S, s_1 \to s_2 \Leftrightarrow \sigma.s_1 \to \sigma.s_2$.

The set of all symmetries of a TS $\mathcal{K}$ is denoted by $Aut(\mathcal{K})$. It is a group under the composition operator $\circ$. Indeed, the identity (neutral element for $\circ$) is obviously a symmetry. The composition of two symmetries is also a symmetry and the inverse of a symmetry is also a symmetry.

In our example with $p = 2$, consider the permutation π that swaps the processes: $\pi(\langle x, y\rangle) = \langle y, x\rangle$ for all $x, y \in \{N, T, C\}$. It is not hard to verify that π is a symmetry.

2.2.2 Reduction under Symmetries

We now detail how symmetries can be used to reduce the relevant part of the state space to be explored in model checking.

Definition 2.3: Symmetric Equivalence Relation
Let $x_1, x_2 \in S$ and G be a subgroup of $Aut(\mathcal{K})$. x_1 and x_2 are said to be symmetric under G, denoted by $x_1 \equiv_G x_2$, if there is a $g \in G$ such that $g.x_1 = x_2$.
$\equiv_G$ is an equivalence relation over S. $[x]_G$ denotes the equivalence class (also called orbit) of x under $\equiv_G$.

$\equiv_G$ being an equivalence relation derives from the definitions of a group and a group action (see Section 2.6.1).

Note that two states are equivalent under $\equiv_G$ if they present the same behavior: since $\equiv_G$ is congruent with the transition relation, transitions between orbits are consistent, and successors of symmetric states are themselves symmetric. We now use $\equiv_G$ to *abstract* the state space.

Definition 2.4: Reduced Transition System
$\tilde{\mathcal{K}} = (\tilde{S}, \tilde{\Delta}, \tilde{S}_0)$ is a reduction of $\mathcal{K}$ w.r.t. a subgroup G of $Aut(\mathcal{K})$ if and only if:

- $\tilde{S} \subseteq S$ and $\forall s \in S, \exists \tilde{s} \in \tilde{S} : s \equiv_G \tilde{s}$,
- $\tilde{S}_0 \subseteq \tilde{S}$ and $\forall g \in G, g.\tilde{S}_0 \subseteq S_0$,
- $\tilde{\Delta} \subseteq \tilde{S} \times \tilde{S}$,
- if $\tilde{s}_1 \in \tilde{S}$ and $(\tilde{s}_1, s_2) \in \Delta$, then there exists $\tilde{s}_2 \in \tilde{S}$ such that $\tilde{s}_2 \equiv_G s_2$ and $(\tilde{s}_1, \tilde{s}_2) \in \tilde{\Delta}$,
- if $(\tilde{s}_1, \tilde{s}_2) \in \tilde{\Delta}$ then there exists $s_2 \in S$ such that $s_2 \equiv_G \tilde{s}_2$ and $(\tilde{s}_1, s_2) \in \Delta$.

A reduction $\tilde{\mathcal{K}}$ of $\mathcal{K}$ w.r.t. G is obtained by removing some of the symmetric behaviors in $\mathcal{K}$. Any state x may be removed to obtain a reduction, as long as there remains a state in $[x]_G$ in the reduction. Therefore, several reductions of different sizes are possible for the same relation $\equiv_G$. Note that $\mathcal{K}$ is a reduction of itself w.r.t G, where no state is removed. At the other end, a reduction $\mathcal{K}$ where it is not possible to remove any other state is isomorphic to the quotient of $\mathcal{K}$ by $\equiv_G$. This smallest reduction is the *quotient state space* of $\mathcal{K}$ by $\equiv_G$ (or by G).

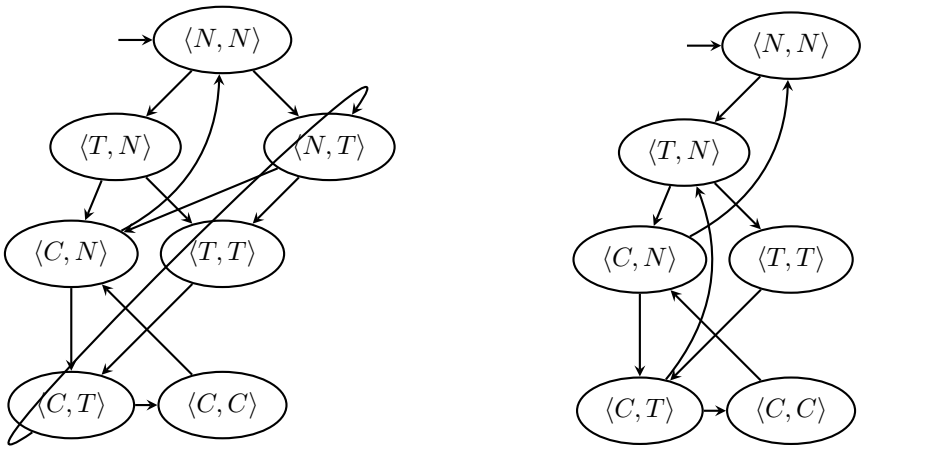
A reduced transition system is smaller than the original state space, yet it contains all relevant behaviors (up to the symmetries) of the initial one. Its smaller size eases the verification of a property.

Thus, Figure 2.2 shows two examples of reduction of the state space of Figure 2.1b under the symmetry that swaps both processes. Note that Figure 2.2b is as small as possible (the quotient state space), whereas Figure 2.2a is not.

Let us now detail how a reduced transition system is interesting for model checking.

We have seen that symmetries must be congruent with the transition relation, so as to ensure that the transition relation between *orbits* is well-defined. Similarly, to ensure logical consistency inside orbits, symmetries must also preserve atomic propositions. More formally, we require that for every symmetry σ and every state $s \in S$, $Q(\sigma.s) = Q(s)$. Thus, all the states in an orbit have the same set of atomic propositions, and Q is well-defined on the orbits.

Therefore, there is a path $(s_1, s_2, \dots)$ in $\mathcal{K}$ if and only if there exists a path $(\tilde{s}_1, \tilde{s}_2, \dots)$ in $\tilde{\mathcal{K}}$ with $Q(s_i) = Q(\tilde{s}_i)$ for all $i \in \mathbb{N}$. Thus, if the symmetries preserve the atomic propositions, then the reduced transition system preserves CTL* formulae [AHI98, CEJS98]. A weaker condition, only requiring the symmetries to preserve certain subformulae, also exists [ES96]. For linear time properties, another method relies on the symmetries of the automaton encoding the (negation of the) formula [AHI98].



(a) non-optimal reduction (b) quotient state space, smallest reduction

Figure 2.2: Reductions of the state space of Figure 2.1b

The symmetry reduction can also be applied with asymmetric properties or systems. In the former case, the state space can be tweaked so as to keep track of asymmetries of atomic propositions [ES96, SG04]. In the latter case, the system is partially symmetric. After a first study on Petri nets [HITZ95], an approach is proposed, that translates asymmetries of the systems into asymmetries of the automaton of the formula to be checked [HIA00]. [ET99, EHT00] weaken the definition of system symmetries, yet still ensuring that the reduction preserves properties. [BDHI05] dynamically adapts the used symmetry group during the walk of the state space, so as to account for asymmetries.

We distinguish symmetries of the system (or structural properties) $Aut(\mathcal{K})$ from the symmetries of a given set of atomic propositions AP, noted $Aut(Q_{AP})$. The considered group of symmetries to build a reduced transition system that preserves the formulae built upon AP is usually the group $G = Aut(\mathcal{K}) \cap Aut(Q_{AP})$. G is also a group of symmetries of the system, that also preserves the atomic propositions. The considered group of symmetries can therefore be adapted to the property at stake. For instance, when checking two formulae ϕ_1 and ϕ_2 built on AP_1 and AP_2 respectively, one should consider the group $Aut(\mathcal{K}) \cap Aut(Q_{AP_1})$ to check ϕ_1 and $Aut(\mathcal{K}) \cap Aut(Q_{AP_2})$ to check ϕ_2.

It also allows an early detection of a trivial (or poor) symmetry group (for instance when $Aut(\mathcal{K}) \cap Aut(Q_{AP}) = \{Id\}$), indicating that symmetry reduction may not be the best approach for the given property.

The attentive reader may have noticed that we define reduced transition systems, and mention that the extreme case corresponds to the quotient state space. The quotient state space is minimal in number of states, offering maximal reduction. Computing this quotient seems to be an appropriate target of symmetry reduction. As we will see later, this quotient state space is however hard to compute, and we prefer the weaker notion of reduced transition systems, as it offers the same guarantee about the preservation of properties, yet may be simpler to compute. We can remember the following trade-off: the smaller the reduced state space, the harder it is to compute. The notion of reduced transition systems allows more flexibility in the size of the state space.

2.2.3 Computation and Representation of Reduced Transition Systems

We show in Algorithm 1 a simple algorithm to compute a reduced state space. The implementation of Algorithm 1 should indicate how symmetric states are chosen (lines 3 and 11).

Algorithm 1: Computation of a reduced TS of $\mathcal{K}$ with respect to G

Input: $\mathcal{K} = (S, \Delta, S_0)$ a TS
Input: G a subgroup of $Aut(\mathcal{K})$
Output: $\tilde{\mathcal{K}} = (\tilde{S}, \tilde{\Delta}, \tilde{S_0})$ a reduced TS of $\mathcal{K}$ w.r.t. G

1 $\tilde{S} \leftarrow \emptyset$, $\tilde{\Delta} \leftarrow \emptyset$, $W \leftarrow \emptyset$
2 **forall the** $s_0 \in S_0$ **do**
3 Choose a $\tilde{s_0} \in S$ such that $s_0 \equiv_G \tilde{s_0}$
4 $\tilde{S_0} \leftarrow \tilde{S_0} \cup \{\tilde{s_0}\}$
5 $W \leftarrow W \cup \tilde{S_0}$
6 **while** $W \neq \emptyset$ **do**
7 Choose any $s \in W$ and set $W \leftarrow W \backslash \{s\}$
8 **if** $s \notin \tilde{S}$ **then**
9 $\tilde{S} \leftarrow \tilde{S} \cup \{s\}$
10 **forall the** s' *such that* $(s, s') \in \Delta$ **do**
11 Choose $\tilde{s}'$ such that $s' \equiv_G \tilde{s}'$
12 $W \leftarrow W \cup \{\tilde{s}'\}$
13 $\tilde{\Delta} \leftarrow \tilde{\Delta} \cup \{\langle s, \tilde{s}' \rangle\}$
14 **return** $\tilde{\mathcal{K}} = (\tilde{S}, \tilde{\Delta}, \tilde{S_0})$

A first approach is to choose a symmetric state that is already in $\tilde{S}$, if any, and use the current state otherwise. However, testing whether two states are symmetric is a hard problem (*i.e.* not in **P**) for most interesting formalisms [Mat79, Pug05, Jun03].

Another approach is thus to use representative states for each orbit, that are not determined by the order in which the states are encountered. Let us thus use a function `repr` that yields a representative for a given state.

Definition 2.5: Representative Function
Let G be a sub-group of $Aut(\mathcal{K})$.
$\texttt{repr} : S \mapsto S$ is a representative function if $\texttt{repr}(x) = \texttt{repr}(y) \implies [x]_G = [y]_G$ for all $x, y \in S$.
`repr` is *canonical* if furthermore $[x]_G = [y]_G \implies \texttt{repr}(x) = \texttt{repr}(y)$ for all $x, y \in S$.

The size of $\tilde{S}$ depends on the used representative function, with two extreme cases:

- if $\texttt{repr}(x) = x$ for all x, then $\tilde{S} = S$ and $\tilde{K} = K$;
- if the representative function is canonical, then the size of $\tilde{S}$ is minimal, and $\tilde{K}$ is the quotient state space.

A variant of this approach uses representatives that are not states of the system. It is notably used in the setting of Symmetric Petri Nets [CDFH90, HKP+09].

Definition 2.6: Generalized Representative Function
Let G be a sub-group of $Aut(\mathcal{K})$, and X an arbitrary set.
$\texttt{repr} : S \mapsto X$ is a representative function if $\texttt{repr}(x) = \texttt{repr}(y) \implies [x]_G = [y]_G$ for all $x, y \in S$.
$\texttt{repr}$ is *canonical* if furthermore $[x]_G = [y]_G \implies \texttt{repr}(x) = \texttt{repr}(y)$ for all $x, y \in S$.

In this setting, the obtained state graph $\tilde{\mathcal{K}}$ is not strictly speaking a reduction of the initial state space $\mathcal{K}$, because $\tilde{S}$ is not included in S. It is nevertheless isomorphic (through $\texttt{repr}$) to a genuine reduction of $\mathcal{K}$.

Canonical representative functions are highly desirable as they minimize the number of states in $\tilde{S}$. Computing such a canonical representative function is however quite hard to achieve, at least as hard as graph isomorphism, not known to have a polynomial solution [CEFJ96, Jun03].

This work focuses on the combination of symmetry reduction and decision diagrams. As will be described later on, decision diagrams are particularly efficient at representing *sets* of states. To take advantage of them, let us try to work with sets of states as much as possible, rather than states individually. Algorithm 2 is a rewriting of Algorithm 1 emphasizing on this set encoding. Thus, we naturally extend $\texttt{succ}$ and $\texttt{repr}$ to take sets as arguments:

- $\texttt{succ} : 2^S \mapsto 2^S$ with $\texttt{succ}(X) = \bigcup_{s \in X} \texttt{succ}(x)$ for all $X \subseteq S$;
- $\texttt{repr} : 2^S \mapsto 2^S$ with $\texttt{repr}(X) = \bigcup_{x \in X} \{\texttt{repr}(x)\}$ for all $X \subseteq S$.

Due to their set representation, one usually does not exactly compute graphs with decision diagrams. We will, from now on, use them to represent sets of states, rather than graphs. The difference is subtle, and lies in the fact that the arcs of the graph are not explicitly represented. They can however be rebuilt efficiently through the application of the successor function. The reduced state space is nothing more than a state space whose transition relation is $\texttt{repr} \circ \texttt{succ}$.

Algorithm 2: Computation of the reduced state space of $\mathcal{K}$ with respect to G

Input: $\mathcal{K} = (S, \Delta, S_0)$ a TS
Input: G a subgroup of $Aut(\mathcal{K})$
Output: $\tilde{S}$ the states of a reduced TS of $\mathcal{K}$ w.r.t. G

```
1 S̃ ← repr(S_0)
2 W ← ∅
3 while W ≠ S̃ do
4 |   W ← S̃
5 |   S̃ ← S̃ ∪ repr(succ(S̃))
6 return S̃
```

We do not detail model checking algorithms here, as they all rely on an exploration (partial or total) of the state space. The transition relation is thus a prerequisite for all of these algorithms. Our aim in this work is to provide the transition relation of the reduced state space, to be used in any model checking algorithm. The algorithm above illustrates the use of this relation in the simple setting of the state space generation.

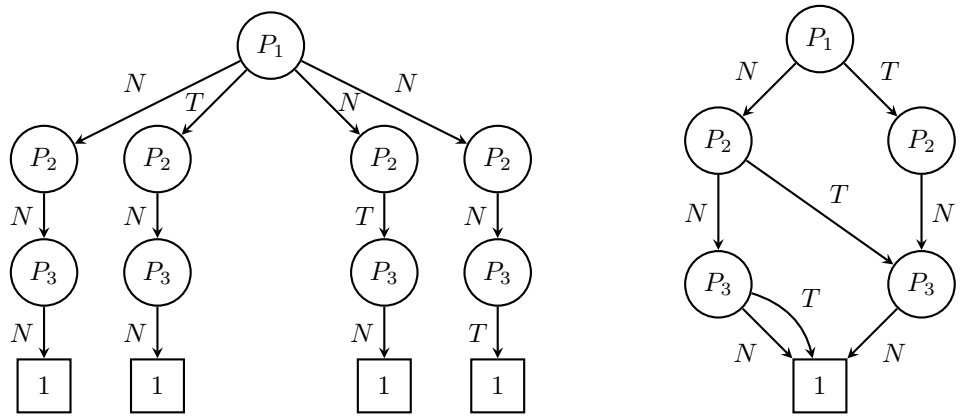
2.3 Decision Diagrams

We now describe how to represent efficiently large sets of states.

We can safely assume that the state of a system is described by a finite set of *state variables*. In our example described above, the state of the system is characterized by two state variables whose common domain (or data type) is the set $\{N, T, C\}$.

2.3.1 Definition of Decision Diagrams

Let us start with an illustrative example. We consider our mutual exclusion example with $p = 3$ processes. Let us assume that we want to represent the initial state $\langle N, N, N\rangle$, along with its immediate successors $\langle T, N, N\rangle$, $\langle N, T, N\rangle$ and $\langle N, N, T\rangle$.



(a) tree representation

(b) shared decision diagram

Figure 2.3: A Data Decision Diagram representing the set $\{\langle N, N, N\rangle; \langle T, N, N\rangle; \langle N, T, N\rangle; \langle N, N, T\rangle\}$

Let us now encode each state with a vector of size 3, as in Figure 2.3a. Each cell of a vector has a name (the one of the corresponding process), and is represented by a node, the content of the cell being represented by a labeled arrow going to the next cell. We add terminal nodes (square boxes labeled 1) at the end of the vectors to mark their ends. Thus, the left-most path in Figure 2.3a encodes the state $\langle N, N, N\rangle$.

We now transform this tree by representing each subtree only once. The result is shown in Figure 2.3b. For instance, all the end-marking nodes are merged into a single one. Similarly, the states $\langle N, N, N\rangle$ and $\langle N, N, T\rangle$ share the same prefix N, N, and their third components are merged into the left-most P_3 node in Figure 2.3b, with two outgoing edges corresponding to the two suffixes N and T.

Note that for each path in Figure 2.3a, there is a path in Figure 2.3b. All the common parts are shared, and the resulting graph is much smaller, thus more compact in memory. Here, Figure 2.3a counts 13 nodes and 12 edges, where Figure 2.3b has only 6 nodes and 8 edges.

In the general case, a system has n state variables, of domain D. A set R of states for this system can be given by its characteristic function $\chi_R : D^n \mapsto \{0, 1\}$. A decision diagram encodes such a function, by representing all elements in D^n, mapping those in R onto the terminal 1, and the others onto the terminal node 0. It suffices to represent only the paths mapped to 1, the other ones being implicitly mapped onto 0. It is thus possible to represent any finite subset of D^n even if D is infinite. This feature is important as it allows to deal with systems with no *a priori* known bound on the values of its state variables (although this bound always exist, as we work only with finite state spaces).

Historically, decision diagrams have been introduced as Binary Decision Diagrams (BDD) [Bry86] to represent boolean functions (with our notation, $D = \{0, 1\}$). Several variants have then been proposed, to represent various kinds of functions. For example, Multi-Valued Decision Diagrams (MDD) [SHMB90] were designed to represent sets of integer vectors ($D = \mathbb{N}$), although the integers must be bounded (so D must in fact be a finite subset of $\mathbb{N}$). Interval Decision Diagrams (IDD) [ST98] are a further variant to handle intervals of the real line. Other examples include Multi-Terminal Binary Decision Diagrams (MTBDD) [CMZ+93], where the output domain of the represented function is not boolean, that have been used for matrix representation. Edge-Valued Decision Diagrams (EVDD) [LS92] are an alternative to MTBDD that is more compact when the number of function values is too high. There also exists several variants such as Zero-Suppressed Binary Decision Diagrams (ZBDD) [Min93], where the value 0 is suppressed along the paths, thus sparing nodes. They are particularly useful at encoding sparse sets of bit vectors. To appropriately rebuild a path, ZBDD assume a fixed (static) ordering of the variables labeling the nodes. Other kinds of decision diagrams use dynamic variable orderings so as to circumvent the variable ordering problem.

All kinds of decision diagrams share common characteristics. Thanks to a canonical representation, nodes of the decision trees are unique in memory, so that common parts of different diagrams are shared. The sharing allows to have a number of paths usually exponential in the representation size (nodes in the diagram), hence the represented has an exponential size in the size of its memory representation. Due to the unique representation of nodes in memory, equality between sets can be tested in constant time. Furthermore, the operations manipulating decision diagrams are memoized, ensuring a complexity which is polynomial in the number of nodes, rather than in the number of paths. It is also well-known that the efficiency of the representation strongly depends on the chosen variable ordering [CGP99].

2.3.2 Data Decision Diagrams (DDD)

This work mainly deals with Data Decision Diagrams (DDD) [CEPA+02]. Like MDD, they are designed to represent sets of integer vectors ($D = \mathbb{N}$), except that D needs not be bounded.

Definition 2.7: DDD
Let Var be a set of variables, and for any ω in Var, let $\mathrm{Dom}(\omega) \subseteq \mathbb{N}$ be the domain of ω. The set $\mathbb{D}$ of DDD is defined inductively by:
$\delta \in \mathbb{D}$ if either $\delta \in \{\mathbf{0}, \mathbf{1}\}$ (terminals) or $\delta = \langle \omega, \alpha \rangle$ with $\omega \in Var$, and $\alpha : \mathrm{Dom}(\omega) \to \mathbb{D}$ is a mapping where only a finite subset of $\mathrm{Dom}(\omega)$ maps onto other DDD than $\mathbf{0}$. By convention, edges that map to the DDD $\mathbf{0}$ are not represented.

DDD do not assume that the variables appear in the same order in all paths, and variables can be repeated. This allows for example to encode lists with variable length. We will however not use this feature, and we assume from now on that each variable appears exactly once in every path of all DDD, and in the same static order. In particular, this ensures that set operations (union, intersection, set difference) are well-defined. Up to a mapping of N, T and C to integers, Figure 2.3b is an example of DDD.

2.3.3 Hierarchical Set Decision Diagrams

[CTM05] introduces Hierarchical Set Decision Diagrams (SDD), that extend DDD by allowing to represent vectors of sets ($D = 2^{D'}$). Since their edges can be labeled by sets, they also can be labeled by decision diagrams (that represent sets), thus introducing hierarchical structure.

Definition 2.8: SDD
Let *Var* be a set of variables, and for any ω in *Var*, let $\mathrm{Dom}(\omega)$ be the domain of ω. The set $\mathbb{S}$ of SDD is defined inductively by:
$\delta \in \mathbb{S}$ if either $\delta \in \{\mathbf{0}, \mathbf{1}\}$ (terminals) or $\delta = \langle \omega, \alpha \rangle$ with $\omega \in Var$ and $\alpha : 2^{\mathrm{Dom}(\omega)} \mapsto \mathbb{S}$ is injective with $\alpha^{-1}(\mathbb{S})$ forming a finite partition of $\mathrm{Dom}(\omega)$ and only one element is mapped to $\mathbf{0}$. By convention, edges mapping to $\mathbf{0}$ are not represented.

Note that a DDD $\delta = \langle \omega, \alpha \rangle$ can easily be turned into a SDD $\delta' = \langle \omega, \alpha' \rangle$ by defining $\alpha'(\alpha^{-1}(d)) = d$ for all $d \in \alpha(\mathrm{Dom}(\omega))$, α' left undefined otherwise. $\{\alpha^{-1}(d) | d \in \alpha(\mathrm{Dom}(\omega))$ is indeed a partition of $\mathrm{Dom}(\omega)$. Since a finite subset of $\mathrm{Dom}(\omega)$ maps to other DDD than $\mathbf{0}$ through α, this partition is finite. It is clear that α' maps at most one element to $\mathbf{0}$.

In Definition 2.8, α represents the edges, each of which labeled by a subset of $\mathrm{Dom}(\omega)$. Since a SDD represents itself a set, the labeling sets can be implemented in terms of SDD, or more generally by any kind of decision diagrams.

2.3.4 Operations on DDD and SDD

We now present operations on DDD and Set Decision Diagrams (SDD).

Since they represent sets, one first need to encode directly on DDD and SDD the natural operations on sets. They indeed support the standard union ($\cup$), intersection ($\cap$) and set difference ($\setminus$), with respect to the set of paths. Note however that complementation is not supported. Since D is not necessarily finite, 2^D is possibly infinite. But DDD can only represent finite sets, so that complementation is not defined for them. Complementation may be defined for SDD (provided the implementation of the edge-labeling sets support complementation), but is not used in our setting. DDD and SDD also offer a concatenation $\delta_1 \cdot \delta_2$ which replaces terminal $\mathbf{1}$ of δ_1 by δ_2. This corresponds to a cartesian product. DDD and SDD support user-defined operations through the notion of *homomorphisms*. A more detailed description of DDD homomorphisms can be found in [CEPA+02].

Recall that the successor function of a system associates to a state the set of its possible successors. Since decision diagrams are efficient at representing sets of states rather than individual states, it becomes essential to extend this function so as to associate to a set of states a set of successors. This is done naturally by defining $\mathtt{succ} : 2^S \mapsto 2^S$ with $\mathtt{succ}(S') = \cup_{s \in S'} \mathtt{succ}(s)$ for all $S' \subseteq S$. We note that this extension guarantees that $\mathtt{succ}(S_1 \cup S_2) = \mathtt{succ}(S_1) \cup \mathtt{succ}(S_2)$ and that $\mathtt{succ}(\emptyset) = \emptyset$. `succ` is thus linear with respect to $\cup$, and is called a homomorphism.

Definition 2.9: Homomorphism
A function $h : \mathbb{D} \mapsto \mathbb{D}$ (resp. $h : \mathbb{S} \mapsto \mathbb{S}$) is a homomorphism if it is linear with respect to the union:

- $h(\mathbf{0}) = \mathbf{0}$;
- $h(\delta_1 \cup \delta_2) = h(\delta_1) \cup h(\delta_2)$.

A homomorphism can thus be defined by its action on DDD (or SDD) paths, rebuilding the resulting DDD is done automatically and is not left to the user. For instance, by defining `succ` on S, one automatically gets the homomorphism `succ` on 2^S.

Homomorphisms are intended to be building blocks for complex operations. Basic ones are hard-coded, and it is possible to combine them through various operators. The most important are the sum ($+$) and the composition ($\circ$):

- $(h_1 + h_2)(\delta) = h_1(\delta) \cup h_2(\delta)$ for all $\delta \in \mathbb{D}$;
- $(h_1 \circ h_2)(\delta) = h_1(h_2(\delta))$ for all $\delta \in \mathbb{D}$.

Let us state some properties of the sum and composition of homomorphisms.

Proposition 2.1.

- *The sum is commutative, and the null homomorphism (constant homomorphism equals to* $\mathbf{0}$*) is its neutral element.*
- *The composition is, in general, not commutative, and the identity Id is its neutral element.*
- *The composition is distributive over the sum.*

Proof. The first point is easily deduced from the properties of the union of decision diagrams (that is the same as the union of sets).

The second point is a well-known property of the composition of functions.

The third point is easily proven. Let h_1, h_2, h_3 be three homomorphisms. Let $\delta \in \mathbb{D}$ (or $\delta \in \mathbb{S}$).

$$\begin{aligned}(h_1 \circ (h_2 + h_3))(\delta) &= h_1((h_2 + h_3)(\delta)) \\ &= h_1(h_2(\delta) \cup h_3(\delta)) \\ &= h_1(h_2(\delta)) \cup h_1(h_3(\delta)) \\ &= (h_1 \circ h_2)(\delta) \cup (h_1 \circ h_3)(\delta) \\ &= ((h_1 \circ h_2) + (h_1 \circ h_3))(\delta)\end{aligned}$$

□

The natural inclusion relation on sets also applies on DDD and SDD. For two DDD (resp. SDD) δ_1 and δ_2, $\delta_1 \subseteq \delta_2$ holds if and only if the set represented by δ_1 is included in the set represented by δ_2.

A homomorphism c is a *selector* if and only if for all $\delta \in \mathbb{D}$ (resp. $\delta \in SDD$), $c(\delta) \subseteq \delta$. This allows to select states satisfying a given condition.

Back to the example of Figure 2.3, assume one wants to find the states such that the two first processes are in the same state. This is implemented by a selector $[P_1 == P_2]$, that, applied on the decision diagram represented on Figure 2.3b, selects the two paths $\langle N, N, N\rangle$ and $\langle N, N, T\rangle$.

We can also define the negation of a selector, noted $\bar{c}$, such that, for all $\delta \in DDD$ (resp. $\delta \in SDD$), $\bar{c}(\delta) = \delta \setminus c(\delta)$. Note that the negation of a selector is also a selector. Selectors are often used to apply homomorphisms conditionally. As a shorthand for "if-then-else", we use $\texttt{IfThenElse}(c, h_1, h_2) = h_1 \circ c + h_2 \circ \bar{c}$, where h_1 and h_2 are homomorphisms and c a selector.

Proposition 2.2. *The composition is commutative on selectors.*

Proof. Let c_1 and c_2 be two selectors. Let $A_1 = \bigcup_{\delta \text{ s.t. } c_1(\delta)=\delta} \delta$. Then, for all $\delta \in \mathbb{D}$ (resp. $\delta \in SDD$), $c_1(\delta) = A_1 \cap \delta$. We define similarly A_2 such that for all $\delta \in \mathbb{D}$ (resp. $\delta \in SDD$),

$c_2(\delta) = A_2 \cap \delta$. Let $\delta \in \mathbb{D}$ (resp. $\delta \in \mathbb{S}$). We have:

$$\begin{aligned}(c_1 \circ c_2)(\delta) &= c_1(c_2(\delta)) & &= c_1(A_2 \cap \delta)\\ &= A_1 \cap (A_2 \cap \delta) & &= A_2 \cap (A_1 \cap \delta)\\ &= c_2(A_1 \cap \delta) & &= c_2(c_1(\delta))\\ &= (c_2 \circ c_1)(\delta)\end{aligned}$$

The key argument is therefore the associativity of the intersection. This holds for all δ, so that c_1 and c_2 commute. □

The *fixpoint* $h^\star$ of a homomorphism, defined as $h^\star(\delta) = h^k(\delta)$ where k is the smallest integer such that $h^k(\delta) = h^{k+1}(\delta)$, is also a homomorphism, provided that such a finite k exists.

Basic homomorphisms are hard-coded in the DDD/SDD library, and available to any user of the library. They include comparisons between two variables as selectors, and assigning a new value to a given variable.

Homomorphisms provide a high-level way of specifying a system's transition relation, but also to directly express various algorithms of state-space exploration. For instance, given a DDD s_0 representing initial states and a homomorphism *succ* representing the transition relation, the set of reachable states can be obtained through the equation $reach = (succ + Id)^\star(s_0)$. Note that this fixpoint exists only if the system has a finite set of reachable states. Similarly, atomic propositions can be encoded as selectors. Combining these selectors allows to encode complex formulae.

As aforementioned, the evaluation of homomorphisms is sped up through memoization. Due to this memoization and the unique tables for the tree nodes, one has to be careful of possibly unnecessary entries in these tables. As a common case, a homomorphism that performs an operation in several steps (through composition for instance) may yield several intermediate results for each step. These intermediate results are stored in the memoization tables but are not reused. Another poor design defines monolithic homomorphisms for complex operations, although these operations have several steps in common that could be factored out into smaller homomorphisms. This way, the intermediate results yielded by these factors are reused among other operations. It also happens when building a DDD (or SDD) in a top-down manner. Assume one first defines δ_1 and δ_2, then uses $\delta_3 = \delta_1.\delta_2$. Then the nodes created for δ_1 are not reused in δ_3, and the unique table is unnecessarily burdened.

Homomorphisms are rendered more efficient by using several techniques, usually enabled automatically, to avoid such overburden. For instance, by detecting at runtime that two homomorphisms affect different variables of a DDD δ, one can deduce that their composition is commutative, and first evaluate the one that affects the bottom-most variables, so as to build the result in a bottom-up fashion. `libddd` [MoV13], the DDD/SDD engine library developed and used at the LIP6, features such runtime rules for an efficient evaluation of homomorphisms.

Another important technique concerns the evaluation of a fixpoint, and is known as *automatic saturation* [HTMK08]. In the same spirit, it relies on the evaluation of the fixpoint on the lower variables of the DDD. The fixpoint on the higher parts is computed only when the fixpoint on the lower parts is already reached. This avoids to evaluate the fixpoint on the higher parts when the result on the lower parts is not computed, not saturated yet. This recursive mechanism relieves the unique and memoization tables from overburden.

2.3.5 Decision Diagrams Bottlenecks: Patterns and Anti-Patterns

We give here some good and bad practice about decision diagrams manipulation, in view of the aforementioned characteristics of DDD and homomorphisms.

The number of paths in a DDD is (in the average case) exponential in the number of nodes in the DDD. Thus, algorithms manipulating DDD should have a complexity related to the number of nodes in the DDD rather than to the number of paths. "Splitting" a DDD in individual paths is always a bad practice when working with DDD.

Here comes the first problem to port algorithms onto symbolic structures: dealing with *sets* of objects is a prerequisite for an algorithm to be efficiently implemented on decision diagrams. For instance, state-of-the-art canonical representative functions take individual states as input, and strongly exploit the structure of a single state to determine its canonical representative [Jun03]. These state-of-the-art algorithms are therefore hardly extensible to take sets of states as input, and will constitute the first challenge in our work. To efficiently combine symmetry reductions and decision diagrams, we first had to design a set-based representative function, as detailed in Chapter 3.

The second challenge resides in an ingenious homomorphism encoding for good performance. Our DDD library automatically rewrites homomorphisms at runtime, and saturation, to improve their evaluation. These are however only activated in a limited number of good patterns. Careful design of operations on DDD can increase the occurrences of such patterns, and considerably improve its performance. On the other hand, if a known pattern is hard to automatically detect, it is also possible to hard-code a specific evaluation mechanism for it. We try to describe here a few guidelines towards this goal, that will be used in the subsequent chapters.

It is rarely a good idea to design big monolithic homomorphisms. We have already mentioned that operations should be split into small building blocks, so as to take advantage of the memoization. These building blocks are then to be combined with sum, composition and fixpoint in order to implement the desired operations.

We have also seen that the library features runtime rewriting for efficient evaluation. Those who design homomorphisms should therefore be aware of these capabilities, to take maximum advantage of them. These mechanisms are mainly based on the detection of commutative homomorphisms (with respect to composition). More precisely, the designer can (and should) tell what variables are affected for the defined homomorphism. The library considers two homomorphisms to be commutative if their relevant sets of variables are disjoint.

When designing the building blocks, one should aim at disjoint sets of affected variables, to maximize the rewriting possibilities. Furthermore, we have seen that the commutativity test aims at building the resulting DDD or SDD in a bottom-up manner. The optimizations of the library actually only work if the sets of affected variables do not overlap. Assume that h_1 affects variables ω_1 and ω_3, and h_2 affects variables ω_2 and ω_4. If the variable ordering is $\omega_1, \omega_2, \omega_3, \omega_4$, the sets overlap. A good ordering would be $\omega_1, \omega_3, \omega_2, \omega_4$. The optimal variable ordering problem for decision diagrams is known to be **NP**-complete [BW96], but a careful analysis of the aforementioned building blocks may lead to a quite good variable ordering.

The commutativity test is quite limited. The designer may use further knowledge about the building blocks in order to hard-code the appropriate combination. For example, if h_1 and h_2 are known to commute but affect the same variables, the library will not detect it. The designer should then decide whether to code the composition as $h_1 \circ h_2$ or $h_2 \circ h_1$. Although it is in general hard to predict which way will perform better, the potential gains are usually worth some analysis. If h_1 also appears as the right-hand side of a composition in another operation, $h_2 \circ h_1$ should make the most of memoization.

Another case concerns the interaction between the sum and composition of homomorphisms. Assume a complex operation is described by a composition of two sums of homomorphisms. Since the composition is distributive over the sum, this composition can be developed. The designer should wonder which version (factored or developed) will yield better performance, especially when some simplifications between the terms are known.

The precise design decisions depend on the particular cases, and it is hard to go further than empirical guidelines. A complete analysis is performed in Section 3.3. Let us sum up the guidelines that have been reached so far:

- find reusable building blocks;
- reduce the set of affected variables of each building block;
- find an appropriate variable ordering that avoids overlapping of these sets;
- hard-code further knowledge (commutativity, simplifications ...).

2.4 Assumptions and Notations

Now that we have presented both the use of symmetry reduction in model checking and decision diagrams, we present the setting in which we conduct our work. In particular, we make some assumptions on the system states, so as to fit both the symmetries and decision diagrams frameworks.

First Assumption

Since decision diagrams represent primarily sets of vectors, we assume that a system state is entirely determined by the value of a fixed finite set of *state variables*, denoted by I. Without loss of generality, we further assume that all variables have the same domain $\mathcal{D}$. Most of the time, $\mathcal{D} = \mathbb{N}$ or $\mathcal{D} = \mathbb{Z}$.

The global state of the system is a *string* on the state variables.

Definition 2.10: String
Let I be a finite linearly ordered index set, and $\mathcal{D}$ a linearly ordered alphabet. A $\mathcal{D}$-string on I is a mapping $s : I \mapsto \mathcal{D}$. The set of all $\mathcal{D}$-strings on I is denoted by $\mathcal{D}^I$.

Alternatively, a $\mathcal{D}$-string on I will also be considered as a vector, whose cells are indexed by I and values in $\mathcal{D}$. This view prevails when encoding sets of strings as DDD.

For systems requiring to dynamically allocate variables, a pool size bound must then be known *a priori*. However, the variables domain $\mathcal{D}$ is *a priori* unbounded, as allowed by DDD. Note that this last assumption is not relevant for the algorithms presented in the subsequent chapters, and they can be ported to any kind of decision diagrams. The lexicographical order on $\mathcal{D}^I$, that is a total order, is guaranteed by the total orders on I and $\mathcal{D}$.

Second Assumption

In the general case, symmetries act arbitrarily on states. In order to be able to express the symmetries action in terms of homomorphisms, we assume that symmetries act by permuting state variables. This fits a wide range of distributed systems, where similar processes are interchangeable.

We claim that this hypothesis holds without loss of generality. As seen in Section 2.6.1, the action of an arbitrary group G on $\mathcal{D}^I$ is defined by a group morphism between G and $Sym(\mathcal{D}^I)$. Let us consider the translation ϕ from $\mathcal{D}^I$ into $\{0,1\}^{(\mathcal{D}^I)}$ that associates to v the vector $\phi(v)$ whose sole non-null component is the one indexed by v. Then, G acts on $\{0,1\}^{(\mathcal{D}^I)}$ by permuting the elements of the vectors. This demonstrates that our hypothesis does not lose generality.

This translation into a boolean vector has however a high complexity. We nevertheless observe that a great majority of symmetries in distributed systems either permute state variables (variable symmetry), or permute the values of a given variable (value symmetry), or are a combination of both. More practically, one translates the vector $v \in \mathcal{D}^I$ into a vector $v' \in \{0,1\}^{(D\times I)}$ such that, for all $d \in D$ and $i \in I$, $v'[(d,i)] = 1$ if, and only if, $v[i] = d$. Both variable and value symmetries (and the combination thereof) act on $v' \in \{0,1\}^{(D\times I)}$ by permuting the elements of the vectors. This case covers all the symmetries we have observed during our work, at a more practical cost than the previous translation.

Third Assumption

Since strings are totally ordered – lexicographically –, the representative function $\texttt{repr}(x) = \min[x]_G$ is often chosen. This choice is motivated by the easy comparison between two vectors (its complexity is much lower than the string orbit problem). This choice is also general enough, as it only requires $\mathcal{D}$ to be linearly ordered with no further requirements.

The problem

Now, our main problem is to compute efficiently $\texttt{repr} : 2^S \mapsto 2^S$ on sets, on top of decision diagrams. We define formally this problem and discuss its theoretical complexity.

The first point is to be able to determine whether two strings belong to the same orbit.

Problem 2.1 (String Orbit Problem)**.** *Given a group $G \lhd Sym(I)$ and two strings $s, s' \in \mathcal{D}^I$, tell whether s and s' are in the same orbit, i.e. $[s]_G = [s']_G$.*

Theorem 2.3. *String Orbit Problem is **NP**, and Graph Isomorphism-hard [CEJS98].*

The exact complexity of the String Orbit Problem is not well determined. It is believed to be an intermediate problem, neither in **P** nor **NP**-hard (unless **P**=**NP**). Several problems related to groups are also known to be computationally equivalent to Graph Isomorphism.

The second problem of interest is the computation of the chosen canonical representative function, formalized in the following problem.

Problem 2.2 (Lexicographical Leader)**.** *Given a group $G \lhd Sym(I)$ and a string $s \in \mathcal{D}^I$, find the smallest string in $[s]_G$.*

Computing a canonical representative function is intuitively harder than the orbit problem, since it requires to find a particular state in an orbit. [BL83] has shown that, if G is given as a set of generators, then the lexicographical leader problem is **NP**-hard, even if G is an abelian 2-group (whose every element is of order 2). [Jun03] goes further, and shows that it is actually $\mathbf{FP}^{\mathbf{NP}}$-complete, even if G is an abelian 2-group.

Theorem 2.4. *Lexicographical Leader is $\boldsymbol{FP}^{\boldsymbol{NP}}$-complete [Jun03].*

Note that this is the same complexity as many well-known optimization problems, such as the traveling salesman problem.

It is interesting to note that computing the lexicographical leader problem on a set amounts to solving several instances of this problem at once. It is to be hoped that some computations can be shared between these several instances, rendering them not much harder than a single one.

2.5 Outline

We have now presented the background for our work and are coming to our contributions. Further details about the concepts used in the thesis are presented in following Section 2.6.

We have seen the potential of symmetry reduction for model checking. Such reduction requires to compute a representative state for each orbit. Chosen to be the lexicographical leader of its orbit, this representative is hard to compute (**NP**-hard). Due to the interest of decision diagrams to represent large sets of states and their successful use in model checking, we propose to compute the representative function on sets (represented by decision diagrams) directly.

Chapter 3 presents a new algorithm towards this end, and discusses its implementation in terms of homomorphisms. This implementation raises a problem for complex operations on decision diagrams (swapping the value of two variables), to which Chapter 4 gives an answer, providing a new mechanism for operations on decision diagrams. Interestingly, it can also be used to encode efficiently the transition relation of a distributed system. Chapter 5 then discusses an implementation for a specific formalism, the Symmetric Nets with Bags.

2.6 Other Preliminaries

We present here a few definitions that will be important, although not at the core of this work. This section can be skipped without hindering the comprehension of most of the thesis, although the notions presented are useful for an in-depth comprehension. We first present some basic notions about groups and group actions. We then recall some complexity results, mostly useful for Chapter 3. We then give a few definitions and notations about multi-sets, especially for Chapter 5.

2.6.1 Groups and Group Actions

We recall some basic definitions, notations and vocabulary about groups, since the set of symmetries of a system is a group.

Definition 2.11: Group
A group $(G, \star)$ is a set G endowed with an associative binary operation $\star$ such that:

- $\star$ has a neutral element $e \in G$: $\forall g \in G, g \star e = e \star g = g$;
- every element of G has an inverse for $\star$: $\forall g \in G, \exists h \in G, g \star h = h \star g = e$.

The group operation will be noted multiplicatively, and the symbol $\star$ often omitted. The inverse of an element g is noted g^{-1}. We recall that the neutral element e is necessarily unique, and that g^{-1} is necessarily unique for all $g \in G$.

A *subgroup* of G is a subset $H \subseteq G$ that has a group structure for $\star$. We note $H \lhd G$.

The *order* of a finite group G is its cardinal. The *order* of an element g of a finite group is the order of $\langle g \rangle$, or, equivalently, the smaller positive integer n such that $g^n = e$.

A group is said to be *abelian* if its operation $\star$ is commutative. An *elementary abelian 2-group* is a group whose every nontrivial element is of order 2. Classification of finitely generated abelian groups shows that an elementary abelian 2-group is isomorphic to $(\mathbb{Z}/2\mathbb{Z})^n$ for some integer n.

It is well-known that the set of permutations (or bijections) of a set X, denoted $Sym(X)$ is a group for the composition operator. $Sym(n)$ is used as a shortcut for $Sym(\{1, \ldots, n\})$.

Let now G be a group, and X an arbitrary set. An *action* of G on X is a mapping $G \times X \mapsto X$, noted . in infix notation, such that:

- $\forall x \in X, e.x = x$;
- $\forall g, h \in G, \forall x \in X, (gh).x = g.(h.x)$.

Alternatively, a group action is defined as a group morphism ϕ from G to $Sym(X)$, such that $g.x = \phi(g)(x)$. Both definitions are equivalent.

Symmetries of systems are usually defined through an appropriate group action. For instance, in our mutual exclusion protocol example, the symmetries derive from the fact that the two processes are symmetric. At this point, we thus consider the group $Sym(\{P_1, P_2\})$, that has two elements: the swap of P_1 and P_2, denoted by π, and the identity. This group naturally acts on the states of the system (that are pairs of states): $\pi.\langle x, y, \rangle = \langle y, x\rangle$ for $x, y \in \{N, T, C\}$. Now consider this action as a group morphism ϕ from $Sym(X)$ to $Sym(S)$. The actual group of symmetries of the system is $\phi(Sym(X))$, the image of $Sym(X)$ by ϕ. We have omitted here transition labels, but the reasoning still holds, since $Sym(X)$ would also act naturally on transition labels.

If G acts on X, we define the relation "is symmetrical to", denoted by $\equiv_G$ on X as: $x \equiv_G y$ if and only if exists $g \in G$ such that $g.x = y$. Following the definitions of a group and a group action, $\equiv_G$ is an equivalence relation:

- reflexivity: $\forall x \in X, e.x = x$;
- transitivity: $\forall x \in X, \forall g, h \in G, g.(h.x) = (gh).x$;
- symmetry: $\forall x \in X, \forall g \in G, g.x = y \Leftrightarrow x = g^{-1}.y$.

The equivalence classes of $\equiv_G$ are also called *orbits*. The orbit of $x \in X$ is noted $[x]_G$.

If $\equiv_G$ has a single orbit on X, the action of G on X is *transitive*. Note that an action is always transitive on an orbit.

G acts *freely* on X if, for all $g, h \in G$, the existence of an $x \in X$ such that $g.x = h.x$ implies that $g = h$. Remark that if G acts freely and transitively on X, then for all $x, y \in X$, there is a unique $g \in G$ such that $g.x = y$.

A group $G \lhd Sym(I)$, where I is a finite set, can be as big as $|I|!$. Efficient algorithms for permutation groups exist, and rely on a few simple assumptions. First of all, it is known that every $G \lhd Sym(I)$ can be succintly represented by a generating set of size $\mathcal{O}(\log |G|)$. A particular generating set, known as *strongly generating set*, or Schreier-Sims representation, of size at most $|I|^2$ exists for any $G \lhd Sym(I)$ [Sim71]. Furthermore, a strongly generating set can be computed in polynomial time for any $G \lhd Sym(I)$ given as a set of generators [FHL80]. This time is polynomial in $|I|$ if the input generating sets are bounded in size by a polyomial in $|I|$. Using the strongly generating set of a group G, it is possible to determine in time polynomial in $|I|$ whether a given permutation g belongs to G [FHL80].

We are interested in algorithms polynomial in $|I|$ rather polynomial in $|G|$. Since the computation of a strongly generating set depends on the number of generators given in input, we assume that symmetry groups are given by "small" sets of generators (of size polynomial in $|I|$).

2.6.2 Complexity Classes

We recall some useful definitions of complexity classes, with not much details though. Complete definitions can be found in any book on complexity theory, such as [Pap03]. We limit ourselves here to the classes relevant to this work, as presented in [Jun03].

The class **P** (resp. **NP**) is made of all decision problems decided by deterministic (resp. non-deterministic) Turing machines in polynomial time. The class **co-NP** is made of all decision problems whose complements are in **NP**.

A search problem is defined through a relation $A \subseteq \Sigma^* \times \Sigma^*$ where Σ is a finite alphabet. The relation A is supposed to be polynomially-balanced, *i.e.* there is a fixed polynomial p such that whenever $(u, v) \in A$, then $|v| \leq p(|u|)$. The *search problem* associated with the relation A is: given an input string $u \in \Sigma^*$, output $v \in \Sigma^*$ such that $(u, v) \in A$, or "no" if there is no such v. A search problem is in the class **FP** if there is a deterministic Turing machine that solves the problem in polynomial time. Similarly, a search problem is in the class $\mathbf{FP}^{\mathbf{NP}}$ if there is a deterministic Turing machine with an access to an **NP** oracle that solves the problem in polynomial time.

Let us recall that a graph is a structure made of nodes and edges connecting these nodes. A graph isomorphism is a bijection between the nodes of a graph G_1 and the nodes of a graph G_2 that preserves the edge relations.

Problem 2.3 (Graph Isomorphism)**.** *Given two graphs G_1 and G_2, tell whether there are isomorphic.*

This problem is one of the first that arises in the field of symmetries. Indeed, a symmetry of a graph G (or automorphism) is a non-trivial isomorphism from G to itself. Several problems related to graph symmetries, and symmetries in general, thus relate to Graph Isomorphism.

Note that Graph Isomorphism is a very interesting problem in complexity. It is one of the few problems in **NP** that are not known to be in **P** nor **NP**-hard.

Although not known to be in **P**, efficient sub-exponential algorithms are known. This problem is not considered to be really intractable, as it becomes polynomial for large classes of graphs.

2.6.3 Bags

We present a few definitions about bags (or multi-sets) that will be of primary interest in Chapter 5.

Definition 2.12: Bag
Let C be a set. A bag on C is a function $b : C \mapsto \mathbb{N}$.
The set of bags on C is noted $Bag(C)$.
For $c \in C$, $b(c)$ is the multiplicity of c in b.

Let $b \in Bag(C)$. The cardinal of b, noted $|b|$, is the number of elements in b: $|b| = \sum_{c \in C} b(c)$. Note that if C is finite, then a bag can also be seen as a string in $\mathbb{N}^C$.

We also define basic operations on bags. Let b_1 and b_2 be two bags on C.

- sum (or union) of two bags: for all $c \in C$, $(b_1 + b_2)(c) = b_1(c) + b_2(c)$;
- inclusion: b_1 is included in b_2, noted $b_1 \leq b_2$, if for all $c \in C$, $b_1(c) \leq b_2(c)$. Note that the bag inclusion is an order relation, and that it differs from the lexicographic order;
- difference between two bags: for all $c \in C$, $(b_1 - b_2)(c) = \max(0, b_1(c) - b_2(c))$. It is usual to require that $b_2 \leq b_1$ before computing $b_1 - b_2$.

To distinguish a set from a bag, sets are noted between curly brackets ($\{, \}$) and bags between double curly brackets ($\{\{, \}\}$).

2.6.4 Partition of an Integer

In Chapter 5, an important notion relies on *integer partition*, that is a way of writing an integer as a sum of positive integers.

Definition 2.13: Integer Partition
An integer partition of a positive integer n is a multi-set on integers whose sum equals n. An element of this bag is called a *part*.

For example, $\{\{1, 1, 2\}\}$ and $\{\{2, 2\}\}$ are two partitions of the integer 4. Note that if P is a partition of the set C, then the cardinals of the cells of P form an integer partition of $|C|$.

Chapter 3

Symmetry Reduction using Decision Diagrams

Gardons-nous de chercher ce qu'on ne peut atteindre.[1]

L'Homme à plaindre (1800)
François Andrieux

We now present the first contribution of this work. This section focuses on the computation of the representative function `repr` with decision diagrams.

Several choices are possible for a canonical representative function. For example, the first encountered state of an orbit can be used as the canonical representative state of this orbit. This approach requires to test, for each newly encountered state, if its orbit has already been encountered, leading to an orbit membership test against all already stored representative states. The orbit problem for strings is in **NP** but also Graph Isomorphism-hard. It is believed to be harder than Graph Isomorphism but not **NP**-complete. This choice of such a canonical representative function has thus a prohibitive computational cost.

Alternatively, the canonical representative of an orbit can be chosen to be deterministically computed. A common choice is to endow the set of states with a linear ordering, and use the minimum of an orbit as its canonical representative. This approach requires to compute the canonical representative for each newly encountered state, but the comparison with already stored states becomes very easy. Computing the lexicographical leader for strings is however a **NP**-hard problem, as expressed by Theorem 2.4.

[Jun03] presents quite efficient algorithms for the computation of the orbit lexicographical leader. Given an input state s, this algorithm searches a symmetry g such that $g.s = \min[s]_G$. It first analyses the input s so as to select a small set C of symmetries, among which is the searched g. It then applies all the candidates $h \in C$ to s and returns the one that leads to the smallest $h.s$. In practice, this algorithm has good performance. It can be extended to compute a non-canonical representative function, by using the candidate symmetries to produce a small set of representative states for the input state. The second approach also gives good results in practice, although it does not compute the quotient state space.

However, this algorithm relies on a deep analysis of the input state with respect to the symmetry group, and takes one input state at a time. It can thus be hardly ported to decision diagrams, as they represent sets of states, so that considering one input state at a time would be counter-productive.

[CEFJ96] attempts to implement a representative function on decision diagrams. It however assumes the classical representation of DD operations. A binary relation on states (such as a transition relation, or a representative function) is nothing more than a big set of pairs of states.

[1] Refrain from searching what cannot be reached.

The classical representation of such a binary relation is simply a decision diagram that encodes this set of pairs, with a duplicate set of states variables (one set per component of the pairs). The main result of [CEFJ96] states that the DD encoding of the representative function becomes exponential, whatever the variable ordering. It thus concludes to the non-practicality of the combination of symmetries and decision diagrams.

We however believe that this limitation can be overcome through the homomorphisms based representation. Our first task, described in Section 3.1 is to design an algorithm able to compute the representative states of several input states, so that it can be implemented on top of decision diagrams. Our original solution does however not escape the theoretical worst-case complexity of the lexicographical leader problem, as shown in Section 3.2. Section 3.3 describes the homomorphism encoding of our new algorithm, and possible optimizations. We conclude this chapter with an assessment in Section 3.4 that demonstrates the practicality of out approach, despite of the limitations expressed in Section 3.2.

3.1 A New Algorithm

Let us recall the assumptions made in the previous chapter. System states are characterized by a fixed finite set I of state variables of domain $\mathcal{D}$. Symmetries of the systems are symmetries of I that act naturally on the states. Let $G \lhd Sym(I)$ be the group of symmetries of the system.

The representative function $\mathtt{repr}_{\text{lex}}$ that associates to each state the minimum of its orbit is a good choice of a canonical representative function. The aim of this chapter is to efficiently compute this function with DDD.

First of all, let us turn this function to a set function:

$$\begin{aligned} \mathtt{repr}_{\text{lex}} : 2^{\mathcal{D}^I} &\mapsto 2^{\mathcal{D}^I} \\ S &\mapsto \{\min[s]_G | s \in S\} \end{aligned}$$

For $g \in G$ and $s \in \mathcal{D}^I$, we say that g *reduces* s if $g.s$ is lexicographically smaller than s. We propose to approximate $\mathtt{repr}_{\text{lex}}$ by iteratively reducing states with a *subset* (not necessarily a subgroup) H of G. Section 3.2 discusses the conditions for this approximation to be exact.

This approximate representative function is denoted by $\mathtt{repr}_H$, and defined by Algorithm 3.

Algorithm 3: Set canonization algorithm

Input: $H \subseteq Aut(\mathcal{K})$
Input: S a set of states of $\mathcal{K}$
Output: $R = \mathtt{repr}_H(S)$

```
1 R ← S
2 repeat
3     for g ∈ H do
4         R' ← {s | s ∈ R, g.s < s}
5         R ← R ∪ g.R'
6         R ← R \ R'
7 until R no longer evolves
8 return R
```

This algorithm takes a set of symmetries H and a set of states S to canonize. Note that the states in S belong to several orbits in the general case. The algorithm iterates over the

permutations of H, applying each one only to the states that it reduces. Since the permutations in H are permutations of the symmetry group G of the system, we are ensured that at each step of the algorithm, each state is either left as is, or mapped to a strictly smaller state belonging to its orbit. Since each orbit has a minimum (its canonical representative) this algorithm is guaranteed to converge.

Admittedly, the algorithm might visit each state of an orbit (in decreasing order, one by one), yielding worst case exponential complexity. It is not surprising since the Lexicographical Leader problem is **NP**-hard. In practice however, this algorithm can be quite efficient. Since it works with a set of states, it solves many instances of the Lexicographical Leader problem at once, hopefully sharing some of the computations.

Let us note that the order in which the permutations of H are considered in the "for" loop does not impact correctness, but may impact performance.

Actually, the choice of H is critical to overall performance of this algorithm. If $H = G$, then this algorithm converges after a single iteration of the outer loop ("repeat"). In other words, for each state, H contains the permutation that maps it to its representative. However, this means that, on the worst case, the size of H is exponential in $|I|$. This is congruent with the observations of [CEFJ96] in which the orbit relation is shown to be exponential in representation size.

On the contrary, when H is small, many iterations may be necessary for the algorithm to converge, but each element of H is likely to reduce many more states. Since the complexity of applying a permutation to a set of states is related to the representation size (in DDD nodes) and not to the number of states in the set, manipulating larger sets lowers the overall complexity.

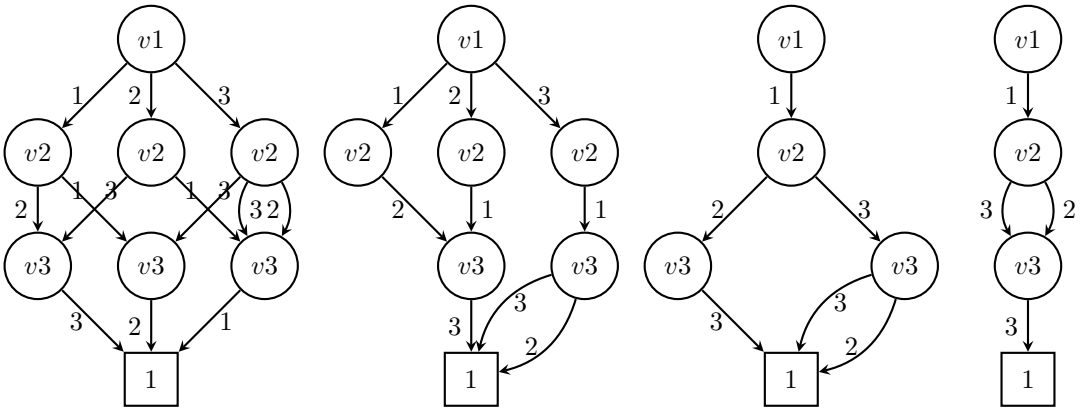
Illustrative example. Let us detail the run of the algorithm on a small illustrative example. Let us assume we have a system with 3 variables $v1$, $v2$ and $v3$ that are all symmetric. We thus consider the symmetry group $G = Sym(\{v1, v2, v3\})$. We choose $H = \{\tau_{1,2}, \tau_{2,3}\}$. Figure 3.1 shows the intermediate steps of the application of $\texttt{repr}_H$ on a set that initially contains the orbit $[(1, 2, 3)]_G$ and the state $(3, 3, 1)$. We focus on the inner loop of Algorithm 3. Each step corresponds to the application of an element g of H to the states reduced by g in the current DD. The first step is to apply $\tau_{2,3}$, that updates the set of states. Then $\tau_{1,2}$ is applied, further reducing the number of states. These two steps are repeated, and we end up with two states. Although not represented here, note that another iteration is necessary at the end of the algorithm to check for convergence.

As we can see through this toy example, each step of the algorithm simultaneously reduces several states. In a single step, each permutation reduces all the states it can, even if they belong to different orbits.

States that belong to the same orbit are progressively collapsed onto their representative. Because of sharing of sub-structures, notice that states $(2, 1, 3)$ and $(2, 3, 1)$ in Figure 3.1a are collapsed onto $(2, 1, 3)$ in Figure 3.1b. $(2, 1, 3)$ is not a canonical representative, but it is smaller than $(2, 3, 1)$. At this step, the two states are merged, allowing to share any subsequent canonization step. In general, each step – with complexity polynomial in the DD size – might merge exponentially many states. This contrasts with explicit approaches that canonize all these states individually.

3.2 A Sufficient Condition for Canonicity

Algorithm 3 defines a representative function $\texttt{repr}_H$, intended to be an approximation of $\texttt{repr}_{\text{lex}}$. We have seen that if $H = G$, then this approximation is actually exact. We here describe a



(a) the orbit of $(1,2,3)$ and the state $(3,3,1)$. (b) after $\tau_{2,3}$ in inner loop (c) after $\tau_{1,2}$ (d) after $\tau_{2,3}$

Figure 3.1: Using $G = Sym(\{v1, v2, v3\})$, we obtain $H = \{\tau_{1,2}, \tau_{2,3}\}$. `set_canonize`$(H)$ applied to (a) successively gives (b), (c), (d). (d) is the set of canonical representatives $\{(1,2,3),(1,3,3)\}$.

sufficient condition on H so that $\texttt{repr}_H = \texttt{repr}_{\text{lex}}$. Let us recall that $\texttt{repr}_H$ is always a valid representative function, and can be used to generate a reduced state space. Its canonicity only ensures that it brings the maximally reduced state space, but possibly at the cost of longer computations.

Looking at Algorithm 3, one sees that, to be sure to reach the minimum of the orbit from any state s, then there must be a permutation in H that reduces s whenever s is not the minimum of its orbit. This is formalized in Definition 3.1.

Definition 3.1: *<-monotonic*
Let $G \lhd Sym(I)$. $H \subseteq G$ is *<-monotonic* w.r.t. G if and only if:

$$\forall s \in \mathcal{D}^I, (\exists g \in G, g.s < s \implies \exists h \in H, h.s < s)$$

It is clear that, if H is *<-monotonic*, then for every state $s \in \mathcal{D}^I$, $\texttt{repr}_H(s) = \min[s]_G$ and thus $\texttt{repr}_H = \texttt{repr}_{\text{lex}}$ and is canonical.

3.2.1 Computing H

We broach here the problem of finding a "good" H for a given group G. We start by complexity results that will limit the search for *<-monotonic* subsets. These negative results will be later balanced by the existence of small and useful *<-monotonic* subsets for commonly encountered groups.

Note that if $|\mathcal{D}| = |I|$ and H is *<-monotonic* for $\mathcal{D}^I$, then H is also *<-monotonic* for any superset $\mathcal{D}'^I$ ($\mathcal{D} \subseteq \mathcal{D}'$). This is due to the fact that at most $|I|$ different elements can appear in a vector of $\mathcal{D}'^I$. Thus, there is a brute force algorithm that computes a *<-monotonic* set H of minimal size with a prohibitive running complexity $\mathcal{O}(|G||I|^{|I|})$.

Also recall that the groups are always given by a "small" set of generators. Therefore, we are interested in the complexity with respect to I rather than to $|G|$. Recall that the size of a group of permutations on I can be as big as $|I|!$, that is computationally prohibitive.

Proposition 3.1. *If $|\mathcal{D}| \geq |I|$, any $<$-monotonic set H w.r.t. a group G is generating set for G.*

Proof. Consider the minimum s of an orbit on which G acts freely (such an orbit exists when $|\mathcal{D}| \geq |I|$). For any $s' \in [s]_G$, there exists a $g \in G$ such that $g.s' = s$. Since G acts freely on $[s]_G$, there is only one such g. Since H is $<$-*monotonic*, g can be written as a product of elements of H, hence $g \in \langle H \rangle$. This holds for every $s' \in [s]_G$. Since G acts freely on $[s]_G$, g covers G when s' covers $[s]_G$. Thus, $g \in \langle H \rangle$ for all $g \in G$, so that $G \subseteq \langle H \rangle$.

Finally, since $H \subseteq G$, thence $\langle H \rangle \subseteq G$, whence $\langle H \rangle = G$. □

Problem 3.1 ($<$-*MONOTONIC* Verification)**.** *Given $H \subseteq Sym(I)$, tell whether H is $<$-monotonic w.r.t. $\langle H \rangle$.*

Problem 3.2 ($<$-*MONOTONIC* Generation)**.** *Given a group G (described with a set of generators), output a subset H of G that is $<$-monotonic.*

$<$-*MONOTONIC* Generation may further require that the output H is of minimal size.

Proposition 3.2. $<$-*MONOTONIC Verification is co-**NP***.

Proof. To prove that H is not $<$-*monotonic*, it suffices to exhibit a permutation g and a string s such that no element of H reduces s (checked in polynomial time), g reduces s (checked in polynomial time) and $g \in \langle H \rangle$ (also in polynomial time, see Section 2.6.1). □

Note however that the polynomial complexity is understood with respect to the size of the input H, and may not be polynomial in the size of I (in case the input H is not bounded in size by a polynomial in $|I|$).

Let us thus explore the minimal size of a $<$-*monotonic* set. We will show that in the general case, the size of $<$-*monotonic* sets can not be bounded by a polynomial in $|I|$.

By contraposition of the Definition 3.1, we see that H is $<$-*monotonic* if and only if:

$$\forall s \in \mathcal{D}^I, (\forall h \in H, h.s \geq s \implies \forall g \in G, g.s \geq s)$$

This characterization resembles those searched for when computing symmetry-breaking predicates for constraint satisfaction problems.

A constraint satisfaction problem P is given by a finite set I of variables, taking values in a given domain $\mathcal{D}$, and a finite set of constraints, that are boolean expressions on variables. The goal of the problem is to find an assignment of the variables (that is a string in $\mathcal{D}^I$) that satisfies all the constraints, if such an assignment exists. Such an assignment is called a *solution* to the problem. A particular case of constraint satisfaction problem is a SAT problem (with input in conjunctive normal form), for which the domain is of size 2.

A symmetry of a constraint satisfaction problem P is a permutation $g \in Sym(I)$, that preserves solutions to the constraints: $s \in \mathcal{D}^I$ is a solution to the problem if and only if $g.s$ is a solution to the problem.

A way to use symmetries in order to reduce the search space for solutions of P is to add to P a new constraint, called a *symmetry-breaking predicate*, that is true only for orbit lexicographical leader strings. The symmetry-breaking predicate thus restricts the set of possible solutions, in

the same way that we want to build a reduced state space for model checking. This approach has been used successfully for instance for SAT solving.

Let us consider that $|\mathcal{D}| = 2$ (we drop our assumption that $|\mathcal{D}| \geq |I|$ for this paragraph only). One can see $\mathcal{D}$ as the set $\{0,1\}$ and a string $s \in \mathcal{D}^I$ as a boolean assignment to $|I|$ boolean variables $(x_i)_{i\in I}$. For $i \in I$ and $g \in G$, let $C(g,i)$ be the clause $(\bigwedge_{j<i} x_j == g.x_j) \implies x_i \leq g.x_i$. Then, for $s \in \mathcal{D}^I$, $g.s \geq s$ if and only if s satisfies the boolean formula $\bigwedge i \in I C(g,i)$. s is the lexicographical leader of its orbit if and only if s satisfies $\Lambda(G,I) =_{def} \bigwedge_{g\in G} \bigwedge_{i\in I} C(g,i)$. Note that $\Lambda(G,I)$ does not depend on s, and is true only on orbit's lexicographical leader strings. $\Lambda(G,I)$ is a possible symmetry-breaking predicate.

It is clear that several clauses in $\Lambda(G,I)$ may be redundant. [LR04] considers eliminating, or pruning, such redundant clauses. However, it states that the number of non-prunable clauses can be exponential.

Theorem 3.3. *[LR04] There are an infinite number of pairs G, I, where $G \lhd Sym(I)$, such that the number of non-prunable clauses in $\Lambda(G,I)$ is c^n for all possible orderings of I, where c is a constant > 1 and $n = |I|$.*
In fact, these groups G have orbits of size ≤ 2 over I and are therefore elementary abelian 2-groups.

We rely on this result to show that there exists $<$-*monotonic* sets of exponential size with respect to $|I|$.

Proposition 3.4. *There exists an infinite number of pairs G, I where $G \lhd Sym(I)$, such that the size of any $<$-monotonic $H \subseteq G$ is greater than $\frac{c^n}{n}$ for all possible orderings of I, where $c > 1$ is a constant and $n = |I|$.*
In fact, this holds when $|\Sigma| = 2$ and G have orbits of size ≤ 2 on I, so that G is an elementary abelian 2-group.

Proof. Let $|\mathcal{D}| = 2$. Let $\mathcal{E}$ be the set of pairs G, I whose existence is given by Theorem 3.3. For $(G,I) \in \mathcal{E}$, let $\mathcal{C}(G,I)$ denote the set of non-prunable clauses in $\Lambda(G,I)$.

Let $H \subseteq G$ be a $<$-*monotonic* set and $s \in \mathcal{D}^I$. On one hand, s is the lexicographical leader of its orbit if and only if $g.s \geq s$ for all $g \in G$, if and only if s satisfies the formula $\Lambda(G,I) = \bigwedge_{g\in G} \bigwedge_{i\in I} C(g,i)$, as explained above.

On the other hand, H is known to be $<$-*monotonic* w.r.t. G. Therefore, s is the lexicographical leader of its orbit if and only if $h.s \geq s$ for all $h \in H$, if and only if s satisfies the formula $\Lambda(H,I) =_{def} \bigwedge_{h\in H} \bigwedge_{i\in I} C(h,i)$.

This holds for all state s, so that $\Lambda(G,I)$ is true on a state s if and only if $\Lambda(H,I)$ is true on s, which establishes the logical equivalence between $\Lambda(G,I)$ and $\Lambda(H,I)$.

Since H is included in G, all the clauses $C(h,i)$, for $h \in H$ and $i \in I$, that appear in $\Lambda(H,I)$ also appear in $\Lambda(G,I)$. Therefore, the clauses $C(g,i)$, for $g \in G - H$ and $i \in I$, that appear in $\Lambda(G,I)$ and not in $\Lambda(H,I)$, are redundant in $\Lambda(G,I)$ and can be pruned. We thus conclude that all the non-prunable clauses, gathered in $\mathcal{C}(G,I)$ appear in $\Lambda(H,I)$.

Let us now count the clauses in $\Lambda(H,I)$. Each $h \in H$ gives rise to at most $n = |I|$ clauses in $\Lambda(H,I)$, so that $|\mathcal{C}(G,I)| \leq |\{C(h,i) | h \in H, i \in I\}| \leq n|H|$.

By Theorem 3.3, there exists a constant $c > 1$ such that $|\mathcal{C}(G,I)| \geq c^n$ for all $(G,I) \in \mathcal{E}$. This leads to the double inequality $c^n \leq |\mathcal{C}(G,I)| \leq n|H|$, whence $\frac{c^n}{n} \leq |H|$.

This holds for any $<$-*monotonic* set H, and whatever the ordering of I. □

Proposition 3.4 shows that the size of $<$-*monotonic* sets is not polynomial in the size of $|I|$.

Theorem 3.5. *The size of $<$-monotonic sets H of minimal size w.r.t. $G \lhd Sym(I)$ cannot be bound by a polynomial in $|I|$.*

This result is quite important as it shows the difficulty to find and use a $<$-*monotonic* set in the general case.

3.2.2 H for Commonly Encountered Groups

Despite Theorem 3.5, small $<$-*monotonic* sets exists for generally encountered groups. Let us recall once more that the $<$-*monotonic* property is only required to get maximal reduction.

Proposition 3.6. *The set of adjacent transpositions is $<$-monotonic for $G = Sym(I)$.*

Proof. Let $s = (s_1, \ldots, s_n) \in \mathcal{D}^I$ be a state, such that $\exists g \in Sym(I), g.s < s$. This means that s is not sorted, and therefore, there exists an index $i \in I$ such that $s_i > s_{i+1}$. Thus $s' = \tau_{i,i+1}.s = (s_1, \ldots, s_{i+1}, s_i, \ldots, s_n) < s$. □

Proposition 3.7. *Let r be the rotation $(2, 3, \ldots, n, 1)$, and $G = \langle r \rangle = \{id, r, r^2, \ldots, r^{|I|-1}\}$. Then G is the only $<$-monotonic set w.r.t. G.*

Proof. For $0 < i \leq |I|$, let $s = (1, 2, \ldots i-1, 0, i+1, \ldots, |I|)$. Then the only rotation in G that reduces s is r^i. □

These two groups are the most frequently encountered groups of symmetries in the literature, as they occur naturally in many symmetric systems. This gives us $<$-*monotonic* sets of size $|I|$ for these two groups. The two properties above are still true when considering groups that act on a subset of the system variables.

When the symmetries of the system arise from several symmetry groups (i.e. symmetries of subsystems), we choose to use the union of their respective $<$-*monotonic* sets.

Let $G = \langle E \cup F \rangle$, and H_E, H_F be $<$-*monotonic* sets w.r.t. E and F respectively. $H_G = H_E \cup H_F$ is $<$-*monotonic* w.r.t. G if E and F act on disjoint sets of variables. Otherwise, we are not ensured that H_G is $<$-*monotonic* w.r.t. G, but it can still be used as a good candidate set for the algorithm.

3.3 Symmetries and Symbolic Structures

Let us now describe how to encode Algorithm 3 in terms of homomorphisms. Following the guidelines of Section 2.3.5, we identify several building blocks that will be combined with homomorphisms operators. After a first attempt, we will rewrite the combination in order to increase performance.

Let us note that, although the encoding is presented for DDD homomorphisms, our algorithm can be used with any kind of decision diagrams.

3.3.1 Implementation with Homomorphisms

States, that are $\mathcal{D}$-strings on I, are naturally represented as a DDD of $n = |I|$ variables. Note that by assumption, n is fixed.

To encode Algorithm 3 using homomorphisms, we need for each permutation $g \in H$:

- $\texttt{red}_g$, a selector that retains states reduced by g, *i.e.* $\texttt{red}_g(S) = \{s \in S | g.s < s\}$;
- $\texttt{apply}_g$, a homomorphism that applies g to a set, *i.e.* $\texttt{apply}_g(S) = \{g.s | s \in S\}$.

In fact, there are no restrictions on the permutations that appear in H. The above operations are therefore defined for every $g \in Sym(I)$, so that the algorithm can be encoded for any input H.

Consider the inner loop (lines 4 to 6) of Algorithm 3. It stores in R' the states in R that are reduced by the current permutation g, then replaces in R these permutations by their reduced version (resulting from the application of g). This inner loop is expressed with the above homomorphisms as $\texttt{IfThenElse}(\texttt{red}_g, \texttt{apply}_g, Id)$. This conditional homomorphism splits its argument in two parts: the states reduced by g on the one hand, and those that are not on the other hand. g is then applied to the former, while the latter is left as is (Id is applied). Then, the result is the union of these two sets.

The walk of all permutations in H is a composition:

$$\bigodot_{g \in G} \texttt{IfThenElse}(\texttt{red}_g, \texttt{apply}_g, Id)$$

Finally, the outer loop guarantees that the inner treatment is applied until R stabilizes. Therefore, the whole algorithm is expressed as a fixpoint:

$$\texttt{set\_canonize}(H) = (\bigodot_{g \in H} \texttt{IfThenElse}(\texttt{red}_g, \texttt{apply}_g, Id))^\star$$

Since convergence is ensured by the fact that each orbit has a minimum, the fixpoint * is well-defined. The homomorphism $\texttt{set\_canonize}(H)$ can be applied to any set of states, yielding their canonical representatives when H is $<$-*monotonic*.

3.3.2 Homomorphism Encoding of $\texttt{red}_g$ and $\texttt{apply}_g$

Although it is possible to encode $\texttt{red}_g$ and $\texttt{apply}_g$ as monolithic homomorphisms, we apply the guidelines edicted in Section 2.3.5, and try to find reusable building blocks. We naturally think of the decomposition of a permutation as a product of transpositions, for several reasons. First of all, a transposition affects only two variables, and this locality is empirically a good property when it comes to decision diagrams. Secondly, the total number of transpositions is quadratic in $|I|$, so we have a guaranteed bound on the number of building blocks used to implement all permutations in H, whose size is not so sharply bounded, as seen in Section 3.2. This reduced number of building blocks is favorable to the memoization mechanisms. The homomorphisms encoding these transpositions will serve as building blocks on top of which all other permutations can be built.

Formally, for $g \in G$, we define the transpositions $\tau_1, \cdots, \tau_n$ and the permutations $g_1, \cdots, g_{n+1}$ co-inductively as follows:

- $g_1 = g$;
- $\forall 1 \leq i \leq n, \tau_i = (i, g_i^{-1}(i))$;

- $\forall 1 \leq i \leq n, g_{i+1} = g_i\tau_i$.

Note that $g_{i+1}(i) = (g_i\tau_i)(i) = g_i(g_i^{-1}(i)) = i$. Based on this, an easy recursion shows that $g_i(j) = j$ for all $j < i$, and in particular $g_{n+1} = id$. The third condition can be rewritten as $\forall 1 \leq i \leq n, g_i = g_{i+1}\tau_i$. Since $g_{n+1} = id$, this shows clearly that $g_i = \tau_n \dots \tau_i$ for all $1 \leq i \leq n$. Hence $g = \tau_n \dots \tau_1$ is the well-known decomposition of g as a product of transpositions.

Consider for example the permutation $g = (2, 3, 1, 4)$. We detail its decomposition into transpositions:

$$\begin{aligned} g_1 &= g = (2,3,1,4) & \tau_1 &= (1,3) \\ g_2 &= g_1\tau_1 = (1,3,2,4) & \tau_2 &= (2,3) \\ g_3 &= g_2\tau_2 = (1,2,3,4) & \tau_3 &= id \end{aligned}$$

Therefore, $g = (2, 3)(1, 3)$.

The homomorphism $\texttt{red}_g$ can then be built from the $\texttt{red}_{\tau_i}$'s.

Lemma 3.8. *$g.s < s$ if and only if:*

- *either $\tau_1.s < s$,*
- *or $\tau_1.s = s$ and $(g\tau_1).s < s$.*

Proof. ($\Rightarrow$) If $g.s < s$, then $s_{g^{-1}(1)} \leq s_1$. If $s_{g^{-1}(1)} < s_1$, then $\tau_1.s < s$. Otherwise, $s_{g^{-1}(1)} = s_1$, thence $\tau_1.s = s$. In this case $(g\tau_1).s = g.(\tau_1.s) = g.s < s$.

($\Leftarrow$) If $\tau_1.s < s$, then $s_{g^{-1}(1)} < s_1$ and thus $g.s < s$.
If $\tau_1.s = s$ and $(g\tau_1).s < s$, then $(g\tau_1).s = g.(\tau_1.s) = g.s < s$. □

At this point, we introduce $\texttt{stab}_g$, a homomorphism to retain states that are unchanged by g, *i.e.* $\texttt{stab}_g(S) = \{s \in S | g.s = s\}$. Using Lemma 3.8 recursively, we obtain:

$$\texttt{red}_g = \sum_{i=1}^{n} \texttt{red}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j} \tag{3.1}$$

For instance, consider again the permutation $g = (2, 3, 1, 4)$. From the above decomposition $g = (2, 3)(1, 3)$, $g.s < s$ if and only if $\tau_1.s < s \vee (\tau_1.s = s \wedge \tau_2.s < s)$, or equivalently if $s_3 < s_1 \vee (s_3 = s_1 \wedge s_2 < s_3)$. Let us note that since position 4 is invariant by g, there are only three nested variable comparisons. Subsequent conditions are trivially simplified away.

When τ is a transposition, $\texttt{stab}_\tau$ and $\texttt{red}_\tau$ are comparisons between two variables. If τ swaps variable x and y, with $x < y$ in I, then $\texttt{red}_\tau$ (resp. $\texttt{stab}_\tau$) selects the paths where $x < y$ (resp. $x = y$). Thus, $\texttt{red}_g$ can be built from variable comparisons, composition (logical and) and sum (logical or) of homomorphisms. Note that when $x = y$, $\texttt{red}_\tau$ is the null homomorphism and $\texttt{stab}_\tau$ is the identity homomorphism. These trivial simplifications are assumed to take place, although we do not explicitly reflect them in the following, for the sake of clarity. Practically, in `libddd`, such simplifications are done automatically, so that we never reflect them in our reasonings.

We note that, for all $g_1, g_2 \in G$:

$$\texttt{apply}_{g_1} \circ \texttt{apply}_{g_2} = \texttt{apply}_{g_1 \circ g_2} \tag{3.2}$$

$$\texttt{stab}_{g_1} \circ \texttt{stab}_{g_2} = \texttt{stab}_{g_1 \circ g_2} \tag{3.3}$$

We thus use the decomposition of g into transpositions to state that:

$$\texttt{apply}_g = \texttt{apply}_{\tau_n} \circ \cdots \circ \texttt{apply}_{\tau_1}$$

$\texttt{apply}_\tau$ when τ is a transposition is simply swapping two variables. The original DDD definition [CEPA+02] includes a general homomorphism to swap two arbitrary variables of a DDD. Chapter 4 also shows how to swap efficiently two variables in a DDD. We thus assume that this basic homomorphism is available.

Note that the same homomorphism bricks can be used to compute the orbit of states, using the equation:

$$orbit(H) = (\bigodot_{g \in H} (\texttt{apply}_g + Id))^\star$$

If $\langle H \rangle = G$, applying $orbit(H)$ to a set of states S returns the set $\bigcup_{s \in S} [s]_G$.

3.3.3 Optimizing the Conditional Statement

We have provided a way to encode Algorithm 3 with homomorphisms, based on the decomposition of a permutation into transpositions. We now optimize this encoding. Indeed, rather than $\texttt{red}_g$ and $\texttt{apply}_g$, we ultimately aim at encoding $\texttt{IfThenElse}(\texttt{red}_g, \texttt{apply}_g, Id)$. Recall Lemma 3.8, and let s be a state such that $g.s < s$ and $\tau_1.s = s$, say $s = \langle 1, 0, 1, 0 \rangle$. When applying $\texttt{IfThenElse}(\texttt{red}_g, \texttt{apply}_g, Id)$ to s, it is selected by $\texttt{red}_g$, then is applied $\texttt{apply}_g$. But since $\tau_1.s = (1,3).s = s$, $\texttt{apply}_g$ starts by swapping two variables (the first and the third here) that have the same value (1 in our example). This unnecessary operation costs time and memory (since it fills the caches). We thus propose a direct encoding of $\texttt{IfThenElse}(\texttt{red}_g, \texttt{apply}_g, Id)$ to avoid such unnecessary swaps.

This new encoding essentially relies on the use of distributivity of homomorphisms operations to simplify away trivial combinations. Recall that a composition of selectors encodes a logical "and", and thus that composition of selectors is commutative. We also note that $\texttt{apply}_g \circ \texttt{stab}_g = \texttt{stab}_g$ for any g. Such a simplification is typically one that cannot be detected by the runtime engine, and has to be hard-coded. We thus obtain:

$$\begin{aligned}
\texttt{apply}_g \circ \texttt{red}_g &= \texttt{apply}_g \circ \sum_{i=1}^{n} (\texttt{red}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) && \text{by Equation (3.1)} \\
&= \sum_{i=1}^{n} (\texttt{apply}_g \circ \texttt{red}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) && \text{by expansion} \\
&= \sum_{i=1}^{n} (\texttt{apply}_g \circ \texttt{red}_{\tau_i} \circ \texttt{stab}_{\tau_{i-1} \circ \cdots \circ \tau_1}) && \text{by Equation (3.3)} \qquad (3.4)
\end{aligned}$$

Let $i \in \{1, \ldots, n\}$. We recall that all selectors commute, by Proposition 2.2, and that

$g = \tau_n \circ \cdots \circ \tau_1$. Thus, the ith term of the sum of Equation (3.4) is rewritten:

$$\begin{aligned}
&\texttt{apply}_g && \circ \texttt{red}_{\tau_i} \circ \texttt{stab}_{\tau_{i-1}\circ\cdots\circ\tau_1} && \\
=&\texttt{apply}_g && \circ \texttt{stab}_{\tau_{i-1}\circ\cdots\circ\tau_1} \circ \texttt{red}_{\tau_i} && \text{by commutativity} \\
=&\texttt{apply}_{\tau_n\circ\cdots\circ\tau_1} && \circ \texttt{stab}_{\tau_{i-1}\circ\cdots\circ\tau_1} \circ \texttt{red}_{\tau_i} && \\
=&\texttt{apply}_{(\tau_n\circ\ldots\tau_i)\circ(\tau_{i-1}\circ\cdots\circ\tau_1)} && \circ \texttt{stab}_{\tau_{i-1}\circ\cdots\circ\tau_1} \circ \texttt{red}_{\tau_i} && \\
=&\texttt{apply}_{\tau_n\circ\cdots\circ\tau_i} \circ \underbrace{\texttt{apply}_{\tau_{i-1}\circ\cdots\circ\tau_1} \circ \texttt{stab}_{\tau_{i-1}\circ\cdots\circ\tau_1}} && \circ \texttt{red}_{\tau_i} && \text{by Equation (3.2)} \\
=&\texttt{apply}_{\tau_n\circ\cdots\circ\tau_i} \circ && \texttt{stab}_{\tau_{i-1}\circ\cdots\circ\tau_1} && \circ \texttt{red}_{\tau_i} \qquad (3.5)
\end{aligned}$$

Further using the fact that $g_i = \tau_n \circ \cdots \circ \tau_i$, we rewrite Equation (3.5) as:

$$\begin{aligned}
\texttt{apply}_g \circ \texttt{red}_{\tau_i} \circ \texttt{stab}_{\tau_{i-1}\circ\cdots\circ\tau_1} &= \texttt{apply}_{\tau_n\circ\cdots\circ\tau_i} \circ \texttt{stab}_{\tau_{i-1}\circ\cdots\circ\tau_1} \circ \texttt{red}_{\tau_i} \\
&= \texttt{apply}_{g_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j} \circ \texttt{red}_{\tau_i} \\
&= \texttt{apply}_{g_i} \circ \texttt{red}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}
\end{aligned}$$

Finally, we obtain for Equation (3.4):

$$\texttt{apply}_g \circ \texttt{red}_g = \sum_{i=1}^{n} (\texttt{apply}_{g_i} \circ \texttt{red}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) \qquad (3.6)$$

Beyond $\texttt{apply}_g \circ \texttt{red}_g$, we aim at an optimization for $\texttt{IfThenElse}(\texttt{red}_g, \texttt{apply}_g, \mathit{Id})$, so that we are also interested in $\overline{\texttt{red}_g}$. Obviously, the sets not reduced by g are those that are either increased by g, or invariant under g. Let $\texttt{incr}_g$ denote the homomorphism that selects only the states increased by g, so that $\texttt{red}_g = \texttt{incr}_g + \texttt{stab}_g$. $\texttt{incr}_g$ is very similar to $\texttt{red}_g$, and the same reasoning as above gives $\texttt{incr}_g = \sum_{i=1}^{n} \texttt{incr}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}$. Thus:

$$\begin{aligned}
&\texttt{IfThenElse}(\texttt{red}_g, \texttt{apply}_g, \mathit{Id}) \\
=&\texttt{apply}_g \circ \texttt{red}_g + \overline{\texttt{red}_g} \\
=&\sum_{i=1}^{n} (\texttt{apply}_{g_i} \circ \texttt{red}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) + \overline{\texttt{red}_g} && \text{by Equation (3.6)} \\
=&\sum_{i=1}^{n} (\texttt{apply}_{g_i} \circ \texttt{red}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) + \texttt{incr}_g + \texttt{stab}_g \\
=&\sum_{i=1}^{n} (\texttt{apply}_{g_i} \circ \texttt{red}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) + \sum_{i=1}^{n} (\texttt{incr}_{\tau_i} \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) + \texttt{stab}_g \\
=&\sum_{i=1}^{n} ((\texttt{apply}_{g_i} \circ \texttt{red}_{\tau_i} + \texttt{incr}_{\tau_i}) \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) + \texttt{stab}_g && \text{by factoring} \\
=&\sum_{i=1}^{n} ((\texttt{apply}_{g_i} \circ \texttt{red}_{\tau_i} + \texttt{incr}_{\tau_i}) \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) + \bigodot_{i=1}^{n} \texttt{stab}_{\tau_i}
\end{aligned}$$

With the convention that $\texttt{red}_{\tau_{n+1}} = \texttt{incr}_{\tau_{n+1}} = Id$, we can finally rewrite

$$\texttt{IfThenElse}(\texttt{red}_g, \texttt{apply}_g, Id) = \sum_{i=1}^{n+1}((\texttt{apply}_{g_i} \circ \texttt{red}_{\tau_i} + \texttt{incr}_{\tau_i}) \circ \bigodot_{j=1}^{i-1} \texttt{stab}_{\tau_j}) \tag{3.7}$$

We thus obtain an optimized homomorphism encoding of Algorithm 3. As we have seen, this optimized encoding has been obtained through a rewriting of the "naive" homomorphisms using factorizations and simplifications that the runtime cannot automatically perform. As already explained, this rewriting avoids some unnecessary homomorphism evaluations, so as to reduce the number of intermediate results that burdens the unique and memoization tables.

3.4 Assessment

Implementation

We have implemented our algorithm, in C++, as an extension of our DDD/SDD-based model checking tool suite `libits`, and that we call DD-Sym. In order to assess our algorithm, we compare our tool to:

- an implementation of Junttila's algorithm for symmetry reduction,
- and a symbolic model checker that uses no symmetry.

Tools

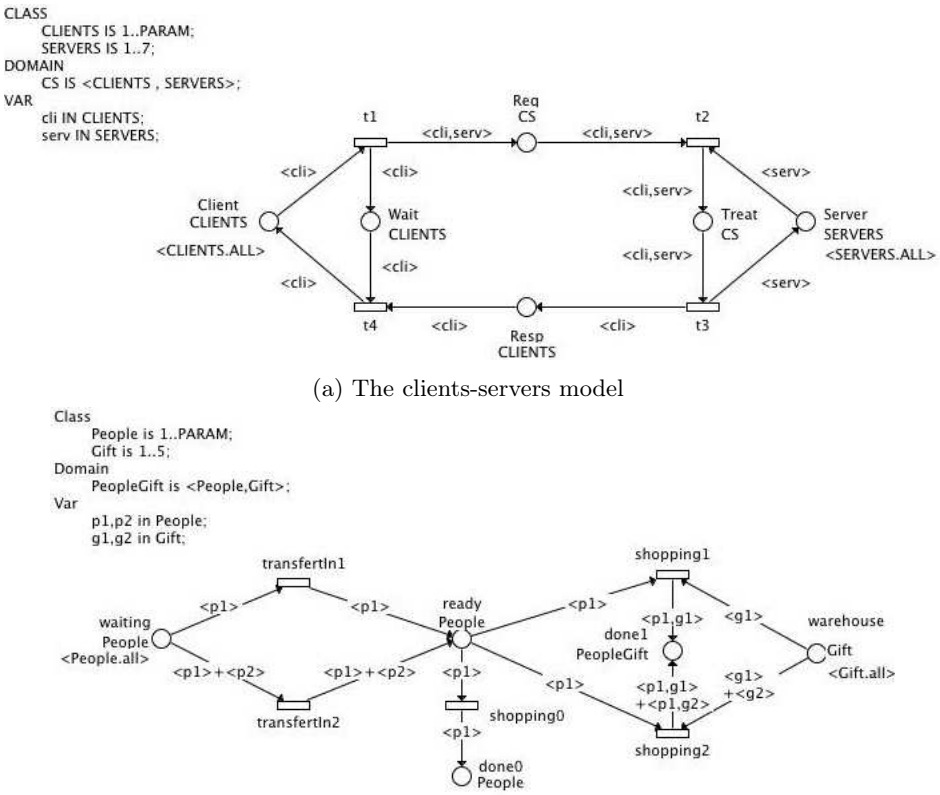
LoLa [Uni12] is a tool with various capabilities oriented towards the analysis of Colored Petri Nets. We are interested here in its computation of the reduced state space of the system. It uses a Schreier-Sims representation of the symmetry group and produces a reduced state space with potentially several representatives per orbit. Its strategy is to avoid the complexity of a canonical representative function, and is supposed to be a good trade-off between this time-consuming process and the size of the state space. It works with explicit data structures, thus its memory consumption grows linearly with the number of representative states. LoLa is a well-maintained and mature software. Despite what we claimed in [CKTMB12], LoLa does not implement the exact Junttila's algorithm, but some variant, as was demonstrated by a comparison with the original implementation by Junttila.

`libits` [MoV13, TMPHK09] is a model checking library that is based on DDD/SDD and that does not use symmetry. Since DD-Sym uses the same DDD/SDD library, this allows to compare the algorithms (with and without symmetry reduction) built on top of the symbolic structures. The encoding (states, transition relation) is also the same as in our prototype DD-Sym.

Our own algorithm may produce several representatives if the provided set H is not $<$-*monotonic* w.r.t. G.

Models

The comparison is based on the generation of the state space (full or reduced depending on the tool) of the following models. They are all Colored Petri Nets, so as to be treated by LoLa. `libits` (as well as DD-Sym) is also able to treat such models. They all have a scaling parameters,



(a) The clients-servers model

(b) The Salestore model

Figure 3.2: Two models used in our assessment

indicated by PARAM in Figure 3.2. Of these three models, only the first one was not designed by us. However, the other ones represent so widely used situations (especially the clients-servers model) that they can hardly be blamed for being biased in our favor.

Software Product Line (SPL) [MCP10]. This model is extracted and then adapted from a case study concerning a software configuration process. Features and configuration options are fully symmetric domains, that do not interact directly. Thus, the union of their respective $<$-*monotonic* sets is $<$-*monotonic* w.r.t. the symmetry group of the model. This is a common case for symmetries, where the global symmetries of the system are the independent combination of local symmetries (here symmetries on two data types). Its Colored Petri Net has a poor graphical representation, due to a quite high number of edges, and is not shown here. The scaling parameter of this model is the number of products.

Clients Servers (CS) [TMIP04]. It models a simple remote procedure call protocol between n clients and p servers sharing a common communication channel. Clients (resp. servers) are considered indistinguishable up to their identity. Thus, we have a full symmetry group on clients and a full symmetry group on servers. We use the union of their respective $<$-*monotonic* sets, although this union is not $<$-*monotonic* w.r.t. the symmetries of the whole system. This is another common case for symmetries, where global symmetries result from the interaction of

Model	Scale	# generated states			time (s)			Memory (MB)		
		LoLa	libits	DD-Sym	LoLa	libits	DD-Sym	LoLa	libits	DD-Sym
SPL	80	486	$3.8685 \cdot 10^{25}$	486	540	135	193	364	273	558
SPL	100	606	$4.0564 \cdot 10^{31}$	606	1,488	238	399	558	411	964
SPL	120	726	$4.2535 \cdot 10^{37}$	726	3,284	389	776	795	589	1,305
SPL	140	—	$4.4601 \cdot 10^{43}$	846	—	581	1,417	—	800	1,311
SPL	160	—	$4.6768 \cdot 10^{49}$	966	—	844	2,472	—	1,014	1,314
SPL	180	—	$4.9040 \cdot 10^{55}$	—	—	1,167	—	—	1,256	—
SPL	200	—	$5.1422 \cdot 10^{61}$	—	—	1,587	—	—	1,560	—
CS	5	22,840	805,284	192	1	4	1.4	14	122	58
CS	6	425,646	11,368,449	448	27	18	3.8	234	383	140
CS	7	3,630,511	157,169,826	1,024	452	67	8.8	1,971	1,232	267
CS	8	—	$2.1307 \cdot 10^{9}$	2,295	—	306	19	—	3,200	521
CS	12	—	—	42,926	—	—	350	—	—	4,004
SaleSt.	5	4,456	71,238	106	0.1	0.93	0.32	3.9	36	17
SaleSt.	10	1,410,608	184,554,369	496	111	37	3.2	689	708	99
SaleSt.	15	—	$2.0763 \cdot 10^{11}$	1,186	—	692	12.5	—	4,190	303
SaleSt.	20	—	—	2,176	—	—	30	—	—	722
SaleSt.	30	—	—	5,056	—	—	154	—	—	2,455
SaleSt.	40	—	—	9,136	—	—	495	—	—	4,194

Table 3.1: Performances of state space generation using LoLa, plain DD (`libits`) and the combination of DD with symmetries.

local symmetries (here on clients and servers). The scaling parameter of this model is the number of clients, as shown on Figure 3.2a.

SaleStore [CBKTM11]. It models a shopping mall where clients can shop for gifts. Clients and gifts form two fully symmetric domains, that interact when a client buys some gifts. Similarly to the clients servers model, we use the union of the $<$-*monotonic* sets. Its scaling parameter is the number of people, as shown on Figure 3.2b.

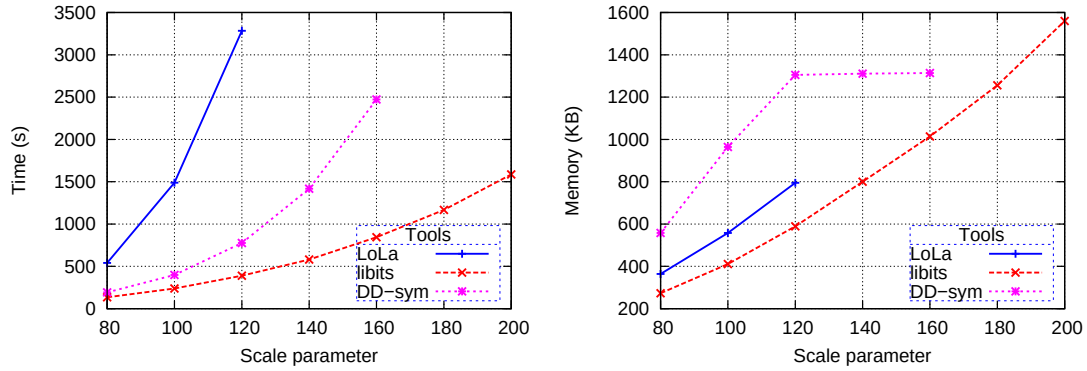
Results and Discussion

Table 3.1 compares the size of the produced state space (reduced state space for LoLa and DD-Sym, full state-space for `libits`) and the time and memory used for its computation, for the three tools. Experiments were run on a Xeon 64bits at 2.6 GHz processor, with a time limitation of 1 hour and memory limit of 5 Gbytes.

We immediately note in Table 3.1 that the three tools do not agree on the number of states. They indeed compute different state spaces: `libits` computes the full state space, whereas LoLa and DD-Sym compute reduced state spaces, and the number of states shown is actually the number of found representative states. Both LoLa and DD-Sym use an algorithm that may lead to several representatives per orbit. In order to control when they achieve full reduction, we have processed small instances of the models with GreatSPN [Gre11] that is guaranteed to perform full reduction. In practice, both LoLa and DD-Sym compute a single representative per orbit for the Software Product Line model. For the two other models, recall that the sets fed to DD-Sym are not $<$-*monotonic*. In spite of this, DD-Sym computes a single representative per orbit for the SaleStore model. On the Clients Servers model, neither LoLa nor DD-Sym achieve maximal reduction, but it is to be noted that LoLa computes many more states than DD-Sym.

On the Software Product Line model, LoLa and DD-Sym fail due to time confinement. `libits` has the best performance on this model, both in time and memory. The comparison of the memory consumption of `libits` and LoLa demonstrates the compactness of the decision diagrams: `libits` represents 10^{35} states in less than 600 MB, whereas LoLa uses almost 800 MB for the computation of 700 states. DD-Sym takes more time and memory for its computations than

`libits`. This demonstrates the cost of the symmetry reduction. DD-Sym seems however to benefit from its DDD encoding, since it performs better than LoLa to compute the same quotient state space.

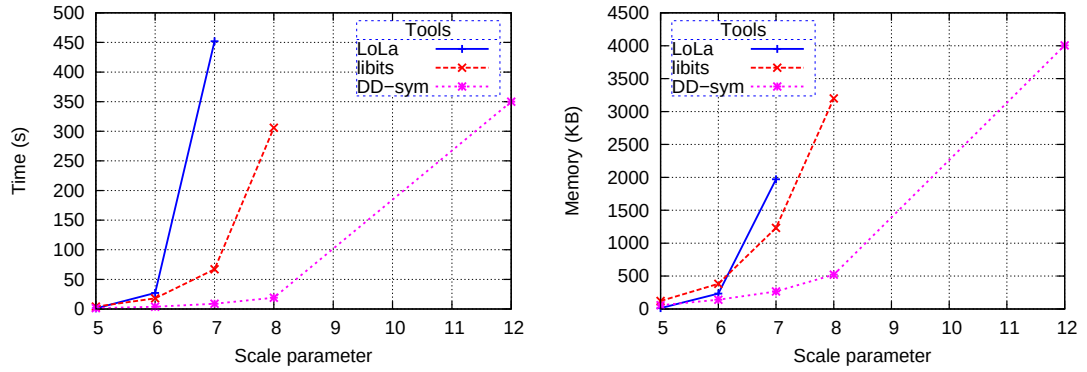


(a) Software Product Line model: time to build the state space.

(b) Software Product Line model: memory to build the state space.

Figure 3.3: Performances on the Software Product Line model

On the two other models, DD-Sym has the best performance, and LoLa the worst one as it fails early due to memory confinement. More specifically, `libits` scales better than LoLa, and DD-Sym scales even better than `libits`, although the performances on the small instances are more in favor of LoLa. This difference of scaling is evident on Figure 3.4, that shows the evolution of performance of the three tools following the scaling parameter, both in time (Figure 3.4a) and in memory (Figure 3.4b). LoLa computes many more representative states than DD-Sym (between 100 and 1000 times more). Its strategy to keep several representatives in order to avoid the complexity of the canonical representative function seems then not-well balanced. We demonstrate here that our method, even with non $<$-*monotonic* sets, is able to compute a much smaller state space than LoLa.



(a) Clients servers model: time to build the state space.

(b) Clients servers model: memory to build the state space.

Figure 3.4: Performances on the clients-servers model

We also demonstrate that decision diagrams can handle a large state space in a reasonable

amount of memory. Although this ability is not very relevant for DD-Sym that reaches only a few thousands states, the underlying mechanisms, especially memoization, contributes to its good performance. On these two last models, our new algorithm outperforms both state-of-the-art symmetry reduction algorithms, and symbolic algorithms, by combining the advantages of both approaches.

Despite the small size (in number of places) of the tested models, these good results should also hold for larger Petri Nets. More places mean more variables in the decision diagrams, and larger state spaces. Such an increase would equally impact the performance of a DD only tool and of a DD with symmetries tool. The impact of larger state spaces is also stronger for explicit tool like LoLa than for symbolic ones.

3.5 Conclusion

We have presented a new algorithm, well adapted to decision diagrams, for the computation of sets of lexicographic leader strings. It is based on successive minimization steps, guided by a subset of the symmetry group given as input. In the context of finding orbit representative states for symmetry reduction, it shows flexibility, the symmetry subset allowing to lead to various representative functions. We exhibit a condition for this set to yield a canonical representative function. We also show that an efficient implementation in terms of homomorphisms is possible, and demonstrate through experiments that such an implementation compares favorably against state-of-the-art tools. This shows that the combination of symmetry reduction and symbolic structures is able to stack the advantages of both individual techniques.

During our description of the homomorphism implementation, we have voluntarily left out the encoding of the swapping of two variables. We define in the next chapter new operations on decision diagrams, that are applied to the encoding of a relation transition, and to the efficient encoding of this swapping.

New Efficient Operations for DD Manipulation

Chapter 4

> Non quia difficilia sunt non audemus, sed quia non audemus difficilia sunt.[1]
>
> *Epistulae Morales ad Lucilium* (64)
> Seneca

We define new operations for decision diagrams manipulation, through the algorithm *EquivSplit*. It evaluates a syntactical expression on a set of states by successive refinements of subsets of the input set of states. During the process, it dynamically reduces the support – the set of affected variables – of the expression, thus overcoming the difficulties encountered by previous approaches. Large supports often result from array manipulation or composition of local effects. When previous methods consider all the potential input states for the transition relation, our algorithm computes the transition relation only for encountered states. Large potential supports due to array manipulations are correctly resolved on-the-fly by *EquivSplit*.

In the previous chapter, we have seen how to encode permutations in homomorphisms. We rely on the existence of a basic homomorphism that swaps the value of two variables. Although such a homomorphism was designed at the very birth of DDD [CEPA+02], the chosen solution was not very efficient because it relied on many intermediate results, thus burdening the caches. Interestingly, *EquivSplit* gives a new, more efficient, implementation for the swap of two variables, thus making an elegant continuation to the previous contribution.

We present this operation *EquivSplit* in its whole generality in this chapter. We start with a few preliminaries in Section 4.1, that set up the general framework for the description of the operation. This description occurs in Section 4.2, with proofs of correctness. The general framework set up in Section 4.1 is then instantiated in Section 4.3 for the evaluation of a transition relation, and in Section 4.4 for the design of a swapping homomorphism. Assessment of the efficiency of this new operation is then presented in Section 4.5.

4.1 Expressions

In practice, a system state is a valuation of the state variables, *i.e.* a string in $\mathcal{D}^X$, and the behavior of the system is described with expressions. Treating such a system using DDD raises the need to evaluate an expression over a *set* of valuations.

We first define in Section 4.1.1 the concepts and notations used throughout the chapter, especially regarding expressions and valuations. The abstract level of these definitions guarantees independence from any concrete syntax. We then give flesh to these definitions with more concrete examples in Section 4.1.3.

[1] It is not because things are difficult that we do not dare, but because we do not dare, things are difficult.

4.1.1 Definitions and Notations

Let Σ be a signature, that is a set of symbols of finite arity. We inductively define the set `Expr` of Σ-*expressions* as $\phi \in$ `Expr` if and only if:

- $\phi \in \Sigma$ of arity 0,
- or $\phi = s(\phi_1, \ldots, \phi_k)$ where $s \in \Sigma$ is of arity k and $\phi_1, \ldots, \phi_k \in Expr$ (ϕ_i is called a sub-expression).

Let $\mathcal{D}$ be a domain for expressions. We assume that $\mathcal{D}$ is embedded in Σ, so that every element of the domain can be referred to syntactically.

Definition 4.1: Interpretation
An *interpretation* I is a function that associates to every symbol $s \in \Sigma$ of arity $k > 0$ a (possibly partial) function $I(s) : \mathcal{D}^k \mapsto \mathcal{D}$, and that maps each symbol of arity 0 to its corresponding element of $\mathcal{D}$.

Intuitively, this formalism captures most programming languages, with pointers and pointer arithmetic. From now on, we assume that there is a finite subset X in $\mathcal{D}$, called *addresses*. The set of addresses X being finite, we note $X = \{x_1, \ldots, x_{|X|}\}$. We assume Σ contains a special symbol δ of arity 1, that allows to access a memory slot given its address. Note that a variable is just a symbolic name for an address. Thus, $I(\delta)$ represents the content of the memory that varies as the program runs. Since we focus on the evolution of the content of the memory, all the interpretations considered from now on are equal for the other symbols (i.e. the operational semantics for the symbols of the language is known and fixed). Let $\mu = I(\delta)$ designate a *valuation*, i.e. the state of the memory. μ is seen as a (partial, when not all memory contents are known) function from X into $\mathcal{D}$. Since all other symbols have a fixed interpretation, an interpretation I can be described by simply providing μ. Furthermore, all symbols interpretations must be complete functions (only the valuation is allowed to be a partial function). Partial interpretations can be completed by adding a special element to $\mathcal{D}$ and mapping the undefined domain onto it. It corresponds to an error or an undefined behavior. Note that the interpretations of all symbols must take into account this new special element.

Definition 4.2: Evaluation
Given an interpretation I, an expression $\phi = s(\phi_1, \ldots, \phi_k)$ ($k \geq 0$) *evaluates* or *reduces* to another expression $eval(I, \phi)$ as follows:

$$eval(I,\phi) = \begin{cases} I(s) \in \mathcal{D} & \text{if } s \text{ is a symbol of arity 0} \\ I(s)(eval(I,\phi_1), \ldots, eval(I,\phi_k)) \in \mathcal{D} & \text{if } eval(I,\phi_i) \in \mathcal{D} \text{ for all } i \text{ and } I(s) \text{ is defined at this point} \\ s(eval(I,\phi_1), \ldots, eval(I,\phi_k)) & \text{otherwise.} \end{cases}$$

If $eval(I, \phi) \in \mathcal{D}$, the evaluation is *complete*.

Notation. We will now abusively denote the evaluation $eval(I, \phi)$ where $I(\delta) = \mu$ by $eval(\mu, \phi)$. If ψ is a (possibly nested) sub-expression of ϕ, $\phi[\psi \leftarrow \theta]$ denotes the expression obtained by substituting the expression θ to ψ in ϕ. Given a valuation μ and a subset of addresses $Y \subseteq X$, $\mu_{|Y}$ denotes the restriction of μ to Y. With these notations, we have, for any variable x, any valuation μ where x is defined, and any expression ϕ: $\phi[\delta(x) \leftarrow \mu(x)] = eval(\mu_{|\{x\}}, \phi)$

We now define an equivalence relation on valuations with respect to the evaluation of an expression. In Section 4.2 this equivalence relation is a key notion, allowing efficient evaluation of expressions on sets of valuations.

Definition 4.3:
Given a subset Y of X and an expression ϕ, for all valuations μ, μ' we define the equivalence relation $\sim_\phi^Y$ as follows:

$$\mu \sim_\phi^Y \mu' \Leftrightarrow eval(\mu_{|Y}, \phi) = eval(\mu'_{|Y}, \phi)$$

Valuations $\mu \neq \mu'$ that are equal on Y make a trivial case of this equivalence.

4.1.2 Support of Expressions

The support of an expression is the set of memory addresses necessary to completely evaluate this expression. Conversely, an expression does not depend on an address if its content does not affect its evaluation. We formally define these notions, and then explain how to partially evaluate an expression until dependencies on a given address are eliminated.

Definition 4.4: Potency to a value
Let $\phi \in \texttt{Expr}$ and $d \in \mathcal{D}$. ϕ is said to be potent to d if there exists a valuation μ such that $eval(\mu, \phi) = d$.
We also denote by $Pot(\phi) = \{x | \phi \text{ is potent to } x\}$ the set of potential values of ϕ.

In other words, an expression ϕ is potent to d if it can be evaluated to d.

Definition 4.5: Dependency on an address
An expression ϕ *does not depend* on an address x if and only if:

$$\forall \mu, \mu' \in \mathcal{D}^X, \mu_{|X\setminus\{x\}} = \mu'_{|X\setminus\{x\}} \implies eval(\mu, \phi) = eval(\mu', \phi)$$

The *support* of an expression ϕ is the set of addresses on which ϕ depends.
An expression that depends on no variable is said to be *constant*.

The difference between the dependence on and the potency to an address may seem subtle. The potency expresses the notion of hidden dependency. Assume an expression ϕ with a sub-expression ψ. If ψ is potent to x, then ϕ depends on x, even though x does not appear explicitly in ϕ. This dependence can be rendered explicit if one decides to evaluate the sub-expression ψ.

Lemma 4.1. *If ϕ is an expression that depends on x, then there exist a sub-expression $\delta(\psi)$ of ϕ such that ψ is potent to x.*
ψ is called an x-expression of ϕ.

Proof. We prove the contraposition. Let ϕ be an expression such that for all its sub-expressions of the form $\delta(\psi)$, there is no valuation μ such that $eval(\mu, \psi) = x$. Let now μ and μ' be two valuations that agree on $X \setminus \{x\}$. By structural induction on ϕ, $eval(\mu, \phi)$ (resp. $eval(\mu', \phi)$) does not depend on the value of $\mu(x)$ (resp. $\mu'(x)$). Hence, $eval(\mu, \phi) = eval(\mu', \phi)$ and we conclude that ϕ does not depend on x. □

Lemma 4.2. *If ϕ contains no nested δ operator, then x is not in the support of $\psi = eval(\mu_{|\{x\}}, \phi)$ for all valuations μ and addresses x. The converse is not true.*

Proof. We also prove this lemma by contraposition. Assume there exists a μ such that ψ has an x-expression ψ'. There exists an x-expression ϕ' of ϕ such that $eval(\mu_{|\{x\}}, \phi') = \psi'$. If ψ' were constant, then, according to Definition 4.2, ψ' would be in $\mathcal{D}$, and since it is an x-expression, ψ' would necessarily be equal to x. Thus, according to Definition 4.2, $\delta(\phi')$ would be replaced by $\mu(\psi') = \mu(x)$ in ψ, so that ψ' would not be a sub-expression of ψ. This is contradictory, and proves that ψ' is not constant. ψ' thus depends on at least an address $y \in X$, and, according to Lemma 4.1, contains an occurrence of δ. It implies that ϕ' also contains an occurrence of δ, showing that ϕ contains nested δ operators.

Let $+$ denote any binary symbol in Σ. If $\phi = \delta(\delta(x) + \delta(y))$ and $\mu(x) + \mu(y) = x$, then $\delta(\mu(x) + \delta(y))$ still depends on x. This counter-example to the converse implication can be extended to a symbol of any arity $n \geq 2$, in case Σ contains no binary symbol. □

Lemma 4.2 states that substituting the content of an address x in ϕ may not completely remove the dependence on x.

4.1.3 Examples of Expressions

To help in visualizing these definitions, let us use as an example a language supporting a C-like syntax. We give concrete examples here for each element defined abstractly above. We consider a language supporting integers and their manipulation operators (arithmetic `+`, `-`, `*` ... as well as bitwise operations `<<`,`>>`,...). The set of considered operators are part of the signature Σ. The domain $\mathcal{D}$ is thus integers. The Σ-expressions are built by syntactic combinations of operators, and the literals 0 or 1 are also (terminal) expressions (as $\mathcal{D}$ is embedded in Σ).

Then, by Definition 4.1, we must provide an interpretation function I that gives the semantics of all the operators which are used in expressions. The interpretation function works with constants; for our example we should provide the integer output value for each of the binary operators given two integers.

Consider now variables of the program "a,b,c". They are seen as symbolic names and mapped to integers (memory addresses), for instance $0, 1, 2$. The special operator δ allows to read the value of such a variable, hence the expression a is interpreted as $\delta(0)$. We add the notion of array of fixed size *tab*, and access to a cell of an array using *tab*[]. Again *tab* is a symbolic name for a variable mapped to an integer, for instance 3 that is the first memory slot occupied by the array. Then *tab*[e] where e is an arbitrary expression is syntactic sugar for $\delta(3 + e)$.

All operators should have complete interpretations: a/b must also be defined when $b = 0$. For this purpose, one or more special constants can be introduced. For a given language manipulating finite types, the definition of the interpretation of most symbols is usually straightforward. We consider that the interpretation of all symbols except δ is fixed throughout the computations. In other words we distinguish the code (all other symbols from the signature) from the data, represented by $I(\delta)$, that may vary as the computation progresses.

Definition 4.2 formalizes partial evaluation of expressions given an interpretation function. For instance, suppose μ only gives the content of memory slot 0, say $\mu(0) = 12$. Let $\phi = add(\delta(0), \delta(1))$ (usually noted $a + b$). Then $eval(\mu, \phi) = add(eval(\mu, \delta(0)), eval(\mu, \delta(1)))$. We have $eval(\mu, \delta(0)) = I(\delta)(0) = \mu(0) = 12$. However, because μ is not defined for address 1, $eval(\mu, \delta(1)) = \delta(1)$. Hence, $eval(\mu, \phi) = add(12, \delta(1))$ (noted $12 + b$).

As an example for Definition 4.3, any two μ, μ' such that $eval(\mu, a+b) = eval(\mu', a+b)$ are equivalent. For instance, if both a and b are in Y, $\mu = (a \leftarrow 0, b \leftarrow 1), \mu' = (a \leftarrow 1, b \leftarrow 0)$ are equivalent. If only a is in Y, μ and μ' are not equivalent, since one yields expression $0 + b$ while the other yields $1 + b$.

4.2 Evaluating Expressions on DDD

In practice, a system state is a valuation of the state variables, *i.e.* a string in $\mathcal{D}^X$, and the behavior of the system is described with expressions. Treating such a system using DDD raises the need to evaluate an expression over a *set* of valuations.

More precisely, given an expression ϕ and a set of valuations V, one needs to compute all the evaluations of ϕ by the valuations in V. To achieve this goal efficiently, we rely on the equivalence relation $\sim_\phi^X$ of Definition 4.3.

Recall that the size of a DDD is often logarithmic in the size of the represented set. The naive approach considers each valuation separately, ending up with a complexity linear in the size of the input set. An efficient solution to this problem should use functions that manipulate the nodes of the data structure representation, so that thanks to caches, the complexity remains proportional to the encoding size.

We propose an algorithm, *EquivSplit*, that partitions a set of valuations (given as a DDD) into equivalence classes with respect to $\sim_\phi^X$. It visits variables in the order given by the DDD, and progressively evaluates the expression. Hence it must work with partial valuations and partially evaluated expressions. The manipulation of equivalence classes allows to always represent sets of states, so that every operation of the algorithm can be implemented symbolically.

We have defined in Section 4.1.2 the notion of dependency on an address. Lemma 4.2 indicates that, if there are no nested δ operators, then the substitution of an address x during a partial evaluation completely removes dependency on x. We use this result to present our algorithm *EquivSplit* in this restricted case to help comprehension in Section 4.2.1. It is then extended to the general case, by introducing another function *SolveSub* in Section 4.2.2. The correction and complexity of these functions are discusses in Section 4.2.4.

4.2.1 Without Nested δ Operators

We first present *EquivSplit* in the case where no nested δ operators occur in expressions. The algorithm *EquivSplit* is shown in Algorithm 4. It builds equivalence classes for $\sim_\phi^X$ dynamically based on successive substitution, refinement and merge steps on a partition of the input set. At step i:

- the substitution step uses the partition according to all possible contents of the current address x_i (directly provided by the DDD encoding of valuations), to evaluate ϕ with each of these values;
- the refinement step refines the partition by recursively evaluating the reduced expressions over addresses $x_{i+1}, \ldots, x_{|X|}$;
- the merge step merges cells of the partition that lead to the same reduced expression over addresses $x_i, \ldots, x_{|X|}$.

At each step i, the goal becomes to remove any dependencies on x_i from the expression ϕ, allowing recursion over $x_{i+1}, \ldots, x_{|X|}$.

From a programming language point of view, forbidding nested δ operators means that all addresses are known at compile time, and that no arithmetic on pointers occurs. By Lemma 4.2, this restriction implies that if ϕ is an expression, x an address and μ a valuation, once ϕ is reduced with $\mu(x)$, it no longer depends on x.

Algorithm 4: *EquivSplit*(ϕ,V,i)

Input: ϕ an expression that does not depend on $x_1, \ldots, x_{i-1}$
Input: V a finite set of valuations
Input: i an integer between 1 and $|X| + 1$
Output: a set of pairs $\{(\phi_1, c_1), \ldots, (\phi_n, c_n)\}$ such that $c_1, \ldots, c_n$ are the equivalence classes of $\sim_\phi^{\{x_i, \ldots x_n\}}$ over V, and for each $1 \leq j \leq n$, $\phi_j = eval(\mu_{|\{x_i, \ldots x_n\}}, \phi)$ for any $\mu \in c_j$.

```
1  if φ is constant then
2      return {(V, φ)}
3  else
4      map < Expr, 2^V > res
5      let α_d = {μ ∈ V | μ(x_i) = d} for d ∈ D
6      foreach α_d ≠ ∅ do
           // Substitution
7          ψ = φ[δ(x_i) ← d]
           // Refinement
8          for (ψ', c') ∈ EquivSplit(ψ, c, i + 1) do
               // Merge
9              res[ψ'] = res[ψ'] ∪ c'
10 return res
```

The base case of the recursion is when ϕ is constant, hence $\sim_\phi^{Y_i}$ has a single equivalence class V (lines 1–2). If $i = |X| + 1$, by the precondition on the input ϕ, ϕ is constant.

The sets $(\alpha_d)_{d \in D}$ partition V into equivalence classes with respect to the value d of x_i (line 5). Note that the symbolic encoding of valuations as DDD naturally provides this partition. To each class α_d, we associate a reduced expression ψ by replacing in ϕ variable x_i by its value d (line 7). Under our simplifying assumption, ψ no longer depends on x_i.

The loop on line 8 refines the partition element α_d (c in the general case) by recursively evaluating ψ on subsequent addresses $(x_{i+1}, \ldots, x_{|X|})$. Since elements from different α_d's may yield the same final value for ϕ, line 9 merges them into the final partition into equivalence classes for $\sim_\phi^{\{x_i, \ldots, x_{|X|}\}}$. Invoking *EquivSplit*$(\phi, V, 0)$ returns the equivalence classes of elements in V with respect to $\sim_\phi^X$.

4.2.2 With Nested δ Operators

We now extend our algorithm to the general case. The precondition on the input ϕ for Algorithm 5 is that ϕ does not depend on $x_1, \ldots, x_{i-1}$. Hence, recursion on line 9 requires that ψ does not depend on $x_1, \ldots, x_i$. The algorithm *SolveSub* (Algorithm 6) addresses this problem by reducing x_i-expressions in θ by looking ahead the values of subsequent addresses $x_{i+1}, \ldots, x_{|X|}$ (line 8). Lemma 4.2 shows that with no nested δ, looking zero addresses ahead suffices to eliminate the dependencies, and falls back to the case of Section 4.2.1. The algorithm *SolveSub* performs this reduction using the look-ahead, and returns a set of pairs $\{(\phi_1, c_1), \ldots, (\phi_n, c_n)\}$ such that

c_j are sets of valuations that agree on a look-ahead reduction ϕ_j of θ and that do not depend on x_i. Besides the call to *SolveSub* on line 8, Algorithm 5 does not differ from Algorithm 4.

Algorithm 5: *EquivSplit*(ϕ,V,i)

Input: ϕ an expression that does not depend on $x_1, \ldots, x_{i-1}$
Input: V a finite set of valuations
Input: i an integer between 1 and $|X| + 1$
Output: a set of pairs $\{(\phi_1, c_1), \ldots, (\phi_n, c_n)\}$ such that $c_1, \ldots, c_n$ are the equivalence classes of $\sim_{\phi}^{\{x_i, \ldots x_n\}}$ over V, and for each $1 \leq j \leq n$, $\phi_j = eval(\mu_{|\{x_i, \ldots x_n\}}, \phi)$ for any $\mu \in c_j$.

```
1  if φ is constant then
2  |  return {(V, φ)}
3  else
4  |  map < Expr, 2^V > res
5  |  let α_d = {μ ∈ V | μ(x_i) = d} for d ∈ D
6  |  foreach α_d ≠ ∅ do
   |  |  // Substitution
7  |  |  θ = φ[δ(x_i) ← d]
   |  |  // to remove nested δ operators
8  |  |  for (ψ, c) ∈ SolveSub(θ, α_d, i) do
   |  |  |  // Refinement
9  |  |  |  for (ψ', c') ∈ EquivSplit(ψ, c, i + 1) do
   |  |  |  |  // Merge
10 |  |  |  |  res[ψ'] = res[ψ'] ∪ c'
11 return res
```

SolveSub, presented in Algorithm 6, computes in res a partition of V, and associates to each cell a simplified expression obtained by partially resolving ϕ, until all dependencies on x_i are removed. tmp is initialized as a single cell associated to ϕ (line 3). At each step of the while loop, an element (ψ, c) of tmp is treated (line 5). If the current expression ψ does not depend on x_i, the pair is moved to res (line 13). Otherwise, we let θ be an x_i-expression of ψ (line 7). Recall, by Lemma 4.1, that such a θ exists and has less nested δ operators than ψ. Any x-expression can be chosen and will lead to a correct result, hence the algorithm has some latitude at this point. Heuristically, to favor merging of partially resolved expressions, it is desirable to first treat x-expressions with a small co-domain (e.g. solve boolean sub-expressions first). Note that this is only possible with some additional knowledge of the signature's interpretation.

Recursion by invoking *EquivSplit* with θ (line 8) refines the cell c according to the value of θ. To each of these refined cells is associated the reduction of ψ obtained by substituting θ by its value (line 9). They are then added to tmp that merges the cells according to the reduced expression ψ' (line 11).

4.2.3 Proofs of Correctness

To prove the correctness of our algorithms, we need some additional definitions and notations.

For $1 \leq i \leq |X| + 1$, let $Y_i = \{x_i, \ldots, x_{|X|}\}$.

We define an order over the expressions for induction on expressions in the proof.

Algorithm 6: *SolveSub*(ϕ,V,i)

Input: ϕ an expression that does not depend on $x_1, \ldots, x_{i-1}$
Input: V a set of valuations that all agree on the value d of x_i
Input: i an integer between 1 and $|X|$
Output: a set of pairs $\{(\phi_1, c_1), \ldots, (\phi_n, c_n)\}$ such that $c_1, \ldots, c_n$ is a partition of V, and for each $1 \leq j \leq n$, ϕ_j is a reduced expression obtained by removing all dependencies on x_i from ϕ, and all valuations in c_j agree on this reduction ϕ_j

```
1  map < Expr, 2^V > res
2  map < Expr, 2^V > tmp
3  tmp[φ] = V
4  while tmp is not empty do
5      (ψ, c) = tmp.pop()
6      if ψ has an x_i-expression then
7          θ = an x_i-expression of ψ
8          for (θ', c') ∈ EquivSplit(θ, c, i) do
9              ψ' = ψ[θ ← θ']
10             ψ' = ψ'[δ(x_i) ← d]
11             tmp[ψ'] = tmp[ψ'] ∪ c'
12     else
           // ψ does not depend on x_i
13         res[ψ] = res[ψ] ∪ c
14 return res
```

Definition 4.6: Order on the Expressions
Let ϕ and ψ be two expressions. $\phi < \psi$ if and only if ϕ has less symbols than ψ.

We define a simple equivalence relation on valuations, that relates valuations if they have the same value on a given address. This is just a formal notation that will help conciseness.

Definition 4.7:
Let $x \in X$. We define the equivalence relation $\wr_x$ over valuations as:

$$\mu \wr_x \mu' \Leftrightarrow \mu(x) = \mu'(x)$$

We now define a rewriting relation on expressions.

Definition 4.8:
Let μ be a valuation, x be an address, ϕ and ϕ' two expressions.
We define $\phi \rightsquigarrow_{x,\mu} \phi'$ if and only if one of the following conditions holds:

- $\phi' = \phi[\psi \leftarrow eval(\mu, \psi)]$, where ψ is an x-expression of ϕ whose support is not empty;
- ϕ contains at least one constant x-expression and $\phi' = eval(\mu_{|\{x\}}, \phi)$.

Informally, $\phi \rightsquigarrow_{x,\mu} \phi'$ if ϕ' is a partial evaluation of ϕ using μ. In particular, ϕ' has less dependencies on x than ϕ.

$\rightsquigarrow^*_{x,\mu}$ denotes the reflexive and transitive closure of $\rightsquigarrow_{x,\mu}$. It is easy to see that if $\phi \rightsquigarrow_{x,\mu} \phi'$, then the number of occurrences of δ in ϕ' is strictly less than in ϕ. Therefore, there is no infinite sequence $\phi \rightsquigarrow_{x,\mu} \phi_1 \rightsquigarrow_{x,\mu} \dots$, and thus, for any expression ϕ, there exists an expression ϕ' that does not depend on x such that $\phi \rightsquigarrow^*_{x,\mu} \phi'$.

We now choose a deterministic criterion for choosing the x-expression ψ in the first case of Definition 4.8. For example, ψ is the first x-expression encountered during a depth-first walk of the syntactic tree of ϕ. This ensures the uniqueness of the above ϕ': for any expression ϕ, there exists a unique expression, denoted by $norm(\mu, x, \phi)$ that does not depend on x and such that $\phi \rightsquigarrow^*_{x,\mu} norm(\mu, x, \phi)$. $norm(\mu, x, \phi)$ is computable by repeatedly evaluating the x-expressions of ϕ chosen according to this criterion.

Definition 4.9:
Given an expression ϕ, we define the equivalence relation $\equiv^x_\phi$ over valuations as follows:

$$\mu \equiv^x_\phi \mu' \Leftrightarrow norm(\phi, x, \mu) = norm(\phi, x, \mu')$$

Note that the choice of the criterion to pick an x-expression of ϕ influences the structure of the equivalence classes of $\equiv^x_\phi$. This does not affect correctness of the algorithms, but may impact their complexities, as already mentioned in Section 4.2.2.

Lemma 4.3. *Let ϕ be an expression, μ be a valuation, and x be an address.*

$$eval(\mu, norm(\mu, x, \phi)) = eval(\mu, \phi)$$

Proof. Let us consider the relation $\rightsquigarrow_\mu = \cup_{y \in X} \rightsquigarrow_{y,\mu}$. If $\phi \rightsquigarrow_\mu \phi'$, then the number of occurrences of δ in ϕ' is strictly less than in ϕ. Therefore, there is no infinite sequence $\phi \rightsquigarrow_\mu \phi_1 \rightsquigarrow_\mu \dots$, and there exists an expression ϕ' that depends on no variable and such that $\phi \rightsquigarrow^*_\mu \phi'$. It is easy to see that $\phi' = eval(\mu, \phi)$. Therefore, since $eval(\mu, \phi)$ is well-defined, ϕ' is unique, and does not depend on the path leading from ϕ to ϕ'.

Similarly, $\phi'' = eval(\mu, norm(\mu, x, \phi))$ depends on no variable and $norm(\mu, x, \phi) \rightsquigarrow^*_\mu \phi''$. Since $\rightsquigarrow_{x,\mu} \subseteq \rightsquigarrow_\mu$, we have $\phi \rightsquigarrow^*_\mu norm(\mu, x, \phi) \rightsquigarrow^*_\mu \phi''$. By unicity of ϕ', we conclude that $\phi' = \phi''$, hence $eval(\mu, norm(\mu, x, \phi)) = eval(\mu, \phi)$. □

Correctness of the algorithms. We prove the correctness of both *EquivSplit* and *SolveSub*, using a double induction on parameters i and ϕ.

EquivSplit(ϕ, V, i) returns a set of couples $\{(\phi_1, c_1), \dots, (\phi_n, c_n)\}$ such that $c_1, \dots, c_n$ are the equivalence classes of $\sim^{Y_i}_\phi$ on V, and $\phi_j = eval(\mu, \phi)$ for any $\mu \in c_j$, provided that ϕ does not depend on $x_1, \dots, x_{i-1}$.

SolveSub(ϕ, V, i) returns a set of couples $\{(\phi_1, c_1), \dots, (\phi_n, c_n)\}$ such that $c_1, \dots, c_n$ are the equivalence classes of $\equiv^{x_i}_\phi$ on V, and $\phi_j = norm(\mu, x_i, \phi)$ for any $\mu \in c_j$, provided that ϕ does not depend on $x_1, \dots, x_{i-1}$. Note that all elements in V must agree on the value of x_i: V must be contained in an equivalence class of $\wr_{x_i}$.

Induction on i. If $i = |X| + 1$, then any expression ϕ that does not depend on $x_1, \dots, x_{|X|}$ is constant. Then both $\sim^{Y_i}_\phi$ and $\equiv^{x_i}_\phi$ are trivial. Furthermore, $eval(\mu, \phi) = \phi$ and $norm(\mu, x_i, \phi) = \phi$ for all valuations μ. *EquivSplit*(ϕ, V, i) and *SolveSub*(ϕ, V, i) both return the singleton $\{(\phi, V)\}$, ensuring their correctness.

Let $1 \leq i \leq |X|$. Let us assume that both *EquivSplit*$(\psi, V, i+1)$ and *SolveSub*$(\psi, V, i+1)$ are correct for any set V and expression ψ. We prove, by structural induction on ϕ, that *EquivSplit*(ϕ, V, i) and *SolveSub*(ϕ, V, i) are correct.

Induction on ϕ. Let ϕ be an expression. If ϕ is a single symbol, it is a constant expression, and the conclusion is the same as above for the base case of the induction on i.

Let us now suppose that ϕ is not reduced to a single symbol, and that (induction hypothesis) *EquivSplit*(ψ, V, i) is correct for all V and $\psi < \phi$, and *SolveSub*(ψ, V, i) is correct for all V and $\psi < \phi$. We first prove that *SolveSub*(ϕ, V, i) is correct, then that *EquivSplit*(ϕ, V, i) is correct.

SolveSub is based on a while loop, that has three invariants, that we prove:

$$\forall (\psi, c) \in tmp, \forall \mu \in c, \phi \rightsquigarrow^*_{x_i,\mu} \psi \tag{4.1}$$

$$\forall (\psi, c) \in res, \forall \mu \in c, \psi = norm(\mu, x_i, \phi) \tag{4.2}$$

$$\text{all the sets stored in } tmp \text{ and } res \text{ form a partition of } V \tag{4.3}$$

These conditions are obviously true when entering the loop for the first time (initializations of lines $1-3$). When entering the loop, a couple (ψ, c) is first removed from tmp (line 5). By (4.1), $\phi \rightsquigarrow^*_{x_i,\mu} \psi$ for all $\mu \in c$. If ψ has no x-expression, then ψ is in normal form and $\psi = norm(\mu, x_i, \phi)$ for all $\mu \in c$. Thus, adding (ψ, c) to res preserves all three invariants (lines $12-13$).

Otherwise, the algorithm picks an x_i-expression θ in ψ (line 7). θ is a proper sub-expression of ψ, and, since ψ does not depend on $x_1, \ldots, x_{i-1}$, neither does θ. Thus, by induction hypothesis, *EquivSplit*(θ, c, i) is correct. Therefore, on line 8, c' is an equivalence class for $\sim^{Y_i}_\theta$ on c and $\theta' = eval(\mu_{|Y_i}, \theta)$ for any $\mu \in c'$. Thus, $\psi' = \psi[\theta \leftarrow eval(\mu_{|Y_i}, \theta)]$ for $\mu \in c'$ is well-defined on c' (line 9). Furthermore, since all valuations in V agree on x_i, then $\psi' = eval(\mu_{|\{x_i\}}, \psi[\theta \leftarrow eval(\mu_{|Y_i}, \theta)])$ is also well-defined on c' (line 10). Furthermore, $\psi \rightsquigarrow_{x_i,\mu} \psi'$ or $\psi \rightsquigarrow^2_{x_i,\mu} \psi'$, and thus $\phi \rightsquigarrow^*_{x_i,\mu} \psi'$ for all $\mu \in c'$. Since the equivalence classes of $\sim^X_\theta$ partition c, adding such pairs (ψ', c') to tmp preserves the three invariants. Since $\psi \rightsquigarrow^+_{x_i,\mu} \psi'$, ψ' is strictly closer to any of its normal forms than ψ. This ensures termination of the loop.

Finally, the loop ends when tmp is empty. Thus, by (4.3), the sets stores in res partition V, and using (4.2), we conclude that res contains exactly the equivalence classes of $\equiv^{x_i}_\phi$ on V.

We now prove that *EquivSplit*(ϕ, v, i) is correct. It is based on three nested loops. The innermost loop has two invariants, that we prove below:

$$\forall (\psi, c) \in res, \forall \mu \in c, \phi = eval(\mu_{|Y_i}, \psi) \tag{4.4}$$

$$\forall (\psi_1, c_1), (\psi_2, c_2) \in res, \phi_1 = \phi_2 \Rightarrow c_1 = c_2 \tag{4.5}$$

These invariants are obviously true before entering the loop on line 6. The α_d's defined on line 5 are the equivalence classes of $\wr_{x_i}$ on V. Thus $\theta = eval(\mu_{|x_i}, \phi)$ (line 7) is well-defined on α_d. Furthermore, $\phi \rightsquigarrow^*_{x_i,\mu} \theta$ for any $\mu \in \alpha_d$. θ is either an expression smaller than ϕ, or ϕ. In both cases, by induction hypothesis or by the proof of correctness of *SolveSub*(ϕ, V, i) above, *SolveSub*(θ, V, i) is correct.

On line 8, c is an equivalence class for $\equiv^{x_i}_\theta$ on α_d and $\psi = norm(\mu, x_i, \theta) = norm(\mu\ , x_i, \phi)$ (since $\phi \rightsquigarrow^*_{x_i,\mu} \theta$) for any $\mu \in c$. Thus ψ does not depend on x_i, and, by induction hypothesis, *EquivSplit*$(\psi, c, i+1)$ is correct. Therefore, on line 9, c' is an equivalence class for $\sim^{Y_{i+1}}_\psi$ on c,

and $\psi' = eval(\mu_{|Y_{i+1}}, \psi)$ for any $\mu \in c'$. Let $\mu \in c'$.

$$\begin{aligned}
\psi' &= eval(\mu_{|Y_{i+1}}, \psi) \\
&= eval(\mu_{|Y_{i+1}}, norm(\mu, x_i, \phi)) \\
&= eval(\mu_{|Y_i}, norm(\mu, x_i, \phi)) && \text{because } norm(\mu, x_i, \phi) \text{ does not depend on } x_i \\
&= eval(\mu_{|Y_i}, \phi) && \text{by Lemma 4.3}
\end{aligned}$$

When inserted in res (line 10), c' is merged to a $(\psi'', c'') \in res$ if, and only if, $\psi'' = \psi'$. By (4.4), $\psi'' = eval(\mu_{|Y_i}, \phi)$ for $\mu \in c''$, so that this insertion in res preserves both invariants.

Recall that α_d (line 6) is an equivalence class for wr_{x_i}. c being an equivalence class for $\equiv_\theta^{x_i}$ on α_d, it is an equivalence class for $\wr_{x_i} \cap \equiv_\theta^{x_i}$ on V. Similarly, c' is an equivalence class for $\sim_\psi^{Y_{i+1}}$ on c, hence an equivalence class of $\wr_{x_i} \cap \equiv_\theta^{x_i} \cap \sim_\psi^{Y_{i+1}}$ on V. The algorithm reaches line 11 when all these equivalence classes of $\wr_{x_i} \cap \equiv_\theta^{x_i} \cap \sim_\psi^{Y_{i+1}}$ on V have been treated and inserted into res. Thus, the union of the sets stored in res is exactly V. By (4.4), they are pairwise disjoint, hence form a partition of V. By (4.4) and (4.5), we conclude that they are exactly the equivalence classes of $\sim_\phi^{Y_i}$, proving the correct result of *EquivSplit*(ϕ, V, i).

4.2.4 Discussion on Complexity

Complexity of *EquivSplit*. In Algorithm 5, the α_d's for the loop on line 6 are already provided by the DDD representation of valuations, so that this loop is just a walk of already computed sets. The main source of complexity in this function lies in the call of *SolveSub*. In the case when ϕ has no nested δ operators, then the loop on line 8 has a single pass. The recursion on line 9 explores the subsequent part of the DDD, so that, using a memoization table, the total complexity of *EquivSplit* is related to the size of the input DDD, rather than to the size of V.

Complexity of *SolveSub*. The look-ahead of *SolveSub*, performed on line 8 of Algorithm 6, refines the α_d in input. This refinement (that builds new decision diagrams) can be arbitrarily fine, and depends on the input expression and the input set of valuations. The overall complexity of *SolveSub* is thus hard to predict and depends on the number of equivalence classes built.

A worst case for our technique would be an expression computing a hash value based on the values in all the memory slots. A perfect hash function would yield equivalence classes limited to singletons, hence encountering exponential worst case complexity (linear in the number of states represented by the DDD). Conversely, expressions with a small codomain (such as boolean expressions) give a small bound on the maximum number of equivalence classes manipulated by the algorithm. A peak effect for symbolic techniques occurs when an intermediate DDD size is proportional to its set size. This may occur anytime a partition element is built, hence finer partitions are more likely to induce a peak effect.

Memoization. A memoization table for *EquivSplit* is built by associating to each ⟨DDD, expression⟩ pair the set {⟨DDD, expression value⟩} that partitions the input DDD into equivalence classes for the input expression. The full evaluation of various statements may thus share the stored results allowing computation of common sub-expressions. Because it contains partial evaluations results, and no specific attempt is made to reconcile combined results, the structure of this cache differs from a decision diagram representing the full effects of transitions, although it allows to reconstruct the same transition information.

Variable Order. Much of the complexity for both of these algorithms depends on the variable ordering used in the DDD encoding. The equivalence classes depend on the order in which x_i's are visited. The representation size of the equivalence classes also strongly depends on this order. Heuristically, orderings that minimize invocations to *SolveSub* reduce the complexity. Limiting the depth of the look-ahead mechanism also helps to build DDD that share existing suffixes.

In our experiments, we adapted the FORCE algorithm [AMS03]. Given a directed hypergraph where weighted edges represent constraints on variables (nodes of the hypergraph), FORCE heuristically computes an ordering on variables that minimizes the total weight. Expressions induce constraints on the variables in their support. By assigning a strong weight to constraints implying invocations to *SolveSub*, and small weight to constraints enforcing locality, we obtained satisfactory results.

Algebraic simplification. Notice that the algebraic structure of $\mathcal{D}$ can be used to further simplify non-constant expressions. In a common and useful case, Σ contains a binary symbol f such that $I(f) : \mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$ has an absorbing element $a \in \mathcal{D}$: for all $x \in \mathcal{D}$, $I(f)(a, x) = a$. In this case, any expression of the form $f(a, \phi)$ can be simplified directly to a with no need to evaluate the sub-expression ϕ. This naturally extends to symbols of arbitrary arity. Examples include the arithmetic multiplication, for which 0 is absorbing, or a conditional expression $\texttt{IfThenElse}(c, \phi, \psi)$, that evaluates either to ϕ or ψ depending on the value of c. Once the sub-expression c is resolved to a constant, only one of ϕ or ψ needs to be evaluated. As shown by this last example, the algebraic structure of D may also help to choose sub-expressions on line 7 of Algorithm 6.

4.3 Application to Assignment Evaluation

We now present how to use our new algorithms to handle assignments of expressions to memory slots.

4.3.1 Evaluating Assignments

The internal state of a software system is made of memory slots. The transition relation is thus described in terms of assignments (especially in imperative programming languages). An assignment is a pair of expressions $\phi \leftarrow \psi$, where ϕ denotes the adress of the affected memory slot and ψ the new value to assign. Allowing ϕ to depend on current memory allows to model dynamic assignments such as `t[i] := 0`.

In our DDD implementation, an assignment is encoded as a homomorphism $\mathbb{D} \mapsto \mathbb{D}$. It evaluates both ϕ and ψ by visiting the input DDD. As variables are encountered, ϕ and ψ are partially evaluated. If dependencies on the current variable are not eliminated (nested δ), *SolveSub* is invoked. At some point, ϕ is reduced to a constant, which is the target of the assignment. When this target is reached, ψ must then be evaluated to a constant which may involve a look ahead using *SolveSub*.

The evaluation of an assignment is described in Algorithm 7. It takes as input an assignment $\phi \leftarrow \psi$, a set of states V and an integer i such that ϕ and ψ do not depend on $x_1, \ldots, x_{i-1}$ and ϕ is potent to neither $x_1, \ldots,$ nor x_{i-1}. This ensures that the assignment does not depend on, nor affect variables $x_1, \ldots x_{i-1}$.

There are three cases to consider:

- ϕ is constant and equals to x_i. In this case, the value of ψ is evaluated on V with *EquivSplit* (if necessary), and x_i is set to the resulting value on each equivalence class.

- ϕ is potent to x_i. The value of ϕ is evaluated on V with *EquivSplit*. On each resulting equivalence class, ϕ is constant and a recursive call to *EvalAssign* will eventually fall in the first case.

- ϕ is not potent to x_i. Then V is refined by eliminating x_i-dependencies from both ϕ and ψ (if necessary), before a recursive call to *EvalAssign* with parameter $i+1$ is done. Note that the order in which ϕ and ψ are considered for the elimination of x_i-dependencies does not matter, thus leaving room for some optimisation.

Optimisations and shortcuts are possible at several points of this algorithm. For instance, if ϕ is not potent to x_i, and ϕ and ψ do not depend on x_i, then the recursive call on line 12 can be done immediately. If ϕ is potent to x_i, the complete evaluation of ϕ is actually not necessary: a partial evaluation sufficient to decide that ϕ is actually not potent to x_i suffices to start considering variable x_{i+1}.

Algorithm 7: EvalAssign

Input: an assignment $\phi := \psi$ where $\phi, \psi \in$ `Expr` such that ϕ and ψ do not depend on x_1, $\ldots$, x_{i-1}, and ϕ is neither potent to x_1, nor x_2, $\ldots$, nor x_{i-1}.
Input: V a finite set of valuations
Output: $W = \{\mu' | \exists \mu \in V, \mu \mapsto_{\phi:=\psi} \mu'\}$

```
1  W ← ∅
2  for α_d do
3      if φ = x_i then
4          for (v, β) ∈ EquivSplit(ψ, α_d, i) do
5              W ← W ∪ [x_i := v](β)
6      else if φ is potent to x_i then
7          for (x, β) ∈ EquivSplit(φ, α_d, i) do
8              W ← W ∪ EvalAssign(x := ψ, β, i)
9      else
           // φ is not potent to x_i
10         for (φ', β) ∈ SolveSub(φ, α_d, i) do
11             for (ψ', γ) ∈ SolveSub(ψ, β, i) do
12                 W ← W ∪ EvalAssign(φ' := ψ', γ, i + 1)
13 return W
```

4.3.2 Example of Assignment Evaluation

Let us illustrate the above mechanisms with a simple example. Consider a system with three state variables that are stored in an array `t`. Like in C, `t` is actually a symbolic name for the address of the first cell, the second cell being at `t + 1` ... Let us now consider the assignment `t[t[0]] := t[1] + t[2]`, to be applied on the set of states $V = \{(0,0,2),(0,0,1),(0,1,1),(1,1,1)\}$. The steps of the algorithm are shown on Figure 4.1, although the sets are not represented with their symbolic representations to ease comprehension.

The algorithm first evaluates which address is to be affected, and resolves the left-hand-side `t[t[0]]`. It yields the address `t[0]` on the subset $V_0 = \{(0,0,2),(0,0,1),(0,1,1)\}$ and `t[1]` on

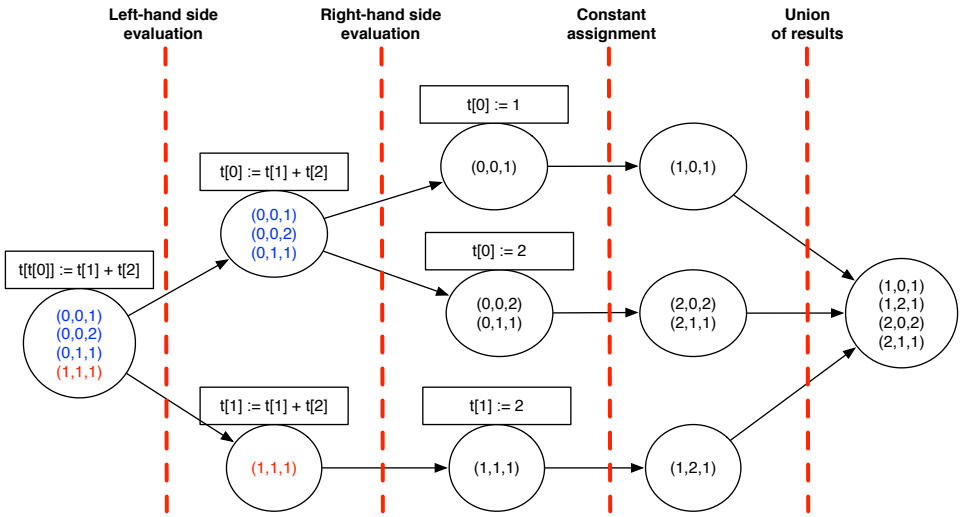


Figure 4.1: Example of the use of *EquivSplit* to evaluate an assignment

$V_1 = \{(1,1,1)\}$. V_0 and V_1 partition V. Now that this partial evaluation is done, the recursive steps consist in applying `t[0] := t[1] + t[2]` on V_0 and `t[1] := t[1] + t[2]` on V_1.

In both cases, the algorithm then evaluates the right-hand side `t[1] + t[2]`. V_0 is thus partitioned in $V_{0,1} = \{(0,0,1)\}$, on which `t[1] + t[2]` is 1, and $V_{0,2} = \{(0,0,2),(0,1,1)\}$ on which `t[1] + t[2]` is 2. On the singleton V_1, `t[1] + t[2]` is always 2.

At this point, we have partitioned the original V into three sets: we now have to apply `t[0] := 1` on $V_{0,1}$, `t[0] := 2` on $V_{0,2}$ and `t[1] := 2` on V_1. These basic assignments (of a given value to a given variable) are basic operations on DD. We thus obtain as results of this three operations the sets $V'_{0,1} = \{(1,0,1)\}$, $V'_{0,2} = \{(2,0,2),(2,1,1)\}$ and $V'_1 = \{(1,2,1)\}$. The final result of the operation is the union of these three sets $V' = \{(1,0,1),(1,2,1),(2,0,2),(2,1,1)\}$.

4.3.3 Reverting Assignments

We also consider the use of the *EquivSplit* algorithm for the evaluation of the reverse transition relation. When checking a property on a model, it is usual to also compute a counter-example when the property does not hold. This counter-example can then be used by the modeler to correct its model.

When checking the property, the state space is explored by firing the transition relation forward. When a set of states that invalidate the property is found, the transition relation is fired backwards until an initial state is reached: this yields a path between an initial state and an invalidating state, that is a counter-example for the property.

Since a state may have an infinite number of predecessors by a statement, the inversion is guided by a set of potential predecessors, usually given by the set of reachable states computed during the forward exploration. Formally, given a set V of states, a set P of potential predecessors and an assignment $\phi := \psi$, the problem is to find the maximal subset of $P' \subseteq P$ such that $\phi := \psi$ applied to P' is included in V (some of the states in V may not have any predecessor by $\phi := \psi$).

Algorithm 8 describes this. It relies on an annex function $computeDomain(P, x) = \{\mu(x) | \mu \in P\}$ that computes all the possible values of an address x in a set of states. In each pass of the inner loop, it computes a set V' of states that potentially have a predecessor. From V', it then

computes a set V'' of potential predecessors of V'. The real predecessors are then obtained by intersecting V'' with the set of potential predecessors P.

Algorithm 8: InvertAssign

Input: an assignment $\phi := \psi$
Input: V a finite set of valuations
Input: P a finite set of (potential) valuations
Output: $W = \{\mu \in P | \exists \mu' \in V, \mu' \mapsto_{\phi:=\psi} \mu\}$

```
1 W ← ∅
2 for (φ', α) ∈ EquivSplit(φ, P, 1) do
3     for (ψ', β) ∈ EquivSplit(ψ, α, 1) do
4         V' ← [δ(φ') == ψ'](V)
5         d ← computeDomain(β, φ')
6         V'' ← [φ' ← choice(d)](V')
7         W ← W ∪ (V'' ∩ β)
8 return W
```

Proof. Let us prove the correctness of Algorithm 8. We know that the α explored in the loop on line 2 partitions P, and that the β in the loop on line 3 partitions the current α. Therefore, it suffices to prove that $V'' \cap \beta = \{\mu \in \beta | \exists \mu' \in V, \mu \mapsto_{\phi:=\psi} \mu'\}$ (line 7).

Recall that for all $\mu \in \beta$, $\mu(\phi) = \phi' \in D$ and $\mu(\psi) = \psi' \in D$. Thus, for $\mu \in \beta$, $\mu \mapsto_{\phi:=\psi} \mu'$ if and only if $\mu'[\phi'] = \psi'$ and $\mu'[x] = \mu[x]$ for all $x \neq \phi'$. $V'' \cap \beta$ is the subset of β also in V''.

Let $\mu \in \beta$. $\mu \in V''$ if and only if there exist $\mu' \in V', \mu'' \in \beta$ such that $\mu[\phi'] = \mu''[\phi']$ and $\mu[x] = \mu'[x]$ for all $x \neq \phi'$. Since μ is already in β, $\mu \in V''$ if and only if there exists $\mu' \in V'$ such that $\mu[x] = \mu_1[x]$ for all $x \neq \phi'$. But $\mu' \in V'$ if and only if $\mu' \in V$ and $\mu'[\phi'] = \psi'$ (line 4). Therefore, $\mu \in V''$ if and only if there exists $\mu_1 \in V$ such that $\mu'[\phi'] = \psi'$ and $\mu[x] = \mu'[x]$ for all $x \neq \phi'$. In this case, $\mu \mapsto_{\phi':=\psi'} \mu'$, and since $\mu \in \beta$, $\mu \mapsto_{\phi:=\psi} \mu'$. Finally, $V'' \cap \beta = \{\mu \in \beta | \mu \in V''\} = \{\mu \in \beta | \exists \mu' \in V, \mu \mapsto_{\phi:=\psi} \mu'\}$. □

Although correct, this algorithm might not be very efficient. Notice for instance that the double evaluation of ϕ and ψ can actually be done simultaneously, as in Algorithm 7. More importantly, the set P of potential predecessors is usually given by the states reached during the forward phase, whereas V is a set of invalidating states. V is therefore a subset of P and is often much smaller than P. For instance, V' (line 4) may be empty for a great number of pairs (ϕ', ψ'), thus rendering the computation of these values on lines 1 and 2 useless.

We observe that when $\mu \mapsto_{\phi:=\psi} \mu'$, μ' and μ differ by only one value. We thus suggest to use V rather than P to evaluate ϕ and ψ, as much as possible.

Let us consider an assignment $\phi := \psi$, and a valuation μ. We know that $\mu \mapsto_{\phi:=\psi} \mu'$ if and only if $\mu'[\mu(\phi)] = \mu(\psi)$ and $\mu'[x] = \mu[x]$ for all $x \neq \mu(\phi)$. Thus, if $x \notin Pot(\phi) \cap Supp(\psi)$, then $\mu'[x] = \mu[x]$. We are thus sure that $\mu_{|X-Pot(\phi)\cap Supp(\psi)}(\theta) = \mu'_{|X-Pot(\phi)\cap Supp(\psi)}(\theta)$ for $\theta \in \{\phi, \psi\}$.

We extend *EquivSplit* and *SolveSub* so that they take another set P as input. Whenever the value of a variable x_i is used to reduce ϕ and ψ, these new versions act as the previous ones if ϕ is not potent to x_i, and rely on the α_d's defined over P rather than V otherwise. When both ϕ and ψ are reduced to constants, we go back to the case illustrated by lines $4-7$ in Algorithm 8. This way, we avoid many useless iterations in loops on lines $1-2$.

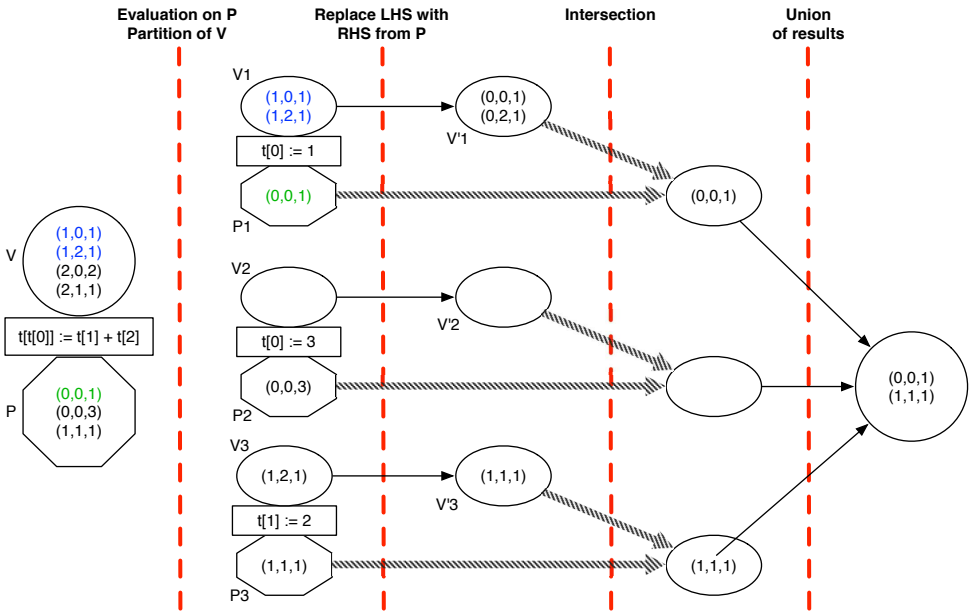


Figure 4.2: Example of an assignment reversion

4.3.4 Example of Assignment Reversion

To illustrate the inversion of assignments, we reuse the example chosen for the evaluation assignment. Let us thus assume that we want to invert the assignment `t[t[0]] := t[1] + t[2]` on the set $V = \{(1,0,1),(1,2,1),(2,0,2),(2,1,1)\}$, knowing that the set of potential states is $P = \{(0,0,1),(0,0,3),(1,1,1)\}$. The run of the algorithm is shown on Figure 4.2.

The algorithm first evaluates both sides of the assignment on P, that yields $P_1 = \{(0,0,1)\}$, with simplified assignment `t[0] := 1`, $P_2 = \{(0,0,3)\}$, with assignment `t[0] := 3`, and $P_3 = \{(1,1,1)\}$ with `t[1] := 2`. To each of these sets, we associate the set of their possible successors in V, *i.e.* the valuations that have the value of the right-hand side at the address given by the left-hand side. For instance, P_1 is associated with $V_1 = \{(1,0,1),(1,2,1)\}$ for which the value of `t[0]` is 1. Similarly, P_2 is paired with $V_2 = \emptyset$ and P_3 with $V_3 = \{(1,2,1)\}$.

We now search in each P_i the possible values for the left-hand side address, and assign them in V_i, obtaining a new set V_i'. V_i' is a thus a set of possible predecessors for V_i, relatively to the information given by P_i. In this case, $V_1' = \{(0,0,1),(0,2,1)\}$: the first component of elements of V_i is replaced by 0, the sole possible value of this first component in P_1. Similarly, we obtain $V_2' = \emptyset$ since V_2 is empty, and $V_3' = \{(1,1,1)\}$. Finally, each V_i' is intersected with the corresponding P_i to obtain the true predecessors of V_i in P_i.

4.4 Application to Symmetries

We now show how to use *EquivSplit* to efficiently swap the values of two variables in a DDD. This problem arose in Chapter 3 for the efficient evaluation of permutations of variables on DDD. The strategy that was adopted is to use the decomposition of a permutation as a product of transpositions. We had supposed then that transpositions evaluation on DD was available,

and we describe in this section how to do it efficiently.

If x and y are two variables that are not adjacent in the DD variable ordering, $swap(x, y)$ has not a good locality. As a DD represents a set, the problem is to carry *efficiently* the values of x to the level y and to carry back the values of y to the level x. As mentioned in Chapter 3, [CEPA+02] gives a homomorphism solution to this problem. This solution uses a homomorphism to carry back the values of the lower variable (say y) to the higher one (say x). However, the evaluation of this homomorphism yields several intermediate DD nodes as a vehicle for the values to be carried, and that are not present in the final result. Such intermediate results that are mostly not used consume space in the caches. *EquivSplit* can be used to evaluate the swap more efficiently.

Suppose that the signature Σ contains a binary symbol $swap$, whose semantics is to swap the content of the two addresses given as arguments. Let x and y be two addresses, and say (since $swap$ is commutative) that x is the higher variable in the DD ordering and y the lower one. We now want to evaluate $swap(x, y)$ using the *EquivSplit* algorithm. We simply define $swap(x, y)[\delta(x) \leftarrow d]$ to be equal to $(x \leftarrow y; y \leftarrow d)$. Using this definition of the substitution, $swap(\phi, \psi)$ can be evaluated as any assignment, using Algorithm 7.

Compared to the previous approach of [CEPA+02], this does not build intermediate nodes to carry out the value of y. Instead, *SolveSub* performs a look-ahead to partition the DD according to the value of y, assigns it to x, then assigns to y the value of x.

Note that the performances presented in the previous chapter rely on an early version of this algorithm to evaluate permutations efficiently. At the time the measures were done, we observed that this early (and not fine-tuned) version yielded better performance that the strategy proposed in [CEPA+02].

4.5 Assessment

The assessment of *EquivSplit* is done in two parts. We expose here its performance when encoding a transition relation. We then demonstrate in Chapter 5 how we have used it for an existing formalism, the Symmetric Nets with Bags. We also present in details the simplifications allowed and the adaptations required by this specific formalism and its performance.

Implementation

To assess our algorithms, we chose to use benchmark models from the BEEM database [Pel07], that are written in the Divine language [BBvR10]. This database was primarily set up as a basis for the comparison between model-checkers. It has already been used for the assessment of various tools, including those we compare to. To this end, we defined an intermediate formalism called Guarded-Action Language (GAL)[1], that can be manipulated symbolically with the algorithms described in this chapter. This formalism defines a system's memory μ using integer variables and fixed size arrays of integers. Its transitions are composed of a guard that is a boolean expression over variables and a sequence of statements that are assignments of expressions to variables (or to cells of an array). A state of a GAL system is defined as the valuation of all variables. A transition is enabled in any state where the guard is true. Firing an enabled transition yields in a single step the successor state obtained by executing the assignments of the transition in an atomic sequence. The semantics are thus globally asynchronous, but sequences of statements are locally synchronous, reflecting the semantics of concurrent systems.

[1] http://move.lip6.fr/software/DDD/gal.php

This small formalism offers a rich signature Σ consisting of all C operators for manipulation of the `int` data type and of arrays (including nested array expressions). There is no explicit support for pointers, though they can be simulated with an array *heap* and indexes into it. It also supports full C-like boolean expressions.

With such features, translation of Divine models into GAL was relatively straightforward. This technical work was done by adapting the code of `LTSmin`'s wrapper for Divine models, where the semantic bridge to a system based on integer variables already existed. Divine is a language for describing processes that communicate through bounded channels, shared variables and/or synchronizations. Channels are modeled using arrays. Synchronizations use a conjunction of local condition as a guard, and a sequence of local effects on each process as action. Priorities (deriving from the "commit" semantics of Divine) are enforced by adding the negation of the disjunction of the guards of higher priority to guards of transitions with lower priority.

Our implementation includes various such optimisations, so as to increase locality of operations and interaction with automatic rewritings and saturation implemented in our DDD library. Since our assignments are encoded as homomorphisms, they benefit from the automatic rewriting rules of [HTMK08]. These rules use the support of the expressions to skip don't-care variables and build clusters of transition effects. Our algorithm can also be implemented on top of other DD libraries. However, using DDD and homomorphisms allowed us to immediately benefit from these features.

Tools

Our implementation in `libits`[2] is compared to classical state-of-the-art approaches, represented by the tools `LTSmin` and `super_prove`.

`LTSmin` **[BvdPW10]** is a tool suite for model checking, that implements state-of-the-art symbolic techniques. It can use several third-party DD libraries, but we configured it to use DDD to allow easier algorithmic comparison. Indeed the state encoding being provided by the same DDD library as ours, the main difference between this tool and ours is the use of *EquivSplit*.

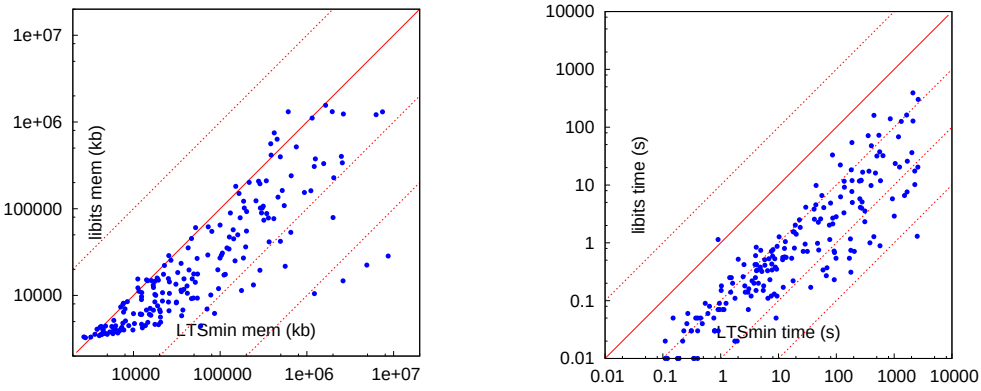
`super_prove` **[SG12]** is a SAT based model checker. It was the winner of the "single safety/bad-state property" track of the HardWare Model Checking Competition from 2010 to 2012, that contains the reachability properties of the BEEM models, translated into SAT instances. It is thus, to our knowledge, the best SAT-solver for this particular benchmark. SAT techniques are very different from those discussed in this chapter, but raw performance comparisons on this benchmark are still possible.

`libits` is a DD-based verification library that uses both hierarchical set decision diagrams and DDD to support model checking (CTL, LTL) of composition of labeled transition systems described symbolically. Transition systems can be described using several input formalisms, such as labeled discrete Time Petri nets. The GAL formalism was embedded in this framework but only uses DDD.

Models

The performance comparison is based on the full set of models from the BEEM database [Pel07], an amount of roughly 400 instances. Here we only report on reachability properties that were

[2] `http://ddd.lip6.fr`



(a) comparison on peak memory

(b) comparison on time

Figure 4.3: `libits` vs `LTSmin`, same variable ordering

also provided in the context of a recent hardware model checking contest (HWMCC'12[3]) as SAT instances. Thanks to previous work on `libits`, our implementation supports full CTL and LTL model checking of Divine models, also these features are not assessed here. All experiments were run on a Xeon 64 bits at 2.6 GHz processor.

Results and Discussion

Detailed results of experiments are presented as scatter plots comparing two tools over the whole benchmark. Each point represents a (model, formula) pair that was tested for reachability with both tools. A point below the diagonal means that `libits` is more efficient than the other tool. Our plots use a logarithmic scale. Lines parallel to the diagonal represent performance ratios of 10, 100 ... (resp. 0.1, 0.01 ...). On Figure 4.4, the red points indicate an unsatisfied property, and a green one a satisfied property.

`libits` vs. `LTSmin`. We compare the performance for the generation of the state space of the models, with 1 hour and 10Gb containment. Statistics of the results are shown in Table 4.1, and more detailed results are shown on Figure 4.3. The results confirm that our *EquivSplit* algorithm performs better than the classical symbolic approach. With the same implementation of DDD and the same variable ordering, our implementation is up to 1000 times faster and 100 times less memory consuming than `LTSmin`. For a dozen models, `LTSmin` is slightly more memory efficient than `libits`, but this can be attributed to side-effects of the garbage collection policy.

`libits` vs. `super_prove`. We compare the performance of both `libits` and `super_prove`, with a containment of 1Gb in memory and 900 seconds wall clock time (`super_prove` uses 4

[3] http://fmv.jku.at/hwmcc12

models tested	treated by			
	`libits`	`super_prove`	both	none
293	264	212	197	22

Table 4.1: `libits` vs `LTSmin`

	# unsat # unsat	mean time unsat (s)	# sat # sat	mean time sat (s)
`libits`	184	14.6	192	8.6
`super_prove`	112	140.6	170	45.1

Table 4.2: `libits` vs `super_prove`: mean runtime

models tested	treated by `libits`	 `super_prove`	 both	 none
456	376	282	258	56

Table 4.3: `libits` vs `super_prove`: number of instances solved

cores while `libits` is mono-threaded). These are the containment settings used in the HWMCC competition. Summary of the results are shown in Tables 4.2 and 4.3, and detailed results are presented in Figure 4.4. We only compared the time usage, since the memory consumption for SAT techniques is usually insignificant. Note that all `libits`'s fails are due to a memory overflow, whereas all `super_prove`'s fails are due to a time overflow.

`libits` treats about 35% more models than `super_prove`. Also, `libits` is quicker than `super_prove` for 80% of the models treated by both tools, with a speed-up factor up to 1000. On the other models, `super_prove`'s speed-up factor ranges up to 100.

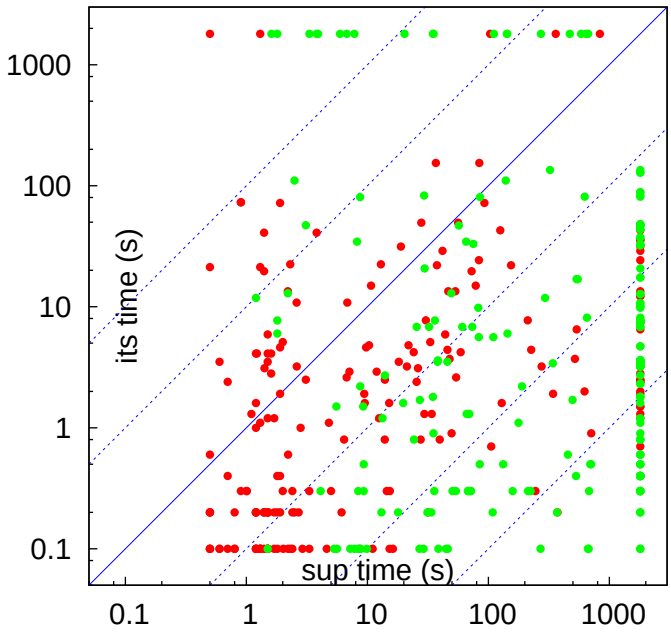


Figure 4.4: Time comparison between `libits` and `super_prove`

The Table 4.2 shows that `libits` runs on average 5 times faster on satisfied properties and 10 times faster on unsatisfied properties than `super_prove`, that stops as soon as it finds a solution for satisfied instances. Our tool regularly interrupts its computation to check whether a solution exists in the states computed so far. When these early checks are deactivated, `libits` is 4 times

faster on satisfied properties and 14 times faster on unsatisfied ones. Unsatisfied instances require both tools to explore the whole reachability graph: these are the hardest problems.

This benchmark shows that state-of-the-art symbolic manipulation of decision diagrams can still outperform the best SAT-based techniques.

4.6 Conclusion

We have described a new algorithm, *EquivSplit*, that enables efficient symbolic manipulation. It uses equivalence relations to avoid explicit manipulation of states. We apply it to the evaluation of symmetries, in a continuation of the previous chapter, and to the evaluation of a transition relation. Assessment on a large third-party benchmark (400 models with state spaces counting up to 4 billions states) shows that, in the latter use, this approach improves existing decision diagram-based techniques, and can outperform SAT-based ones.

Our algorithm supports arbitrary signatures (languages), and can be used with any type of decision diagrams. Beyond its interest for the evaluation of symmetry reductions, it represents an improvement for the manipulation of symbolic structures in general.

Chapter 5

Application to a specific Formalism: Symmetric Nets with Bags

We present the symmetry-DD combination on a particular formalism: Symmetric Nets with Bags (SNB). They are a variant of Colored Petri Nets, in which the symmetries act on *data* stored by state variables. Recall that the state of a system is a string of $\mathcal{D}^I$ where I is a set of state variables, and $\mathcal{D}$ their common domain. So far, symmetries are permutations of I whose action naturally extends to $\mathcal{D}^I$.

This chapter differs from the previous ones: symmetries in SNB act on $\mathcal{D}$, and their action is then extended to $\mathcal{D}^I$. An appropriate transformation of SNB make them fit in the previous framework, so that there is no theoretical difference. It however changes the computation of a representative function, and we have to design an appropriate encoding of states and a dedicated algorithm for the computation of a representative function.

There is another difference in this chapter with respect to the previous ones. We focus here on the use of Hierarchical Set Decision Diagrams (SDD). They extend DDD in the following manner: edges can now be labeled with sets rather than integers. This allows for hierarchy: since a SDD represents itself a set, an edge can be labeled by a SDD. We will exploit this capability to render the hierarchy in SNB, to obtain very compact encodings.

We first present the Symmetric Nets with Bags in Section 5.1. We then describe the symmetry reduction process for this formalism in Section 5.2, and the implementation in terms of SDD in Section 5.3. An assessment is finally presented in Section 5.4.

5.1 Symmetric Nets with Bags

We first present in this section the formalism of Symmetric Nets with Bags.

5.1.1 Petri Nets

Petri nets (PN) are a formalism to represent concurrent systems, whose foundations were introduced by [Pet62]. A Petri net is made of a finite number of places, containing tokens, and transitions that move tokens from places to places. The state of a Petri Net is given by the number of tokens in each of its places, and can be represented by a $\mathbb{N}$-string indexed by the places. The transition relation of the system is described through the net transitions, that establish arithmetical dependencies between places.

The transitions describe the transition relation as arithmetical relations between states, so that the underlying transition system is quite straightforward.

Definition 5.1: Petri net
A Petri net N is a tuple $\langle P, T, W^-, W^+, M_0 \rangle$ where:

- P is a finite set of *places*;
- T is a finite set of *transitions*, $P \cap T = \emptyset$;
- $W^- : T \times P \mapsto \mathbb{N}$ is the *pre-relation*, describing how many tokens a given transition consumes from a given place;
- $W^+ : T \times P \mapsto \mathbb{N}$ is the *post-relation*, describing how many tokens a given transition produces in a given place;
- $M_0 : P \mapsto \mathbb{N}$ is the initial *marking*.

When there are enough tokens for a transition to consume, it can *fire* which leads to a new marking.

Definition 5.2: Petri net Semantics
Let $N = \langle P, T, W^-, W^+, M_0 \rangle$ be a Petri net.
Its semantics is a transition system $\langle S, \Delta, \rangle$ where:

- states are markings $S = \mathbb{N}^P$;
- M_0 is the initial state;
- $\langle M, t, M' \rangle \in \Delta$ if, and only if, for all $p \in P$, $W^-(t)(p) \leq M(p)$ and $M'(p) = M(p) - W^-(t)(p) + W^+(t)(p)$.

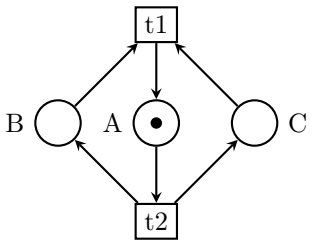


Figure 5.1: Example of Petri net

A very simple Petri net is presented on Figure 5.1. It has 3 places A, B, C and 2 transitions $t1$ and $t2$. Usually, the number of tokens moved by a transition t from or to a place p is symbolized by an arc between p and t, labeled with the number of moved tokens. If no label, it implicitly means that a single token is to be moved. The net is represented with a marking of one token in the place A and none in places B and C. With this marking, the transition $t2$, that only consumes one token from place A, is *enabled*. If fired, it will lead to a new marking where the place A contains no token and the places B and C contain one token each, thus enabling $t1$.

5.1.2 Colors and Symmetric Nets

Although Petri nets are well-suited to model concurrency, the data contained in a place is limited. Colored Petri Nets (CPN) [Jen87] have been introduced to solve this limitation. Tokens can have a *color*, and a place may contain tokens of different colors. This allows to represent more complex systems by keeping the net compact. Intuitively, a color class represents a data type, the colors inside the class representing the different values of this type.

An alternative syntax, emphasizing on symmetries, has been proposed by [CDFH90]. As shown in [CDFH91], both formalisms have only slight differences, but have the same expressive power.

Definition 5.3: Symmetric Nets (SN)
A Symmetric Net N is a tuple $\langle P, T, C, J, \phi, W^-, W^+ \rangle$ where:

- P is a finite set of places;
- T is a finite set of transitions, $P \cap T \neq \emptyset$;
- $C = \{C_1, \ldots, C_n\}$ a finite set of pair-wise disjoint finite color classes. A color class C_i may be partitioned in static subclasses $C_i = \cup_{q=1}^{n_i} D_{i,q}$;
- $J : P \cup T \mapsto Bag(\{1 \ldots n\})$ associates to each node r a color domain $C(r) = C_{J(r)}$. A domain is thus always a cartesian product of color classes.
- W^-, W^+ are the input and output arc expressions. For all $p \in P$ and $t \in T$, $W^-(p,t)$ and $W^+(p,t)$ are mappings from $C_{J(t)}$ to $Bag(C_{J(p)})$;
- ϕ associates to each transition t a predicate, or guard, $\phi(t) : C_{J(t)} \mapsto \{\top, \bot\}$.

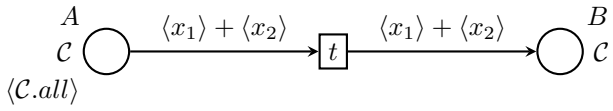


Figure 5.2: Example of a Symmetric Net

A marking M of a Symmetric Net N associates to each place p a place marking $M(p) \in Bag(C_{J(p)})$, that is a multi-set of tokens in its color domain.

The input and output arc expressions are represented by labels on arcs. Recall that $W^+(p,t)$ and $W^-(p,t)$ are mappings from $C_{J(t)}$ to $Bag(C_{J(p)})$. The SN syntax for arc expressions rely on $|J(t)|$ formal variables – one per each component of $C_{J(t)}$ – and constant colors. It mainly features an operator for tuple creations, multiplication by an integer to represent the multiplicity of an element, sum and difference of bags (see Section 2.6.3), color increment and decrement (in the case color classes are ordered).

A transition instance $\langle t, c \rangle$ (for a transition t and $c \in C_{J(t)}$) is *enabled* in a marking M if for all places p, $W^-(p,t)(c) \leq M(p)$, and $\phi(t)(c) = \top$. If enabled, the transition instance $\langle t, c \rangle$ can be fired which leads to a new marking M' such that, for all places p, $M'(p) = M(p) - W^-(p,t)(c) + W^+(p,t)(c)$.

An example of Symmetric Net is presented on Figure 5.2. It features two places A and B with the same domain $\mathcal{C}$. Initially, A is marked with $\langle \mathcal{C}.all \rangle$, that designates the bag that contains each element of $\mathcal{C}$ once. B is initially empty. The transition t moves two tokens, designated by the formal variables x_1 and x_2 from A to B. Its domain is implicit here, and is the product of the domain of its formal variables, that is $\mathcal{C} \times \mathcal{C}$. Its guard, implicit, is true by default.

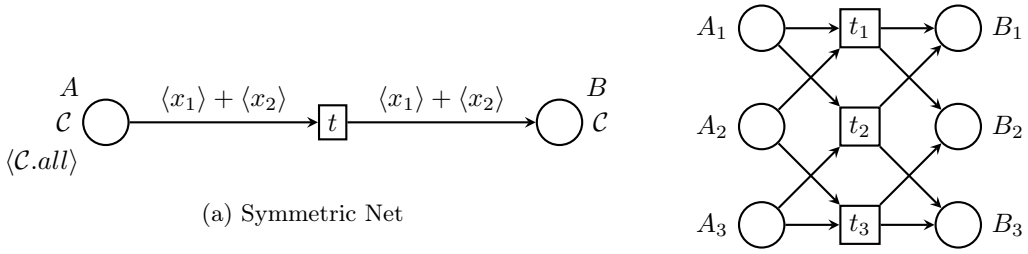
These definitions are very similar to the definition of Petri nets. Whereas a Petri net place contains an integer, a SN place contains a multi-set of colors. A Petri net can be seen as a SN with a single color class, of size 1, that is the domain of all places and transitions. Interestingly, despite this inclusion, Symmetric Nets are not more expressive than Petri nets.

Bags on colors are defined as functions that associate to each color a multiplicity. Therefore, a place marked with a bag on colors can be unfolded into one Petri net place for each color. Following this idea, unfolding a SN into a Petri net with the same semantics is quite straightforward, and shows that SN are actually as expressive as Petri nets.

Definition 5.4: SN Unfolding
Let $N = \langle P, T, C, J, \phi, W^-, W^+ \rangle$ be a SN. We define its Petri net unfolding $N' = \langle P', T', W'^-, W'^+ \rangle$ as:

- P' is the set of pairs $\langle p, c \rangle$ where $p \in P$ and $c \in C_{J(p)}$;
- T' is the set of pairs $\langle t, c \rangle$ where $t \in T$, $c \in C_{J(t)}$ and $\phi(t)(c) = \top$;
- for all $p \in P$, $t \in T$, $c_p \in C_{J(p)}$ and $c_t \in C_{J(t)}$ such that $\phi(t)(c) = \top$, $W'^-(\langle p, c_p \rangle, \langle t, c_t \rangle) = W^-(p, t)(c_t)(c_p)$ and $W'^+(\langle p, c_p \rangle, \langle t, c_t \rangle) = W^+(p, t)(c_t)(c_p)$;

One should then see Colored Petri Nets (CPN) and SN as an alternative, usually more compact, syntax for Petri nets. Note that the unfolding can lead to a net with a size exponential in the size of the original colored net, and may not be of practical interest for model checking.



(a) Symmetric Net

(b) Petri Net unfolding

Figure 5.3: An example of a Symmetric Net and its unfolding

An example of SN and its unfolding is presented in Figure 5.3. The net on Figure 5.3a has two places A and B, and a single transition t, that takes two tokens from place A (represented by $\langle x_1 \rangle$ and $\langle x_2 \rangle$) and moves them to place B. Figure 5.3b shows its unfolding. For each place (A and B) and each color, there is a place in the unfolding (A_1, A_2, A_3, B_1, B_2 and B_3). The transition t is unfolded in three transitions (t_1, t_2 and t_3), one for each way of choosing two elements ($\langle x_1 \rangle$ and $\langle x_2 \rangle$) among three.

5.1.3 Symmetric Nets with Bags

Symmetric Nets with Bags (SNB) extend Symmetric Nets by allowing to manipulate bags of tokens directly. In particular, "super-tokens" containing a bag of tokens can be created. Therefore, in SNB, the domain of a node is of the form $\prod_{i \in I} C_i \times \prod_{i \in I'} Bag(C_i)$. Arc expressions are extended with a bag creation operator, that creates "super-tokens", and a bag projection operator, to use the content of such a "super-token" as a bag that can be added to a place marking. Similarly, guards now allow the comparison between bags and bag sizes. Guards of transitions must however ensure a finite number of transition instances. With this limitation, SNB can be unfolded in a SN, so that the expressive power of SNB is the same as SN. Their complete definition can be found in [HKP+09].

The SaleStore example Let us present SNB by means of a simple example, the SaleStore (see Figure 5.4). People enter the sale store through an **airlock** with a maximal capacity of two (of course, only a single person may enter too). Then, people may buy items (at most two, but possibly zero if none fits their need) and leave with the acquired items. Let us note that this example has two scalable parameters: **P**, the number of involved people in the system, and **G**, the number of available gifts in the warehouse.

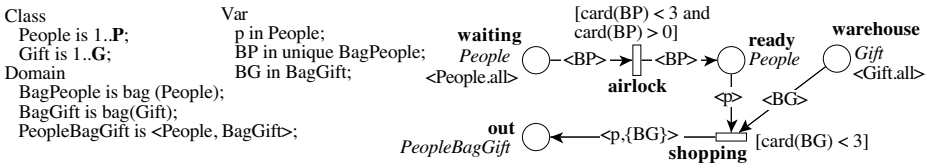


Figure 5.4: The SaleStore example modelled with a SNB

The model in Figure 5.4 illustrates most of the main features of SNB. First, there are several color types giving the place domains: simple color types like *People* or *Gift* are called classes, while bags such as *BagPeople* or *BagGift* and cartesian products such as *PeopleBagGift* are built upon basic color classes.

Variables which are formal parameters for transition binding are declared in the *Var* section. A basic variable such as p can be bound to represent any element of *People*. A variable such as BP represents a multiset (or bag) of *People*; since it is tagged by the `unique` keyword, it can actually only be bound to a subset of *People* (each element in BP appears at most once). Variable BG is not tagged with the `unique` keyword; it could be bound to any multiset of gifts (if the warehouse was configured to contain several instances of each gift for instance).

Transition guards can be used to constrain the cardinality of a bag variable: the constraint $[card(BP) < 3 \text{ and } card(BP) > 0]$ on **airlock** models its capacity of at most 2 people (the airlock does not operate empty), while the constraint on **shopping** bounds the number of gifts bought in the store by each person.

Equivalence with Symmetric Nets SNB have the same expressiveness as Symmetric Nets but allow for more compact modeling. Like colored nets, which can be seen as an abbreviation of P/T nets, SNB can also be unfolded into an equivalent SN. Figure 5.5 shows such an unfolding. Transitions **airlock** and **shopping** are replicated for each possible cardinality of the bag variable they can instantiate. Place **out** in the SNB is unfolded according to the various domains obtained by "flattening" the bag token of the *PeopleBagGift* type. Hence, the greater the bound on the cardinality, the more places in the unfolded SN. Note that modeling anything the customers do once they are out with their gifts would be very cumbersome in the SN.

However, designers must be careful when defining guards and initial markings so that the model remains finite if unfolded to SN. For example, if **shopping** had no **warehouse** place as input and no guard, an infinite number of bags could be generated (leading to an infinite unfolding). The main advice for the designers is to always ensure that the cardinality of the bag-variables is bounded.

Advantages of SNB From Figures 5.4 and 5.5, it is obvious that with bag manipulations, SNB provide a much more compact and natural way to model systems than SN.

Another advantage of SNB is to allow the production of a more compact quotient reachability graph in number of edges. This is due to the use of bag variables which better express the symmetries of possible bindings of variables to values. For instance, when choosing two *People*

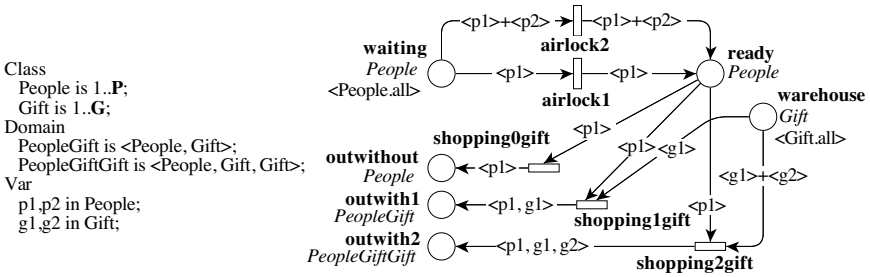


Figure 5.5: Unfolding of the SNB presented in Figure 5.4 into a SN

from **waiting**, **airlock** in the SNB allows $P \times (P-1)$ bindings (*i.e.* choose two from P), while **airlock** in the SN allows $2 \times P \times (P-1)$ because variables $p1$ and $p2$ can independently be bound to any element of *People*. Since computing successors in a quotient graph is a costly operation (due to the canonization procedure), this aspect may heavily impact the performance of analysis tools.

5.2 Symmetry Reduction for SN and SNB

Let us consider a SN N and its Petri net unfolding N' as defined by Definition 5.4. Let us consider a permutation $g \in \prod_{i \in I} Sym(C_i)$. g acts naturally on colors. Furthermore, it preserves the color classes, and acts then naturally on tuples of colors. Let $D = \prod_{j \in J} C_{i_j}$ be a cartesian product of color classes (for instance a domain of a node). Let $c = (c_j)_{j \in J} \in D$, $g.c = (g(c_j))_{j \in J}$. Since $g(C_i) \subseteq C_i$ for all $i \in I$, $g.c$ is also an element of D. This action extends naturally on bags on colors, and finally on markings of the SNB N.

The action on color tuples also extends straightforwardly on the places and transitions of the unfolding N'. Considering this last action, one can see g as a permutation of $P' \cup T'$. It is indeed a symmetry if g preserves the value of guards and is regular with respect to the arc expressions.

In practice, the color classes are partitioned in *static subclasses*, as mentioned in Definition 5.3. These subclasses are intended to define the symmetries of the model. Structural analysis of the guards and arc expressions can be used to infer such static subclasses [TMDM03].

From now on, we assume that any $g \in \prod_{i \in I} \prod_{q=1}^{n_i} Sym(D_{i,q})$ is a symmetry of the net.

Due to the particular structure of the symmetry group, [CDFH97] proposes an original representative function for orbits, that does not strictly fit our Definition 2.5. Let us first extend the notion of representative function to the notion of invariant.

5.2.1 Invariants

A (canonical) representative function associates to each orbit a (unique) representative state that belongs to this orbit. Such representatives are then used to efficiently compare orbits. However, as defined in Definition 2.6, the representative need not be a state: one may chose any arbitrary set for representatives, provided that they allow for efficient orbit comparison.

We thus define an invariant to be a function constant on an orbit.

Definition 5.5: Invariant
Let X and Y be two arbitrary sets, and $G \lhd Sym(X)$.
A G-invariant is a function $\zeta : X \mapsto Y$ such that for all x and y in X, $[x]_G = [y]_G \implies$

$\zeta(x) = \zeta(y)$.
A G-invariant is canonical if, furthermore, for all x and y in X, $\zeta(x) = \zeta(y) \implies [x]_G = [y]_G$.

Note that a (generalized) representative function is canonical if and only if it is an invariant. Conversely, an invariant is canonical if and only if it is a (generalized) representative function. Thus, the notions of canonical invariant and canonical representative function coincide.

5.2.2 Symbolic Marking

The representation of an orbit according to [CDFH97], also called a *symbolic marking*, is such a canonical invariant. We describe the construction for SN, to ease the notations. The construction and the proof that it is a canonical invariant extends naturally to SNB.

Let M be a marking of a SN. Let us define an equivalence relation $\sim_M$ on colors, that depends on M: $c \sim_M c'$ if and only if c and c' belong to the same static subclass and M is left unchanged by swapping c and c'. This guarantees that c and c' appear in the same location: in the same places, with the same multiplicities, in the same "super-tokens" ...

Let us now consider a symmetry g of the net. We remark that $g.c \sim_{g.M} g.c'$ if and only if $c \sim_M c'$. Each static subclass C is partitioned by $C_{/\sim_M}$, and $\{\{|c|, c \in C_{/\sim_M}\}\}$ is a partition of $|C|$. We then define $\zeta(M)$ the function that associates to each static subclass C the partition of $|C|$. The parts of these partitions are called *dynamic subclasses*. We denote by D the set of dynamic subclasses, and by $\mathcal{C}$ the set of all colors (*i.e.* the disjoint union of all color classes).

Proposition 5.1. *ζ is a G-invariant on markings.*

Proof. Consider a marking M and a symmetry g. Consider a static subclass C. Recall that, by definition of the static subclasses, $g(C) = C$. If we note $C_{/\sim_M} = \{D_1, \ldots, D_p\}$, then $C_{/\sim_{g.M}} = \{g(D_1), \ldots, g(D_p)\}$. Since g is bijective, we can then conclude that $\{\{|c|, c \in C_{/\sim_M}\}\} = \{\{|c|, c \in C_{/\sim_{g.M}}\}\}$. Therefore $\zeta(M)(C) = \zeta(g.M)(C)$, and this holds for every static subclass C, so that $\zeta(M) = \zeta(g.M)$. □

Let us illustrate this construction with a small example. Consider the partial marking (only places **waiting**, **warehouse** and **out** are represented) of the SaleStore example of Figure 5.4:

$$M = waiting(P_1) + warehouse(G_0) + out(\langle P_0, \{G_1\}\rangle)$$

$\sim_M$ is easy to compute: P_0, P_1, G_0 and G_1 are alone in their respective equivalence classes. Other elements of P (say $P_2, \ldots, P_n$) are in the same equivalence class, and other elements of G (say $G_2, \ldots, G_p$) are in the same equivalence class. Therefore, $\zeta(M)$ indicates that M defines two dynamic subclasses of size 1 and one of size $n-1$ for P, and two of size 1 and one of size $p-1$ for G.

The invariant ζ is however not canonical. Let us now define the symbolic marking itself. By definition of the dynamic subclasses, $\mathcal{C}_{/\sim_M}$ and D have the same cardinality. Choose a bijection $\phi_M : \mathcal{C}_{/\sim_M} \mapsto D$, such that $\phi_M(c) = |c|$ for all $c \in \mathcal{C}_{/\sim_M}$. This associates every classes of $\sim_M$ to a dynamic subclass of the corresponding cardinality.

For a color a, we note $\eta_M(a) = \phi_M([a]_{\sim_M})$. For $c = (c_1, \ldots, c_{|J(p)|}) \in C_{J(p)}$, we note $\eta_M(c) = (\eta_M(c_1), \ldots, \eta_M(c_{|J(p)|}))$. Let $\eta_M(C_{J(p)}) = \{\eta_M(c) | c \in C_{J(p)}\}$.

Definition 5.6: Symbolic Marking
The symbolic marking $\tilde{M}(p)$ of place p is a bag on $\eta_M(C_{J(p)})$:

$$\forall c \in C_{J(p)}, \tilde{M}(p)(\eta_M(c)) = M(p)(c)$$

Let $c = (c_1, \ldots, c_{|J(p)|})$ and $c' = (c'_1, \ldots, c'_{|J(p)|})$ be two elements of $C_{J(p)}$ such that $\eta_M(c) = \eta_M(c')$. For $i \in \{1; \ldots; |J(p)|\}$, $\eta_M(c_i) = \phi_M([c_i]_{/\sim_M})$. Since ϕ_M is bijective, and $\eta_M(c_i) = \eta_M(c'_i)$, c_i and c'_i necessarily belong to the same equivalence class: $c_i \sim_M c'_i$. This holds for all $i \in \{1; \ldots; |J(p)|\}$, so that $M(p)(c) = M(p)(c')$. This shows that $\tilde{M}(p)(\eta_M(c))$ is well-defined, for all $c \in C_{J(p)}$.

Strictly speaking, the symbolic marking is the pair $\langle \zeta(M), \tilde{M} \rangle$. We have already seen that ζ is a G-invariant on markings. However, due to the choice of ϕ_M during the construction of $\tilde{M}$, the symbolic marking may not be a G-invariant.

We go back to our example of the partial marking $M = waiting(P_1) + warehouse(G_0) + out(\langle P_0, \{G_1\}\rangle)$. Let note the dynamic subclasses Z_0^P, Z_1^P (both of size 1), Z_2^p (of size $n-1$) for P, and Z_0^G, Z_1^G (both of size 1) and Z_2^G (of size $p-1$) for G. Several symbolic markings are possible, namely:

$$\begin{aligned}
&waiting(Z_1^P) + warehouse(Z_0^G) + out(\langle Z_0^P, \{Z_1^G\}\rangle)\\
&waiting(Z_1^P) + warehouse(Z_1^G) + out(\langle Z_0^P, \{Z_0^G\}\rangle)\\
&waiting(Z_0^P) + warehouse(Z_0^G) + out(\langle Z_1^P, \{Z_1^G\}\rangle)\\
&waiting(Z_0^P) + warehouse(Z_1^G) + out(\langle Z_1^P, \{Z_0^G\}\rangle)
\end{aligned}$$

5.2.3 Symbolic Marking as a Canonical Representative Function

Let us find out under which condition the symbolic marking is an invariant.

Let g be a symmetry of the net, and $M' = g.M$. Recall that $\tilde{M}'$ depends on the choice of a bijection $\phi_{M'}$ between $\mathcal{C}_{/\sim_{M'}}$ and D. We first choose such a $\phi_{M'}$ such that, for all $c \in \mathcal{C}_{/\sim_{M'}}, \phi_{M'}(c) = |c|$. Thus, for all $c = \langle c_1, \ldots, c_{|J(p)|}\rangle \in C_{J(p)}$:

$$\begin{aligned}
\tilde{M}'(p)(\eta_{M'}(g.c)) &= M'(p)(g.c)\\
&= M(p)(c) && \text{by definition of } g\\
&= \tilde{M}(p)(\eta_M(c)) && (5.1)
\end{aligned}$$

Equation (5.1) clearly shows that $\tilde{M}' = \tilde{M}$ if and only if $\eta_{M'} \circ g = \eta_M$.

Let $c = \langle c_1, \ldots, c_{|J(p)|}\rangle \in C_{J(p)}$.

$$\begin{aligned}
(\eta_{M'} \circ g)(c) &= \eta_{M'}(g.c)\\
&= \phi_{M'}(\langle [g.c_1]_{\sim_{M'}}, \ldots, [g.c_{|J(p)|}]_{\sim_{M'}}\rangle) && \text{by definition of } \eta_M\\
&= \phi_{M'}(\langle g.[c_1]_{\sim_M}, \ldots, g.[c_{|J(p)|}]_{\sim_M}\rangle)\\
&= (\phi_{M'} \circ g)(\langle [c_1]_{\sim_M}, \ldots, [c_{|J(p)|}]_{\sim_M}\rangle) && \text{by factoring } g \text{ out} \quad (5.2)
\end{aligned}$$

Let us consider $\psi = \phi_{M'} \circ g \circ \phi_M^{-1}$, a permutation of D. We further rewrite Equation (5.2):

$$\begin{aligned}
(\eta_{M'} \circ g)(c) &= (\phi_{M'} \circ g)(\langle [c_1]_{\sim_M}, \ldots, [c_{|J(p)|}]_{\sim_M}\rangle)\\
&= (\phi_{M'} \circ g)(\langle \phi_M^{-1}(\eta_M(c_1)), \ldots, \phi_M^{-1}(\eta_M(c_{|J(p)|}))\rangle)\\
&= (\phi_{M'} \circ g \circ \phi_M^{-1} \circ \eta_M)(c_1, \ldots, c_{|J(p)|}) && \text{for } \phi_M^{-1} \circ \eta_M = id\\
&= (\psi \circ \eta_M)(c)
\end{aligned}$$

Thus, $\eta_{M'} \circ g = \psi \circ \eta_M$, whence $\eta_{M'} \circ g = \eta_M$ if and only if ψ is the identity. Note that the set of all possible ψ is a subgroup Ψ of $Sym(D)$, namely the group of permutations that respect the cardinalities of the parts. This holds for any g, and since Ψ does not depend on g, we obtain the following result: Ψ acts transitively on the set $\mathbf{M}$ of possible symbolic markings for M.

Interestingly, if `repr` is a representative function on the symbolic markings, then the pair $\langle \zeta(M), \texttt{repr}(\tilde{M}) \rangle$ is a canonical invariant on markings. Through the action of Ψ, we get back to already visited concepts. It is clear that Ψ is exactly the set $\{\phi_{g.M} \circ g \circ \phi_M^{-1}\}$ where g, ϕ_M and $\phi_{g.M}$ range over their respective domains. Let us define the action of Ψ on markings. Let us (abusively) note $M \circ \eta$ the marking such that $(M \circ \eta)(p)(c) = M(p)(\eta(c))$ for all place p and $c \in C_{J(p)}$. For all marking M, $\psi.M$ is the marking such that $(\psi.M) \circ (\eta_{g.M} \circ g) = M \circ (\psi \circ \eta_M)$ for some g, ϕ_M and $\phi_{g.M}$ such that $\psi = \phi_{g.M} \circ g \circ \phi_M^{-1}$ (such elements always exist).

Therefore, considering our example of the previous section, the retained canonical symbolic marking is $\tilde{M} = waiting(Z_0^P) + warehouse(Z_0^G) + out(\langle Z_1^P, \{Z_1^G\}\rangle)$.

To sum up, the symbolic marking is unique up to a reindexing of the dynamic subclasses of the same cardinality. [CDFH97] suggests to index the dynamic subclasses such that $\tilde{M}$ is lexicographically minimal.

Thus, the construction of a symbolic marking for the orbit of M is performed as follows:

- compute $\zeta(M)$;
- choose a bijection ϕ between $C_{J(p)}/\sim_M$ and D;
- build $\tilde{M}$ according to ϕ;
- find a canonical representative of the orbit of $\tilde{M}$ under the action Ψ (that only depends on $\zeta(M)$).

5.2.4 Symbolic Firing

We have presented the representation of orbits as symbolic markings. We now describe how the transitions can be fired on such markings.

The key idea is to avoid to find an ordinary marking from the symbolic marking, fire the transition explicitly, and compute the symbolic marking of the result. Instead, the variables of the transitions are assigned appropriate dynamic subclasses, and the transition is then fired directly on the symbolic marking. This requires some additional technical details. A variable can only be assigned a dynamic subclass of size 1. A transition is then fired in three steps:

- split each dynamic class D into $|D|$ dynamic classes of size 1;
- find all possible symbolic instantiations of the transition;
- fire the instantiated transition directly on the symbolic marking;
- canonize the result.

During the canonization of the resulting symbolic marking, one has first to merge previously split dynamic subclasses, then find its canonical representative.

The implementation of these steps mainly relies on the *EquivSplit* algorithm presented in Chapter 4. Historically, the proper definition of *EquivSplit* occurred later than the SNB implementation; although the mechanisms and concepts are the same, its precise design differs somehow from what is presented in the previous chapter, and is specialized for the SNB framework.

We now describe the implementation of the various steps:

- **split** dynamic classes into classes of size 1;
- **instantiate** the transitions;
- **fire** the instantiated transitions symbolically;
- canonize the result, which implies several steps as explained above:
  - **merge** the dynamic classes of size 1 into bigger ones (recompute properly ζ);
  - **reindex** the dynamic classes to get the lexicographically smaller marking.

Split. This operation is done through a simple homomorphism that rewrites each dynamic class D in the marking into $|D|$ dynamic classes of size 1. This rewriting takes care of the proper naming of dynamic classes, to avoid name overlapping. It does not present difficulties and does not use *EquivSplit*.

Instantiate. This step aims to find all the possible instantiations for a given transition. It is primarily based on the partition of static subclasses into dynamic subclasses. Thus, it does need to visit the symbolic markings, and is not an operation to be implemented in terms of homomorphisms.

Several optimizations are possible, so as to avoid redundant or pointless instantiations. By using the information on the dynamic subclasses before the step of splitting, several redundancies can be eliminated. For instance, if z is a dynamic subclass of size 2, split into z_1 and z_2, then binding a variable x to z_1 is not different from binding to z_2.

Another possible optimization retains only the instantiations that are relevant for the considered markings. It consists of exploring the markings to eliminate instantiations that would not be fireable, typically because of a lack of the appropriate tokens in one place. Because it necessitates information on the markings, this optimization is implemented in terms of homomorphism. In the fashion of *EquivSplit*, it associates to each marking the possible instantiations, gathering the markings that share the same set of possible instantiations. These can be computed by successive refinements, place by place. Due to the high number of sets of instantiations, there is a risk to individualize the markings that would incur performance loss. The common strategy is to compute an over-approximation of the set of possible instantiations, to avoid too much individualization of DD paths. Elimination of non-fireable instantiations is then performed during the firing step, as explained next.

Fire. This step only requires to visit the markings, removing and adding tokens according to a given instantiation of a transition. In case a place does not contain enough tokens to be removed, it maps the path onto **0** in the DD, thus indicating that the marking represented by this path has no successor by the current instantiated transition. This step does not present difficulties, and does not require to use *EquivSplit*.

Merge. This step consists in recomputing $\sim_M$ so as to reform the proper dynamic subclasses that have previously been split. We first define the equivalence relation $\sim_M^{P'}$ on dynamic subclasses, for a $P' \subseteq P$ a subset of places: $z_1 \sim_M^{P'} z_2$ if M restricted to P' is invariant by swapping z_1 and z_2. We note that $\sim_M^{P_1 \cup P_2}$ is the intersection of $\sim_M^{P_1}$ and $\sim_M^{P_2}$. We also note that $\sim_M^{P} = \sim_M$. This allows to compute $\sim_M$ by successive refinements, place by place. This operation thus follows the scheme used in *EquivSplit*, where the expression is evaluated by successive refinements, variable by variable.

The dynamic subclasses that are equivalent under $\sim_M$ can then be merged. We first compute the relation $\sim_M$ for each marking, gathering markings that have the same $\sim_M$. Once $\sim_M$ is computed, we use it to merge the dynamic subclasses; this implies to update the cardinalities information, and to rename the classes in the markings. This last step has no particular difficulty and is very similar to the split operation above.

Reindex. This step ensures that the found symbolic markings are the lexicographical leaders of their respective orbits. Due to the encoding, the action of Ψ is not a permutation of positions in a vector, so that the algorithm of Chapter 3 cannot be used. The sorting is performed place by place as follows: it minimizes lexicographically the marking of the current place, propagates the possible re-indexation on the subsequent places, and remembers the classes whose index is fixed by the minimization of the place. These fixed indices restrict the possible permutations to be used, place by place: if all Ψ can be used on the first place, it eventually becomes trivial when advancing in the marking.

5.2.5 Example of the Implementation

We now illustrate how the steps defined above work on a simple example. Let us consider the simple net presented in Figure 5.6. It features three places, A, B and C that have the same domain $\mathcal{D}$. We assume that $\mathcal{D} = \{1, 2, 3, 4, 5\}$ has five elements. The single transition t takes two elements x and y from place A and moves one of them to B, and the other one to C.

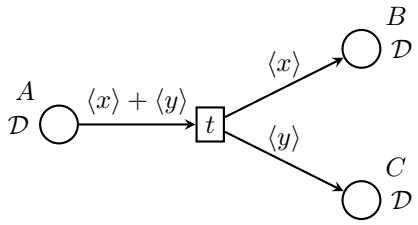


Figure 5.6: Example of Symmetric Net

Let us illustrate the firing of transition t on the two symbolic markings:

$$M_1 = A(Z_1) + B() + C() \qquad \text{with } |Z_1| = 5$$
$$M_2 = A(Z_1) + B(Z_2) + C(Z_3) \qquad \text{with } |Z_1| = 3, |Z_2| = |Z_3| = 1$$

Split. It is a rewriting operation that leads to the new markings, where all dynamic subclasses have size 1:

$$M_1^s = A(Z_1 + Z_2 + Z_3 + Z_4 + Z_5) + B() + C()$$
$$M_2^s = A(Z_1 + Z_2 + Z_3) + B(Z_4) + C(Z_5)$$

Instantiate. This operation assigns a dynamic subclass (of size 1) to each of the formal parameters of the transition, namely x and y. Since there are 5 dynamic subclasses, we obtain 25 possible instantiations.

At this point, we can use a first optimization that uses the information on dynamic subclasses before the splitting step to eliminate redundant instantiations. For instance, in the two markings above, Z_1, Z_2 and Z_3 result from the split of a single subclass. Therefore, permuting them does

not change the markings. So it is not relevant to bind a variable twice with them. Associating, say, x with Z_1 and y with Z_2 has the same effect as associating x with Z_2 and y with Z_3. Similarly, binding both x and y to Z_1 has the same effect as binding them both to Z_2. One can further argue that the same holds for Z_4 and Z_5 for the marking M_1. The argument holds, but it would lead to different sets of instantiations for both markings, what we want to avoid here to keep the example simple. In this case, it would also lead to an individualization of the markings, that may hinder the performance of the underlying DD.

The second optimization to be used deduces forbidden instantiations from the markings. For instance, binding x and y to the same subclass requires, for the instantiation to be fireable, that this subclass appears twice in place A. Since all dynamic subclasses appear exactly once in the net, such bindings have no successor and can be dropped.

We thus end up with seven instantiations, to be applied on both markings:

$$\begin{array}{ccc} & x : Z_1, y : Z_2 & \\ x : Z_1, y : Z_4 & x : Z_1, y : Z_5 & x : Z_4, y : Z_1 \\ x : Z_4, y : Z_5 & x : Z_5, y : Z_1 & x : Z_5, y : Z_4 \end{array}$$

Further note that all these instantiations are equivalent for the marking M_1^s.

Fire. Now that the instantiations are computed, we just have to fire them. As explained above, it is easy to design a homomorphism that removes and adds the appropriate tokens in each place, and maps the path to **0** if there are not enough tokens in the place.

We finally obtain the following markings, where all dynamic subclasses have size 1 (we have removed equivalent bindings to be fired on M_1^s but one, to limit the number of markings to be considered):

$$\begin{aligned} M_1^f &= A(Z_3 + Z_4 + Z_5) + B(Z_1) + C(Z_2) \\ M_2^f &= A(Z_2 + Z_3 + Z_5) + B(Z_1) + C(Z_4) \\ M_3^f &= A(Z_3) + B(Z_1 + Z_4) + C(Z_2 + Z_5) \end{aligned}$$

Merge. In this step, we compute place by place the relation $\sim_M$ for all the markings.

$$\begin{aligned} \sim_{M_1^f} &= \{\{Z_1\}, \{Z_2\}, \{Z_3, Z_4, Z_5\}\} \\ \sim_{M_2^f} &= \{\{Z_1\}, \{Z_2, Z_3, Z_5\}, \{Z_4\}\} \\ \sim_{M_3^f} &= \{\{Z_1, Z_4\}, \{Z_2, Z_5\}, \{Z_3\}\} \end{aligned}$$

We then merge the equivalent subclasses, and rename them so as to keep the subclasses sorted by cardinality:

$$\begin{array}{ll} M_1^m = A(Z_1) + B(Z_2) + C(Z_3) & \text{with } |Z_1| = 3, |Z_2| = |Z_3| = 1 \\ M_2^m = A(Z_1) + B(Z_3) + C(Z_2) & \text{with } |Z_1| = 3, |Z_2| = |Z_3| = 1 \\ M_3^m = A(Z_3) + B(Z_2) + C(Z_1) & \text{with } |Z_1| = |Z_2| = 1, |Z_3| = 1 \end{array}$$

Note that several possibilities may be possible during the renaming. It is not important here: the choice between the different renaming is resolved in the next step, to get the lexicographical leaders.

Reindex. This step performs the final sort of the dynamic classes to get the lexicographical leaders. Let us describe how it runs to sort M_3^m. In place A, only the subclass Z_3 appears. Since no other subclass has the same cardinality, no permutation is possible. The algorithm remembers that Z_3 is now fixed and moves to the next place. In B, the subclass Z_2 can be swapped with the subclass Z_1 that has the same cardinality. Doing so would reduce the marking of B ($Z_1 < Z_2$). Since Z_1 and Z_2 are not marked as fixed, the algorithm swaps them, propagates this swap to subsequent places, and marks Z_1 as fixed. C contains now Z_2 (after the swapping), and there is no other non-fixed subclass of the same cardinality to be swapped with. The algorithm thus terminates.

M_1^m is minimal lexicographical and stays the same. M_2^m is not lexicographically minimal, and is turned to M_1^m by this step. We conclude that M_1^m and M_2^m were in the same orbit. We finally obtain the two symbolic markings:

$$M_1^r = A(Z_1) + B(Z_2) + C(Z_3) \qquad \text{with } |Z_1| = 3, |Z_2| = |Z_3| = 1$$
$$M_2^r = A(Z_3) + B(Z_1) + C(Z_2) \qquad \text{with } |Z_1| = |Z_2| = 1, |Z_3| = 1$$

as successors of M_1 and M_2 by transition t.

5.3 SDD Representation of the Symbolic Markings

We come to the representation of (symbolic) markings using SDD. This representation is the main factor for the performance of our implementation. The use of symbolic markings reduces the memory print of a single marking. The presence of bags in tokens allows for hierarchical sharing in decision diagrams, as will be explained later. Finally, the use of bags and of symbolic firing both reduce the number of instantiations for transitions, and reduce the amount of operations to be performed to compute the state space.

Let us recall the hierarchical possibilities of the SDD. Since their edges are labelled with sets, and they themselves represent sets, SDD edges can be labelled with SDD. This hierarchy allows to share (in unique tables) SDD nodes that appear at different levels of hierarchy.

This feature is exploited in the SDD encoding of the SNB markings. The marking of a place is a bag of tokens. Bag tokens are encoded using another level of hierarchy. This allows to share the representation of bag-tokens with place markings in the favorable cases, as will be shown in our example.

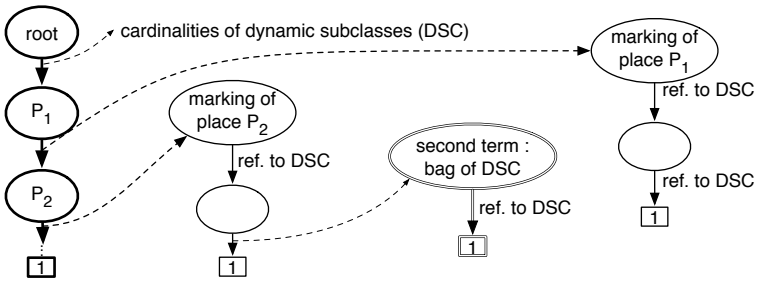


Figure 5.7: Principle of SNB symbolic markings encoding in SDD

Let us illustrate our encoding scheme with Figure 5.7. A symbolic marking is represented by:

- the identification of dynamic subclass cardinalities,

- the symbolic content of each place (as a cartesian product of dynamic subclasses).

Our encoding presents three levels. The first one (bold) corresponds to the structure of the SNB and lists its places. The second level (thin) is obtained from the arcs between the nodes encoding places and describe their symbolic marking. The third level (double thin) stands to encode bag tokens. As both second and third levels represent bags, they may share a given description (its interpretation is then handled by the homomorphism that operates on the structure).

Figure 5.8 shows an example of this capability for two partial markings (only places **waiting**, **warehouse** and **out** are represented) of the SaleStore example of Figure 5.4:

$$M_1 = waiting(P_1) + warehouse(G_0) + out(\langle P_0, \{G_1\}\rangle)$$
$$M_2 = waiting(P_1) + warehouse(G_1) + out(\langle P_0, \{G_0\}\rangle)$$

We assume that P_0 and P_1 are dynamic subclasses in *People* while G_0, G_1 are dynamic subclasses in *Gift*.

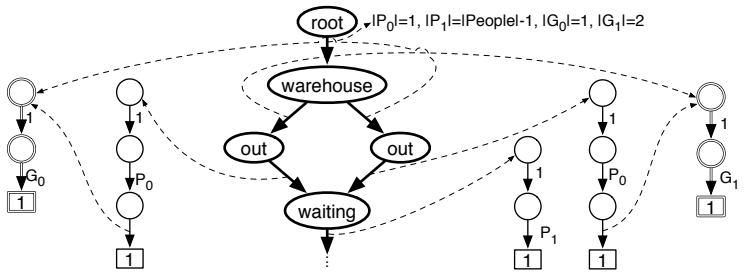


Figure 5.8: Example of encoding for two markings with shared parts

This figure shows the encoding of these two partial markings. The three levels of hierarchy are clearly visible. We also observe that the markings G_0 (and G_1) are shared at two different levels. The dotted path from the arcs below place **out** corresponds to a bag inside a bag, while from place **warehouse** they are simply a bag.

5.4 Assessment

Implementation

This encoding has been used in a prototype of a symmetry-reduced state space generator for SNB, named Crocodile [CBKTM11]. Written in C++, its first version relies on `libddd`. A second version, that better handles several features of SNB has been developed in 201, and uses premises of the operations defined in the previous chapter.

Tools

GreatSPN [Gre11]. GreatSPN has become the reference implementation of the quotient state graph for SN. We only use the SN features of GreatSPN, but it also handles stochastic nets[1]. It computes the quotient state space of the input SN, using symbolic markings to represent orbits. It is a mature tool that has been developed for over than twenty years. It relies on explicit data structures to store the state space. GreatSPN also computes the size of the non-reduced state space.

[1] Well-Formed Petri Nets [CDFH91] introduce stochastic features that are not embedded in SNB.

Crocodile [CBKTM11]. As explained above, Crocodile computes the quotient state space, representing orbits with symbolic markings. It relies on SDD and DDD (through the `libddd`), using the encoding described above. Since SNB encompass SN, it handles both formalisms.

Model

We compare the performance of both tools for the computation of the quotient state space of the model SaleStore described in Section 5.1. We use several values for the two scaling parameters (size of types *People* and *Gift*). The SNB of Figure 5.4 was processed by Crocodile, while its corresponding SN was processed by both Crocodile and GreatSPN. This enables a separate evaluation of the gain brought by the encoding compared to the one coming from the use of bags in tokens.

Executions were operated on a 32bits 3.2GHz Intel processor machine with 3Gbyte of memory, running Linux.

Results and Discussion

Table 5.1 summarizes the collected informations. Columns show:

- P, the size of class *People*,
- G, the size of class *Gift*,
- the size (number of nodes) for both the quotient state graph and the state space (*i.e.* the corresponding non-reduced state space),
- for SNB (with Crocodile) and SN (with both Crocodile and GreatSPN): the number of firings performed to build the quotient state graph, consumed memory[2] and execution time in seconds.

We first note that, for all instances, the two tools and the two versions – SN and SNB – agree on the size of the quotient state space.

The main difference between the SNB and its unfolded SN is the number of symbolic transitions. Transition instantiations with bags avoid redundancy, and lead to the analysis of less successors, thus less canonizations (as illustrated in Table 5.1 and Figure 5.9).

Table 5.1 also shows the very low number of firings that Crocodile needs to build the SNB quotient state compared to GreatSPN for the SN one. Working on explicit data structures, the canonization algorithm in GreatSPN must be repeated for each new discovered state, whereas Crocodile factorizes out some of the computation when canonization several states at a time. This has a dramatic impact on GreatSPN execution. This impact can be noticed in Figure 5.9a that shows execution time for $|People| = 6$ and $|Gift|$ varying from 2 to 15.

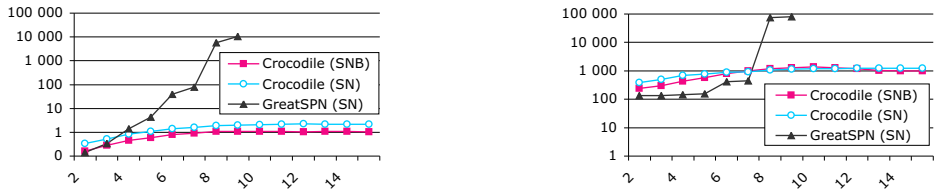
The benefits of the decision diagram based representation is also highlighted when comparing GreatSPN and Crocodile running on SN. For small values of P and G, shared parts of the quotient state graph are not sufficient to compensate the initial cost of the decision diagram structure, this becomes false when G grows larger than 8 in Table 5.1, thus leading to consequent gain in memory usage. This is also visible in Figure 5.9b that shows the evolution of memory consumption for $|People| = 6$ and $|Gift|$ varying from 2 to 15.

Due to the combinatorial explosion of firings, GreatSPN runs out of memory when G is greater than 9. However, Crocodile is still able to process the example for large values such as

[2] MOVF means memory overflow (around 2.03 Gbytes on our experiment machine).

P	G	Number of Nodes		SNB (Crocodile)			SN (Crocodile)			SN (GreatSPN)		
		Quot. Graph	Ordinary Space	# of Fir.	Mem. (MB)	Time (s.)	# of Fir.	Mem. (MB)	Time (s.)	# of Fir.	Mem. (MB)	Time (s.)
5	6	116	6.7×10^{05}	285	0.50	0.4	361	0.70	0.7	10 430	0.41	17
5	7	120	2.8×10^{06}	296	0.60	0.4	377	0.75	0.7	60 850	0.43	30
5	8	124	1.1×10^{07}	303	0.68	0.4	388	0.78	0.8	99 920	71.96	2 041
5	9	125	4.2×10^{07}	305	0.65	0.4	392	0.78	0.9	3 497 661	77.63	3 128
5	10	126	1.5×10^{08}	306	0.57	0.4	394	0.78	0.9	—	MOVF	—
5	50	126	4.2×10^{16}	306	0.57	0.5	394	0.80	1.1	—	MOVF	—
6	6	180	4.1×10^{06}	496	0.79	0.7	652	0.88	1.2	28 612	0.42	40
6	7	190	2.0×10^{07}	527	0.98	0.8	695	0.93	1.3	146 775	0.45	78
6	8	200	9.2×10^{07}	550	1.20	0.9	730	1.09	1.6	289 301	73.86	5 858
6	9	204	4.2×10^{08}	561	1.28	0.9	745	1.13	1.7	10 589 107	79.53	10 565
6	10	208	1.9×10^{09}	568	1.36	1.0	756	1.18	1.8	—	MOVF	—
6	11	209	7.8×10^{09}	570	1.28	0.9	760	1.18	1.8	—	MOVF	—
6	12	210	3.1×10^{10}	571	1.18	0.9	762	1.23	1.9	—	MOVF	—
6	50	210	1.0×10^{19}	571	1.00	1.1	762	1.25	2.1	—	MOVF	—
7	6	260	2.3×10^{07}	744	1.28	1.2	967	1.24	2.0	64 531	0.45	84
7	7	280	1.2×10^{08}	811	1.58	1.5	1 052	1.41	2.4	304 504	0.88	197
7	8	300	6.7×10^{08}	864	1.91	1.8	1 127	1.62	2.9	680 594	75.75	12 024
7	9	310	3.2×10^{09}	896	2.14	2.0	1 168	1.73	3.2	22 553 532	81.42	23 548
7	10	320	1.8×10^{10}	919	2.15	2.1	1 200	1.99	3.5	—	MOVF	—
7	14	330	8.3×10^{12}	940	1.98	2.0	1 232	2.13	4.0	—	MOVF	—
7	100	330	4.2×10^{27}	940	1.81	2.7	1 232	2.17	4.9	—	MOVF	—
8	6	356	1.2×10^{08}	1 084	1.30	1.8	1 448	1.61	3.04	123 639	0.49	152
8	7	390	7.1×10^{08}	1 208	1.97	2.33	1 606	1.86	3.73	587 084	0.95	495
8	8	425	4.3×10^{09}	1 312	2.60	3.02	1 754	2.24	4.63	1 496 928	77.65	21 300
8	9	445	2.5×10^{10}	1 382	3.13	3.47	1 845	2.38	5.2	40 063 379	83.33	43 946
8	10	465	1.5×10^{11}	1 436	3.50	3.93	1 924	2.53	5.84	—	MOVF	—
8	16	495	2.9×10^{15}	1 511	3.57	4.02	2 032	2.65	6.97	—	MOVF	—
8	100	495	3.1×10^{31}	1 511	3.26	5.04	2 032	2.74	8.34	—	MOVF	—
20	40	10 626	3.2×10^{51}	41 529	1 402	6 017	57 051	1 150	5 423	—	MOVF	—

Table 5.1: Compared performance of Crocodile and GreatSPN on state space generation.



(a) Computation time in seconds

(b) Memory consumption in Kbyte

Figure 5.9: Memory and time measures for $|People| = 6$ and $|Gift|$ varying from 2 to 15

20 peoples with 40 gifts (last line in the table, asymptot point in the quotient state space for this systems configuration).

Let us notice two points for this model. First, the combinatorial explosion dramatically increases every two increments of G. This is due to the maximum bound of gifts to be bought (see guard in transition **shopping**, that bounds this value to 2). This increase can be directly observed in the charts of Figure 5.9. Second, the number of symbolic markings stabilizes when $|Gift| \geq 2 \times |People|$. When $|Gift|$ is just below the stabilization value ($2 \times |People|$), the SDD structure is not fully dense; beyond this value, the sharing in the SDD structure is maximized. Reaching this point of maximal sharing results in a slight decrease of memory consumption for Crocodile.

State Space Analysis

So far, Crocodile provides analysis of reachability properties. Such properties are constraints that can be checked during state space generation. This does not bring extra complexity (just a constant due to the property evaluation). Evaluation of a reachability property is done using the following scheme:

- translation of the property into constraints c on the symbolic markings (expressed as a SDD),
- for each new symbolic state s, comparing the canonical representation of s with c (since both are SDD, this is a fast operation).

Once a state verifying the property is found, the tool must reexecute the state space generation algorithm to store the list of symbolic firings leading to the identified state. Thus, verification of a reachability property may lead to building the state space twice in the worst case.

Another possibility is to revert the transition relation as explained in the previous chapter. For SNB however, transition instantiations, especially with bags, would require much technical work to properly and efficiently invert the transition relation and was beyond the scope of the thesis.

5.5 Conclusion

We have described that the symmetry reduction on top of decision diagrams can also be operated in a specific framework, that does not perfectly fit our assumptions in Section 2.4. It shows that this combination can be implemented in various contexts.

Through the experiments, we have demonstrated that our method applied to SN and SNB is a technique that can outperform state-of-the-art tools, even when they are tuned towards this specific formalism.

Chapter 6

Conclusion and Perspectives

En toute bonne chose, il faut considérer la fin.[1]

Le Renard et le Bouc (1668)
Jean de la Fontaine

Summary

In this work, we have dealt with the combination of the symmetry reduction with decision diagrams towards efficient model checking. This combination is motivated by the struggle against the combinatorial explosion that arises in model checking.

Both techniques have been independently quite well studied, and have shown their usefulness for model checking of distributed systems. On the one hand, symmetries often arise for such systems, and allow to build a reduced state space with the same properties as of the original one. This reduction fights directly the explosion by allowing the verification process to occur on a smaller state space. On the other hand, decision diagrams yield compact representation of large sets of states, and are especially well-suited for globally asynchronous locally synchronous (GALS) systems, which is often the case for distributed systems. This compact representation handles better the explosion by optimizing the usage of the physical memory during the verification process.

Combining Symmetries and Decision Diagrams. The combination of both techniques still raises some problems. Specifically, since decision diagrams represent sets of states compactly, it is necessary to design algorithms that handle sets as input. State-of-the-art algorithms for symmetry reduction rather rely on a single state input. This first problem is addressed by our first contribution (Chapter 3), as an algorithm fitted for decision diagrams to compute lexicographical leader strings. This algorithm relies on a subset of the symmetries H that is used to successively reduce the input states until stabilization. The algorithm always computes a representative function that can be used to build a reduced state space, although it may not be canonical. Depending on the choice of H, various degrees of reduction can be achieved, rendering the algorithm quite flexible. We give conditions on H to ensure maximal reduction: the $<$-*monotonic* condition. It is also proven that the size of $<$-*monotonic* cannot be bounded by a polynomial. This negative result is however balanced by the fact that $<$-*monotonic* sets of reasonable size exist for common symmetry groups. Implementation of our flexible algorithm on top of decision diagrams in terms of homomorphisms is also discussed, and an optimized implementation is proposed. Its efficiency is then demonstrated experimentally.

[1] In every matter we should mind the end.

New Operations for Decision Diagrams. Our second contribution (Chapter 4) goes beyond and describes how to efficiently implement set-based algorithms on decision diagrams through our algorithm *EquivSplit*. It relies on the computation of finer and finer equivalence classes over the input set, during the successive evaluation of a syntactical expression. Thus, the elements of the input set are not individualized, so as to take advantage of the underlying symbolic data structures. It happens to be well-suited to implement high-level transition relations, both forward and backward, of distributed systems by decision diagrams. We also apply them for the efficient swapping of two variables in a decision diagrams, thus implementing the transpositions that we use to encode our first contribution. Implemented in our model checking tool suite, we demonstrate its efficiency and competitiveness against other symbolic and non-symbolic state-of-the-art tools (GreatSPN).

Application to a Specific Formalism. Finally, all the concepts described in the thesis are applied to a specific formalism: Symmetric Nets with Bags (Chapter 5). Although it does not perfectly fit our original hypotheses, we demonstrate that the commonly used representation for orbits can also be computed with our algorithms. Most computations in this framework rely on the algorithmic concept of *EquivSplit*. Whereas the two previous contributions focus on Data Decision Diagrams, this application uses Hierarchical Set Decision Diagrams (SDD). Once again, the implementation of our algorithms shows to be quite efficient and able to outperform the state-of-the-art tool for this formalism.

Perspectives

Application to Hierarchical Decision Diagrams. Although our algorithms are implementable on any kind of decision diagrams, we have only tested Data Decision Diagrams and, to a lesser extent, Set Decision Diagrams. The *EquivSplit* algorithm for instance has not yet been extended to SDD. On SDD, *EquivSplit* should associate sets of values to each equivalence class. This does not fit the applications we had in mind, namely the transition relation of BEEM models, or permutations of variables. This lack of application refrained us from extending our algorithms to SDD, although there is no theoretical obstacle to this SDD implementation.

Building this SDD extension raises other problems, such as the variable ordering, so as to take advantage of the hierarchy. Previous work in the research of hierarchical ordering for SDD [HKPAE12] showed the very high cost of non-locality in this setting.

Guarded-Action Language. The algorithm *EquivSplit* evaluates a syntactical expression on a decision diagram. In order to use this algorithm for the evaluation of high-level transition relations, we have designed an intermediate language, acting as a semantic assembler for decision diagrams (see Section 4.5): the Guarded-Action Language (GAL). When translating benchmark models to GAL, we have noticed that a sequence of assignments often leads to unnecessary intermediate computations. A common example is a conditional statement (say `x<0?x++:x--`), that occurs in the sequence after an assignment to variable `x`. The value assigned to `x` in this assignment could be propagated to the conditional statement, so as to simplify it as soon as possible, without need to read the value of `x` once more. This is a single example of many situations we have identified in the BEEM benchmark alone.

We think there is room here for syntactical, static simplification, to occur before the translation to homomorphisms, in order to avoid such unnecessary intermediate computations. Such simplifications should be ideally based on GAL, or a proper adaptation/extension of it, and rely on techniques from formal calculus, and/or code optimization from the theory of compilation.

Further steps can considered, such as to use this simplification engine dynamically; this may however conflict with the memoization mechanisms.

Symmetry Detection. Another question has not been addressed in our work: the detection of symmetries. The main limits to our experiments was the detection of the symmetries of the systems, done by hand. An automatic detection of symmetries would be preferable. The symmetries can be quite straightforwardly computed directly on the full state space. Due to the combinatorial explosion, this method is rarely a good idea. One should rather detect the symmetries directly from the model. Several works towards this automatic detection in various settings exist: Symmetric Nets [TMDM03], graphs [Mat79], constraint satisfactory problems [Pug05], Promela [DM05] ... The main method consists in building a graph from the system, whose symmetries correspond to symmetries of the system. Therefore, the theoretical complexity of symmetry detection is strongly related to Graph Isomorphism. Although in general manageable, this complexity is to be understood in terms of the size of the built graph, whose size may be exponential in the size of the model. Furthermore, it is quite hard to design a graph whose symmetries capture *exactly* all the symmetries of the graph. It often happens that the size of the graph is related to the exhaustivity of the symmetry detection: the bigger the size, the more complete the detection. The extreme case corresponds to the detection of symmetries directly on the state space, as discussed above.

Bibliography

[AHI98] Khalil Ajami, Serge Haddad, and Jean-Michel Ilié. Exploiting symmetry in linear time temporal logic model checking: One step beyond. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 52–67, 1998.

[AMO99] Mark D. Aagaard, Thomas F. Melham, and John W. O'Leary. Xs are for trajectory evaluation, booleans are for theorem proving. In *Correct Hardware Design and Verification Methods*, pages 202–218. Springer, 1999.

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Model Checking Software*, pages 146–162. Springer, 2006.

[AMS03] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 116–119. ACM, 2003.

[BBvR10] Jiří Barnat, Luboš Brim, Milan Češka, and Petr Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC)*, pages 4–7. IEEE, 2010.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320. ACM, 1999.

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for CTL. In *Logic in Computer Science, 1995. LICS'95. Proceedings., Tenth Annual IEEE Symposium on*, pages 388–397. IEEE, 1995.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} States and beyond. *Information and computation*, 98(2):142–170, 1992.

[BDHI05] Souheib Baarir, Claude Dutheillet, Serge Haddad, and Jean-Michel Ilié. On the use of exact lumpability in partially symmetrical well-formed nets. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 23–32. IEEE, 2005.

[BHR84] Stephen D. Brookes, Charles A.R. Hoare, and Andrew W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599, 1984.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *Computer Aided Verification*, pages 403–418. Springer, 2000.

[BL83] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 171–183. ACM, 1983.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[BTY97] Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. On-the-fly symbolic model checking for real-time systems. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 25–34. IEEE, 1997.

[BvdPW10] Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and symbolic reachability. In *Computer Aided Verification*, pages 354–359. Springer, 2010.

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *Computers, IEEE Transactions on*, 45(9):993–1002, 1996.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM (JACM)*, 30(2):323–342, 1983.

[CBKTM11] Maximilien Colange, Souheib Baarir, Fabrice Kordon, and Yann Thierry-Mieg. Crocodile: a symbolic/symbolic tool for the analysis of symmetric nets with bags. *Applications and Theory of Petri Nets*, pages 338–347, 2011.

[CBKTM13] Maximilien Colange, Souheib Baarir, Fabrice Kordon, and Yann Thierry-Mieg. Towards Distributed Software Model-Checking using Decision Diagrams. In *25th International Conference on Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 830–845. Springer Verlag, July 2013.

[CDFH90] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. On well-formed coloured nets and their symbolic reachability graph. In *11th International Conference on Application and Theory of Petri Nets*. Springer, 1990.

[CDFH91] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. Stochastic well-formed coloured nets and multiprocessor modelling applications. In *High-level Petri Nets*, pages 504–530. Springer, 1991.

[CDFH97] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. A symbolic reachability graph for coloured Petri nets. *Theoretical Computer Science*, 176(1-2):39–65, 1997.

[CE81] Edmund M. Clarke and Allen E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY*, volume 131 of *LNCS*. Springer-Verlag, 1981.

[CEFJ96] Edmund M Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1):77–104, 1996.

[CEJS98] Edmund M Clarke, E. Emerson, S. Jha, and A. Sistla. Symmetry reductions in model checking. In *Computer Aided Verification*, pages 147–158. Springer, 1998.

[CEPA+02] Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data decision diagrams for Petri net analysis. *Application and Theory of Petri Nets 2002*, pages 129–158, 2002.

[CES86] Edmund M. Clarke, E. Allen Emerson, and Prasad A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CGLR96] James Crawford, Matthew Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning -International Conference-*, pages 148–159. Morgan Kaufmann Publishers, 1996.

[CGP99] Edmund M Clarke, Orna Grumberg, and D.A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CHKP12] Maximilien Colange, Lom Messan Hillah, Fabrice Kordon, and Pierre Parutto. Extreme Symmetries in Complex Distributed Systems: the Bag-Oriented Approach. In *Decvelopment, Operation and Management of Large-Scale Complex IT Systems, 17th Monterey Workshop, Revised Selected Papers*, volume 7539 of *LNCS*, pages 330–352. Springer, 2012.

[CKTMB12] Maximilien Colange, Fabrice Kordon, Yann Thierry-Mieg, and Souheib Baarir. State Space Analysis using Symmetries on Decision Diagrams. In *12th International Conference on Application of Concurrency to System Design (ACSD'2012)*, pages 164–172, Hamburg, Germany, June 2012. IEEE Computer Society.

[CMS03] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. Saturation unbound. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 379–393. Springer Berlin Heidelberg, 2003.

[CMZ+93] Edmund M. Clarke, Kenneth L. McMillan, Xudong Zhao, Masahiro Fujita, and Jerry Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Design Automation, 1993. 30th Conference on*, pages 54–60. IEEE, 1993.

[Cou90] Jean-Michel Couvreur. The general computation of flows for coloured nets. In *11th International Conference on Application and Theory of Petri Nets*, pages 204–223. Springer, 1990.

[CP95] Søren Christensen and Laure Petrucci. Modular state space analysis of coloured Petri nets. In Giorgio De Michelis and Michel Diaz, editors, *Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 201–217. Springer, 1995.

[CTM05] Jean-Michel Couvreur and Yann Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. *Formal Techniques for Networked and Distributed Systems-FORTE 2005*, pages 443–457, 2005.

[CVWY93] Costas Courcoubetis, Moshe Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Computer-Aided Verification*, pages 129–142. Springer, 1993.

[DM05] Alastair F. Donaldson and Alice Miller. Automatic symmetry detection for model checking using computational group theory. In *FM 2005: Formal Methods*, pages 481–496. Springer, 2005.

[EHT00] E Allen Emerson, John W Havlicek, and Richard J Trefler. Virtual symmetry reduction. In *Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on*, pages 121–131. IEEE, 2000.

[EJP97] E Allen Emerson, Somesh Jha, and Doron Peled. Combining partial order and symmetry reductions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 19–34. Springer, 1997.

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal methods in system design*, 9(1-2):105–131, 1996.

[Esp97] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[ET99] E. Emerson and R. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. *Conference on Correct Hardware Design and Verification Methods*, pages 142–156, 1999.

[EW03] E. Emerson and T. Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. *Correct Hardware Design and Verification Methods*, pages 216–230, 2003.

[FFK+01] Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Vardi, and Zijiang Yang. Is there a best symbolic cycle-detection algorithm? In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 420–434. Springer, 2001.

[FHL80] Merrick Furst, John Hopcroft, and Eugene M. Luks. Polynomial-time algorithms for permutation groups. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 36–41. IEEE, 1980.

[Gre11] GreatSPN. Petri nets suite. `http://www.di.unito.it/~greatspn`, 2011.

[HIA00] Serge Haddad, Jean-Michel Ilié, and Khalil Ajami. A model checking method for partially symmetric systems. In Tommaso Bolognesi and Diego Latella, editors, *Proceedings of IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'XIII) and Protocol Specification, Testing and Verification (PSTV'XX)*, volume 183 of *IFIP Conference Proceedings*, pages 121–136, Pisa, Italy, oct 2000. Kluwer Academic Publishers.

[HITZ95] Serge Haddad, Jean-Michel Ilié, Mohamed Taghelit, and Belhassen Zouari. Symbolic reachability graph and partial symmetries. In *Application and Theory of Petri Nets 1995*, pages 238–257. Springer, 1995.

[HKP+09] Serge Haddad, Fabrice Kordon, Laure Petrucci, Jean-François Pradat-Peyre, and Nicolas Treves. Efficient state-based analysis by introducing bags in Petri nets color domains. In *American Control Conference, 2009. ACC'09.*, pages 5018–5025. IEEE, 2009.

[HKPAE12] Silien Hong, Fabrice Kordon, Emmanuel Paviot-Adet, and Sami Evangelista. Computing a hierarchical static order for decision diagram-based representation from P/T nets. In *Transactions on Petri Nets and Other Models of Concurrency V*, pages 121–140. Springer, 2012.

[Hoa69] Charles A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] Charles A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HTMK08] Alexandre Hamez, Yann Thierry-Mieg, and Fabrice Kordon. Hierarchical set decision diagrams and automatic saturation. *Applications and Theory of Petri Nets*, pages 211–230, 2008.

[Jen87] Kurt Jensen. Coloured Petri nets. *Petri nets: central models and their properties*, pages 248–299, 1987.

[Jun03] Tommi Junttila. *On the symmetry reduction method for Petri Nets and similar formalisms.* PhD thesis, Helsinki University of Technology, Espoo, Finland, 2003.

[LR04] Eugene M. Luks and Amitabha Roy. The complexity of symmetry-breaking formulas. *Annals of Mathematics and Artificial Intelligence*, 41(1):19–45, 2004.

[LS92] Yung-Te Lai and Sarma Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 608–613. IEEE Computer Society Press, 1992.

[Mat79] Rudolf Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–136, 1979.

[MCP10] Radu Muschevici, David Clarke, and Jose Proenca. Feature Petri nets. In *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)*, volume 2, 2010.

[Min93] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Design Automation, 1993. 30th Conference on*, pages 272–277. IEEE, 1993.

[MoV13] MoVe team. The `libddd` home page. `http://move.lip6.fr/software/DDD`, 2013.

[NM94] Masato Notomi and Tadao Murata. Hierarchical reachability graph of bounded Petri nets for concurrent-software analysis. *Software Engineering, IEEE Transactions on*, 20(5):325–336, 1994.

[Pap03] Christos H. Papadimitriou. *Computational complexity.* John Wiley and Sons Ltd., 2003.

[PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer aided verification*, pages 377–390. Springer, 1994.

[Pel07] Radek Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *Model Checking Software, 14th Int'l SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, 1962.

[Pug05] Jean-François Puget. Automatic detection of variable and value symmetries. In *Principles and practice of constraint programming-CP 2005*, pages 475–489. Springer, 2005.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[RAB+95] Rajeev K. Ranjan, Adnan Aziz, Robert K. Brayton, Bernard Plessier, and Carl Pixley. Efficient BDD algorithms for FSM synthesis and verification. *IWLS95, Lake Tahoe, CA*, 253:254, 1995.

[SG04] A. Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(4):702–734, 2004.

[SG12] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, release 12/10/06. http://www.eecs.berkeley.edu/~alanmi/abc/, 2012.

[SH09] Martin Schwarick and Monika Heiner. CSL model checking of biochemical networks with interval decision diagrams. In *Computational Methods in Systems Biology*, pages 296–312. Springer, 2009.

[SHMB90] Arvind Srinivasan, T. Ham, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 92–95. IEEE, 1990.

[Sim71] Charles C. Sims. Computation with permutation groups. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 23–28. ACM, 1971.

[SRB02] Fabio Somenzi, Kavita Ravi, and Roderick Bloem. Analysis of symbolic SCC hull algorithms. In *Formal Methods in Computer-Aided Design*, pages 88–105. Springer, 2002.

[ST98] Karsten Strehl and Lothar Thiele. Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 686–692. ACM, 1998.

[TMDM03] Yann Thierry-Mieg, Claude Dutheillet, and Isabelle Mounier. Automatic symmetry detection in well-formed nets. In *Applications and Theory of Petri Nets 2003*, pages 82–101. Springer, 2003.

[TMIP04] Yann Thierry-Mieg, Jean-Michel Ilié, and Denis Poitrenaud. A symbolic symbolic state space representation. *Formal Techniques for Networked and Distributed Systems–FORTE 2004*, pages 276–291, 2004.

[TMPHK09] Yann Thierry-Mieg, Denis Poitrenaud, Alexandre Hamez, and Fabrice Kordon. Hierarchical set decision diagrams and regular models. *Tools and Algorithms for the Construction and Analysis of Systems*, 5505:1–15, 2009.

[Uni12] Universität Rostock. The `LoLa` home page. `http://www.informatik.uni-rostock.de/tpp/lola/`, 2012.

[Val93] Antti Valmari. *Compositional state space generation*. Springer, 1993.

[VAM96] François Vernadat, Pierre Azéma, and François Michel. Covering step graph. *Application and theory of Petri nets 1996*, pages 516–535, 1996.

[WG93] Pierre Wolper and Patrice Godefroid. Partial-order methods for temporal verification. In *CONCUR'93*, pages 233–246. Springer, 1993.

Index