Faculté des Sciences
Département d'Informatique

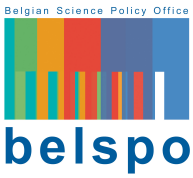
Université Libre de Bruxelles

New Algorithms and Data Structures for the Emptiness Problem of Alternating Automata

Nicolas Maquet

Thèse présentée en vue de l'obtention du grade de Docteur en Sciences

Directeur: Prof. Jean-François Raskin



New Algorithms and Data Structures for the Emptiness Problem of Alternating Automata

Nicolas Maquet

Thèse réalisée sous la direction du Professeur Jean-François Raskin et présentée en vue de l'obtention du grade de Docteur en Sciences. La défense publique de cette thèse a eu lieu le 3 mars 2011 à l'Université Libre de Bruxelles, devant le jury composé de:

- Parosh Aziz Abdulla (Uppsala University, Suède)
- Bernard Boigelot (Université de Liège, Belgique)
- Véronique Bruyère (UMons, Belgique)
- Jean Cardinal (Université Libre de Bruxelles, Belgique)
- Gilles Geeraerts (Université Libre de Bruxelles, Belgique)
- Jean-François Raskin (Université Libre de Bruxelles, Belgique)

Résumé

Ce travail traite de l'étude de nouveaux algorithmes et structures de données dont l'usage est destiné à la *vérification de programmes*. Les ordinateurs sont de plus en plus présents dans notre vie quotidienne et, de plus en plus souvent, ils se voient confiés des tâches de nature *critique pour la sécurité*. Ces systèmes sont caractérisés par le fait qu'une panne ou un *bug* (erreur en jargon informatique) peut avoir des effets potentiellement désastreux, que ce soit en pertes humaines, dégâts environnementaux, ou économiques. Pour ces systèmes critiques, les concepteurs de systèmes industriels prônent de plus en plus l'usage de techniques permettant d'obtenir une *assurance formelle* de correction.

Une des techniques de vérification de programmes les plus utilisées est le *model checking*, avec laquelle les programmes sont typiquement *abstraits* par une machine a états finis. Après cette phase d'abstraction, des *propriétés* (typiquement sous la forme d'une formule de logique temporelle) peuvent êtres vérifiées sur l'abstraction à espace d'états fini, à l'aide d'outils de vérification automatisés. Les *automates alternants* jouent un rôle important dans ce contexte, principalement parce que plusieurs logiques temporelles peuvent êtres traduites efficacement vers ces automates. Cette caractéristique des automates alternants permet de réduire le model checking des logiques temporelles à des questions sur les automates, ce qui est appelé *l'approche par automates du model checking*.

Dans ce travail, nous étudions trois nouvelles approches pour l'analyse (le test du vide) des automates alternants sur mots finis et infinis. Premièrement, nous appliquons l'approche par antichaînes (utilisée précédemment avec succès pour l'analyse d'automates) pour obtenir de nouveaux algorithmes pour les problèmes de satisfaisabilité et du model checking de la logique temporelle linéaire, via les automates alternants. Ces algorithmes combinent l'approche par antichaînes avec l'usage des ROBDD, dans le but de gérer efficacement la combinatoire induite par la taille exponentielle des alphabets d'automates générés à partir de LTL. Deuxièmement, nous développons de nouveaux algorithmes d'abstraction et raffinement pour les automates alternants, combinant l'usage des antichaînes et de l'*interprétation abstraite*, dans le but de pouvoir traiter efficacement des automates de grande taille. Enfin, nous définissons une nouvelle structure de données, appelée LVBDD[1], qui permet un encodage efficace des fonctions de transition des automates alternants sur alphabets symboliques. Tous ces travaux ont fait l'objet d'implémentations et ont été validés expérimentalement.

[1] Lattice-Valued Binary Decision Diagrams

Abstract

This work studies new algorithms and data structures that are useful in the context of *program verification*. As computers have become more and more ubiquitous in our modern societies, an increasingly large number of computer-based systems are considered *safety-critical*. Such systems are characterized by the fact that a failure or a *bug* (computer error in the computing jargon) could potentially cause large damage, whether in loss of life, environmental damage, or economic damage. For safety-critical systems, the industrial software engineering community increasingly calls for using techniques which provide some *formal assurance* that a certain piece of software is correct.

One of the most successful program verification techniques is *model checking*, in which programs are typically *abstracted* by a finite-state machine. After this abstraction step, *properties* (typically in the form of some temporal logic formula) can be checked against the finite-state abstraction, with the help of automated tools. *Alternating automata* play an important role in this context, since many temporal logics on words and trees can be efficiently translated into those automata. This property allows for the reduction of model checking to automata-theoretic questions and is called the *automata-theoretic approach to model checking*.

In this work, we provide three novel approaches for the analysis (emptiness checking) of alternating automata over finite and infinite words. First, we build on the successful framework of *antichains* to devise new algorithms for LTL satisfiability and model checking, using alternating automata. These algorithms combine antichains with reduced ordered binary decision diagrams in order to handle the exponentially large alphabets of the automata generated by the LTL translation. Second, we develop new abstraction and refinement algorithms for alternating automata, which combine the use of antichains with *abstract interpretation*, in order to handle ever larger instances of alternating automata. Finally, we define a new symbolic data structure, coined *lattice-valued binary decision diagrams* that is particularly well-suited for the encoding of transition functions of alternating automata over symbolic alphabets. All of these works are supported with experimental evaluations that confirm the practical usefulness of our approaches.

Acknowledgements

First and foremost, I would like to show my gratitude to Jean-François Raskin, my thesis advisor. His technical excellence, his rigor, and his sheer enthusiasm are perhaps the qualities that Jean-François is best known for among his peers. As one of his doctoral students, I would like to testify also of his human qualities that have made these four years at his side not only instructive and fruitful, but also enjoyable and fulfilling. Most of all, I thank him for his his patience, his availability, and his daily cheerfulness.

Next, I want to heartily thank my coauthors. These are, in no particular order, Laurent Doyen, Martin De Wulf, Gilles Geeraerts, Pierre Ganty, Gabriel Kalyon and Tristan Le Gall. The research material that went into this Ph.D. thesis is very much the fruit of a collaborative work, and I'm very thankful for their work and support.

I would like to thank all my colleagues form the Computer Science Department and the Verification Group, for making the department a lively and welcoming place. Special thanks to my faithful office neighbor, Frédéric Servais, for his biting humor and support, and to Emmanuel Filiot for being the most Belgian French I have ever met. A big thank you also to the three secretaries of the Department: Maryka Peetroons, Véronique Bastin and Pascaline Browaeys.

I sincerely thank all the members of the thesis' jury comittee: Parosh Aziz Abdulla, Bernard Boigelot, Véronique Bruyère, Jean Cardinal, Gilles Geeraerts and Jean-François Raskin, for their careful reading of this work, and for suggesting improvements.

Finally, I would like to express my deepest gratitude to all of those who have offered me their love and friendship over the years. In the end, I am what they give me, more than anything else, for which I will be forever grateful. In particular, and most importantly, I would like to thank Amandine, for her continued love and support.

À mes frangin(e)s[2]

[2]Comme d'autres, plus sages, l'ont dit avant moi: ça leur fera une belle jambe ...

Illustration: Caroll, Lewis: "Alice's Adventures in Wonderland" (1865)

Contents

Chapter 1

Introduction

Today, computers are ubiquitous, and are a necessary tool of modern life. It is quite amazing to realize how quickly and radically computers have changed the way we live. The computers of today do an incredibly vast range of tasks for us: they help us communicate with each other, they help us pay our bills, they help us brake more efficiently on wet roads. But computers not only make what *was possible* easier, more importantly they also allow to perform tasks which, not too long ago, *were considered impossible*. Without computers, modern medicine would not be possible. Without computers, modern physics would not be possible. The list goes on and on.

1.1 Context

In the last 40 years or so, our society has become increasingly dependent on computers and their programs. Also, computers are more and more entrusted with *safety-critical* tasks such as guiding airplanes and rockets, controlling medical and industrial equipment, etc. When the cost of a potential software failure (whether human, environmental, or economic) becomes prohibitive, the question of *software correctness* becomes highly important. There have been many well-publicized stories of catastrophic software failures leading to the loss of lives and / or enormous economic damage. Traditional software engineering rely on *testing* to remove bugs (programming

mistakes, design flaws) from programs. However, testing can only detect the *presence* of errors in a computer program, but it cannot show that the program is *correct* in any formal sense. More and more, certain applications call for some *formal assurance* that a piece of software meets a specification written in a formal language.

Since the early works of Robert Floyd and Tony Hoare in the late 1960's, the field of *program verification* has gathered much research effort. The ultimate goal of this discipline is to devise *proofs* for the correctness of programs. However, it has become quickly apparent that finding such proofs is extremely hard, usually much more so than writing the program itself! Moreover, program proofs are often very long and tedious, which makes them very difficult to compute by hand. For those reasons, a lot of research has been put into the development of *automatic methods* for software verification. The idea is to use computer programs to find and / or validate proofs of correctness of other programs.

The field of computer-aided verification is faced with a fundamental *undecidability* problem. In general, checking whether a computer program satisfies its specification is undecidable. In fact, the Rice theorem tells us that all non trivial properties of computer programs are undecidable. *Computer-aided verification techniques* sidestep this undecidability barrier by exploiting *sound abstractions* which *over-approximate* the behavior of the original program. The idea is to substitute the concrete program with an abstract, more simple version, in such a way that any property that holds on the abstraction also holds on the concrete program. Note that this technique is sound but not complete in general, so there is no contradiction with the general undecidability of program verification.

A particularly fruitful and popular abstraction of computer programs is the *finite state machine*, or *finite automaton*. Finite-state abstractions are useful for modeling programs and properties where the intricacies of the system lies more in the control-state combinatorial rather than in the manipulation of complex data. Many systems fall into this category, such as network protocols, distributed control systems, embedded or reactive systems, and so on. The field of *model checking* encompasses the study of

efficient algorithms and data structures for the analysis of finite-state systems. In this context, a program is represented by a finite-state abstraction $\mathcal{M}$ that describes a *language* of *possible behaviors*, denoted $L(\mathcal{M})$. On the other hand, a *property* φ is written down in some formal logic (temporal or otherwise) that describes a language of *correct behaviors*, denoted $L(\varphi)$. For a system $\mathcal{M}$ and a property φ the *model checking problem* asks whether $\mathcal{M} \models \varphi$, that is whether $L(M) \subseteq L(\varphi)$. If all the behaviors of a sound approximation of a program $\mathcal{P}$ are included in those defined by a property φ, then we say that $\mathcal{P}$ is *provably-correct* with respect to φ.

The primary challenge faced by the model checking community is the *state space explosion problem*. Finite-state abstractions of real-world programs typically have a very large number of states, and much research effort has been put into dealing with this challenge.

In this work, we develop new algorithms for the analysis of *alternating automata* over finite and infinite-words. The concept of *alternation* for automata was first studied by Chandra and Stockmeyer in 1976. In their seminal work, the authors define the *alternating Turing machine*, as a generalization of non-deterministic Turing machines, whose set of states is partitioned between *existential* and *universal*. In an existential state, such a machine makes a non-deterministic guess to select the next state. In a universal state, however, the machine launches as many copies of itself as there are successor states, and accepts only if each of these "computation branches" are accepting. The motivation for studying alternating Turing machines originally came from research in computational complexity. Chandra and Stockmeyer show, among other results, that the class of problems that can be solved in *polynomial space* is the same than the class of problems that can be solved in *alternating polynomial time* (i.e., PSpace = APTime).

Similarly to alternating Turing machines, *alternating automata* generalize non-deterministic automata. Alternating automata have been defined on finite and infinite-words and on finite and infinite trees. The use of alternating automata is a key component of the *automata-theoretic approach* to model checking. In this approach, the property φ that need to be verified is

translated into an automaton, which reduces the model checking problem to automata-theoretic questions. One of the key strengths of alternating automata is that several temporal logics can be translated to alternating automata in *linear-time*. Efficient decision procedures for those automata are thus very desirable, particularly in the field of model checking.

1.2 Contributions

The purpose of this thesis is to find new efficient algorithms for the analysis of alternating automata. Our aim is to develop algorithms which analyze alternating automata directly, without resorting to an explicit and costly translation to non-deterministic automata. We summarize here the main contributions of this work.

In recent research [DDHR06, DR07, DR10], Raskin et al. have developed new algorithms for the analysis of finite automata which are based on *antichains*. The idea of antichains is to exploit simulation preorders that exist by construction on the state spaces of exponential subset constructions. Many such simulation preorders can be found in automata-theoretic problems and can be used to facilitate the evaluation of fixed-point expressions based on the pre and post operators. In this work, we apply the framework of antichains to the *satisfiability* and *model checking* problems of the linear temporal logic (LTL, for short) over symbolic Kripke structures[1]. To perform the analysis of LTL, we use the linear-time translation to alternating automata to reduce these problems to language-emptiness questions on automata. However, the translation from LTL to alternating automata creates automata with alphabets of exponential size. To efficiently handle the combinatorial entailed by the exponential alphabets, we combine *reduced ordered binary decision diagrams* [Bry86] (ROBDD, for short) with antichains in a new set of algorithms. Note that the use of antichains and ROBDD for LTL satisfiability is only useful in a heuristic sense. The problem being PSpace-Complete, our algorithms cannot strictly improve on the existing theoretically optimal solutions. Therefore, in order to validate

[1]Symbolic Kripke structures are, in brief, symbolically-encoded finite-state machines.

our approach, we demonstrate its efficiency with empirical evaluations on a number of meaningful benchmarks.

Our second contribution is the development and implementation of an abstraction / refinement framework for alternating automata. The motivation for this work was to investigate how the use of *abstract interpretation* [CC77], together with the framework of *antichains* could yield more efficient algorithms for alternating automata. Our framework uses the set of *partitions* of the set of states of an alternating automaton as basis for the abstraction. This work has a number of interesting characteristics. First, our refinement is not based on *counter-examples* as is usually the case [CGJ+03] but rather builds on recent research which uses fixed-point guided refinement instead [CGR07, RRT08]. Second, contrary to other refinement approaches, our abstract domain does not need to be *strictly refined* at each successive abstraction / refinement step. Rather, our algorithm always keeps the abstract domain *as coarse as possible* with respect to its current knowledge of the system. Finally, we have implemented our approach and compared its efficiency on a number of benchmark examples. Our experimental evaluations reveal that, while the overhead of the abstraction refinement is not negligible, it can greatly outperform concrete antichain algorithms when the number of locations in the alternating automaton grows very large.

Our final contribution is the definition and study of an new symbolic data structure, coined *lattice-valued binary decision diagrams* (LVBDD, for short) for the efficient representation of functions of the form $2^{\mathbb{P}} \mapsto \mathcal{L}$ where $\mathbb{P}$ is a finite set of Boolean propositions and $\mathcal{L}$ is a lattice. LVBDD are similar to previously-defined data structures in the literature such as *multi-terminal binary decision diagrams* [FMY97], or *algebraic decision diagrams* [BFG+97] which can be used to encode functions of the form $2^{\mathbb{P}} \mapsto D$ where D is an arbitrary domain. LVBDD differ from these existing structures by exploiting the fact that the co-domain of the represented functions is a *structured set*, to achieve more compact representations. We argue in this work, and experimentally demonstrate, that LVBDD are particularly well-suited for the encoding of the *transition functions of alternating automata* over symbolic

alphabets. We define two distinct semantics for LVBDD and prove that they are incomparable in general. Also, we define two *normal forms* for LVBDD, and we define algorithms for computing the greatest lower bound (*meet*) and least upper bound (*join*) of their representations. Our algorithms are accompanied with complete and *generic* proofs of soundness which are applicable to LVBDD over any (finite and distributive) lattice. We also provide a study of the worst-case complexity of LVBDD manipulation algorithms. Finally, we have implemented our LVBDD data structure in the form of a C++ template library and made it available to the research community. Using this implementation, we have experimentally compared the efficiency of LVBDD and ROBDD in the context of solving LTL satisfiability with alternating automata and antichains. Our experiments reveal that, in that context, LVBDD can outperform ROBDD by several orders of magnitude.

1.3 Plan of the Thesis

The remainder of this thesis is organized into the following chapters:

- In **Chapter 2: Preliminaries** we recall some basic notions about numbers, sets, languages and automata, and so on. This chapter fixes the formal notations used throughout the thesis.

- In **Chapter 3: Antichain Approaches to Automata Emptiness** we summarize (a part of) the framework of antichains for the analysis of automata. Since this framework is used in all the remaining chapters, it has been summarized in this chapter in order to avoid repetitions. Note that we have taken the liberty to slightly adapt the theory for the needs of this dissertation.

- In **Chapter 4: LTL Satisfiability and Model Checking Revisited** we apply the framework of antichains to the particular case of LTL satisfiability and model checking. We study algorithms which combine ROBDD with antichains in the analysis of alternating automata, and discuss an empirical evaluation of these algorithms.

- In **Chapter 5: Fixed Point-Guided Abstraction Refinement** we present abstraction / refinement algorithms designed to handle alternating automata with very large numbers of states. The abstraction is based on the set of partitions of the set of states, and uses a refinement strategy that is not based on counter-examples. An experimental evaluation of the technique is discussed, comparing our approach to "classical" non-abstract antichain algorithms.

- In **Chapter 6: Lattice-Valued Binary Decision Diagrams** we study a new symbolic data structure for the efficient encoding of functions of the form $2^{\mathbb{P}} \mapsto \mathcal{L}$, where $\mathbb{P}$ is a finite set of Boolean propositions and $\mathcal{L}$ is a lattice. We define algorithms for manipulating LVBDD and prove their soundness and complexity properties. Also, we discuss an empirical comparison of LVBDD and ROBDD in the context of the analysis of alternating automata over symbolic alphabets.

- In **Chapter 7: Conclusions** we draw some conclusions and provide some thoughts on potential future work.

1.4 Chronology & Credits

Here are some chronological and credit comments on the contents of each chapters.

- The contents of Chapter 3 is not my original work, and is due to Doyen and Raskin [DR07, DR09, DR10].

- The work of Chapter 4 has been published in the proceedings of TACAS 2008 [DDMR08b] in collaboration with Doyen, De Wulf, and Raskin. A "tool-paper" describing the corresponding implementation has been published in the proceedings of ATVA 2008 [DDMR08a] in collaboration with the same authors.

- The contents of Chapter 5 is the result of a collaboration with Ganty and Raskin, which has been published in the proceedings of CIAA 2009 [GMR09]. An extended version of this work was published in a

special issue of *Theoretical Computer Science* [GMR10] in collaboration with the same authors.

- The work presented in Chapter 6 has been done in collaboration with Geeraerts, Kalyon, Le Gall, and Raskin, and has been published in the proceedings of ATVA 2010 [GKL+10].

Chapter 2

Preliminaries

In this chapter, we review the basic notions used throughout the rest of the thesis.

2.1 Sets, Functions, and Relations

We denote the set of *natural numbers* $\{0, 1, 2, \ldots\}$ by $\mathbb{N}$ and the set of *positive natural numbers* $\{1, 2, 3, \ldots\}$ by $\mathbb{N}_0$. We denote the set of *integer numbers* $\mathbb{N} \cup \{-1, -2, \ldots\}$ by $\mathbb{Z}$. The empty set is denoted $\varnothing$. For any set S we denote by 2^S the *powerset* of S, i.e., the set of subsets of S. We denote by $|S| \in \mathbb{N} \cup \{+\infty\}$ the *cardinality* of a set S. For any subset $X \subseteq S$ of a set S, we denote by $\overline{X} \stackrel{\mathsf{def}}{=} S \setminus X$ the *complement* of X in S, when S is clear from the context. In what follows, we sometimes make use of Church's *lambda notation* to anonymously define functions. For instance, $\lambda\, x \cdot x + 1$ is such an anonymous "lambda-style" definition. For a function $f : A \mapsto B$, we call A the *domain* of f, and B the *codomain* of f, which are denoted $\mathsf{dom}(f)$ and $\mathsf{codom}(f)$, respectively. The *image* of a function $f : A \mapsto B$ is the set $\mathsf{img}(f) \stackrel{\mathsf{def}}{=} \{f(a) \mid a \in A\}$.

2.1.1 Binary Relations

A *binary relation* over a set S is a set of pairs $R \subseteq S \times S$. A binary relation R is *reflexive* if and only if for each $s \in S$ we have that $\langle s, s \rangle \in R$; it is

symmetric if and only if for each pair $\langle s_1, s_2\rangle \in R$ we have that $\langle s_2, s_1\rangle \in R$; it is *antisymmetric* if and only if for each pair $\langle s_1, s_2\rangle \in R$ such that $\langle s_2, s_1\rangle \in R$ it is also the case that $s_1 = s_2$; it is *total* if and only for each pair $s_1 \in S, s_2 \in S$ we either have that $\langle s_1, s_2\rangle \in R$ or $\langle s_2, s_1\rangle \in R$; finally, it is *transitive* if and only if for each triple s_1, s_2, s_3 such that $\langle s_1, s_2\rangle \in R$ and $\langle s_2, s_3\rangle \in R$, it is also the case that $\langle s_1, s_3\rangle \in R$.

Binary relations are often denoted using non-alphabetic symbols and the infix notation. Let $\sim \subseteq S \times S$; the expression $s_1 \sim s_2$ is equivalent to $\langle s_1, s_2\rangle \in \sim$, and $s_1 \not\sim s_2$ is equivalent to $\langle s_1, s_2\rangle \notin \sim$. To make the notations more intuitive, we always use symmetric symbols for symmetric relations and vice-versa. Moreover, the infix notation allows to use symbol relations backwards, e.g., the expression $s_1 \preceq s_2$ is equivalent to $s_2 \succeq s_1$ for any binary relation $\preceq$.

Let $R \subseteq S \times S$ be a binary relation; we denote by R^0 the set of pairs $\{\langle s, s\rangle \mid s \in S\}$, and for every $i \in \mathbb{N}_0$ we denote by R^i the relation:

$$\left\{\langle s_1, s_3\rangle \in S \times S \;\middle|\; \exists\, s_2 \in S : \langle s_1, s_2\rangle \in R^{i-1} \wedge \langle s_2, s_3\rangle \in R\right\}$$

The *transitive closure* of R, denoted R^+, is the relation:

$$\left\{\langle s_1, s_2\rangle \in S \times S \;\middle|\; \exists\, i \in \mathbb{N}_0 : \langle s_1, s_2\rangle \in R^i\right\}$$

The *reflexive and transitive closure* of R, denoted R^*, is the relation $R^0 \cup R^+$. A *preorder* is a reflexive and transitive binary relation. A preorder that is also symmetric is an *equivalence*. A preorder that is also antisymmetric is a *partial order*.

2.1.2 Partially Ordered Sets

A *partially ordered set* (or *poset*, for short) is a pair $\langle S, \preceq\rangle$ where $\preceq \subseteq S \times S$ is a *partial order*. A set $X \subseteq S$ is called a *chain* if and only if for every pair of elements $x_1 \in X, x_2 \in X$ we have that either $x_1 \preceq x_2$ or $x_2 \preceq x_1$. Dually, a set $X \subseteq S$ is called an *antichain* if and only if for every pair of elements $x_1 \in X, x_2 \in X$ we have that whenever $x_1 \preceq x_2$ or $x_2 \preceq x_1$ then $x_1 = x_2$. We denote by $\mathsf{Chains}[S, \preceq]$ and $\mathsf{Antichains}[S, \preceq]$ the *set of chains* and *set of antichains* of the partially ordered set $\langle S, \preceq\rangle$, respectively. The

height (respectively *width*) of a partially ordered set is the cardinality of its *largest chain* (respectively *largest antichain*).

Let $\langle S, \preceq \rangle$ be a partially ordered set. The *upward-closure* of a set $X \subseteq S$ is $\uparrow X = \{s \in S \mid \exists\, x \in X \colon s \succeq x\}$. Conversely, the *downward-closure* of a set $X \subseteq S$ is $\downarrow X = \{s \in S \mid \exists\, x \in X \colon s \preceq x\}$. A set $X \subseteq S$ is *upward-closed* (respectively *downward-closed*) if and only if $X = \uparrow X$ (respectively $X = \downarrow X$). We respectively denote by $\mathsf{UCS}[S, \preceq]$ and $\mathsf{DCS}[S, \preceq]$ the set of upward- and downward-closed subsets of the partially ordered set $\langle S, \preceq \rangle$. For any set $X \subseteq S$, the set of *minimal elements* of X is the antichain $\lfloor X \rfloor = \{x \in X \mid \forall\, x' \in X \colon x' \preceq x \Rightarrow x' = x\}$; similarly, the set of *maximal elements* of X is the antichain $\lceil X \rceil = \{x \in X \mid \forall\, x' \in X \colon x' \succeq x \Rightarrow x' = x\}$. Note that $\lfloor X \rfloor$ and $\lceil X \rceil$ are always antichains since, by definition, they cannot contain two distinct and comparable elements.

A useful property of upward and downward-closed sets is that upward and downward-closedness is preserved by union and intersection. This is formalized by the following lemma.

Lemma 2.1.1 ($\cup$ and $\cap$ Preserve Closedness for Partial Orders)**.** *Let $\langle S, \preceq \rangle$ be a partially ordered set. For every pair of upward-closed sets $U_1, U_2 \subseteq S$, the sets $U_1 \cup U_2$ and $U_1 \cap U_2$ are both upward-closed. Likewise, for every pair of downward-closed sets $D_1, D_2 \subseteq S$, the sets $D_1 \cup D_2$ and $D_1 \cap D_2$ are both downward-closed.*

The following theorem formalizes the notion that minimal and maximal antichains are *canonical representations* of upward- and downward-closed sets, respectively.

Theorem 2.1.2 (Antichain Representation of Closed Sets)**.** *For any partially ordered set $\langle S, \preceq \rangle$ and $X \subseteq S$, we have that X is upward-closed if and only if $X = \uparrow \lfloor X \rfloor$. Similarly, X is downward-closed if and only if $X = \downarrow \lceil X \rceil$.*

Let $\langle S, \preceq \rangle$ be a partially ordered set and $X \subseteq S$. An element $s \in S$ is an *upper bound* of X if and only if $\forall\, x \in X \colon s \succeq x$; dually, s is a *lower bound* of X if and only if $\forall\, x \in X \colon s \preceq x$. The set of upper bounds (respectively lower bounds) of X is denoted by $\mathsf{UB}[S, \preceq](X)$ (respectively $\mathsf{LB}[S, \preceq](X)$). The *least upper bound* of X, denoted $\mathsf{LUB}[S, \preceq](X)$, is the element s such that

$\lfloor \mathsf{UB}[S, \preceq](X) \rfloor = \{s\}$, and is undefined when $|\lfloor \mathsf{UB}[S, \preceq](X) \rfloor| \neq 1$. Dually, the *greatest lower bound* of X, denoted $\mathsf{GLB}[S, \preceq](X)$, is the element s such that $\lceil \mathsf{LB}[S, \preceq](X) \rceil = \{s\}$, and is undefined when $|\lceil \mathsf{LB}[S, \preceq](X) \rceil| \neq 1$.

In the context of partially ordered sets, an *isomorphism* is a one-to-one correspondence (i.e., a bijection) between two posets that is *order preserving*. Formally, the bijective function $\sigma\colon S_1 \mapsto S_2$ is an isomorphism between the posets $\langle S_1, \preceq_1 \rangle$ and $\langle S_2, \preceq_2 \rangle$ if and only if $\forall\, s \in S_1, s' \in S_1\colon (s \preceq_1 s') \Leftrightarrow (\sigma(s) \preceq_2 \sigma(s'))$. We say that two partially ordered sets are *isomorphic* if and only if there exists an isomorphism between them.

2.1.3 Lattices

A *join-semilattice* (respectively *meet-semilattice*) is a poset set $\langle S, \preceq \rangle$ such that, for every pair of elements $s_1 \in S, s_2 \in S$, $\mathsf{LUB}[S, \preceq](\{s_1, s_2\})$ (respectively $\mathsf{GLB}[S, \preceq](\{s_1, s_2\})$) is defined. A *lattice* is a poset that is both a join-semilattice and a meet-semilattice.

The least upper bound of two lattice elements s_1, s_2 is called the *join* of these elements, and is denoted $s_1 \sqcup s_2$; the greatest lower bound of s_1, s_2 is called the *meet* of these elements, and is denoted $s_1 \sqcap s_2$.

A lattice $\langle S, \preceq \rangle$ is said to be *complete* if and only if both $\mathsf{LUB}[S, \preceq](X)$ and $\mathsf{GLB}[S, \preceq](X)$ are defined for *every* set $X \subseteq S$. Notice that finite lattices are always complete. A lattice $\langle S, \preceq \rangle$ is *bounded* if and only if we have that both $\mathsf{LUB}[S, \preceq](S)$ and $\mathsf{GLB}[S, \preceq](S)$ are defined, in which case they are denoted $\top$ and $\bot$, respectively. Again, finite lattices are trivially always bounded.

A lattice $\langle S, \preceq \rangle$ is *distributive* if and only if the following two equalities hold for all triples $s_1, s_2, s_3 \in S$.

$$(s_1 \sqcup s_2) \sqcap s_3 = (s_1 \sqcap s_3) \sqcup (s_2 \sqcap s_3)$$

$$(s_1 \sqcap s_2) \sqcup s_3 = (s_1 \sqcup s_3) \sqcap (s_2 \sqcup s_3)$$

An element s of a lattice $\langle S, \preceq \rangle$ is *join-irreducible* if and only if $s \neq \bot$ and $\forall\, s'_1 \in S, s'_2 \in S\colon (s'_1 \sqcup s'_2 = s) \Rightarrow (s'_1 = s \vee s'_2 = s)$. Dually, s is *meet-irreducible* if and only if $s \neq \top$ and $\forall\, s'_1 \in S, s'_2 \in S\colon (s'_1 \sqcap s'_2 = s) \Rightarrow (s'_1 = s \vee s'_2 = s)$. In short, $s \notin \{\bot, \top\}$ is join-irreducible (respectively

meet-irreducible) whenever it is not the join (respectively not the meet) of two distinct lattice elements. The set of join-irreducible (respectively meet-irreducible) elements of a lattice $\langle S, \preceq \rangle$ is denoted $\mathsf{JIR}[S, \preceq]$ (respectively $\mathsf{MIR}[S, \preceq]$).

We call a *lattice of sets* any collection of sets that forms a lattice using subset inclusion as partial order. Simply put, any collection of sets that is closed under union and intersection is a lattice of sets.

We can now recall Birkhoff's Representation Theorem for finite distributive lattices. This important result in lattice theory states that every finite distributive lattice is *isomorphic* to a lattice of sets.

Theorem 2.1.3 (Birkhoff's Representation Theorem [Bir67])**.** *Any finite distributive lattice* $\langle S, \preceq \rangle$ *is isomorphic to the lattice of* downward-closed sets of its join-irreducible elements*, or formally, the lattice* $\langle \mathsf{DCS}[\mathsf{JIR}[S, \preceq], \preceq], \subseteq \rangle$*. The corresponding isomorphism is* $\sigma(s) = \{x \in \mathsf{JIR}[S, \preceq] \mid x \preceq s\}$ *with* $\sigma^{-1}(X) = \mathsf{LUB}[S, \preceq](X)$*. Conversely, any finite distributive lattice* $\langle S, \preceq \rangle$ *is isomorphic to the lattice of* upward-closed sets of its meet-irreducible elements*, or formally, the lattice* $\langle \mathsf{UCS}[\mathsf{MIR}[S, \preceq], \preceq], \subseteq \rangle$*. In this case the corresponding isomorphism is the function* $\sigma(s) = \{x \in \mathsf{MIR}[S, \preceq] \mid x \succeq s\}$ *with* $\sigma^{-1}(X) = \mathsf{GLB}[S, \preceq](X)$*.*

An illustration of Birkhoff's theorem is depicted at Figure 2.1.

The set of upward-closed subsets of a finite join-semilattice is a finite and distributive lattice. Likewise, the set of downward-closed subsets of a finite meet-semilattice is a finite and distributive lattice.

Theorem 2.1.4 (Lattice of Upward/Downward-Closed Sets)**.** *For any join-semilattice* $\langle S_\sqcup, \preceq \rangle$ *we have that* $\langle \mathsf{UCS}[S_\sqcup, \preceq], \subseteq \rangle$ *is a finite and distributive lattice. For any meet-semilattice* $\langle S_\sqcap, \preceq \rangle$ *we have that* $\langle \mathsf{DCS}[S_\sqcup, \preceq], \subseteq \rangle$ *is a finite and distributive lattice.*

The lattice of upward-closed subsets of a finite join-semilattice is isomorphic to a lattice of minimal antichains. Symmetrically, the lattice of downward-closed subsets of a finite meet-semilattice is isomorphic to a lattice of maximal antichains.

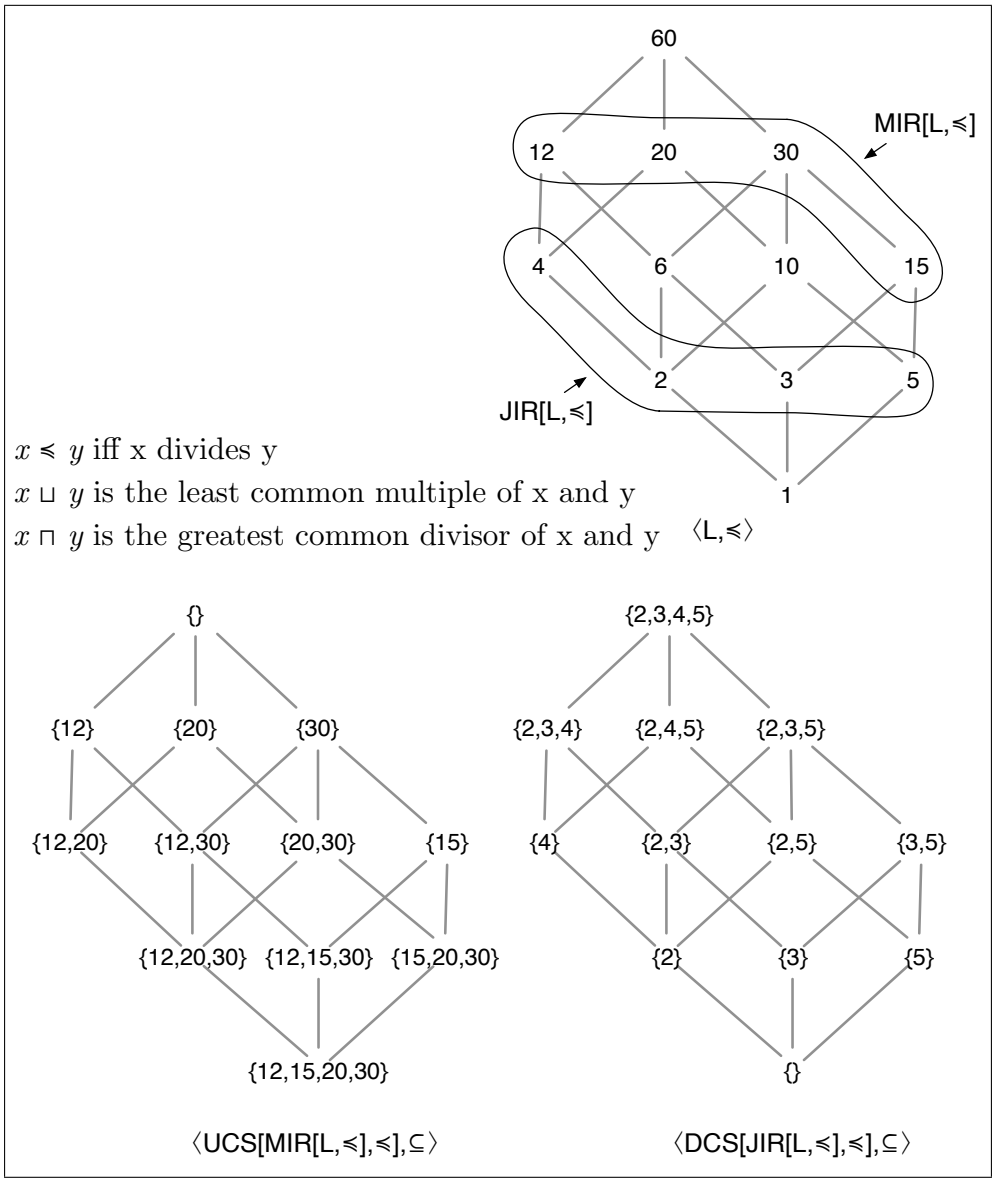


Figure 2.1: Illustration of Birkhoff's representation theorem on the *lattice of divisors* of 60. The set of divisors of 60 is partially ordered by the *divides* relation. The join and meet are the *least common multiple* and *greatest common divisor* operations, respectively.

Theorem 2.1.5 (Lattice of Minimal Antichains)**.** *Let $\langle S, \preceq \rangle$ be a finite join-semilattice, and for any $X, Y \in \mathsf{Antichains}[S, \preceq]$ let $X \sqsubseteq Y$ iff $\forall\, x \in X\ \exists\, y \in Y : x \succeq y$. The pair $\langle \mathsf{Antichains}[S, \preceq], \sqsubseteq \rangle$ is a finite and distributive lattice, such that for any $X, Y \in \mathsf{Antichains}[S, \preceq]$:*

$$\begin{aligned} X \sqcup Y &= \lfloor X \cup Y \rfloor \\ X \sqcap Y &= \lfloor \{ x \sqcup y \mid x \in X, y \in Y \} \rfloor \\ \bot &= \varnothing \\ \top &= \lfloor S \rfloor \end{aligned}$$

Moreover, the function $\lfloor \cdot \rfloor : \mathsf{UCS}[S, \preceq] \mapsto \mathsf{Antichains}[S, \preceq]$ is a lattice isomorphism between $\langle \mathsf{UCS}[S, \preceq], \subseteq \rangle$ and $\langle \mathsf{Antichains}[S, \preceq], \sqsubseteq \rangle$.

Theorem 2.1.6 (Lattice of Maximal Antichains)**.** *Let $\langle S, \preceq \rangle$ be a finite meet-semilattice, and for any $X, Y \in \mathsf{Antichains}[S, \preceq]$ let $X \sqsubseteq Y$ iff $\forall\, x \in X\ \exists\, y \in Y : x \preceq y$. The pair $\langle \mathsf{Antichains}[S, \preceq], \sqsubseteq \rangle$ is a finite and distributive lattice, such that for any $X, Y \in \mathsf{Antichains}[S, \preceq]$:*

$$\begin{aligned} X \sqcup Y &= \lceil X \cup Y \rceil \\ X \sqcap Y &= \lceil \{ x \sqcap y \mid x \in X, y \in Y \} \rceil \\ \bot &= \varnothing \\ \top &= \lceil S \rceil \end{aligned}$$

Moreover, the function $\lceil \cdot \rceil : \mathsf{DCS}[S, \preceq] \mapsto \mathsf{Antichains}[S, \preceq]$ is a lattice isomorphism between $\langle \mathsf{DCS}[S, \preceq], \subseteq \rangle$ and $\langle \mathsf{Antichains}[S, \preceq], \sqsubseteq \rangle$.

2.1.4 Functions over Lattices and Fixed Points

We now recall some definitions regarding functions over lattices.

Let $\langle S, \preceq \rangle$ be a lattice; a function $f \colon S \mapsto S$ is *monotonic* if and only if $\forall\, s \in S, s' \in S \colon (s \preceq s') \Rightarrow (f(s) \preceq f(s'))$. Moreover, the function f is *continuous* if and only if either S is a finite set, or for every non-empty set $X \subseteq S$, we have that $f(\mathsf{LUB}[S, \preceq](X)) = \mathsf{LUB}[S, \preceq](\{ f(x) \mid x \in X \})$. Note that every continuous function is necessarily monotonic. In this thesis, we shall only manipulate functions over finite lattices, so the continuity requirement will always be trivially satisfied by monotone functions.

A *fixed point* of a function $f\colon S \mapsto S$ over the lattice $\langle S, \preceq \rangle$ is an element $s \in S$ such that $f(s) = s$. We denote by $\mathsf{FP}[S, \preceq](f)$ the set of fixed points of f over the lattice $\langle S, \preceq \rangle$.

We can now recall the well-known Knaster-Tarski theorem on continuous functions over complete lattices.

Theorem 2.1.7 (Knaster-Tarski Theorem)**.** *For every continuous function $f\colon S \mapsto S$ over a complete lattice $\langle S, \preceq \rangle$, the set of fixed point $\mathsf{FP}[S, \preceq](f)$ forms a complete lattice with the partial order $\preceq$.*

Since complete lattices are necessarily non-empty[1], it follows from the Knaster-Tarski theorem that every continuous function over a complete lattice admits both a *least fixed point* and a *greatest fixed point*. We denote these fixed points by $\mathsf{LFP}[S, \preceq](f) = \mathsf{GLB}[S, \preceq](\mathsf{FP}[S, \preceq](f))$ and $\mathsf{GFP}[S, \preceq](f) = \mathsf{LUB}[S, \preceq](\mathsf{FP}[S, \preceq](f))$, respectively.

Let $f\colon S \mapsto S$ be a continuous function over the complete lattice $\langle S, \preceq \rangle$. We define f^0 to be the identity function, i.e., $\forall\, s \in S\colon f^0(s) = s$, and for any $i \in \mathbb{N}_0$ we define f^i recursively as $f^i(s) = f(f^{i-1}(s))$.

Algorithm 2.1 depicts the practical iterative computation of fixed points. In both algorithms, the least or greatest fixed point is computed by repeatedly applying the function (beginning either with $\top$ or $\bot$).

2.1.5 Galois Connections

In this thesis, we use *Galois connections* [CC77] to formalize the approximation of alternating automata. A Galois connection is a pair of functions $\alpha : \mathcal{C} \mapsto \mathcal{A}$, $\gamma : \mathcal{A} \mapsto \mathcal{C}$ connecting two *domains* $\mathcal{A}$ and $\mathcal{C}$, which are both complete lattices. The complete lattice $\langle \mathcal{C}, \sqsubseteq_{\mathcal{C}} \rangle$ is called the *concrete domain*, while $\langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$ is called the *abstract domain*. Therefore, the function $\gamma : \mathcal{A} \mapsto \mathcal{C}$ is called the *concretization* function, and $\alpha : \mathcal{C} \mapsto \mathcal{A}$ is called the *abstraction* function.

Definition 2.1.8 (Galois Connection [CC77])**.** *Let $\langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$ and $\langle \mathcal{C}, \sqsubseteq_{\mathcal{C}} \rangle$ be two complete lattices, and two functions $\alpha : \mathcal{C} \mapsto \mathcal{A}$ and $\gamma : \mathcal{A} \mapsto \mathcal{C}$. The*

[1] At the very least, $\mathsf{LUB}(\varnothing)$ and $\mathsf{GLB}(\varnothing)$ must evaluate to a lattice element.

Algorithm 2.1: Computation of least and greatest fixed points.

inputs : a complete lattice $\langle S, \preceq \rangle$ and a continuous function $f \colon S \mapsto S$
output: $\mathsf{LFP}[S, \preceq](f)$

```
1  begin computeLFP(S, ⪯, f)
2      s ← ⊥ ;
3      do
4          s′ ← s ;
5          s ← f(s) ;
6      while s ≠ s′ ;
7      return s ;
8  end
```

inputs : a complete lattice $\langle S, \preceq \rangle$ and a continuous function $f \colon S \mapsto S$
output: $\mathsf{GFP}[S, \preceq](f)$

```
9  begin computeGFP(S, ⪯, f)
10     s ← ⊤ ;
11     do
12         s′ ← s ;
13         s ← f(s) ;
14     while s ≠ s′ ;
15     return s ;
16 end
```

4-tuple $(\alpha, \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle, \langle \mathcal{C}, \sqsubseteq_{\mathcal{C}} \rangle, \gamma)$ *forms a* Galois connection*, which is denoted* $\langle \mathcal{C}, \sqsubseteq_{\mathcal{C}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$*, if and only if (both statements are equivalent):*

$$\forall\, c \in \mathcal{C}\ \forall a \in \mathcal{A} \colon \alpha(c) \sqsubseteq_{\mathcal{A}} a \Leftrightarrow c \sqsubseteq_{\mathcal{C}} \gamma(a)$$
$$\forall\, c \in \mathcal{C} \colon c \sqsubseteq_{\mathcal{C}} \gamma \circ \alpha(c) \text{ and } \forall\, a \in \mathcal{A} \colon \alpha \circ \gamma(a) \sqsubseteq_{\mathcal{A}} a$$

The lattice orderings in both domains represent the relative *precision* of domain values. We say that an abstract value $a \in \mathcal{A}$ *represents* a concrete

value $c \in \mathcal{C}$ whenever $\alpha(c) \sqsubseteq_{\mathcal{A}} a$, or equivalently whenever $c \sqsubseteq_{\mathcal{C}} \gamma(a)$. Furthermore, we say that a concrete value c is *exactly represented* by a Galois connection whenever $\gamma \circ \alpha(c) = c$, that is whenever c is represented without loss of precision.

Lemma 2.1.9 (Properties of Galois Connections)**.** *For every Galois connection* $\langle \mathcal{C}, \sqsubseteq_{\mathcal{C}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$ *the following properties hold:*

- α *and* γ *are monotone functions ;*
- *for every* $c_1, c_2 \in \mathcal{C}$ *we have that* $\alpha(c_1 \sqcup c_2) = \alpha(c_1) \sqcup \alpha(c_2)$ *;*
- *for every* $a_1, a_2 \in \mathcal{A}$ *we have that* $\gamma(a_1 \sqcap a_2) = \gamma(c_1) \sqcap \gamma(c_2)$ *;*
- $\alpha \circ \gamma \circ \alpha = \alpha$ *and* $\gamma \circ \alpha \circ \gamma = \gamma$ *;*
- $\alpha = \lambda\, c : \mathsf{GLB}(\{a \mid c \sqsubseteq_{\mathcal{C}} \gamma(a)\})$ *;*
- $\gamma = \lambda\, a : \mathsf{LUB}(\{c \mid \alpha(c) \sqsubseteq_{\mathcal{A}} a\})$ *.*

We now provide some intuitions on some of the properties above.

The properties $c \sqsubseteq_{\mathcal{C}} \gamma \circ \alpha(c)$ and $\alpha \circ \gamma(a) \sqsubseteq_{\mathcal{A}} a$ show the *soundness* of the abstraction: a concrete value c should be at least as precise than the concretization of the abstraction of c, and similarly, the abstraction of the concretization of an abstract value a should be at least as precise as a.

The last two properties of the above lemma show that the abstraction function and the concretization function are both univocally determined by the other. What is more, these properties states that Galois connections are always such that α is *maximally precise*, since it maps each concrete value c to the unique most precise value a that represents c (that is, $c \sqsubseteq_{\mathcal{C}} \gamma(a)$).

A Galois connection $\langle \mathcal{C}, \sqsubseteq_{\mathcal{C}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$ for which $\gamma \circ \alpha$ is the identity function is called a *Galois insertion*.

The following theorem shows how it is possible to approximate the abstraction of a concrete domain fixed point, by carrying out the fixed point computation in the abstract domain. In general, this abstract fixed point computation is less precise than the abstraction of the concrete fixed point. In practice, it is often not possible to do better however, when the concrete fixed point cannot be computed directly for undecidability or intractability reasons.

Theorem 2.1.10 (Abstraction of Fixed Points [CC77])**.** *For any Galois connection* $\langle \mathcal{C}, \sqsubseteq_{\mathcal{C}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$*, and for any monotone function* $f : \mathcal{C} \mapsto \mathcal{C}$ *we have that:*

$$\alpha(\mathsf{LFP}[\mathcal{C}, \sqsubseteq_{\mathcal{C}}](f)) \sqsubseteq_{\mathcal{A}} \mathsf{LFP}[\mathcal{A}, \sqsubseteq_{\mathcal{A}}](\alpha \circ f \circ \gamma)$$
$$\alpha(\mathsf{GFP}[\mathcal{C}, \sqsubseteq_{\mathcal{C}}](f)) \sqsubseteq_{\mathcal{A}} \mathsf{GFP}[\mathcal{A}, \sqsubseteq_{\mathcal{A}}](\alpha \circ f \circ \gamma)$$

2.2 Boolean Logic

In this section, we quickly recall a number of definitions and notations regarding Boolean logic that are needed in the sequel.

2.2.1 Boolean Functions

We denote the set of *Boolean truth values* as $\mathbb{B} \overset{\mathsf{def}}{=} \{\mathsf{true}, \mathsf{false}\}$. In general, a set of logical truth values can contain more than two elements (e.g., to accommodate a third, uncertain possibility), and can even be infinite (e.g., the real numbers between 0 and 1); Boolean logic however, deals only with the binary world of *true* and *false*.

Let $\mathbb{P}$ be a finite set of elements called *propositions*. A *valuation* over $\mathbb{P}$ is a set $v \subseteq \mathbb{P}$ which identifies a *truth assignment* of each proposition in $\mathbb{P}$; by convention, the propositions in v are seen as assigned to true and the propositions in $\mathbb{P} \setminus v$ are seen as assigned to false. For any valuation $v \in 2^{\mathbb{P}}$ and proposition $p \in \mathbb{P}$, we use the notations $v|_{p=\mathsf{true}} \overset{\mathsf{def}}{=} v \cup \{p\}$ or $v|_{p=\mathsf{false}} \overset{\mathsf{def}}{=} v \setminus \{p\}$ interchangeably.

A *Boolean function* is a function f of type $f \colon 2^{\mathbb{P}} \to \mathbb{B}$ for some finite set of propositions $\mathbb{P}$. We refer to the cardinality $|\mathbb{P}|$ as the *arity* of a Boolean function. For any Boolean function $f \colon 2^{\mathbb{P}} \to \mathbb{B}$, with $p \in \mathbb{P}$ and $t \in \mathbb{B}$, we define $f|_{p=t}$ to be the function $f' \colon 2^{\mathbb{P}} \to \mathbb{B}$ such that $f'(v) = f(v|_{p=t})$ for every valuation $v \in 2^{\mathbb{P}}$. We denote the set of Boolean functions over $\mathbb{P}$ by $\mathsf{BF}(\mathbb{P})$.

The *efficient representation and manipulation* of Boolean functions is a central problem in both computer science and engineering. The most ex-

plicit, and the simplest representation of a Boolean function is the *truth table*, which contains one row for each of the possible $2^{|\mathbb{P}|}$ inputs. The remainder of this section reviews two more interesting representations of Boolean functions, namely *propositional Boolean formulas* and *reduced ordered binary decision diagrams*.

2.2.2 Propositional Boolean Formulas

Let us first formally define the syntax of propositional Boolean formulas.

Definition 2.2.1 (Syntax of Propositional Boolean Formulas)**.** *Let $\mathbb{P}$ be a finite set of Boolean propositions, and let $p \in \mathbb{P}$. The set of syntactically correct* propositional Boolean formulas *is defined recursively with the following grammar:*

$$\varphi ::= \mathsf{false} \mid \mathsf{true} \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2$$

We denote the set of propositional Boolean formulas over $\mathbb{P}$ by $\mathsf{BL}(\mathbb{P})$.

The above definition introduces basic *logical connectives*. The symbol $\neg$ is the unary *negation*, $\vee$ is the *disjunction* connective and $\wedge$ is the *conjunction* connective. Additionally, we define the following *syntactic shorthands*: $(\varphi \Rightarrow \psi) \stackrel{\mathsf{def}}{=} (\neg\varphi \vee \psi)$ is the *logical implication* connective (it reads "φ *implies* ψ"), and $(\varphi \Leftrightarrow \psi) \stackrel{\mathsf{def}}{=} (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$ is the *logical equivalence* connective (it reads "φ *if and only if* ψ").

As stated previously, propositional Boolean formulas are syntactic representations of Boolean functions. The represented Boolean function is obtained by induction on the structure of the formula, as formalized in the following definition.

Definition 2.2.2 (Semantics of Propositional Boolean Formulas)**.** *Let $\mathbb{P}$ be a finite set of Boolean propositions, and φ be a propositional Boolean formula over $\mathbb{P}$. The* semantics *of the formula φ, denoted $[\![\varphi]\!]$, is the Boolean function $f : 2^{\mathbb{P}} \mapsto \mathbb{B}$ such that, for every valuation $v \in 2^{\mathbb{P}}$:*

- $[\![\mathsf{true}]\!](v) = \mathsf{true}$*;*
- $[\![\mathsf{false}]\!](v) = \mathsf{false}$*;*

- $[\![p]\!](v) = \mathsf{true}$ *iff* $p \in v$ *(assuming* $p \in \mathbb{P}$*);*
- $[\![\neg\varphi]\!](v) = \mathsf{true}$ *iff* $[\![\varphi]\!](v) = \mathsf{false}$*;*
- $[\![\varphi_1 \vee \varphi_2]\!](v) = \mathsf{true}$ *iff* $[\![\varphi_1]\!](v) = \mathsf{true}$ *or* $[\![\varphi_2]\!](v) = \mathsf{true}$ *or both;*
- $[\![\varphi_1 \wedge \varphi_2]\!](v) = \mathsf{true}$ *iff both* $[\![\varphi_1]\!](v) = \mathsf{true}$ *and* $[\![\varphi_2]\!](v) = \mathsf{true}$.

We see from the above definition that propositional Boolean formulas associate to each valuation of their propositions the value true or false. When $[\![\varphi]\!](v) = \mathsf{true}$ we say that the valuation v *satisfies* the formula φ, which we denote more compactly by $v \models \varphi$. Conversely, when $[\![\varphi]\!](v) = \mathsf{false}$ then v does not satisfy φ which is simply written $v \not\models \varphi$. A formula φ is said to be *satisfiable* iff it is satisfied by at least by one valuation, or equivalently iff $[\![\varphi]\!] \neq \lambda v \cdot \mathsf{false}$. Dually, a formula φ is *valid* iff it is satisfied by every valuation, or equivalently iff $[\![\varphi]\!] = \lambda v \cdot \mathsf{true}$. Deciding whether a given propositional Boolean formula is satisfiable or valid are fundamental NP-Complete and co-NP-Complete problems, respectively [Coo71]. We denote by $\mathsf{Sat}(\varphi) \stackrel{\mathsf{def}}{=} \{v \in 2^{\mathbb{P}} \mid v \models \varphi\}$ the set of valuations which satisfy a formula φ.

2.2.3 Positive Propositional Boolean Formulas

In the remainder of this thesis, we will often manipulate *positive* propositional Boolean formulas, i.e., formulas which do not contain negation symbols. We denote by $\mathsf{BL}^+(\mathbb{P})$ the set of positive propositional Boolean formulas over $\mathbb{P}$.

The interesting property of positive propositional Boolean formulas is that the set of valuations that makes the formula true is always upward-closed. This is formalized by the following Lemma.

Lemma 2.2.3 (BL^+ Formulas Have Upward-Closed Valuation Sets)**.** *Let* $\mathbb{P}$ *be a finite set of Boolean propositions and let* $\varphi \in \mathsf{BL}^+(\mathbb{P})$*. The set of valuations* $\mathsf{Sat}(\varphi)$ *is upward-closed for* $\subseteq$*.*

Proof. We proceed by induction over the structure of the formula φ. For the base case, we can have $\varphi = \mathsf{true}$ and $\mathsf{Sat}(\varphi) = 2^{\mathbb{P}}$, or $\varphi = \mathsf{false}$ and $\mathsf{Sat}(\varphi) = \varnothing$, which are both upward-closed for $\subseteq$. The final base case is $\varphi = p$, with $p \in \mathbb{P}$, for which $\mathsf{Sat}(\varphi) = \{v \subseteq \mathbb{P} \mid p \in v\}$ is also upward-closed. For the

inductive case, we either have that $\varphi = \varphi_1 \vee \varphi_2$ and $\mathsf{Sat}(\varphi) = \mathsf{Sat}(\varphi_1) \cup \mathsf{Sat}(\varphi_2)$, or $\varphi = \varphi_1 \wedge \varphi_2$ and $\mathsf{Sat}(\varphi) = \mathsf{Sat}(\varphi_1) \cap \mathsf{Sat}(\varphi_2)$. By the induction hypothesis, we know that both $\mathsf{Sat}(\varphi_1)$ and $\mathsf{Sat}(\varphi_2)$ are upward-closed, and since union and intersection preserve upward-closedness by Lemma 2.1.1, we have that $\mathsf{Sat}(\varphi)$ is upward-closed for $\subseteq$ in both cases. □

2.2.4 Reduced Ordered Binary Decision Diagrams

Formulas offer a simple and compact representation of Boolean functions but lack the crucial property of *canonicity*, which is of great practical importance, as it allows for example to easily test for equivalence (more on that later). Truth tables are canonical but are always exponential in $|\mathbb{P}|$, no matter the represented function, so they cannot be used for practical purposes. *Reduced ordered binary decision diagrams* [Bry86] (ROBDD, for short) provide a canonical, graph-based representation of Boolean functions that has proven to be extremely efficient in a number of applications, particularly in the field of *symbolic model checking* [BCM+90].

Definition 2.2.4 (Ordered Binary Decision Diagrams)**.** *Let* $\mathbb{P} = \{p_1, \ldots, p_k\}$ *be a totally ordered set of Boolean propositions. An* ordered binary decision diagram *(*OBDD*, for short) over* $\mathbb{P}$ *is* (i) *either a* terminal OBDD $\langle \mathsf{index}(n), \mathsf{val}(n) \rangle$ *where* $\mathsf{index}(n) = k + 1$ *and* $\mathsf{val}(n) \in \{\mathsf{true}, \mathsf{false}\}$*; or* (ii) *a* non-terminal OBDD $\langle \mathsf{index}(n), \mathsf{lo}(n), \mathsf{hi}(n) \rangle$*, where* $1 \leq \mathsf{index}(n) \leq k$ *and* $\mathsf{lo}(n)$ *and* $\mathsf{hi}(n)$ *are (terminal or non-terminal)* OBDD *such that* $\mathsf{index}(\mathsf{hi}(n)) > \mathsf{index}(n)$ *and* $\mathsf{index}(\mathsf{lo}(n)) > \mathsf{index}(n)$*. We denote the set of syntactically correct* OBDD *over* $\mathbb{P}$ *by* $\mathsf{OBDD}(\mathbb{P})$*.*

Additionally, for every $n \in \mathsf{OBDD}(\{p_1, \ldots, p_k\})$ with $\mathsf{index}(n) = i$, we denote by $\mathsf{prop}(n)$ the proposition p_i.

It is easy to see that the syntactic requirements enforced on OBDD by the definition above are such that all OBDD are essentially rooted directed acyclic graphs, with non-leaf nodes having an out-degree of 2 and leaf nodes being labeled by a truth value.

In the sequel, we refer to OBDD also as "OBDD node", or simply "node". This abuse of language is justified by the fact that OBDD*s* are

uniquely identified by their root node. For any non-terminal node n, we call $\mathsf{hi}(n)$ and $\mathsf{lo}(n)$ the *high-child* and *low-child* of n, respectively. Additionally, the set $\mathsf{nodes}(n)$ of an OBDD n is defined recursively as follows. If n is terminal, then $\mathsf{nodes}(n) = \{n\}$. Otherwise $\mathsf{nodes}(n) = \{n\} \cup \mathsf{nodes}(\mathsf{lo}(n)) \cup \mathsf{nodes}(\mathsf{hi}(n))$. The number of (unique) nodes of an OBDD is denoted by $|n|$.

The semantics of ordered binary decision diagrams is naturally defined in terms of the *if-then-else* function, which we denote ite. For any $p \in \mathbb{P}$, $f, g : 2^{\mathbb{P}} \mapsto \mathbb{B}$ we have that $[\![\mathsf{ite}(p, f, g)]\!] = \lambda v \cdot \text{if } p \in v \text{ then } f(v) \text{ else } g(v)$.

Definition 2.2.5 (Semantics of OBDD)**.** *Let $\mathbb{P} = \{p_1, \ldots, p_k\}$ be a totally ordered set of Boolean propositions and let $d \in \mathsf{OBDD}(\mathbb{P})$. The semantics of the ordered binary decision diagram d, denoted $[\![d]\!]$, is a Boolean function $f : 2^{\mathbb{P}} \mapsto \mathbb{B}$ defined recursively over the structure of d:*

- *if d is a terminal node then $[\![d]\!] = [\![\mathsf{val}(d)]\!]$;*
- *otherwise, let $i = \mathsf{index}(d)$, $[\![d]\!] = \mathsf{ite}(p_i, [\![\mathsf{lo}(d)]\!], [\![\mathsf{hi}(d)]\!])$.*

We can see from the above definitions that OBDD are non-canonical. *Reduced* OBDD (or ROBDD for short) provide additional syntactic requirements to enforce canonicity. The definition of ROBDD relies on the notion of isomorphism between OBDD, which we now define formally.

Definition 2.2.6 (Isomorphism Between OBDD)**.** *Let $d_1, d_2 \in \mathsf{OBDD}(\mathbb{P})$ be two* OBDD *over $\mathbb{P}$. An* isomorphism *between d_1 and d_2 is a bijection $\sigma : \mathsf{nodes}(d_1) \mapsto \mathsf{nodes}(d_2)$ such that for every $n_1 \in \mathsf{nodes}(d_1)$ we have that:*

- $\mathsf{index}(n_1) = \mathsf{index}(\sigma(n_1))$*;*
- $\sigma(\mathsf{lo}(n_1)) = \mathsf{lo}(\sigma(n_1))$*;*
- $\sigma(\mathsf{hi}(n_1)) = \mathsf{hi}(\sigma(n_1))$*;*
- $\mathsf{val}(n_1) = \mathsf{val}(\sigma(n_1))$ *if n_1 is terminal.*

Two OBDD*s are* isomorphic *iff there exists an isomorphism between them.*

We can now formally define *reduced* OBDD.

Definition 2.2.7 (Reduced OBDD)**.** *Let $\mathbb{P} = \{p_1, \ldots, p_k\}$ and $d \in \mathsf{OBDD}(\mathbb{P})$. We say that the* OBDD *d is* reduced *iff it satisfies the following constraints:*

- *d does not contain two distinct isomorphic* OBDD*s;*
- *for every $n \in \mathsf{nodes}(d)$ we have that $\mathsf{lo}(n) \neq \mathsf{hi}(n)$.*

We denote the set of syntactically correct ROBDD *over $\mathbb{P}$ by* $\mathsf{ROBDD}(\mathbb{P})$.

Bryant [Bry86] showed that ROBDD*s* are a *canonical* data structure for Boolean functions, assuming a total ordering of the Boolean propositions. This is formalized by the following theorem.

Theorem 2.2.8 (Canonicity of ROBDD [Bry86])**.** *Let $\mathbb{P} = \{p_1, \ldots, p_k\}$ be a totally ordered set of Boolean propositions, and let $f : 2^{\mathbb{P}} \mapsto \mathbb{B}$ be a Boolean function over $\mathbb{P}$. There exists a* ROBDD *$d \in \mathsf{ROBDD}(\mathbb{P})$ such that $[\![d]\!] = f$ and for every $d' \in \mathsf{ROBDD}(\mathbb{P})$ if $[\![d']\!] = f$ then d and d' are isomorphic.*

For every Boolean formula $\varphi \in \mathsf{BL}(\mathbb{P})$, we denote by $\mathsf{BuildROBDD}(\varphi)$ the ROBDD $d \in \mathsf{ROBDD}(\mathbb{P})$ such that $[\![d]\!] = [\![\varphi]\!]$.

The practical importance of canonicity is very large. Efficient ROBDD implementations very often enforce *structural equality* between isomorphic ROBDD, so that two ROBDD are equivalent if and only if they have the same address. One can therefore check for satisfiability or validity of an ROBDD simply by inspecting its root node. More importantly, memory-level canonicity means that ROBDD can be implemented as *hashable* values; this allows to use caching techniques and *dynamic programming* algorithms.

As expected, we denote the disjunction and conjunction of two ROBDD d_1, d_2 respectively by $d_1 \vee d_2$ and $d_1 \wedge d_2$, and we denote the negation of an ROBDD d by $\neg d$ (the formal definitions of these operations are obvious and omitted here). Likewise, for an ROBDD $d \in \mathsf{ROBDD}(\mathbb{P})$ and $p \in \mathbb{P}$, we use the syntax $d|_{p=\mathsf{true}}$ and $d|_{p=\mathsf{false}}$ in the expected way. An additional ROBDD operation that we use throughout this work is *existential quantification*, defined below.

Definition 2.2.9 (ROBDD existential quantification)**.** *Let $\mathbb{P} = \{p_1, \ldots, p_k\}$ be a totally ordered set of Boolean propositions, and let $d \in \mathsf{ROBDD}(\mathbb{P})$ be an* ROBDD *over $\mathbb{P}$. The* existential quantification *of d with respect to a set $S \subseteq \mathbb{P}$, denoted $\exists\, S \cdot d$, is defined by $\exists\, S \cdot d \stackrel{\mathsf{def}}{=} \bigvee_{p \in S} (d|_{p=\mathsf{false}} \vee d|_{p=\mathsf{true}})$.*

2.3 Finite Automata

In this section, we review the basic formalisms needed to *represent formal languages* and *encode computation systems*.

2.3.1 Alphabets, Words, and Languages

Let Σ be a finite non-empty set that we call *alphabet* and the elements of which we call *letters*. A *finite word* is a finite sequence $w = \sigma_0, \ldots, \sigma_{n-1}$ of letters from Σ. The word of length zero is denoted ϵ. We denote by Σ^* the set of all finite words over Σ. A *finite-word language* is a finite or infinite set of finite words, i.e., a set $L \subseteq \Sigma^*$. We also call Σ^* the *finite-word universal language* as it contains all words of finite length, and we call the empty set of words $\varnothing$ the *empty language*.

These notions are extended to infinite words in the following natural way. An *infinite word* is an infinite sequence $w = \sigma_0, \sigma_1, \ldots$ of letters from Σ. We denote by Σ^ω the set of all infinite words over Σ. An *infinite-word language* is a (finite or infinite) set of infinite words, i.e., a set $L \subseteq \Sigma^\omega$. We call Σ^ω the *infinite-word universal language*.

The *length* of a word w is denoted $|w|$; the length of a infinite word is simply $+\infty$. The *position* of a letter in a finite or infinite word is a natural number, the first letter being at position zero. We denote the letter of position i in a finite or infinite word w by $w[i]$. For any word w such that $|w| > i$, we denote by $w[i \ldots]$ the *suffix* of the word w, starting at and including position i. Note that a suffix has always a strictly positive length.

In this thesis we do not consider languages that contain both finite and infinite words.

2.3.2 Finite State Machines and Automata

On of the most fundamental tools to represent formal languages is the *finite state machine* (FSM for short). An FSM is essentially a graph whose vertices are called *states*, and whose edges are called *transitions*, and which identifies a set of *initial states* along with a set of *final states* (non-initial and non-final states are simply called *states*). Additionally, the transitions of an FSM are

labeled by symbols taken from a finite alphabet.

Definition 2.3.1 (Finite State Machine)**.** *A finite state machine is a tuple* $\langle Q, \Sigma, I, \rightarrow, F \rangle$ *where:*

- $Q = \{q_1, \ldots, q_n\}$ *is a finite set of* states*;*
- $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ *is a finite* alphabet*;*
- $I \subseteq Q$ *is a finite set of* initial states*;*
- $\rightarrow \subseteq Q \times \Sigma \times Q$ *is a labeled* transition relation*;*
- $F \subseteq Q$ *is a finite set of* final states*.*

The transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ of FSM is inherently *non deterministic*, in the sense that there can be several edges labeled with the same alphabet symbol which go from a single state to several distinct states. Contrarily, *deterministic* FSM are such that $\forall\, q \in Q, \forall\, \sigma \in \Sigma\colon |\, \{q' \mid \langle q, \sigma, q' \rangle \in \rightarrow\}\, | \leq 1$. Moreover, deterministic FSM must have exactly one initial state, i.e., $|I| = 1$.

An FSM is *complete* iff $\forall\, q \in Q, \forall\, \sigma \in \Sigma\colon |\, \{q' \mid \langle q, \sigma, q' \rangle \in \rightarrow\}\, | \geq 1$. In other words, a complete FSM contains no deadlocks, i.e., each state has always at least one outgoing transition *for every alphabet symbol*. Without loss of generality we shall assume in the remainder of this thesis that all FSM that we manipulate are always complete, unless otherwise stated.

Finite state machines admit either finite or infinite executions called *runs*. A run of an FSM is a finite or infinite path inside the machine which follows the transition relation. Runs which originate from the initial state are called *initial*. Moreover, every run of a finite state machine is classified as either *accepting* or *rejecting* (non-accepting). In the case of finite runs, the acceptance criterion is simple: a finite run is accepting iff it ends in a final state. For infinite runs, we consider the so-called *Büchi* acceptance condition. A run is accepting for Büchi iff it visits final states *infinitely often*. Many other acceptance conditions have been studied in automata theory, such as Streett, Rabin, Parity, generalized Büchi, co-Büchi and others. In this thesis, we will be mainly concerned with the Büchi acceptance condition for infinite words. We can now formally define the notion of runs of an FSM.

Definition 2.3.2 (Run of an FSM)**.** *Let* $M = \langle Q, \Sigma, I, \rightarrow, F \rangle$ *be an* FSM *and let* $w \in \Sigma^* \cup \Sigma^\omega$ *be a word. A* run *of the finite state machine* M *over*

w is a sequence of states $\rho \in Q^* \cup Q^\omega$ such that $|\rho| = |w| + 1$ and for every position $i \in \{0, \ldots, |w| - 1\}$ we have that $\langle \rho[i], w[i], \rho[i+1] \rangle \in \to$. A run ρ is initial *iff $\rho[0] \in I$. The set of initial runs of M over w is denoted* $\mathrm{Init}(M, w)$. *A* finite *run ρ over a finite word w is* accepting *iff $\rho[|w|] \in F$. For any infinite run ρ, we denote by $\mathrm{Inf}(\rho) \subseteq Q$ the set of states which appear infinitely often in ρ. Notice that for any infinite run ρ, $\mathrm{Inf}(\rho)$ contains at least one state. An infinite run ρ is* accepting for Büchi *iff $\mathrm{Inf}(\rho) \cap F \neq \varnothing$.*

The completeness assumption for FSM implies that for every word $w \in \Sigma^* \cup \Sigma^\omega$ a deterministic finite state machine admits *exactly one initial run* over w. Deterministic FSM therefore admit two natural language interpretations: one for finite words, and one for infinite words.

In the case of *non-deterministic* FSM, each finite or infinite word corresponds to at least one, but possibly many runs. For non-deterministic FSM, we therefore duplicate each language interpretation in terms of *existential* or *universal* acceptance. Intuitively, a word is accepted in an existential language interpretation if and only if the machine admits at least one initial accepting run on that word; dually, a word w is accepted in the universal language interpretation if and only if every initial run over w is accepting.

We now formally define each of the six language interpretations of finite state machines that we consider in this thesis.

Definition 2.3.3 (Language of an FSM)**.** *Let $M = \langle Q, \Sigma, I, \to, F \rangle$ be a finite state machine. The* existential finite-word language *of M is the set of finite words $L^{\exists}_{\mathsf{F}}(M) = \{w \in \Sigma^* \mid \exists\, \rho \in \mathrm{Init}(M, w) : \rho \text{ is accepting}\}$. The* universal finite-word language *of M is the set of finite words $L^{\forall}_{\mathsf{F}}(M) = \{w \in \Sigma^* \mid \forall\, \rho \in \mathrm{Init}(M, w) : \rho \text{ is accepting}\}$. If the machine M is deterministic, we define the* deterministic finite-word language *of M as the set of finite words $L^{\mathsf{D}}_{\mathsf{F}}(M) = L^{\exists}_{\mathsf{F}}(M) = L^{\forall}_{\mathsf{F}}(M)$. The* existential Büchi language *of M is the set of infinite words $L^{\exists}_{\mathsf{B}}(M) = \{w \in \Sigma^\omega \mid \exists \rho \in \mathrm{Init}(M, w) : \rho \text{ is accepting for Büchi}\}$. The* universal Büchi language *of M is the set of infinite words $L^{\forall}_{\mathsf{B}}(M) = \{w \in \Sigma^\omega \mid \forall\, \rho \in \mathrm{Init}(M, w) : \rho \text{ is accepting for Büchi}\}$. If the machine M is deterministic, we define the* deterministic Büchi language *of M as the set of infinite words $L^{\mathsf{D}}_{\mathsf{B}}(M) = L^{\exists}_{\mathsf{B}}(M) = L^{\forall}_{\mathsf{B}}(M)$.*

FSM language interpretation	Automaton	Language Class
deterministic finite-word : $L_{\mathsf{F}}^{\mathsf{D}}$	DFA	regular
existential finite-word : $L_{\mathsf{F}}^{\exists}$	NFA	regular
universal finite-word : $L_{\mathsf{F}}^{\forall}$	UFA	regular
deterministic Büchi : $L_{\mathsf{B}}^{\mathsf{D}}$	DBA	*deterministic* ω-regular
existential Büchi : $L_{\mathsf{B}}^{\exists}$	NBA	ω-regular
universal Büchi : $L_{\mathsf{B}}^{\forall}$	UBA	*deterministic* ω-regular

Table 2.1: Language interpretations for finite state machines.

A *finite state automaton* is a finite state machine paired with a language interpretation. For instance, a *non-deterministic finite automaton* (NFA) is a non-deterministic finite state machine interpreted existentially over finite words[2]; a *universal Büchi automaton* (UBA) is a non deterministic finite state machine interpreted universally with a Büchi acceptance condition, and so on. The correspondence between language interpretations for finite state machines and automata is depicted in Table 2.1. We denote the language of an automaton A simply by $L(A)$, as the language interpretation is given by the automaton type.

The six language interpretations of Definition 2.3.3 correspond to three distinct *language classes*. Deterministic, non-deterministic and universal finite automata all have an expressive power that is equivalent to the class of *regular languages*. In the case of ω-languages (i.e., languages over infinite words), deterministic and universal Büchi automata are strictly less expressive than their non-deterministic counterpart; the class of languages that are representable with NBA are called the ω-regular languages, while DBA and UBA are only complete for the class of *deterministic ω-regular languages*.

[2]In the literature, the adjective *non-deterministic* has two distinct meanings: the first is the syntactic criterion of finite state machines that we defined previously; the second is the *semantic* criterion of existentially quantifying over runs to obtain a language interpretation. In this thesis we use the term for both meanings.

2.3.3 Alternating Automata

We now formally define the mathematical object that will be the entire focus of this thesis: the *alternating automaton*. The concept of *alternation* in computer science was introduced in the early 1980's by Chandra et al. [CKS81] as a generalization of the existential and universal language interpretations for finite state machines[3].

One possible way of defining alternating finite automata is to partition the set of states into *existential* and *universal* states. In this view, an *alternating* language interpretation of the underlying FSM appears naturally: the quantification over initial accepting runs alternates between existential and universal quantifiers depending on the nature of the current state.

A less intuitive but more convenient way of defining alternating automata is to *augment* the syntax of finite state machines more invasively by replacing the transition relation by an *alternating transition function* that maps each state and alphabet symbol to a *positive Boolean formula* over the set of states Q. This allows each state to be either fully existential (by using only disjunctions) or fully universal (by using only conjunctions), but it is also possible to freely mix conjunctions and disjunctions, which makes for a convenient and clean syntax.

Definition 2.3.4 (Alternating Automaton)**.** *An* alternating automaton *is a tuple* $\langle Q, \Sigma, q_0, \delta, F \rangle$ *where:*

- $Q = \{q_1, \ldots, q_n\}$ *is a finite* set of states*;*
- $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ *is a finite* alphabet*;*
- $q_0 \in Q$ *is the* initial state*;*
- $\delta : Q \times \Sigma \mapsto \mathsf{BL}^+(Q)$ *is an* alternating transition function*;*
- $F \subseteq Q$ *is a set of* final states*.*

In contrast with finite state machines which have linear runs called paths, the runs of alternating automata are defined as rooted, directed acyclic graphs.

[3]The work of Chandra et al in [CKS81] actually goes much beyond finite state machines. The paper explores many aspects of alternation and studies the properties of *alternating Turing machines*, *alternating pushdown automata* and more.

Definition 2.3.5 (Run of an Alternating Automaton)**.** *Let* $A = \langle Q, \Sigma, q_0, \delta, F \rangle$ *by an alternating automaton and let* $w \in \Sigma^* \cup \Sigma^\omega$ *be a word. A* run *of* A *over* w *is directed acyclic graph* $G = \langle V, \to \rangle$ *where:*

- $V \subseteq Q \times \mathbb{N}$ *is a finite or infinite set of vertices. The vertex* $\langle q, i \rangle$ *represents an active state in* Q *of depth* i*. Also,* G *is bounded by the size of* w *such that* $\langle q, i \rangle \in V$ *implies* $i \leq |w|$*;*
- G *is rooted by* $\langle q_0, 0 \rangle$*, i.e., the initial state at depth 0;*
- $\to \subseteq V \times V$ *is such that:*
  1. $\langle q, i \rangle \to \langle q', j \rangle$ *implies* $j = i + 1$*;*
  2. $\forall \langle q, i \rangle \in V, i < |w| : \{ q' \mid \langle q, i \rangle \to \langle q', i + 1 \rangle \} \models \delta(q, w[i])$.

We denote by $\mathrm{Runs}(A, w)$ *the set of runs of* A *over* w*.*

In the definition above, the positive Boolean formulas of the alternating transition function express both existential and universal types of transitions. To construct a run over a word w, each vertex $\langle q, i \rangle$ must be connected to a set of children vertices of the form $\langle q', i + 1 \rangle$; the set of possible sets of children vertices is identified by the *set of valuations* of the positive Boolean formula $\delta(q, w[i])$.

Definition 2.3.6 (Language of Alternating Automata)**.** *Let* $A = \langle Q, \Sigma, q_0, \delta, F \rangle$ *by an alternating automaton, and let* $G = \langle V, \to \rangle$ *be a run of* A *over some word* $w \in \Sigma^* \cup \Sigma^\omega$*. A* branch *of the run* G *is a sequence of states* $\beta \in Q^* \cup Q^\omega$ *such that* $\beta[0] = q_0$ *and* $\forall\, i \in \mathbb{N}, i < |w| : \langle \langle \beta[i], i \rangle, \langle \beta[i + 1], i + 1 \rangle \rangle \in \to$*. We denote by* $\mathrm{Branches}(G)$ *the set of branches of a run* G*. The* alternating finite-word language *of* A *is the set of finite words* $L^{\mathsf{A}}_{\mathsf{F}}(A) \stackrel{\mathsf{def}}{=} \{ w \in \Sigma^* \mid \exists\, G \in \mathrm{Runs}(A, w), \forall\, \beta \in \mathrm{Branches}(G) : \beta[|w|] \in F \}$*. Let* $\mathrm{Inf}(\beta) \subseteq Q$ *be the set of states which occur infinitely along an infinite branch* $\beta \in Q^\omega$*. The* alternating Büchi language *of* A *is the set of infinite words* $L^{\mathsf{A}}_{\mathsf{B}}(A) \stackrel{\mathsf{def}}{=} \{ w \in \Sigma^\omega \mid \exists\, G \in \mathrm{Runs}(A, w), \forall\, \beta \in \mathrm{Branches}(G) : \mathrm{Inf}(\beta) \cap F \neq \varnothing \}$.

The language interpretations for alternating automata are similar to their counterpart for finite state machines. Over finite words, a run is accepting iff every branch ends in a final state; in the case of infinite words, a run is accepting for Büchi iff every branch encounters a final state infinitely often.

We define one alternating automaton type for each language interpretation: *alternating finite automata* (AFA, for short) are alternating automata interpreted over finite words, *alternating Büchi automata* (ABA) are interpreted over infinite words. It is well-known that AFA accept exactly the regular languages, and that ABA are complete for the class of ω-regular languages [CKS81, MH84].

2.3.4 Automata Decision Problems

We consider several decision problems for automata. The *emptiness problem* asks, for an automaton A, whether $L(A) = \varnothing$. The *finite-word* (respectively *infinite-word*) *universality problem* asks, for an automaton A, whether $L(A) = \Sigma^*$ (respectively $L(A) = \Sigma^\omega$). The *language inclusion problem* asks, for two automata A_1, A_2, whether $L(A_1) \subseteq L(A_2)$.

Classical solutions to the universality and language inclusion problems typically use a form of reduction to the emptiness problem. Indeed, for any automaton A, one can construct an automaton A' (possibly of another type[4]) such that $L(A') = \overline{L(A)}$ Clearly, the automaton A is universal if and only if A' is empty. Likewise, for any pair of automata A_1, A_2, we have that $L(A_1) \subseteq L(A_2)$ if and only if $L(A_1) \cap \overline{L(A_2)} = \varnothing$.

Solving the emptiness problem for DFA and NFA is easy: one only needs to check for the existence of a path between the initial state and a final state, which is an NLogSpace-Complete problem [Pap94]. For UFA and AFA, a time-optimal solution is to translate the input automaton into an equivalent NFA and check for emptiness. This translation to NFA blows up exponentially in the worst case. In fact, its is well-known that both problems are PSpace-Complete [MS72]. The existence of an equivalent NFA for UFA and AFA is formalized by the following theorem

Theorem 2.3.7. *(Translation from* UFA *or* AFA *to* NFA*) For any universal or alternating automaton A with k states, there exists an* NFA *A' with at most 2^k states such that $L(A) = L(A')$. Moreover, for every $k \in \mathbb{N}_0$, there*

[4]For some types of automata, resorting to another automaton type for complementation is necessary, e.g., NBA cannot be complemented directly.

exists a universal (and therefore also alternating) automaton A such that the smallest NFA *A' such that $L(A) = L(A')$ has 2^k states.*

As this translation from AFA to NFA is central to the remainder of this thesis, we now define this construction in detail.

Definition 2.3.8 (Subset Construction)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* AFA*. The* subset construction *of A is the* NFA $\mathsf{SC}(A) = \langle Q', \Sigma, I, \to, F'\rangle$ *with:*

- $Q' = 2^Q$;
- $I = \{c \subseteq Q \mid q_0 \in c\}$;
- $\to = \left\{\langle c_1, \sigma, c_2\rangle \;\middle|\; c_2 \models \bigwedge_{q\in c_1} \delta(q,\sigma)\right\}$;
- $F' = 2^F$.

Lemma 2.3.9. *(Correctness of Subset Construction [CKS81]) For every* AFA *A, the subset construction* NFA *$A' = \mathsf{SC}(A)$ is such that $L(A') = L(A)$.*

We now turn our attention to the emptiness problem for automata over infinite words. In the case of DBA and NBA, solving emptiness is again easy: it is sufficient to check for the existence of a path between the initial state and a final state, and to check that this final state is part of a cycle. The problem of deciding the existence of such a *lasso path* is also NLogSpace-Complete. For UBA and ABA, deciding emptiness has the same complexity than their finite-word counterparts and is also PSpace-Complete. As for the finite-word case, both UBA and ABA can be translated into equivalent NBA, as formalized by the following theorem due to Miyano and Hayashi [MH84].

Theorem 2.3.10. *(Translation from* UBA *or* ABA *to* NBA*) For any universal or alternating Büchi automaton A with k states, there exists an* NBA *A' with at most $(2^k)^2$ states such that $L(A) = L(A')$.*

Definition 2.3.11 (Miyano Hayashi Construction)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* ABA*. To alleviate the notations, we introduce the following shorthand: $x \overset{\sigma}{\rightsquigarrow} y \overset{\mathsf{def}}{=} y \models \bigwedge_{q\in x} \delta(q,\sigma)$. The* Miyano Hayashi construction *of A is the* NBA $\mathsf{MH}(A) = \langle Q', \Sigma, I, \to, F'\rangle$ *with:*

- $Q' = 2^Q \times 2^Q$;
- $I = \{(\{q_0\}, \varnothing)\}$;
- $\rightarrow = \left\{\langle (s_1, \varnothing), \sigma, (s_2, s_2 \setminus F)\rangle \;\middle|\; s_1 \overset{\sigma}{\rightsquigarrow} s_2\right\} \cup \left\{\langle (s_1, o_1 \neq \varnothing), \sigma, (s_2, o_2 \setminus F)\rangle \;\middle|\; s_1 \overset{\sigma}{\rightsquigarrow} s_2,\ o_1 \overset{\sigma}{\rightsquigarrow} o_2,\ o_2 \subseteq s_2\right\}$;
- $F' = 2^Q \times \{\varnothing\}$.

Lemma 2.3.12. *(Correctness of* MH *Construction) For any* ABA A*, the Miyano Hayashi construction* NBA $A' = \mathsf{MH}(A)$ *is such that* $L(A') = L(A)$.

2.3.5 Fixed Points for Automata Emptiness

In this thesis, we develop new algorithms for the emptiness of alternating automata based on the translations to non-deterministic automata. To decide the emptiness of non-deterministic automata, one must either find an accepting and initial path, or find an accepting and initial lasso path. When non-deterministic automata are given explicitly, a classical and optimal solution for these path-finding problems is to use *depth-first search* and *nested depth-first search* [HPY96], which are both linear-time algorithms. In this thesis however, we consider algorithms which solve the emptiness problem using *breadth-first search* (BFS, for short) and *repeated breadth-first search*. This class of algorithms enjoy elegant descriptions in terms of *fixed points of monotonic functions* that contain *state space operators*.

In the context of finite state machines, we call a *state space operator* any monotonic function of the form $f\colon 2^Q \mapsto 2^Q$, where Q is the set of states of the FSM. The operators that we use in this thesis are pre, post, cpre, and cpost.

Definition 2.3.13 (Statespace operators for FSM)**.** *Let* $M = \langle Q, \Sigma, I, \rightarrow, F\rangle$ *be a finite state machine. We define the following* state space operators*:*

$$
\begin{aligned}
\mathsf{pre}_\sigma[M](X) &= \{q_1 \in Q \mid \exists \langle q_1, \sigma, q_2\rangle \in \rightarrow : q_2 \in X\} \\
\mathsf{post}_\sigma[M](X) &= \{q_2 \in Q \mid \exists \langle q_1, \sigma, q_2\rangle \in \rightarrow : q_1 \in X\} \\
\mathsf{cpre}_\sigma[M](X) &= \{q_1 \in Q \mid \forall \langle q_1, \sigma, q_2\rangle \in \rightarrow : q_2 \in X\} \\
\mathsf{cpost}_\sigma[M](X) &= \{q_2 \in Q \mid \forall \langle q_1, \sigma, q_2\rangle \in \rightarrow : q_1 \in X\} \\
\mathsf{pre}[M](X) &= \bigcup_{\sigma\in\Sigma} \mathsf{pre}_\sigma[M](X) \\
\mathsf{post}[M](X) &= \bigcup_{\sigma\in\Sigma} \mathsf{post}_\sigma[M](X) \\
\mathsf{cpre}[M](X) &= \bigcap_{\sigma\in\Sigma} \mathsf{pre}_\sigma[M](X) \\
\mathsf{cpost}[M](X) &= \bigcap_{\sigma\in\Sigma} \mathsf{post}_\sigma[M](X)
\end{aligned}
$$

The pre and post operators respectively compute the set of *predecessors* and the set of *successors* of a set of states. The operation $\mathsf{cpre}(X)$ computes the set of *controllable predecessors*[5] (or *unavoidable predecessors*) of X, i.e., the set of states which *cannot avoid* going into X in one step. Likewise, the operation $\mathsf{cpost}(X)$ computes of set of *controllable successors* (or *unavoidable successors*) of X, i.e. the set of states which *surely come from* a state in X, at the previous step.

Definition 2.3.14 (Shorthands for Common Least Fixed Points)**.** *Some commonly used least fixed points of functions that are based on* pre *and* post *are abbreviated as follows:*

$$
\begin{aligned}
\mathsf{pre}^*(X) &\stackrel{\mathsf{def}}{=} \mathsf{LFP}(\lambda Y \cdot \mathsf{pre}(Y) \cup X) \\
\mathsf{pre}^+(X) &\stackrel{\mathsf{def}}{=} \mathsf{LFP}(\lambda Y \cdot \mathsf{pre}(Y) \cup \mathsf{pre}(X)) \\
\mathsf{post}^*(X) &\stackrel{\mathsf{def}}{=} \mathsf{LFP}(\lambda Y \cdot \mathsf{post}(Y) \cup X) \\
\mathsf{post}^+(X) &\stackrel{\mathsf{def}}{=} \mathsf{LFP}(\lambda Y \cdot \mathsf{post}(Y) \cup \mathsf{post}(X))
\end{aligned}
$$

The following backward fixed-point computes the set of states of an FSM which can reach a final state in zero or more steps.

Definition 2.3.15 (Backward BFS)**.** *Let $M = \langle Q, \Sigma, I, \rightarrow, F\rangle$ be an* FSM*. The* backward-reachable *states of M, denoted $\mathsf{B}^*(M)$, are defined as follows:*

$$
\begin{aligned}
\mathsf{B}^0(M) &= F \\
\mathsf{B}^i(M) &= \mathsf{B}^{i-1}(M) \cup \mathsf{pre}(\mathsf{B}^{i-1}(M)) \\
\mathsf{B}^*(M) &= \mathsf{LFP}(\lambda X \cdot \mathsf{pre}(X) \cup F) = \mathsf{pre}^*(F)
\end{aligned}
$$

[5]The terminology of *controllable predecessors* is borrowed from game theory, where this operator is used to synthesize strategies on game graphs.

Symmetrically, the following forward fixed point computes the set of states of an FSM which can be reached from an initial state in zero or more steps.

Definition 2.3.16 (Forward BFS)**.** *Let* $M = \langle Q, \Sigma, I, \to, F \rangle$ *be an* FSM*. The* forwards-reachable *states of* M*, denoted* $\mathsf{F}^*(M)$*, are defined as follows:*

$$\begin{aligned} \mathsf{F}^0(M) &= I \\ \mathsf{F}^i(M) &= \mathsf{F}^{i-1}(M) \cup \mathsf{post}(\mathsf{F}^{i-1}(M)) \\ \mathsf{F}^*(M) &= \mathsf{LFP}(\lambda X \cdot \mathsf{post}(X) \cup I) = \mathsf{post}^*(I) \end{aligned}$$

We call *cycling state* a state of an FSM which belongs to a non-trivial cycle (a cycle of length at least 1). The following nested backward fixed point computes the set of final states which can reach a cycling final state in zero steps or more.

Definition 2.3.17 (Repeated Backward BFS)**.** *Let* $M = \langle Q, \Sigma, I, \to, F \rangle$ *be an* FSM*. The* repeatedly backward-reachable *final states of* M*, denoted* $\mathsf{BB}^*(M)$*, are defined as follows:*

$$\begin{aligned} \mathsf{BB}^0(M) &= F \\ \mathsf{BB}^i(M) &= F \cap \mathsf{pre}^+(\mathsf{BB}^{i-1}(M)) \\ \mathsf{BB}^*(M) &= \mathsf{GFP}(\lambda X \cdot \mathsf{pre}^+(X) \cap F) \end{aligned}$$

Finally, the following nested forward fixed point computes the set of final states which are (i) reachable from an initial state in zero steps or more, and (ii) reachable from a final cycling state in zero steps or more.

Definition 2.3.18 (Repeated Forward BFS)**.** *Let* $M = \langle Q, \Sigma, I, \to, F \rangle$ *be an* FSM*. The* repeatedly forward-reachable *final states of* M*, denoted* $\mathsf{FF}^*(M)$*, are defined as follows:*

$$\begin{aligned} \mathsf{FF}^0 &= F \cap \mathsf{post}^*(I) \\ \mathsf{FF}^i &= F \cap \mathsf{post}^+(\mathsf{FF}^{i-1}) \\ \mathsf{FF}^* &= \mathsf{GFP}(\lambda X \cdot \mathsf{post}^+(X) \cap F \cap \mathsf{post}^*(I)) \end{aligned}$$

We can now formalize how these fixed point algorithms relate to the emptiness decision problems for NFA and NBA.

Theorem 2.3.19. *(Language Emptiness of* NFA *with Fixed Points) Let* $A = \langle Q, \Sigma, I, \to, F \rangle$ *be an* NFA*. We have that* $L(A) = \varnothing$ *if and only if* $\mathsf{B}^* \cap I = \varnothing$ *if and only if* $\mathsf{F}^* \cap F = \varnothing$*.*

Theorem 2.3.20. *(Language Emptiness of* NBA *with Fixed Points) Let* $A = \langle Q, \Sigma, I, \to, F \rangle$ *be an* NBA*. We have that* $L(A) = \varnothing$ *if and only if* $\mathsf{pre}^*(\mathsf{BB}^*) \cap I = \varnothing$ *if and only if* $\mathsf{FF}^* = \varnothing$*.*

2.4 Kripke Structures and LTL

2.4.1 Kripke Structures

A *propositional Kripke structure* is a finite-state model of a system, which describes how binary properties of that system can evolve over time. Syntactically, Kripke structures are very similar to the finite state machines we have defined previously. They differ only in by following aspects: first, they do not have final states, and second, their transition function is not labeled with alphabet symbols. Instead, they contain an additional *labeling function*, which maps each state to a *valuation* over a finite set of Boolean propositions.

Definition 2.4.1 (Propositional Kripke Structure)**.** *A propositional Kripke structure is a tuple* $\mathcal{K} = \langle Q, I, \mathbb{P}, \mathcal{L}, \to \rangle$ *where:*

- $Q = \{q_1, \ldots, q_n\}$ *is a finite set of* states*;*
- $\mathbb{P} = \{p_1, \ldots, p_k\}$ *is a finite set of* Boolean propositions*;*
- $I \subseteq Q$ *is a finite set of* initial states*;*
- $\to \subseteq Q \times Q$ *is a* transition relation*;*
- $\mathcal{L} \colon Q \to 2^{\mathbb{P}}$ *is* labeling function*.*

The relation $\to$ *must be complete, i.e.,* $\forall\, q \in Q\ \exists\, q' \in Q \colon \langle q, q' \rangle \in \to$*.*

We have seen previously that a run of a finite state machine associates a path to a given word. Kripke structures do effectively the opposite, they associate a word to a given path. This is formalized by the following definition.

Definition 2.4.2 (Runs and Language of a Kripke Structure)**.** *Let* $\mathcal{K} = \langle Q, I, \mathbb{P}, \mathcal{L}, \rightarrow \rangle$ *be a Kripke structure. A* run *of* $\mathcal{K}$ *is a sequence* $\rho \in Q^* \cup Q^\omega$ *of states such that* $|\rho| = |w| + 1$ *and for every position* $i \in \{0, \ldots, |w| - 1\}$ *we have that* $\langle \rho[i], \rho[i+1] \rangle \in \rightarrow$*. A run* ρ *is* initial *iff* $\rho[0] \in I$*. The labeling function is extended to runs in the following natural way:* $\forall\, i \colon \mathcal{L}(\rho)[i] = \mathcal{L}(\rho[i])$*. We denote the set of runs of a Kripke structure* $\mathcal{K}$ *by* $\mathrm{Runs}(\mathcal{K})$*. The* language *of* $\mathcal{K}$*, denoted* $L(\mathcal{K})$ *is defined as the set of finite or infinite words* $\{\mathcal{L}(\rho) \mid \rho \in \mathrm{Runs}(\mathcal{K}) \wedge \rho \text{ is initial}\}$*.*

The reader will notice that there are actually two distinct semantics embedded in the previous definition: a *finite-word* semantics and an *infinite-word* semantics for Kripke structures. As we have stated previously, we do not consider languages which contain both finite and infinite words, so the expression $L(\mathcal{K})$ must be understood as either a subset of Σ^* or Σ^ω ($\Sigma = 2^{\mathbb{P}}$ in this case), but this will be clear from the context. Notice that since Kripke structures do not have an acceptance condition, they define finite-word languages which are *prefix-closed*.

2.4.2 Linear Temporal Logic

Definition 2.4.3 (Syntax of Linear Temporal Logic)**.** *Let* $\mathbb{P}$ *be a finite set of propositions, and let* $p \in \mathbb{P}$*. The set of syntactically correct* formulas of the linear temporal logic *over* $\mathbb{P}$ *is defined recursively using the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathsf{X}\,\varphi \mid \varphi_1\,\mathsf{U}\,\varphi_2$$

We denote the set of formulas of the linear temporal logic over $\mathbb{P}$ *by* $\mathsf{LTL}(\mathbb{P})$*.*

We can see from the definition above that the linear temporal logic is a syntactic proper superset of propositional Boolean logic, which introduces extra *temporal modalities*. These are the *next-time* modality X, and the *until* modality U. Additionally, we define the following syntactic shortcuts: $\mathsf{F}\,\varphi \stackrel{\mathsf{def}}{=} \mathsf{true}\,\mathsf{U}\,\varphi$ is the *finally* modality (it reads "*finally* φ"), and $\mathsf{G}\,\varphi \stackrel{\mathsf{def}}{=} \neg\mathsf{F}\,\neg\varphi$ is the *globally* modality (it reads "*globally* φ").

Definition 2.4.4 (Semantics of Linear Temporal Logic)**.** *Let* $\Sigma = 2^{\mathbb{P}}$*. The* semantics *of an LTL formula* φ*, which we denote by* $[\![\varphi]\!]$*, is a subset of* $\Sigma^* \cup \Sigma^\omega$ *as defined by the following semantic rules. Let* $w \in \Sigma^* \cup \Sigma^\omega$*.*

- $w \in [\![p]\!]$ *iff* $p \in w[0]$
- $w \in [\![\neg\varphi]\!]$ *iff* $w \notin [\![\varphi]\!]$
- $w \in [\![\varphi_1 \wedge \varphi_2]\!]$ *iff* $w \in [\![\varphi_1]\!]$ *and* $w \in [\![\varphi_2]\!]$
- $w \in [\![\varphi_1 \vee \varphi_2]\!]$ *iff* $w \in [\![\varphi_1]\!]$ *or* $w \in [\![\varphi_2]\!]$ *or both*
- $w \in [\![\mathsf{X}\,\varphi]\!]$ *iff* $|w| > 1$ *and* $w[1\ldots] \in [\![\varphi]\!]$
- $w \in [\![\varphi_1 \,\mathsf{U}\, \varphi_2]\!]$ *iff* $\exists\, i < |w| : w[i\ldots] \in [\![\varphi_2]\!] \wedge\ \forall\, j < i : w[j\ldots] \in [\![\varphi_1]\!]$

Similar to the case of Kripke structures, two semantics are embedded in the previous definition: a *finite-word* semantics and an *infinite-word* semantics. Again, we do not consider languages which contain both finite and infinite words, so the expression $[\![\varphi]\!]$ must be understood as either a subset of Σ^* or Σ^ω depending on the context.

Chapter 3

Antichain Approaches to Automata Emptiness

In this chapter, we present a general framework for solving the reachability and repeated reachability problems on finite-state systems. This framework exploits simulation preorders that exist by construction on the state space of some classes of FSM, like the ones obtained by subset construction. Simulation preorders are used to efficiently evaluate the reachability and repeated reachability fixed point expressions by maintaining, at every iteration of the fixed point, only the minimal or maximal states with respect to the simulation. When the simulation preorder is a partial order (we always assume that is the case in this thesis), such set of minimal or maximal elements form an *antichain* of states that is a canonical representation. In the second part of the chapter, we instantiate the antichains framework to the subset construction of AFA and the Miyano-Hayashi construction of ABA.

3.1 Introduction

Finite-state machines and automata are essential tools for the formal specification and verification of programs. For instance, in the *automata-theoretic approach to program verification* [VW86, VW94, Kur94], both the system and the property to verify are given as automata. For a system modeled as an automaton A, and a property modeled as an automaton B, the ver-

ification problem translates naturally to the language inclusion question $L(A) \subseteq L(B)$. The intuition is that a word of the automaton corresponds to a particular behavior of the system, so that the language inclusion question reads as : *"is every behavior of the program model included in the set of allowed behaviors of the specification ?"*. The classical approach to that problem is to solve the emptiness question $L(A \otimes \overline{B}) \stackrel{?}{=} \varnothing$, where $\overline{B}$ is the *complement automaton* of B, and $\otimes$ is the *synchronous product* automata-theoretic operation corresponding to language intersection.

In the case where A and B are given as non-deterministic automata over finite words, the non-emptiness of $L(A \otimes \overline{B})$ is classically solved by the *determinization* of B using the *subset construction*. Once B is determinized, the complement automaton is obtained simply by switching accepting and non-accepting states. Even when carefully implemented (e.g., see [TV05] for a survey), the subset construction needed by the determinization step may blow up exponentially. This is unavoidable in general, since the language inclusion problem for finite automata is known to be PSpace-Complete [MS72].

When the automata A and B are given as non-deterministic automata over *infinite* words, things unfortunately get more difficult than in the finite word case [Mic88]. Büchi first proved in the early 1960's that NBA can be complemented [Büc62], but his procedure entailed a doubly exponential blowup in the worst case. Later, Safra proposed a $2^{\mathcal{O}(n \log n)}$ optimal construction [Saf88], which found many theoretical applications, but this construction is notoriously complex and very difficult to implement [ATW06]. More than forty years after Büchi's seminal work, specifications in the form of NBA with more 10 states were still largely considered intractable [GKSV03].

In recent years, so-called *antichain*-based algorithms have dramatically improved the situation both on the finite-word [DDHR06, ACH$^+$10] and the infinite-word [DR07, DR10] cases, and have been shown to outperform classical constructions by orders of magnitude. The main idea of these antichain-based algorithms is to exploit simulation preorders that exist by construction on the FSM generated by the exponential subset constructions. Using these simulation preorders, it is possible to solve the reachability and repeated reachability problems more efficiently by pruning the correspond-

ing fixed point evaluations, and consider only minimal or maximal elements (that is, an antichain). The core ideas of the antichain framework for finite automata can be succinctly summarized by the two following self-dual statements. For *any* finite state machine we have that:

the pre *operator preserves forward-simulation downward-closedness;*

the post *operator preserves backward-simulation upward-closedness.*

A preorder $\preceq_{\mathsf{f}}$ is a *forward simulation* of an FSM if for every letter σ, whenever $q_1 \xrightarrow{\sigma} q_3$ and $q_2 \preceq_{\mathsf{f}} q_1$, there exists q_4 such that $q_2 \xrightarrow{\sigma} q_4$ and $q_4 \preceq_{\mathsf{f}} q_3$. When $q_1 \preceq_{\mathsf{f}} q_2$ we say that q_1 forward-simulates q_2. Consider a finite state machine whose state space is equipped with a forward simulation $\preceq_{\mathsf{f}}$. In that context, a $\preceq_{\mathsf{f}}$-downward-closed set of states is a set of states of the FSM comprising a finite number of states, along with every other state by which they can be forward-simulated. Now consider a predecessor state q of a state q' that belongs to a $\preceq_{\mathsf{f}}$-downward-closed set X. By definition of forward-simulation, every state which simulates q can lead to a state which simulates q', and that state necessarily belongs to X since it is downward-closed. Therefore, it is easy to see that the set of predecessors of a $\preceq_{\mathsf{f}}$-downward-closed set is again a $\preceq_{\mathsf{f}}$-downward-closed set. By dualizing the arguments, we can show that the set of successors of an $\succeq_{\mathsf{b}}$-upward-closed set of states is again $\succeq_{\mathsf{b}}$-upward-closed.

Thanks to this powerful *closedness-preserving* property of pre and post with respect to simulation preorders, it is possible to evaluate fixed point expressions on FSM such that every iteration is upward- or downward-closed. We can therefore canonically (see Theorem 2.1.2) represent each of those sets by using antichains, a representation that is *exponentially more compact* than the underlying upward- or downward-closed set in the best case.

Related Works In about five years of existence, the research topic of *antichains for program verification* has yielded a number of contributions. Antichain algorithms have been developed to solve *games of imperfect information* [DDR06, RCDH07, BCD+08], which are two-player games played on graphs, in which the players do not have perfect knowledge of the opponent's positions, and have to rely on imperfect *observations*. Most antichain algorithms developed so far improve on classical automata-theoretic

questions, such as the *universality problem for* NFA [DDHR06], the *universality and language inclusion problems for* NBA and the *emptiness problem* of ABA [DR07, DR09], and the *universality and language inclusion problems of non-deterministic tree automata* [BHH+08]. Antichain algorithms have also been developed for solving the *satisfiability and model checking problems for* LTL [DDMR08b] (the topic of the next chapter), and for solving the *realizability problem* for LTL [FJR09]. Most recently, new provably better antichain algorithms have been devised, that use a combination of antichains and simulations [ACH+10, DR10].

Structure of the Chapter In Section 3.2 we formally define the notions of forward and backward simulations on FSM, and we define and prove the aforementioned closedness-preserving properties of pre and post. In Section 3.3 we provide generic algorithms for decomposing the computation of pre and post and for manipulating antichains. In Section 3.4 we define fixed points on the lattice of antichains of states to solve the reachability and repeated reachability problems for FSM, both in forward and backward directions. In Section 3.5 we instantiate the antichains framework on the emptiness problem for AFA and the emptiness problem of ABA. Finally, in Section 3.6 we provide some discussion.

3.2 Simulation Preorders of Finite Automata

Definition 3.2.1 (Simulation Preorders of an FSM)**.** *For any finite state machine $M = \langle Q, \Sigma, I, \rightarrow, F \rangle$, we define two notions of simulation:*

- *A preorder $\preceq_{\mathsf{f}} \subseteq Q \times Q$ is a* forward simulation *of M iff for all $\sigma \in \Sigma$, for all $q_1, q_2, q_3 \in Q$, if $q_2 \preceq_{\mathsf{f}} q_1$ and $\langle q_1, \sigma, q_3 \rangle \in \rightarrow$ then there exists $q_4 \in Q$ such that $\langle q_2, \sigma, q_4 \rangle \in \rightarrow$ and $q_4 \preceq_{\mathsf{f}} q_3$. The expression "$x \preceq_{\mathsf{f}} y$" reads "x forward-simulates y".*
- *A preorder $\succeq_{\mathsf{b}} \subseteq Q \times Q$ is a* backward simulation *of M iff for all $\sigma \in \Sigma$, for all $q_1, q_2, q_3 \in Q$, if $q_2 \succeq_{\mathsf{b}} q_1$ and $\langle q_3, \sigma, q_1 \rangle \in \rightarrow$ then there exists $q_4 \in Q$ such that $\langle q_4, \sigma, q_2 \rangle \in \rightarrow$ and $q_4 \succeq_{\mathsf{b}} q_3$. The expression "$x \succeq_{\mathsf{b}} y$" reads "x backward-simulates y".*

Remark 3.2.2. *The rationale behind the orientation of $\preceq_{\mathsf{f}}$ and $\succeq_{\mathsf{b}}$ comes from the fact that $\subseteq$ and $\supseteq$ are respectively forward and backward simulation preorders on several classical subset constructions, making for a useful mnemonic.*

Remark 3.2.3. *In this thesis, we restrict ourselves to the case where simulation preorders are also antisymmetric (and are hence* partial orders*).*

We say that a simulation preorder over $Q \times Q$ is *compatible* with a set of states $X \subseteq Q$ if for all $q_1, q_2 \in Q$, if $q_1 \in X$ and q_2 (forward or backward) simulates q_1, then $q_2 \in X$. Compatibility is therefore equivalent to *closedness* in the following sense: a forward simulation $\preceq_{\mathsf{f}}$ is compatible with X iff X is downward-closed for $\preceq_{\mathsf{f}}$; a backward simulation $\succeq_{\mathsf{b}}$ is compatible with X iff X is upward-closed for $\succeq_{\mathsf{b}}$.

Lemma 3.2.4 (The union of two simulations is a simulation)**.** *Let $M = \langle Q, \Sigma, I, \to, F\rangle$ be a finite state machine. For any pair of forward or backward simulations $\preceq_1, \preceq_2 \subseteq Q \times Q$ of M we have that $\preceq \stackrel{\mathsf{def}}{=} \preceq_1 \cup \preceq_2$ is again a (respectively forward or backward) simulation preorder of M.*

Corollary 3.2.5 (FSM admit coarsest backward and forward simulations)**.** *Let $M = \langle Q, \Sigma, I, \to, F\rangle$ be a finite state machine. There exists a unique coarsest backward simulation preorder of M, and a unique coarsest forward simulation preorder of M.*

Remark 3.2.6. *It is not the case that the union of two partial orders remains a partial order. Therefore, contrary to simulation preorders, there does not exist unique backward and forward coarsest simulation partial orders.*

It is well-known that finding the coarsest simulation preorder of an FSM requires quadratic time in the number of states [HHK95]. Therefore, it may seem counterproductive to look for simulation preorders in order to solve the reachability or repeated reachability problems, that have a *linear* complexity in the number of states. However, there exist simulation preorders that can be characterized and exploited algorithmically with only minimal computation. More formally, the antichain algorithms exploit simulation

preorders such that similarity between two states can be computed in *logarithmic time* in the number of states of an FSM. In other words, antichain algorithms exploit simulation preorders which can be computed in polynomial time with respect to the number of states of the *source automaton* of the worst-case exponential translation. Contrary to other uses of simulations in automata theory which typically exploit maximally coarse simulations, antichain algorithms exploit simulation preorders which are not maximally coarse, but we do not compute them since they exist by construction.

Since all of the theory presented in this chapter is applicable to both backward and forward algorithms, many results and properties come in pairs. In order to avoid the duplication of proofs, we exploit the fact that for every FSM M, we can construct a *reverse* FSM M^{-1} such that pre and post are reversed, and such forward simulations become backward simulations and vice versa.

Definition 3.2.7 (Reverse FSM)**.** *For any* FSM $M = \langle Q, \Sigma, I, \rightarrow, F \rangle$*, the* reverse FSM *of* M*, denoted* M^1*, is such that* $M^{-1} = \langle Q, \Sigma, I, \rightarrow^{-1}, F \rangle$ *such that:* $\rightarrow^{-1} = \{\langle q', \sigma, q \rangle \mid \langle q, \sigma, q' \rangle \in \rightarrow\}$.

Theorem 3.2.8 (Duality of M^{-1})**.** *For every* FSM $M = \langle Q, \Sigma, I, \rightarrow, F \rangle$*, and for any set of states* $X \subseteq Q$ *we have that* $\mathsf{post}[M](X) = \mathsf{pre}[M^{-1}](X)$ *and* $\mathsf{pre}[M](X) = \mathsf{post}[M^{-1}](X)$*. Also, the preorder* $\preceq_{\mathsf{f}} \subseteq Q \times Q$ *is a forward simulation of* M *iff* $\preceq_{\mathsf{f}}^{-1}$ *is a backward simulation of* M^{-1}*. Similarly,* $\succeq_{\mathsf{b}} \subseteq Q \times Q$ *is a backward simulation of* M *iff* $\succeq_{\mathsf{b}}^{-1}$ *is a forward simulation of* M^{-1}*. Finally, it is trivial that for any preorder* $\preceq$*, every* $\preceq$*-downward-closed set in* M *is* $\preceq^{-1}$*-upward-closed in* M^{-1}*, and every* $\succeq$*-upward-closed set in* M *is* $\succeq^{-1}$*-downward-closed in* M^{-1}*.*

The consequence of Theorem 3.2.8 is as follows. Let $\mathcal{T}$ be a theorem of FSM (i.e., a property that is true for every FSM), and which involves pre, a forward simulation $\preceq_{\mathsf{f}}$ and $\preceq_{\mathsf{f}}$-downward-closed sets. Thanks to Theorem 3.2.8 we know that the dual $\mathcal{T}^{-1}$, obtained by rephrasing $\mathcal{T}$ such that post replaces pre and so on, is also a theorem of FSM. In the sequel we use Theorem 3.2.8 in order to avoid the duplication of proofs.

We now formalize the intuitions given in the introduction of this chapter. The crucial property of simulation preorders is as follows. For every

finite state machine and for every letter σ, the pre_σ state space operator preserves downward-closedness for all forward simulations, and the post_σ state space operator preserves upward-closedness for all backward simulations. We show that the reverse property also holds; for every finite state machine and for every letter σ, every preorder for which pre_σ preserves downward-closedness is a forward simulation, and every preorder for which post_σ preserves upward-closedness is a backward simulation.

Lemma 3.2.9. *Let $M = \langle Q, \Sigma, I, \to, F\rangle$ be a finite state machine. A preorder $\preceq_{\mathsf{f}} \subseteq Q \times Q$ is a forward simulation of M iff for every $\sigma \in \Sigma$ and for every $\preceq_{\mathsf{f}}$-downward-closed set of states $X \subseteq Q$, the set $\mathsf{pre}_\sigma(X)$ is $\preceq_{\mathsf{f}}$-downward-closed.*

Proof. $\boxed{\to}$ Let $\preceq_{\mathsf{f}} \subseteq Q \times Q$ be a forward simulation on M, let $X \subseteq Q$ be a $\preceq_{\mathsf{f}}$-downward-closed set, and let $\sigma \in \Sigma$. First, if $\mathsf{pre}_\sigma(X)$ is empty, then it is trivially downward-closed. Otherwise, let $q_1 \in \mathsf{pre}_\sigma(X)$ such that $\langle q_1, \sigma, q_3\rangle \in \to$ for some $q_3 \in X$, and let $q_2 \preceq_{\mathsf{f}} q_1$. By the forward simulation $\preceq_{\mathsf{f}}$ we know that there exists $q_4 \in X$ such that $\langle q_2, \sigma, q_4\rangle \in \to$ with $q_4 \preceq_{\mathsf{f}} q_3$. Finally, since $q_3 \in X$ and X is downward-closed, we know that $q_4 \in X$ and therefore that $q_2 \in \mathsf{pre}_\sigma(X)$, showing that $\mathsf{pre}_\sigma(X)$ is also downward-closed.

$\boxed{\leftarrow}$ Assume that $\mathsf{pre}_\sigma(X)$ is $\preceq_{\mathsf{f}}$-downward-closed for every $\preceq_{\mathsf{f}}$-downward-closed set $X \subseteq Q$ and for every $\sigma \in \Sigma$, for some preorder $\preceq_{\mathsf{f}} \subseteq Q \times Q$. Let $q_1, q_2, q_3 \in Q$ and $\sigma \in \Sigma$ such that $\langle q_1, \sigma, q_3\rangle \in \to$ and $q_2 \preceq_{\mathsf{f}} q_1$. Clearly, we have that $q_1 \in \mathsf{pre}_\sigma(\mathop{\downarrow}\{q_3\})$ and therefore that $q_2 \in \mathsf{pre}_\sigma(\mathop{\downarrow}\{q_3\})$. Since $q_2 \in \mathsf{pre}_\sigma(\mathop{\downarrow}\{q_3\})$, we know that there exists $q_4 \in \mathop{\downarrow}\{q_3\}$ such that $\langle q_2, \sigma, q_4\rangle \in \to$ and $q_4 \preceq_{\mathsf{f}} q_3$, showing that $\preceq_{\mathsf{f}}$ is a forward simulation on M. □

Theorem 3.2.10 (pre preserves forward simulation downward-closedness)**.** *Let $M = \langle Q, \Sigma, I, \to, F\rangle$ be a finite state machine. For any forward simulation $\preceq_{\mathsf{f}} \subseteq Q \times Q$ and for any $\preceq_{\mathsf{f}}$-downward-closed set $X \subseteq Q$, we have that $\mathsf{pre}(X)$ is $\preceq_{\mathsf{f}}$-downward-closed.*

Proof. Let $\preceq_{\mathsf{f}} \subseteq Q \times Q$ be a forward simulation on M, let $X \subseteq Q$ be a $\preceq_{\mathsf{f}}$-downward-closed set. We know that $\mathsf{pre}(X) \stackrel{\mathsf{def}}{=} \bigcup_{\sigma\in\Sigma} \mathsf{pre}_\sigma(X)$ (Definition 2.3.13), we know that $\cup$ preserves downward-closedness (Lemma 2.1.1),

and that $\mathsf{pre}_\sigma(X)$ is $\preceq_\mathsf{f}$-downward-closed for any $\sigma \in \Sigma$ (Lemma 3.2.9), which clearly shows that $\mathsf{pre}(X)$ is $\preceq_\mathsf{f}$-downward-closed. □

Remark 3.2.11. *It is important to note that if* $\mathsf{pre}(X)$ *is downward-closed for every downward-closed set* X*, it does not imply that the corresponding preorder is a forward simulation. Consider an* FSM $M = \langle Q, \Sigma, I, \to, F\rangle$ *with* $Q = \{1, 2\}$*,* $\Sigma = \{a, b\}$*, and* $\to = \{\langle 1, a, 1\rangle, \langle 1, b, 2\rangle\}$*, and let* $\preceq_\mathsf{f}$ *be the preorder such that* $1 \preceq_\mathsf{f} 1, 2 \preceq_\mathsf{f} 2$ *and* $2 \preceq_\mathsf{f} 1$*. Clearly,* M *is such that* $\mathsf{pre}(X)$ *is downward closed for every downward-closed set* $X \subseteq \{1, 2\}$*, but* $\preceq_\mathsf{f}$ *is not a forward simulation on* M *(2 does not forward-simulate 1).*

Lemma 3.2.12. *Let* $M = \langle Q, \Sigma, I, \to, F\rangle$ *be a finite state machine. A preorder* $\succeq_\mathsf{b} \subseteq Q \times Q$ *is a backward simulation of* M *iff for every* $\succeq_\mathsf{b}$*-upward-closed set of states* $X \subseteq Q$ *and for every* $\sigma \in \Sigma$*, the set* $\mathsf{post}_\sigma(X)$ *is* $\succeq_\mathsf{b}$*-upward-closed.*

Proof. Dual of Lemma 3.2.9 by Theorem 3.2.8. □

Theorem 3.2.13 (post preserves backward simulation upward-closedness)**.** *Let* $M = \langle Q, \Sigma, I, \to, F\rangle$ *be a finite state machine. For any backward simulation* $\succeq_\mathsf{b} \subseteq Q \times Q$ *and for any* $\succeq_\mathsf{b}$*-upward-closed set* $X \subseteq Q$*, we have that* $\mathsf{post}(X)$ *is* $\succeq_\mathsf{b}$*-upward-closed.*

Proof. Dual of Theorem 3.2.10 by Theorem 3.2.8. □

Remark 3.2.14. *As has been shown previously for* pre*, if* $\mathsf{post}(X)$ *is upward-closed for every upward-closed set* X*, it does not imply that the corresponding preorder is a backward simulation.*

3.3 Symbolic Computations with Antichains

We have seen that, for appropriate simulations, the successors of an upward-closed set of states of an FSM is again an upward-closed set, and the predecessors of a downward-closed set of states is again a downward-closed set. Antichain-based algorithms exploit this property of pre and post by relying on the *symbolic* computation of state space operators, that is by computing the antichain of successors or predecessors of an antichain *directly*, avoiding

to explicitly manipulate the underlying upward or downward-closed sets. Such symbolic version of pre and post are respectively denoted $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$.

Definition 3.3.1 (Operators $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$)**.** *Let M be the* FSM $\langle Q, \Sigma, I, \to, F\rangle$*, and let $\preceq_{\mathsf{f}} \subseteq Q \times Q$, $\succeq_{\mathsf{b}} \subseteq Q \times Q$ be forward and backward simulation preorders of M, respectively. The* symbolic pre *and* post*, respectively denoted $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$, are defined as:*

$$\widehat{\mathsf{pre}}(X) \stackrel{\mathsf{def}}{=} \lceil \mathsf{pre}(\downarrow X)\rceil \qquad (\preceq_{\mathsf{f}})$$

$$\widehat{\mathsf{post}}(X) \stackrel{\mathsf{def}}{=} \lfloor \mathsf{post}(\uparrow X)\rfloor \qquad (\succeq_{\mathsf{b}})$$

Note that Definition 3.3.1 above should be understood as a *specification* of the $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ operators. In practice, we want to avoid the evaluation of the upward- or downward-closure operation, and evaluate the operator directly on the antichain.

Lemma 3.3.2 (Correctness of $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$)**.** *Let $M = \langle Q, \Sigma, I, \to, F\rangle$ be a finite state machine. Let $\preceq_{\mathsf{f}} \subseteq Q \times Q$ be a forward simulation on M with $X \in \mathsf{Antichains}[Q, \preceq_{\mathsf{f}}]$, and let $\succeq_{\mathsf{b}} \subseteq Q \times Q$ be a backward simulation on M with $Y \in \mathsf{Antichains}[Q, \succeq_{\mathsf{b}}]$. The operators $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ are such that:*

$$\textstyle\downarrow \widehat{\mathsf{pre}}(X) = \mathsf{pre}(\downarrow X) \qquad (\preceq_{\mathsf{f}})$$

$$\textstyle\uparrow \widehat{\mathsf{post}}(Y) = \mathsf{post}(\uparrow Y) \qquad (\succeq_{\mathsf{b}})$$

Proof. By Definition 3.3.1 we know that $\downarrow \widehat{\mathsf{pre}}(X) = \downarrow \lceil \mathsf{pre}(\downarrow X)\rceil$. Since $\preceq_{\mathsf{f}}$ is a forward simulation, we know that $\mathsf{pre}(\downarrow X)$ is $\preceq_{\mathsf{f}}$-downward-closed (Theorem 3.2.10), and therefore that $\downarrow \lceil \mathsf{pre}(\downarrow X)\rceil = \mathsf{pre}(\downarrow X)$ (Theorem 2.1.2). The symmetrical property of $\widehat{\mathsf{post}}$ is obtained by Theorem 3.2.8. □

The practical efficiency of antichain algorithms relies heavily on efficient implementations of the symbolic operators $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$. Naturally, the particulars of the computation of $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ strongly depend on how the underlying FSM is defined, and what simulation is used. In the context of antichain algorithms, we do not make the hypothesis that the FSM over which $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ is given explicitly (in fact, we assume that it is given *implicitly*). Therefore, we cannot define general algorithms for computing $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$.

Short of defining general algorithms for $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$, we provide in the remainder of this section two useful general properties of $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ that can be used to derive efficient implementations. Firstly, the computation of $\widehat{\mathsf{pre}}(X)$ and $\widehat{\mathsf{post}}(X)$ can be *decomposed* into individual and independent computations for each element of the antichain X, and for each letter of the alphabet. This is formalized by the following proposition.

Proposition 3.3.3. *Let $M = \langle Q, \Sigma, I, \to, F\rangle$ be an* FSM. *Let $\preceq_{\mathsf{f}} \subseteq Q \times Q$ be a forward simulation on M with $X \in \mathsf{Antichains}[Q, \preceq_{\mathsf{f}}]$, and let $\succeq_{\mathsf{b}} \subseteq Q \times Q$ be a backward simulation on M with $Y \in \mathsf{Antichains}[Q, \succeq_{\mathsf{b}}]$. The operators $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ are such that:*

$$\widehat{\mathsf{pre}}(X) = \bigsqcup_{\sigma\in\Sigma} \bigsqcup_{x\in X} \lceil \mathsf{pre}_\sigma(\downarrow\{x\}) \rceil \qquad (\preceq_{\mathsf{f}})$$

$$\widehat{\mathsf{post}}(X) = \bigsqcup_{\sigma\in\Sigma} \bigsqcup_{x\in X} \lfloor \mathsf{post}_\sigma(\uparrow\{x\}) \rfloor \qquad (\succeq_{\mathsf{b}})$$

The proposition above states that, to compute the antichain of predecessors or successors of an antichain, one can compute the antichain of predecessors or successors individually for each element of the antichain and for each letter. This decomposition yields several advantages.

First, it simplifies the symbolic computation by breaking it down into smaller and more manageable parts. Second and more importantly, it allows to potentially speed up implementations by using caching techniques. Since the smaller computations are independent of each other, their result can be cached and reused later on. In our experiments, typical instances lead to cache-hit rates of more than 70%, bringing dramatic performance improvements.

Finally the decomposition of Proposition 3.3.3 can be further simplified when the underlying simulation preorder is *bidirectional* (i.e., both forward and backward). This is formalized by the following proposition.

Proposition 3.3.4. *For every finite state machine $M = \langle Q, \Sigma, I, \to F\rangle$, for every set of states $X \subseteq Q$, for every forward simulation $\preceq \subseteq Q \times Q$ on M such that $\succeq$ is a backward simulation, and for every $\sigma \in \Sigma$ we have that:*

$$\lceil \mathsf{pre}_\sigma(\downarrow X) \rceil = \lceil \mathsf{pre}_\sigma(X) \rceil \qquad (\preceq)$$

$$\lfloor \mathsf{post}_\sigma(\uparrow X) \rfloor = \lfloor \mathsf{post}_\sigma(X) \rfloor \qquad (\succeq)$$

Thanks to Proposition 3.3.3 and Proposition 3.3.4, whenever the underlying simulation preorder is bidirectional it is possible to reduce the symbolic predecessors or successors of an antichain to a series of steps which only involve the computation of the predecessors or successors of a *single state* for a *single letter*.

We now turn to the proofs of Proposition 3.3.3 and Proposition 3.3.4. The first step involves showing that both pre_σ and post_σ distribute over set union.

Lemma 3.3.5 (pre_σ and post_σ distribute over $\cup$)**.** *For any finite state machine $M = \langle Q, \Sigma, I, \to, F \rangle$, for any $\sigma \in \Sigma$ and $X, Y \subseteq Q$ we have that:*

$$\mathsf{pre}_\sigma(X \cup Y) = \mathsf{pre}_\sigma(X) \cup \mathsf{pre}_\sigma(Y)$$
$$\mathsf{post}_\sigma(X \cup Y) = \mathsf{post}_\sigma(X) \cup \mathsf{post}_\sigma(Y)$$

Proof. These results follow naturally from the definitions of pre_σ and post_σ (Definition 2.3.13):

$$\begin{aligned}
\mathsf{pre}_\sigma(X \cup Y) &= \{q_1 \in Q \mid \exists \langle q_1, \sigma, q_2 \rangle \in \to : q_2 \in (X \cup Y)\} \\
&= \{q_1 \in Q \mid \exists \langle q_1, \sigma, q_2 \rangle \in \to : q_2 \in X\} \\
&\quad \cup \{q_1 \in Q \mid \exists \langle q_1, \sigma, q_2 \rangle \in \to : q_2 \in Y\} \\
&= \mathsf{pre}_\sigma(X) \cup \mathsf{pre}_\sigma(Y)
\end{aligned}$$

$$\begin{aligned}
\mathsf{post}_\sigma(X \cup Y) &= \{q_2 \in Q \mid \exists \langle q_1, \sigma, q_2 \rangle \in \to : q_1 \in (X \cup Y)\} \\
&= \{q_2 \in Q \mid \exists \langle q_1, \sigma, q_2 \rangle \in \to : q_1 \in X\} \\
&\quad \cup \{q_2 \in Q \mid \exists \langle q_1, \sigma, q_2 \rangle \in \to : q_1 \in Y\} \\
&= \mathsf{post}_\sigma(X) \cup \mathsf{post}_\sigma(Y)
\end{aligned}$$

□

In order to prove Proposition 3.3.3, we must show that the downward and upward-closures of antichains can be decomposed into a union of *cones*, that is a union of the upward or downward-closure of singleton sets.

Lemma 3.3.6 (Decomposition of upward and downward-closure)**.** *Let* $\langle S, \preceq \rangle$ *be a partially ordered set. For any antichain* $X \in \mathsf{Antichains}[S, \preceq]$ *we have that:*

$$\downarrow X = \bigcup_{x \in X} \downarrow\{x\} \qquad\qquad \uparrow X = \bigcup_{x \in X} \uparrow\{x\}$$

Proof. Both properties are easily shown, as follows:

$$\downarrow X = \{s \in S \mid \exists\, x \in X \colon s \preceq x\} = \bigcup_{x \in X} \{s \in S \mid s \preceq x\} = \bigcup_{x \in X} \downarrow\{x\}$$

$$\uparrow X = \{s \in S \mid \exists\, x \in X \colon s \succeq x\} = \bigcup_{x \in X} \{s \in S \mid s \succeq x\} = \bigcup_{x \in X} \uparrow\{x\}$$

□

Proof of Proposition 3.3.3. The first property is shown as follows:

$$\begin{aligned}
\widehat{\mathsf{pre}}(X) &= \lceil \downarrow \widehat{\mathsf{pre}}(X) \rceil && (\widehat{\mathsf{pre}}(X) \text{ is an antichain}) \\
&= \lceil \mathsf{pre}(\downarrow X) \rceil && \text{(Lemma 3.3.2)} \\
&= \left\lceil \bigcup_{\sigma \in \Sigma} \mathsf{pre}_\sigma(\downarrow X) \right\rceil && \text{(Definition 2.3.13)} \\
&= \left\lceil \bigcup_{\sigma \in \Sigma} \mathsf{pre}_\sigma(\bigcup_{x \in X} \downarrow\{x\}) \right\rceil && \text{(Lemma 3.3.6)} \\
&= \left\lceil \bigcup_{\sigma \in \Sigma} \bigcup_{x \in X} \mathsf{pre}_\sigma(\downarrow\{x\}) \right\rceil && \text{(Lemma 3.3.5)} \\
&= \bigsqcup_{\sigma \in \Sigma} \bigsqcup_{x \in X} \lceil \mathsf{pre}_\sigma(\downarrow\{x\}) \rceil && \text{(Theorem 2.1.6)}
\end{aligned}$$

The second property is obtained by Theorem 3.2.8. □

We now provide formal proofs for the equalities of Proposition 3.3.4.

Lemma 3.3.7 ($\downarrow \mathsf{pre}_\sigma(X) = \mathsf{pre}_\sigma(\downarrow X)$ for Bidirectional Simulations)**.** *For every finite state machine* $M = \langle Q, \Sigma, I, \to F \rangle$*, for every set of states* $X \subseteq Q$*, for every forward simulation on* $\preceq \subseteq Q \times Q$ *on* M *such that* $\succeq$ *is a backward simulation, and for every* $\sigma \in \Sigma$ *we have that:*

$$\downarrow \mathsf{pre}_\sigma(X) = \mathsf{pre}_\sigma(\downarrow X) \qquad\qquad (\preceq)$$

Proof. $\boxed{\subseteq}$ Let $q_2 \in \downarrow \mathsf{pre}_\sigma(X)$, and let us show that $q_2 \in \mathsf{pre}_\sigma(\downarrow X)$. We know that there exist $q_3 \in X$ and $q_1 \in \mathsf{pre}_\sigma(X)$ such that $\langle q_1, \sigma, q_3 \rangle \in \rightarrow$ and $q_2 \preceq q_1$. Because $\preceq$ is a forward simulation, we know that there exists $q_4 \preceq q_3$ such that $\langle q_2, \sigma, q_4 \rangle \in \rightarrow$. Since $q_3 \in X$ and $q_4 \preceq q_3$ we know that $q_4 \in \uparrow X$ and therefore that $q_2 \in \mathsf{pre}_\sigma(\downarrow X)$.

$\boxed{\supseteq}$ Let $q_3 \in \mathsf{pre}_\sigma(\downarrow X)$, and let us show that $q_3 \in \downarrow \mathsf{pre}_\sigma(X)$. We know that there exist $q_1 \in \downarrow X$ and $q_2 \in X$ such that $q_2 \succeq q_1$ and $\langle q_3, \sigma, q_1 \rangle \in \rightarrow$. Because $\succeq$ is a backward simulation, we know that there exists $q_4 \succeq q_3$ such that $\langle q_4, \sigma, q_2 \rangle \in \rightarrow$. Since $q_2 \in X$ we have that $q_4 \in \mathsf{pre}_\sigma(X)$, and because $q_4 \succeq q_3$ we have that $q_3 \in \downarrow \mathsf{pre}_\sigma(X)$. □

Lemma 3.3.8 ($\uparrow \mathsf{post}_\sigma(X) = \mathsf{post}_\sigma(\uparrow X)$ for Bidirectional Simulations)**.** *For every finite state machine $M = \langle Q, \Sigma, I, \rightarrow F \rangle$, for every set of states $X \subseteq Q$, for every forward simulation on $\preceq \subseteq Q \times Q$ on M such that $\succeq$ is a backward simulation, and for every $\sigma \in \Sigma$ we have that:*

$$\uparrow \mathsf{post}_\sigma(X) = \mathsf{post}_\sigma(\uparrow X) \qquad (\preceq)$$

Proof. Dual of Lemma 3.3.7 by Theorem 3.2.8. □

Proof of Proposition 3.3.4. Both equalities are immediate from Lemma 3.3.7 and Lemma 3.3.8 and the fact that $\mathsf{post}_\sigma(\uparrow X)$ and $\mathsf{pre}_\sigma(\downarrow X)$ are respectively upward and downward-closed by Lemma 3.2.9 and Lemma 3.2.12. □

3.4 Fixed Points on the Lattice of Antichains

In this section, we show how to use backward and forward simulation preorders to compute fixed point expressions on the lattice of antichains. We show in this section that these antichain fixed points closely mimic their counterpart fixed points on the lattice of sets of states, and can therefore be used to decide the emptiness of automata.

Definition 3.4.1 (Shorthands for Maximal Antichains LFP)**.** *Some commonly used least fixed points of functions that are based on $\widehat{\mathsf{pre}}$ and the lattice*

of maximal antichains are abbreviated as follows:

$$\widehat{\mathsf{pre}}^*(X) \stackrel{\mathsf{def}}{=} \mathsf{LFP}(\lambda Y \cdot \widehat{\mathsf{pre}}(Y) \sqcup X)$$
$$\widehat{\mathsf{pre}}^+(X) \stackrel{\mathsf{def}}{=} \mathsf{LFP}(\lambda Y \cdot \widehat{\mathsf{pre}}(Y) \sqcup \widehat{\mathsf{pre}}(X))$$

Definition 3.4.2 (Shorthands for Minimal Antichains LFP)**.** *Some commonly used least fixed points of functions that are based on* $\widehat{\mathsf{post}}$ *and the lattice of minimal antichains are abbreviated as follows:*

$$\widehat{\mathsf{post}}^*(X) \stackrel{\mathsf{def}}{=} \mathsf{LFP}(\lambda Y \cdot \widehat{\mathsf{post}}(Y) \sqcup X)$$
$$\widehat{\mathsf{post}}^+(X) \stackrel{\mathsf{def}}{=} \mathsf{LFP}(\lambda Y \cdot \widehat{\mathsf{post}}(Y) \sqcup \widehat{\mathsf{post}}(X))$$

We begin by adapting the basic backward breadth-first search fixed point of Definition 2.3.15 to antichains. This new fixed point is evaluated on the lattice of maximal antichains of sets of states (see Theorem 2.1.6) and assumes the use of a forward simulation preorder that is compatible with the set of final states. Note that we can exploit Theorem 2.1.6 since $\langle 2^Q, \subseteq \rangle$ is a lattice, and therefore it is also a meet-semilattice.

Definition 3.4.3 (Backward BFS with Maximal Antichains)**.** *Let* $M = \langle Q, \Sigma, I, \to, F \rangle$ *be a finite state machine, and let* $\preceq_{\mathsf{f}} \subseteq Q \times Q$ *be a forward simulation on* M *that is compatible with* F*. The backward antichain fixed point of* M*, denoted* $\widehat{\mathsf{B}}^*(M)$*, is defined as follows:*

$$\widehat{\mathsf{B}}^0(M) = \lceil F \rceil \qquad (\preceq_{\mathsf{f}})$$
$$\widehat{\mathsf{B}}^i(M) = \widehat{\mathsf{B}}^{i-1}(M) \sqcup \widehat{\mathsf{pre}}(\widehat{\mathsf{B}}^{i-1}(M))$$
$$\widehat{\mathsf{B}}^*(M) = \mathsf{LFP}(\lambda X \cdot \widehat{\mathsf{pre}}(X) \sqcup \lceil F \rceil) = \widehat{\mathsf{pre}}^*(\lceil F \rceil) \qquad (\preceq_{\mathsf{f}})$$

We now show how this backward antichain fixed point is related to its classical counterpart, and how it can be used to decide NFA emptiness.

Lemma 3.4.4. *For every* FSM M*,* $i \in \mathbb{N}$*, we have that* $\downarrow \widehat{\mathsf{B}}^i(M) = \mathsf{B}^i(M)$*.*

Proof. We proceed by induction. Since $\preceq_{\mathsf{f}}$ is compatible with F we know that F is downward-closed. Hence $\downarrow \widehat{\mathsf{B}}^0(M) = \big\downarrow \lceil F \rceil = F = \mathsf{B}^0(M)$. Let $i \in \mathbb{N}_0$ and let us assume that for every $j \in \mathbb{N}$ such that $0 \leq j < i$ we have that $\downarrow \widehat{\mathsf{B}}^j(M) = \mathsf{B}^j(M)$. By Lemma 3.3.2 and the induction hypothesis we

know that $\widehat{\mathsf{pre}}(\widehat{\mathsf{B}}^{i-1}(M)) = \left\lceil \mathsf{pre}(\downarrow \widehat{\mathsf{B}}^{i-1}(M)) \right\rceil = \lceil \mathsf{pre}(\mathsf{B}^{i-1}(M)) \rceil$ and that $\widehat{\mathsf{B}}^{i-1}(M) = \lceil \mathsf{B}^{i-1}(M) \rceil$. From the definition of $\widehat{\mathsf{B}}^{i}(M)$ we have therefore that:

$$\begin{aligned}
\downarrow \widehat{\mathsf{B}}^{i}(M) &= \downarrow \left(\left\lceil \mathsf{B}^{i-1}(M) \right\rceil \sqcup \left\lceil \mathsf{pre}(\mathsf{B}^{i-1}(M)) \right\rceil \right) \\
&= \downarrow \left\lceil \mathsf{B}^{i-1}(M) \right\rceil \cup \downarrow \left\lceil \mathsf{pre}(\mathsf{B}^{i-1}(M)) \right\rceil \\
&= \mathsf{B}^{i-1}(M) \cup \mathsf{pre}(\mathsf{B}^{i-1}(M)) \\
&= \mathsf{B}^{i}(M)
\end{aligned}$$

□

Corollary 3.4.5 (Backward NFA Emptiness with Maximal Antichains)**.** *For every* NFA $A = \langle Q, \Sigma, I, \rightarrow, F \rangle$*, we have that* $\downarrow \widehat{\mathsf{B}}^{*} \cap I = \varnothing$ *iff* $\mathsf{B}^{*} \cap I = \varnothing$ *iff* $L(A) = \varnothing$*.*

Unsurprisingly, the classical forward breadth-first search fixed point can be similarly adapted using the lattice of minimal antichains of sets of states, thanks to Theorem 2.1.5. Again, we can exploit Theorem 2.1.5 since $\langle 2^{Q}, \subseteq \rangle$ is a lattice, and therefore it is also a join-semilattice. This new antichain fixed point is based on the $\widehat{\mathsf{post}}$ state space operator and assumes the usage of a backward simulation preorder that is compatible with the set of initial states.

Definition 3.4.6 (Forward BFS with Minimal Antichains)**.** *Let* $M = \langle Q, \Sigma, I, \rightarrow, F \rangle$ *be a finite state machine, and let* $\succeq_{\mathsf{b}} \subseteq Q \times Q$ *be a backward simulation on* M *that is compatible with* I*. The forward antichain fixed point of* M*, denoted* $\widehat{\mathsf{F}}^{*}(M)$*, is defined as follows:*

$$\begin{aligned}
&\widehat{\mathsf{F}}^{0}(M) = \lfloor I \rfloor && (\succeq_{\mathsf{b}}) \\
&\widehat{\mathsf{F}}^{i}(M) = \widehat{\mathsf{F}}^{i-1}(M) \sqcup \widehat{\mathsf{post}}(\widehat{\mathsf{F}}^{i-1}(M)) \\
&\widehat{\mathsf{F}}^{*}(M) = \mathsf{LFP}(\lambda X \cdot \widehat{\mathsf{post}}(X) \sqcup \lfloor I \rfloor) = \widehat{\mathsf{post}}^{*}(\lfloor I \rfloor) && (\succeq_{\mathsf{b}})
\end{aligned}$$

Again, we show that each iteration of this new antichain-based forward fixed point closely matches its classical counterpart, and can therefore be used to decide NFA emptiness.

Lemma 3.4.7. *For every* FSM M, $i \in \mathbb{N}$, *we have that* $\uparrow\widehat{\mathsf{F}}^i(M) = \mathsf{F}^i(M)$.

Proof. We proceed by induction. Since $\succeq_{\mathsf{b}}$ is compatible with I we know that I is upward-closed. Hence $\uparrow\widehat{\mathsf{F}}^0(M) = \uparrow\lfloor I \rfloor = I = \mathsf{F}^0(M)$. Let $i \in \mathbb{N}_0$ and let us assume that for every $j \in \mathbb{N}$ such that $0 \leq j < i$ we have that $\uparrow\widehat{\mathsf{F}}^j(M) = \mathsf{F}^j(M)$. By Lemma 3.3.2 and the induction hypothesis we know that $\widehat{\mathsf{post}}(\widehat{\mathsf{F}}^{i-1}(M)) = \left\lfloor \mathsf{post}(\uparrow\widehat{\mathsf{F}}^{i-1}(M)) \right\rfloor = \lfloor \mathsf{post}(\mathsf{F}^{i-1}(M)) \rfloor$ and that $\widehat{\mathsf{F}}^{i-1}(M) = \lfloor \mathsf{F}^{i-1}(M) \rfloor$. From the definition of $\widehat{\mathsf{F}}^i(M)$ we have therefore that:

$$\begin{aligned}
\uparrow\widehat{\mathsf{F}}^i(M) &= \uparrow\left(\left\lfloor \mathsf{F}^{i-1}(M) \right\rfloor \sqcup \left\lfloor \mathsf{post}(\mathsf{F}^{i-1}(M)) \right\rfloor\right) \\
&= \uparrow\left\lfloor \mathsf{F}^{i-1}(M) \right\rfloor \cup \uparrow\left\lfloor \mathsf{post}(\mathsf{F}^{i-1}(M)) \right\rfloor \\
&= \mathsf{F}^{i-1}(M) \cup \mathsf{post}(\mathsf{F}^{i-1}(M)) \\
&= \mathsf{F}^i(M)
\end{aligned}$$

□

Corollary 3.4.8 (Forward NFA Emptiness with Minimal Antichains)**.** *For every* NFA $A = \langle Q, \Sigma, I, \rightarrow, F \rangle$, *we have that* $\uparrow\widehat{\mathsf{F}}^* \cap F = \varnothing$ *iff* $\mathsf{F}^* \cap F = \varnothing$ *iff* $L(A) = \varnothing$.

We now proceed to adapt the doubly nested fixed points of Definition 2.3.17 and Definition 2.3.18 to antichains. We begin with the backward repeated breadth-first search fixed point. Like the simple backward fixed point of Definition 3.4.3, it is evaluated on the lattice of maximal antichains of sets of states, and assumes the usage of a forward simulation preorder that is compatible with the set of final states.

Definition 3.4.9 (Repeated Backward BFS with Maximal Antichains)**.** *Let* $M = \langle Q, \Sigma, I, \rightarrow, F \rangle$ *be an* FSM, *and let* $\preceq_{\mathsf{f}} \subseteq Q \times Q$ *be a forward simulation of* M *that is compatible with* F. *The backward nested antichain fixed point of* M, *denoted* $\widehat{\mathsf{BB}}^*(M)$, *is defined as follows:*

$$\widehat{\mathsf{BB}}^0(M) = \lceil F \rceil \qquad (\preceq_{\mathsf{f}})$$

$$\widehat{\mathsf{BB}}^i(M) = \lceil F \rceil \sqcap \widehat{\mathsf{pre}}^+(\widehat{\mathsf{BB}}^{i-1}(M)) \qquad (\preceq_{\mathsf{f}})$$

$$\widehat{\mathsf{BB}}^*(M) = \mathsf{GFP}(\lambda X \cdot \lceil F \rceil \sqcap \widehat{\mathsf{pre}}^+(X)) \qquad (\preceq_{\mathsf{f}})$$

Using the soundness proof of the simple backward antichain fixed point, showing how this doubly nested backward antichain fixed point is related to the classical fixed point poses no extra difficulty.

Lemma 3.4.10 (Soundness of $\widehat{\mathsf{BB}}^i$)**.** *For every* FSM M*, for every* $i \in \mathbb{N}$*, we have that* $\downarrow\widehat{\mathsf{BB}}^i(M) = \mathsf{BB}^i(M)$*.*

Proof. We proceed by induction. Since $\preceq_{\mathsf{f}}$ is compatible with F we know that F is downward-closed. Hence $\downarrow\widehat{\mathsf{BB}}^0(M) = \left\downarrow\lceil F\rceil\right. = F = \mathsf{BB}^0(M)$. Let $i \in \mathbb{N}_0$ and let us assume that for every $j \in \mathbb{N}$ such that $0 \leq j < i$ we have that $\downarrow\widehat{\mathsf{BB}}^j(M) = \mathsf{BB}^j(M)$. It follows from the proof of Lemma 3.4.4 that for any maximal antichain X we have that $\widehat{\mathsf{pre}}^+(X) = \lceil \mathsf{pre}^+(\downarrow X)\rceil$. Therefore we have that:

$$\begin{aligned}
\downarrow\widehat{\mathsf{BB}}^i &= \downarrow\Big(\lceil F\rceil \sqcap \Big\lceil \mathsf{pre}^+(\downarrow\widehat{\mathsf{BB}}^{i-1}(M))\Big\rceil\Big) \\
&= \downarrow\lceil F\rceil \cap \downarrow\Big\lceil \mathsf{pre}^+(\mathsf{BB}^{i-1}(M))\Big\rceil \\
&= F \cap \mathsf{pre}^+(\mathsf{BB}^{i-1}(M)) \\
&= \mathsf{BB}^i(M)
\end{aligned}$$

□

Corollary 3.4.11 (Backward NBA Emptiness with Maximal Antichains)**.** *For every* NBA $A = \langle Q, \Sigma, I, \to, F\rangle$*, we have that* $\widehat{\mathsf{pre}}^*(\widehat{\mathsf{BB}}^*(M)) \cap I = \varnothing$ *iff* $\mathsf{pre}^*(\mathsf{BB}^*) \cap I = \varnothing$ *iff* $L(A) = \varnothing$*.*

Defining an antichain version of the doubly nested forward breadth-first search fixed point is almost as straightforward as its backward counterpart. The only difference lies in the fact that both the initial and the final states appear in the fixed point, so they both need to be compatible with the simulation preorder (i.e., be upward-closed).

Definition 3.4.12 (Repeated Forward BFS with Minimal Antichains)**.** *Let* $M = \langle Q, \Sigma, I, \to, F\rangle$ *be an* FSM*, and let* $\succeq_{\mathsf{b}} \subseteq Q \times Q$ *be a backward simulation of* M *that is compatible with* I *and* F*. The forward nested antichain*

fixed point of M, denoted $\widehat{\mathsf{FF}}^*(M)$, is defined as follows:

$$\widehat{\mathsf{FF}}^0(M) = \widehat{\mathsf{post}}^*(\lfloor I \rfloor) \sqcap \lfloor F \rfloor \qquad (\succeq_\mathsf{b})$$
$$\widehat{\mathsf{FF}}^i(M) = \widehat{\mathsf{post}}^+(\widehat{\mathsf{FF}}^{i-1}(M)) \sqcap \lfloor F \rfloor \qquad (\succeq_\mathsf{b})$$
$$\widehat{\mathsf{FF}}^*(M) = \mathsf{GFP}(\lambda X \cdot \widehat{\mathsf{post}}^+(X) \sqcap \lfloor F \rfloor) \qquad (\succeq_\mathsf{b})$$

Showing the soundness of this forward nested fixed point is straightforward, using the proof of its simply nested counterpart.

Lemma 3.4.13 (Soundness of $\widehat{\mathsf{FF}}^i$)**.** *For every* FSM *M, for every $i \in \mathbb{N}$, we have that $\uparrow\widehat{\mathsf{FF}}^i(M) = \mathsf{FF}^i(M)$.*

Proof. We proceed by induction. We know that I and F are both upward-closed with respect to $\succeq_\mathsf{b}$. It follows from the proof of Lemma 3.4.7 that for any minimal antichain X we have that $\widehat{\mathsf{post}}^*(X) = \lfloor \mathsf{post}^*(\uparrow X) \rfloor$. Therefore for the base case we have that:

$$\begin{aligned}\uparrow\widehat{\mathsf{FF}}^0(M) &= \uparrow\left(\left\lfloor \mathsf{post}^*(\uparrow\lfloor I \rfloor)\right\rfloor \sqcap \lfloor F \rfloor\right)\\ &= \uparrow\lfloor \mathsf{post}^*(I)\rfloor \cap \uparrow\lfloor F \rfloor\\ &= \mathsf{post}^*(I) \cap F\\ &= \mathsf{FF}^0(M)\end{aligned}$$

For the inductive case, let $i \in \mathbb{N}_0$ and let us assume that for every $j \in \mathbb{N}$ such that $0 \leq j < i$ we have that $\uparrow\widehat{\mathsf{FF}}^j(M) = \mathsf{FF}^j(M)$. Again, it follows from the proof of Lemma 3.4.7 that for any minimal antichain X we have that $\widehat{\mathsf{post}}^+(X) = \lfloor \mathsf{post}^+(\uparrow X) \rfloor$. By the induction hypothesis we have that:

$$\begin{aligned}\uparrow\widehat{\mathsf{FF}}^i(M) &= \uparrow\left(\left\lfloor \mathsf{post}^+(\uparrow\widehat{\mathsf{FF}}^{i-1}(M))\right\rfloor \sqcap \lfloor F \rfloor\right)\\ &= \uparrow\lfloor \mathsf{post}^+(\mathsf{FF}^{i-1}(M))\rfloor \cap \uparrow\lfloor F \rfloor\\ &= \mathsf{post}^+(\mathsf{FF}^{i-1}(M)) \cap F\\ &= \mathsf{FF}^i(M)\end{aligned}$$

□

Corollary 3.4.14 (Forward NBA Emptiness with Minimal Antichains)**.** *For every* NBA *$A = \langle Q, \Sigma, I, \to, F\rangle$, we have that $\widehat{\mathsf{FF}}^*(M) = \varnothing$ iff $\mathsf{FF}^* = \varnothing$ iff $L(A) = \varnothing$.*

3.5 Antichains and Alternating Automata

In this section, we apply the antichains framework to the problem of alternating automata emptiness. In the previous sections, we have presented a generic framework that exploits simulation preorders to evaluate fixed point expressions on the lattice of antichains of sets of states. We have stated that the usefulness of this approach relies on the fact that the simulation preorder does not need to be computed explicitly, but rather exists "by construction".

We now instantiate the antichains framework to the context of alternating automata and the corresponding translations to non-deterministic automata. We identify appropriate simulation preorders that exist by construction in the subset construction and the Miyano Hayashi construction, and we show how the symbolic $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ state space operators can be efficiently computed.

3.5.1 Antichains for Alternating Finite Automata

It is easy to show that the subset ordering between NFA states in the subset construction for AFA is a simulation preorder, both forward and backward. As shown in the proof, it is a natural consequence of the fact that AFA transitions are described by positive Boolean formulas, whose set of valuations is always $\subseteq$-upward-closed (see Lemma 2.2.3).

Lemma 3.5.1 ($\supseteq$ / $\subseteq$ is a backward / forward simulation on SC)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* AFA *and let* $\mathsf{SC}(A) = A' = \langle Q', \Sigma, I, \to, F'\rangle$ *be the subset construction* NFA *of A. We have that $\subseteq$ is a forward simulation for A', and $\supseteq$ is a backward simulation for A'.*

Proof. Let us show that $\subseteq$ is a forward simulation. Let $c_1, c_2, c_3 \in Q'$ with $c_2 \subseteq c_1$ and $\langle c_1, \sigma, c_3\rangle \in \to$ for some $\sigma \in \Sigma$. Recall from Definition 2.3.8 that $\to = \left\{\langle c_1, \sigma, c_2\rangle \;\middle|\; c_2 \models \bigwedge_{q\in c_1} \delta(q,\sigma)\right\}$. Therefore, we know that $c_3 \models \bigwedge_{q\in c_1} \delta(q,\sigma)$ and since $c_2 \subseteq c_1$ we also have that $c_3 \models \bigwedge_{q\in c_2} \delta(q,\sigma)$, and thus that $\langle c_2, \sigma, c_3\rangle \in \to$. To show that $\supseteq$ is a backward simulation, let $c_1, c_2, c_3 \in Q'$ with $c_2 \supseteq c_1$ and $\langle c_3, \sigma, c_1\rangle \in \to$ for some $\sigma \in \Sigma$. Again, this implies that $c_1 \models \bigwedge_{q\in c_3} \delta(q,\sigma)$ and since $c_2 \supseteq c_1$ we also have that

$c_2 \models \bigwedge_{q \in c_3} \delta(q, \sigma)$ since $\mathsf{Sat}(\bigwedge_{q \in c_3} \delta(q, \sigma))$ is upward-closed by Lemma 2.2.3, and therefore $\langle c_3, \sigma, c_2 \rangle \in \rightarrow$. □

Now that we have an appropriate bidirectional simulation preorder on SC, we proceed with the precise definition of algorithms for computing $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$. We use the decomposition properties of Proposition 3.3.3 and Proposition 3.3.4, so it is sufficient to provide algorithms to compute the functions $\lceil \mathsf{pre}_\sigma(\cdot) \rceil$ and $\lfloor \mathsf{post}_\sigma(\cdot) \rfloor$ for singletons. It important to note that in terms of complexity, these two algorithms are not symmetrical. The $\widehat{\mathsf{pre}}$ operator is computable in quadratic time with respect to the AFA, while the $\widehat{\mathsf{post}}$ algorithm may construct an antichain of exponential length.

Lemma 3.5.2 (Computation of $\lceil \mathsf{pre}_\sigma(\cdot) \rceil$ on SC)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F \rangle$ be an AFA and let $\mathsf{SC}(A) = A' = \langle Q', \Sigma, I, \rightarrow, F' \rangle$ be the subset construction NFA of A. For every $q' \in Q'$, we have that:*

$$\lceil \mathsf{pre}_\sigma(\{q'\}) \rceil = \{\{q \in Q \mid q' \models \delta(q, \sigma)\}\} \qquad (\subseteq)$$

This operation can be performed in time $\mathcal{O}(|Q||\delta|)$, where $|\delta|$ is the number of Boolean connectives of the largest formula in δ.

Proof. We proceed as follows:

$$\begin{aligned} q'_1 \in \mathsf{pre}_\sigma(\{q'_2\}) \Leftrightarrow \langle q'_1, \sigma, q'_2 \rangle \in \rightarrow \;&\Leftrightarrow\; q'_2 \models \bigwedge_{q \in q'_1} \delta(q, \sigma) \\ &\Leftrightarrow \forall\, q \in q'_1 : q'_2 \models \delta(q, \sigma) \\ &\Leftrightarrow q'_1 \subseteq \{q \in Q \mid q'_2 \models \delta(q, \sigma)\} \end{aligned}$$

To conclude, we have $q'_1 \in \lceil \mathsf{pre}_\sigma(\{q'_2\}) \rceil \Leftrightarrow q'_1 = \{q \in Q \mid q'_2 \models \delta(q, \sigma)\}$. Finally, deciding whether $q'_2 \models \delta(q, \sigma)$ can be done in linear time with respect to the number of Boolean connectives in $\delta(q, \sigma)$, so the computation of $\{q \in Q \mid q'_2 \models \delta(q, \sigma)\}$ can be done in time $\mathcal{O}(|Q||\delta|)$. □

Lemma 3.5.3 (Computation of $\lfloor \mathsf{post}_\sigma(\cdot) \rfloor$ on SC)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F \rangle$ be an AFA and let $\mathsf{SC}(A) = A' = \langle Q', \Sigma, I, \rightarrow, F' \rangle$ be the subset construction NFA of A. For every $q' \in Q'$, we have that:*

$$\lfloor \mathsf{post}_\sigma(\{q'\}) \rfloor = \left\lfloor [\![\bigwedge_{q \in q'} \delta(q, \sigma)]\!] \right\rfloor \qquad (\supseteq)$$

The resulting antichain may be of exponential size in the worst case.

Proof. We proceed as follows:

$$q_2' \in \mathsf{post}_\sigma(\{q_1'\}) \Leftrightarrow \langle q_1', \sigma, q_2'\rangle \in \rightarrow \;\Leftrightarrow\; q_2' \models \bigwedge_{q \in q_1'} \delta(q,\sigma)$$
$$\Leftrightarrow q_2' \in [\![\bigwedge_{q \in q_1'} \delta(q,\sigma)]\!]$$

We therefore have that $q_2' \in \lfloor \mathsf{post}_\sigma(\{q_1'\}) \rfloor \Leftrightarrow q_2' \in \left\lfloor [\![\bigwedge_{q \in q_1'} \delta(q,\sigma)]\!] \right\rfloor$.
To prove the space lower bound on the computation of $\left\lfloor [\![\bigwedge_{q \in q_1'} \delta(q,\sigma)]\!] \right\rfloor$, consider an AFA $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ with $Q = \{q_0, \ldots, q_{2k}\}$ for some $k \in \mathbb{N}_0$ and $\delta(q_0,\sigma) = \bigwedge_{i=1}^{k}(q_{2i} \vee q_{2i-1})$ for some $\sigma \in \Sigma$. It is easy to see that $\mathsf{SC}(\mathsf{A})$ is such that $|\lfloor \mathsf{post}_\sigma(\{\{q_0\}\})\rfloor| = 2^k$. □

Contrary to the case of $\widehat{\mathsf{pre}}$, for which Lemma 3.5.2 gives an immediate description of the resulting antichain, it is not immediately obvious how to compute $\widehat{\mathsf{post}}$ from Lemma 3.5.3. Practical implementations can achieve this in multiple ways, the most immediate one being to precompute an antichain $\hat{\delta}(q,\sigma) \stackrel{\mathsf{def}}{=} \lfloor [\![\delta(q,\sigma)]\!] \rfloor$ for each state q and each letter σ. These precomputed antichains enjoy a natural inductive definition, as shown in the following Lemma.

Lemma 3.5.4 (Recursive computation of $\hat{\delta}(q,\sigma)$)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* AFA *and let $\hat{\delta}(q,\sigma) \stackrel{\mathsf{def}}{=} \lfloor [\![\delta(q,\sigma)]\!] \rfloor$. We have that $\hat{\delta}(q,\sigma) = \mathsf{antichain}(\delta(s,\sigma))$ such that:*

$$\begin{aligned} \mathsf{antichain}(q) &= \{\{q\}\} \\ \mathsf{antichain}(\phi_1 \vee \phi_2) &= \mathsf{antichain}(\phi_1) \sqcup \mathsf{antichain}(\phi_2) \\ \mathsf{antichain}(\phi_1 \wedge \phi_2) &= \mathsf{antichain}(\phi_1) \sqcap \mathsf{antichain}(\phi_2) \end{aligned}$$

The precomputation of each antichain $\hat{\delta}$ allows for the straightforward computation of $\lfloor \mathsf{post}_\sigma(\cdot)\rfloor$, as shown by this next Lemma.

Lemma 3.5.5. *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* AFA*, let $\hat{\delta}(q,\sigma) \stackrel{\mathsf{def}}{=} \lfloor [\![\delta(q,\sigma)]\!] \rfloor$, and let $\mathsf{SC}(A) = A' = \langle Q', \Sigma, I, \rightarrow, F'\rangle$ be the subset construction* NFA *of A. For every $q' \in Q'$, we have that:*

$$\lfloor \mathsf{post}_\sigma(\{q'\})\rfloor = \bigsqcap \left\{ \hat{\delta}(q,\sigma) \;\middle|\; q \in q' \right\}$$

3.5.2 Antichains for Alternating Büchi Automata

Recall from Definition 2.3.11 that for each alternating Büchi automaton, the Miyano-Hayashi construction (MH, for short) creates a language-equivalent non-deterministic Büchi automaton whose state space is *the set of pairs of sets of states* of the ABA, with the additional constraint that the second set be include in the first. By convention, we denote such NBA states as (s, o) pairs, with s and o being sets of states of the ABA, and $o \subseteq s$.

To obtain a simulation preorder on MH, we generalize the subset ordering to pairs of sets. However, since the set of final MH states are the pairs whose o component is empty, we strengthen the double inclusion ordering with the added requirement that only final or non-final pairs be comparable. This ensures that our simulation is compatible with the set of final states.

Definition 3.5.6 ($\preceq_{\mathsf{MH}}$ on MH of ABA)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* ABA *and let $\mathsf{MH}(A) = \langle Q', \Sigma, I, \to, F'\rangle$ be the Miyano-Hayashi construction* NBA *of A. We define the following partial order $\preceq_{\mathsf{MH}} \subseteq Q' \times Q'$:*

$$\langle s_1, o_1\rangle \preceq_{\mathsf{MH}} \langle s_2, o_2\rangle \text{ iff } \begin{cases} s_1 \subseteq s_2 \wedge \\ o_1 \subseteq o_2 \wedge \\ o_1 = \varnothing \text{ iff } o_2 = \varnothing \end{cases}$$

Lemma 3.5.7 ($\preceq_{\mathsf{MH}}$ is a forward simulation on MH)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* ABA *and let $\mathsf{MH}(A) = A' = \langle Q', \Sigma, I, \to, F'\rangle$ be the Miyano-Hayashi construction* NBA *of A. We have that $\preceq_{\mathsf{MH}}$ is a forward simulation for A'.*

Proof. Let us show that $\preceq_{\mathsf{MH}}$ is a forward simulation for A'. Recall from Definition 2.3.11 that:

$$\begin{aligned} x \overset{\sigma}{\rightsquigarrow} y &\overset{\mathsf{def}}{=} y \models \bigwedge_{q \in x} \delta(q, \sigma) \\ \to &= \left\{ \langle (s_1, \varnothing), \sigma, (s_2, s_2 \setminus F)\rangle \;\middle|\; s_1 \overset{\sigma}{\rightsquigarrow} s_2 \right\} \cup \\ &\quad \left\{ \langle (s_1, o_1 \neq \varnothing), \sigma, (s_2, o_2 \setminus F)\rangle \;\middle|\; s_1 \overset{\sigma}{\rightsquigarrow} s_2,\ o_1 \overset{\sigma}{\rightsquigarrow} o_2,\ o_2 \subseteq s_2 \right\} \end{aligned}$$

Let $(s_1, o_1), (s_2, o_2), (s_3, o_3) \in Q'$ with $(s_2, o_2) \preceq_{\mathsf{MH}} (s_1, o_1)$ and such that $\langle (s_1, o_1), \sigma, (s_3, o_3)\rangle \in \to$ for some $\sigma \in \Sigma$. Therefore, we know that $s_1 \overset{\sigma}{\rightsquigarrow} s_3$ and that either $o_1 = \varnothing$ and $o_3 = s_3 \setminus F$, or $o_1 \neq \varnothing$ and $o_1 \overset{\sigma}{\rightsquigarrow} o_3'$ for some

$o'_3 \subseteq Q$ with $o_3 = o'_3 \setminus F$. We deal with each case separately. If $o_1 = \varnothing$, then $o_2 = \varnothing$ and since $s_2 \subseteq s_1$ then we know that $s_2 \overset{\sigma}{\rightsquigarrow} s_3$ and therefore that $\langle (s_2, o_2 = \varnothing), \sigma, (s_3, o_3 = s_3 \setminus F) \rangle \in \rightarrow$. Otherwise, $o_1 \neq \varnothing$ and $o_2 \neq \varnothing$ and since both $s_2 \subseteq s_1$ and $o_2 \subseteq o_1$ we know that $s_2 \overset{\sigma}{\rightsquigarrow} s_3$ and $o_2 \overset{\sigma}{\rightsquigarrow} o'_3$ and therefore that $\langle (s_2, o_2), \sigma, (s_3, o_3 = o'_3 \setminus F) \rangle \in \rightarrow$. □

Let $\mathsf{MH}(A) = \langle Q, \Sigma, I, \rightarrow, F \rangle$ for some alternating automaton A. It is important to note that the partial order $\preceq_{\mathsf{MH}}^{-1}$ is *not a backward simulation* on $\mathsf{MH}(A)$ in general, because states of the form $\langle s, o \rangle \in Q$ such that $o \cap F \neq \varnothing$ never have predecessors, but it may be that $\langle s, o \setminus F \rangle \preceq_{\mathsf{MH}} \langle s, o \rangle$ and that $\langle s, o \setminus F \rangle$ has a predecessor.

In order to make $\succeq_{\mathsf{MH}} \overset{\mathsf{def}}{=} \preceq_{\mathsf{MH}}^{-1}$ a backward simulation on MH, we extend the Miyano-Hayashi construction by artificially adding transitions such that the set of successors in the NBA is always $\succeq_{\mathsf{MH}}$-upward-closed. This operation preserves the language of the MH NBA.

Definition 3.5.8 (Extended MH Construction)**.** *Let A be an* ABA *and let* $\mathsf{MH}(A) = \langle Q, \Sigma, I, \rightarrow, F \rangle$ *be the* MH *construction of A. The* extended Miyano Hayashi construction *of A is the* NBA $\mathsf{MH}^{\mathsf{ext}}(A) = \langle Q, \Sigma, I, \twoheadrightarrow, F \rangle$ *such that:*

$$\twoheadrightarrow = \rightarrow \cup \{ \langle (s, o), \sigma, (s'', o'') \rangle \mid \langle (s, o), \sigma, (s', o') \rangle \in \rightarrow,\ (s'', o'') \succeq_{\mathsf{MH}} (s', o') \}$$

Naturally, in order to use the $\mathsf{MH}^{\mathsf{ext}}$ definition, we must show that it defines the same infinite-word language than MH.

Lemma 3.5.9. *For every* ABA *A, we have that* $L(\mathsf{MH}(A)) = L(\mathsf{MH}^{\mathsf{ext}}(A))$.

Proof. Since $\mathsf{MH}^{\mathsf{ext}}(A)$ is an NBA with more transitions than $\mathsf{MH}(A)$, we naturally have that $\mathsf{MH}^{\mathsf{ext}}(A) \supseteq \mathsf{MH}(A)$. We now show the reverse inclusion. Let $w \in L(\mathsf{MH}^{\mathsf{ext}}(A))$ such that ρ is an infinite accepting run of $\mathsf{MH}^{\mathsf{ext}}(A)$ on w. From the definition of $\mathsf{MH}^{\mathsf{ext}}$ and since $\preceq_{\mathsf{MH}}$ is a forward simulation, we can construct an infinite run ρ' of $\mathsf{MH}(A)$ on w such that $\rho'[0] = \rho[0]$ and for every $i \in \mathbb{N}_0 : \rho'[i] \preceq_{\mathsf{MH}} \rho[i]$. Also, whenever $\rho[i]$ is accepting we have that $\rho'[i]$ is accepting, meaning that ρ' is an accepting run on w, and therefore $w \in L(\mathsf{MH}(A))$. □

Thanks to our extended definition of the Miyano-Hayashi construction, we can readily show that $\preceq_{\mathsf{MH}}$ and $\succeq_{\mathsf{MH}}$ are forward and backward simulations on $\mathsf{MH}^{\mathsf{ext}}$, respectively.

Lemma 3.5.10 ($\succeq_{\mathsf{MH}}$ / $\preceq_{\mathsf{MH}}$ is a backward / forward simulation on $\mathsf{MH}^{\mathsf{ext}}$)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* ABA *and let $\mathsf{MH}^{\mathsf{ext}}(A) = A' = \langle Q', \Sigma, I, \to, F'\rangle$ be the extended Miyano-Hayashi construction* NBA *of A. We have that $\preceq_{\mathsf{MH}}$ is a forward simulation for A' and that $\succeq_{\mathsf{MH}}$ is a backward simulation for A'.*

Proof. We begin by showing that $\preceq_{\mathsf{MH}}$ is a forward simulation on A'. Let $\langle (s_1, o_1), \sigma, (s_3, o_3)\rangle \in \to$ and let $(s_2, o_2) \in Q'$ such that $(s_2, o_2) \preceq_{\mathsf{MH}} (s_1, o_1)$. From the definition of $\mathsf{MH}^{\mathsf{ext}}$ we know that there exists $(s'_3, o'_3) \in Q'$ with $(s'_3, o'_3) \preceq_{\mathsf{MH}} (s_3, o_3)$ such that $\langle (s_1, o_1), \sigma, (s'_3, o'_3)\rangle$ is a transition of $\mathsf{MH}(A)$. Since $\preceq_{\mathsf{MH}}$ is a forward simulation on $\mathsf{MH}(A)$, we know that there exists $(s_4, o_4) \preceq_{\mathsf{MH}} (s'_3, o'_3)$ such that $\langle (s_2, o_2), \sigma, (s_4, o_4)\rangle$ is a transition of $\mathsf{MH}(A)$, and thus also of $\mathsf{MH}^{\mathsf{ext}}$, since $(s_4, o_4) \preceq_{\mathsf{MH}} (s'_3, o'_3) \preceq_{\mathsf{MH}} (s_3, o_3)$. Finally, the fact that $\succeq_{\mathsf{MH}}$ is a backward simulation on A' is a direct consequence of Lemma 3.2.12, since $\mathsf{post}_\sigma(X)$ is clearly $\succeq_{\mathsf{MH}}$-upward-closed for all X. □

Now that we have defined an appropriate bidirectional simulation preorder on the Miyano-Hayashi state space, we can provide practical algorithms to compute the symbolic $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ state space operators. Similarly than for AFA, we make use of Proposition 3.3.3 and Proposition 3.3.4 to decompose the symbolic operators. For technical reasons, we do not make use of Proposition 3.3.4 in the case of $\widehat{\mathsf{pre}}$, and we therefore provide algorithms for computing $\lceil \mathsf{pre}_\sigma(\downarrow\cdot)\rceil$ on MH (it makes the proof simpler, and is not needed to obtain an efficient algorithm) and $\lfloor \mathsf{post}_\sigma(\cdot)\rfloor$ on $\mathsf{MH}^{\mathsf{ext}}$.

Lemma 3.5.11 (Computation of $\lceil \mathsf{pre}_\sigma(\downarrow\cdot)\rceil$ on MH)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* ABA *and let $\mathsf{MH}(A) = A' = \langle Q', \Sigma, I, \to, F'\rangle$ be the Miyano-Hayashi construction* NBA *of A. Recall from Lemma 3.5.7 that $\preceq_{\mathsf{MH}}$ is a forward*

simulation on A'. For every $(s', o') \in Q'$ such that $o' \subseteq (s' \setminus F)$, we have:

$$\left\lceil \mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\}) \right\rceil = \begin{cases} \{\langle o, \varnothing\rangle, \langle s, o\rangle\} & \text{if } o \neq \varnothing \\ \{\langle \varnothing, \varnothing\rangle\} & \text{otherwise} \end{cases}$$
$$\text{with } s = \{q \in Q \mid s' \models \delta(q, \sigma)\}$$
$$o = \{q \in Q \mid o' \cup (s' \cap F) \models \delta(q, \sigma)\}$$

Proof. Recall from Definition 2.3.11 that $x \overset{\sigma}{\rightsquigarrow} y \overset{\mathsf{def}}{=} y \models \bigwedge_{q \in x} \delta(q, \sigma)$. Recall also from Lemma 3.5.2 that for any $x \subseteq Q$, whenever $x \overset{\sigma}{\rightsquigarrow} s'$ we have that $x \subseteq s$ and whenever $x \overset{\sigma}{\rightsquigarrow} o' \cup (s' \cap F)$ we have that $x \subseteq o$.

(1) To begin, we show that for any $\langle s_1, o_1\rangle \in \mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\})$ we have either $\langle o, \varnothing\rangle \succeq_{\mathsf{MH}} \langle s_1, o_1\rangle$ or $\langle s, o\rangle \succeq_{\mathsf{MH}} \langle s_1, o_1\rangle$. Let $\langle s_1, o_1\rangle \overset{\sigma}{\to} \langle s_2, o_2\rangle$ with $\langle s_2, o_2\rangle \preceq_{\mathsf{MH}} \langle s', o'\rangle$.

case 1 : $o_1 = \varnothing$ In this first case, we must show that $\langle o, \varnothing\rangle \succeq_{\mathsf{MH}} \langle s_1, o_1\rangle$. If $o_1 = \varnothing$ then $o_2 = s_2 \setminus F$ and $s_1 \overset{\sigma}{\rightsquigarrow} s_2$. We have that $s_2 \setminus F = o_2 \subseteq o'$, and also that $s_2 \cap F \subseteq s' \cap F$. Therefore $(s_2 \cap F) \cup (s_2 \setminus F) = s_2 \subseteq o' \cup (s' \cap F)$. Since $s_1 \overset{\sigma}{\rightsquigarrow} s_2$ and $s_2 \subseteq o' \cup (s' \cap F)$, we have that $s_1 \overset{\sigma}{\rightsquigarrow} o' \cup (s' \cap F)$. Since whenever $x \overset{\sigma}{\rightsquigarrow} o' \cup (s' \cap F)$ we have that $x \subseteq o$, for any $x \subseteq Q$, we know that $s_1 \subseteq o$.

case 2 : $o_1 \neq \varnothing$ In this second case, we must show that $\langle s, o\rangle \succeq_{\mathsf{MH}} \langle s_1, o_1\rangle$. If $o_1 \neq \varnothing$ then $s_1 \overset{\sigma}{\rightsquigarrow} s_2$ and $o_1 \overset{\sigma}{\rightsquigarrow} o_3$, for some $o_3 \subseteq s_2$ such that $o_2 = o_3 \setminus F$. Since $s_1 \overset{\sigma}{\rightsquigarrow} s_2$ and $s_2 \subseteq s'$ we know that $s_1 \overset{\sigma}{\rightsquigarrow} s'$ and therefore that $s_1 \subseteq s$. Also, since $o_3 \subseteq s_2$ and $o_2 = o_3 \setminus F$ we have that $o_3 \subseteq (o_3 \setminus F) \cup (o_3 \cap F) \subseteq o_2 \cup (s_2 \cap F) \subseteq o' \cup (s' \cap F)$. Therefore, since $o_1 \overset{\sigma}{\rightsquigarrow} o_3$ and $o_3 \subseteq o' \cup (s' \cap F)$ we know that $o_1 \overset{\sigma}{\rightsquigarrow} o' \cup (s' \cap F)$, which implies that $o_1 \subseteq o$.

(2) We now show that $\langle \varnothing, \varnothing\rangle \in \mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\})$, and that if $o \neq \varnothing$ then $\{\langle o, \varnothing\rangle, \langle s, o\rangle\} \subseteq \mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\})$.

Showing that $\langle \varnothing, \varnothing\rangle \in \mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\})$ is easy, since $\varnothing \overset{\sigma}{\rightsquigarrow} x$ for any state $x \subseteq Q$. Clearly we have $\langle \varnothing, \varnothing\rangle \overset{\sigma}{\to} \langle o', o'\rangle$ and also that $\langle o', o'\rangle \preceq_{\mathsf{MH}} \langle s', o'\rangle$.

Let us now show that if $o \neq \varnothing$ then $\langle o, \varnothing\rangle \xrightarrow{\sigma} \langle o' \cup (s' \cap F), o'\rangle$ and $\langle o' \cup (s' \cap F), o'\rangle \preceq_{\mathsf{MH}} \langle s', o'\rangle$. We clearly have that $o \overset{\sigma}{\rightsquigarrow} o' \cup (s' \cap F)$ and $(o' \cup (s' \cap F)) \setminus F = o'$, thus $\langle o, \varnothing\rangle \xrightarrow{\sigma} \langle o' \cup (s' \cap F), o'\rangle$. Since $o' \subseteq s'$ we have that $o' \cup (s' \cap F) \subseteq s'$ and therefore that $\langle o' \cup (s' \cap F), o'\rangle \preceq_{\mathsf{MH}} \langle s', o'\rangle$.

Finally, we show that if $o \neq \varnothing$ then $\langle s, o\rangle \xrightarrow{\sigma} \langle s', o'\rangle$. This is trivial since we have that $s \overset{\sigma}{\rightsquigarrow} s'$ and $o \overset{\sigma}{\rightsquigarrow} o' \cup (s' \cap F)$, and $(o' \cup (s' \cap F)) \setminus F = o'$.

To conclude the proof, we know by (1) that every element of $\mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\})$ is $\succeq_{\mathsf{MH}}$-smaller than either $\langle o, \varnothing\rangle$ or $\langle s, o\rangle$. We know by (2) that $\langle \varnothing, \varnothing\rangle \in \mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\})$ and if $o \neq \varnothing$ then $\{\langle o, \varnothing\rangle, \langle s, o\rangle\} \subseteq \mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\})$. Therefore, if $o \neq \varnothing$ then $\lceil \mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\})\rceil = \{\langle o, \varnothing\rangle, \langle s, o\rangle\}$. If $o = \varnothing$ then $\lceil \mathsf{pre}_\sigma(\downarrow\{\langle s', o'\rangle\})\rceil = \{\langle \varnothing, \varnothing\rangle\}$. □

Lemma 3.5.12 (Computation of $\lfloor \mathsf{post}_\sigma(\cdot)\rfloor$ on $\mathsf{MH}^{\mathsf{ext}}$)**.** *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* ABA *and let $\mathsf{MH}^{\mathsf{ext}}(A) = A' = \langle Q', \Sigma, I, \to, F'\rangle$ be the extended Miyano-Hayashi construction* NFA *of A. Recall from Lemma 3.5.10 that $\preceq_{\mathsf{MH}}$ and $\succeq_{\mathsf{MH}}$ are respectively forward and backward simulations on A'. For every $(s, o) \in Q'$, such that $o \neq \varnothing$ we have that:*

$$\lfloor \mathsf{post}_\sigma(\{(s, \varnothing)\})\rfloor = \{(s', s' \setminus F) \mid s' \in S\} \qquad (\succeq_{\mathsf{MH}})$$

$$\lfloor \mathsf{post}_\sigma(\{(s, o)\})\rfloor = \{(s', o' \setminus F) \mid s' \in S,\ o' \in O\} \qquad (\succeq_{\mathsf{MH}})$$

$$S = \left\lfloor [\![\bigwedge_{q \in s} \delta(q, \sigma)]\!] \right\rfloor \qquad (\supseteq)$$

$$O = \left\lfloor [\![\bigwedge_{q \in o} \delta(q, \sigma)]\!] \right\rfloor \qquad (\supseteq)$$

Proof. We show the first equality. From the definition of $\mathsf{MH}^{\mathsf{ext}}$ we have that:

$$\mathsf{post}_\sigma(\{(s, \varnothing)\}) = \uparrow\left\{(s', s' \setminus F) \;\middle|\; s \overset{\sigma}{\rightsquigarrow} s'\right\} \qquad (\succeq_{\mathsf{MH}})$$

To compute $\lfloor \mathsf{post}_\sigma(\{(s, \varnothing)\})\rfloor$ we proceed as follows:

$$\left\lfloor \uparrow\left\{(s', s' \setminus F) \;\middle|\; s \overset{\sigma}{\rightsquigarrow} s'\right\}\right\rfloor = \left\{(s', s' \setminus F) \;\middle|\; s' \in \left\lfloor [\![\bigwedge_{q \in s} \delta(q, \sigma)]\!] \right\rfloor\right\}$$

For the second equality, the definition of $\mathsf{MH}^{\mathsf{ext}}$ is such that:

$$\mathsf{post}_\sigma(\{(s,o)\}) = \underbrace{\uparrow\left\{(s', o' \setminus F) \;\middle|\; s \overset{\sigma}{\rightsquigarrow} s',\ o \overset{\sigma}{\rightsquigarrow} o'\right\}}_{\alpha}$$

Finally, to compute $\lfloor \mathsf{post}_\sigma(\{(s,o)\}) \rfloor$ we proceed as follows:

$$\lfloor \alpha \rfloor = \left\{ (s', o' \setminus F) \;\middle|\; s' \in \left\lfloor [\![\bigwedge_{q \in s} \delta(q,\sigma)]\!] \right\rfloor \wedge o' \in \left\lfloor [\![\bigwedge_{q \in o} \delta(q,\sigma)]\!] \right\rfloor \right\}$$

□

Due to the strong similarities, it is easy to see that the computation of $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ on the MH state space have the same worst-case complexity than their counterpart for SC. Therefore, $\widehat{\mathsf{pre}}$ can be computed in quadratic time with respect to the size of the ABA, while $\widehat{\mathsf{post}}$ may evaluate to antichains of exponential size.

3.6 Discussion

In this chapter, we have presented an antichain-based framework for solving the reachability and repeated reachability problems for FSM, and we have shown how this framework can be used to develop simple and elegant algorithms for the emptiness problem of alternating automata. We have shown in the chapter how the classical reachability and repeated reachability fixed point expressions can be lifted onto the *lattice of antichains of states*, and how such antichain-based fixed points can be computed efficiently by evaluating pre and post *directly over antichains*, that is without the need to consider the corresponding upward- or downward-closure.

The framework presented in this chapter can be (and has been) improved in a number of ways. For instance, *alternating simulations* can computed in polynomial time on the AFA and ABA, and used to obtain provably-better antichain algorithms for emptiness [DR10]. Also, the decomposition of the computation of pre and post relies on *alphabet enumeration*, which makes it unsuitable for handling automata over exponentially large alphabets; this issue is the main focus of the next chapter.

Chapter 4

LTL Satisfiability and Model Checking Revisited

In this chapter, we develop new antichain-based algorithms for the satisfiability and model checking problems for LTL, by using the reduction to alternating automata emptiness. These new algorithms are based on the antichain framework described in the previous chapter, and use binary decision diagrams to handle the exponentially large alphabets of the alternating automata produced by the translation from LTL. In order to experimentally validate our new algorithms, we have implemented them in a prototype tool called ALASKA. We also provide a detailed comparison of the performance of our antichain-based algorithms with the NuSMV model checker, on a selection of satisfiability and model checking benchmarks, with encouraging results.

4.1 Introduction

Since the seminal work of Pnueli in the late 1970's [Pnu77], the linear temporal logic has been an essential tool for the specification of temporal properties of programs. The *model checking problem* [CES86, CGP99, Mer01] for LTL asks, for a finite state machine M and an LTL formula φ, whether $M \models \varphi$, that is whether $L(M) \subseteq [\![\varphi]\!]$. The *satisfiability problem* for LTL asks, for an LTL formula φ, whether $[\![\varphi]\!] \neq \varnothing$. The satisfiability problem

naturally reduces to model checking, as any LTL formula φ is satisfiable iff $M^{\mathcal{U}} \not\models \neg\varphi$, where $M^{\mathcal{U}}$ is a *universal* FSM which accepts all words. Thanks to this reduction, tools which can solve the model checking problem can also easily be used to check for satisfiability.

In the *automata-based approach to* LTL [VW86, VW94, Var08], the negated LTL formula $\neg\varphi$ is first translated into a language-equivalent non-deterministic automaton $A_{\neg\varphi}$, which enables to reduce the model checking problem $M \models \varphi$ to the language emptiness problem $L(M) \cap L(A_{\neg\varphi}) = \varnothing$. This elegant algorithmic solution has triggered a large research effort, and is now supported by several efficient tools, called LTL *model checkers*.

Model checkers for LTL can be broadly classified in two categories, *explicit* and *symbolic*. Explicit model checkers, such as Spin [Hol97] or SPOT [DP04], explore the synchronous product state space $M \otimes A_{\neg\varphi}$ explicitly, and search for an accepting word (that is, a specification violation) with graph-traversal algorithms such as *nested depth-first search* [HPY96]. On the other hand, symbolic model checkers such as Cadence SMV [McM99], NuSMV [CCG+02], or vis [BHSV+96] represent sets of states and the transition relation of the $M \otimes A_{\neg\varphi}$ product by using ROBDD [CGH94].

The translation from LTL to non-deterministic automata is *worst-case exponential*, and a large body of research is dedicated to optimizing this translation [DGV99, GO01, SB00, Fri03, Tau03]. Current tools for translating LTL formulas into non-deterministic automata have become very sophisticated and it is unlikely that dramatic improvements will be made in that field in the foreseeable future. However, only explicit model checkers are required to perform an *explicit* translation of the LTL formula, as symbolic model checkers can encode the underlying non-deterministic automaton *symbolically* with Boolean constraints [CGH94]. In [RV07], Rozier and Vardi have extensively compared symbolic and explicit approaches for LTL satisfiability checking, and found that the symbolic approach scales better.

Efficient decision methods for LTL satisfiability are highly desirable. The process of writing formal specifications is difficult and error-prone, just like writing implementations is difficult and error prone. Furthermore, model checkers often do not provide the designer with any useful information when the program model does satisfy its specification. Negative responses from

model checkers are usually accompanied with a counter-example trace which can be analyzed and checked, but no such output can be produced for positive instances. Yet positive answers from model checkers can be erroneous when the specification or program model contain bugs. In the face of this, the model checking community has advocated for the use of procedures and tools for "debugging" formal specifications [Kup06]. In the case of LTL, one such *sanity check* is to systematically check that both φ and $\neg\varphi$ are satisfiable formulas. This can be applied on every subformula of the specification. In order for these sanity checking procedures to be usable in practice, we need efficient algorithms for LTL satisfiability.

Contributions The contributions presented in this chapter are as follows. First, we adapt the antichain framework presented in the previous chapter to alternating automata over *symbolic* alphabets. The decomposition of pre and post presented in the previous chapter (see Proposition 3.3.3) is impractical when the alphabet is of the form $2^{\mathbb{P}}$ (due to its exponential size). To cope with this, we introduce a way of combining ROBDD-based techniques with antichain algorithms, taking advantage of the strengths of ROBDD*s* for Boolean reasoning. Also, we extend the combination of ROBDD*s* and antichains to the model checking of LTL specifications over symbolic Kripke structures. Finally, we have implemented and extensively tested our new algorithms and compared their performance with NuSMV. While previous evaluations of antichain algorithms [DDHR06, DR07] were performed on randomly generated models, we experiment here our new algorithms on more concrete satisfiability and model checking examples. Most of our examples are taken from [RV07] and [STV05], where they are presented as benchmarks to compare model checking algorithms. In our experiments, we found that our new algorithms can outperform standard classical symbolic algorithms of the highly optimized industrial-level tool NUSMV for both satisfiability and model checking.

Related Works In [STV05], Vardi et al. propose a *hybrid approach* to LTL model checking: the system is represented symbolically using ROBDD, but the LTL formula is translated explicitly into an NBA. Their method has the nice property to partition the usually huge symbolic state space of

the model into pieces that are associated to each state of the NBA (this heuristic is called *property-driven partitioning* in [RV07]). Our approach also gains from this interesting feature, but in contrast to Vardi et al., we do not need the expensive *a priori* construction of the explicit NBA from the LTL formula.

In [MS03, HKM05] Merz et al. propose a new algorithm for explicit-state LTL model checking that does not rely on the expensive LTL to NBA translation. Their algorithm relies on a computation of the strongly connected components of the underlying NBA in an "on-the-fly" fashion, by using a variant of Tarjan's algorithm. This process of SCC detection is interleaved with the exploration of the explicit-state model, and preliminary experiments indicate that their algorithm compares favorably to Spin.

Structure of the Chapter In Section 4.2 we define *symbolic* alternating automata, which are alternating automata over symbolic alphabets. In Section 4.3 we recall the linear translation of LTL to alternating automata, both for finite words and infinite words. In Section 4.4 we describe how the transition function of symbolic alternating automata can be compactly encoded with ROBDD, and how we can use ROBDD to represent closed sets in the state spaces of the subset construction for AFA, and the Miyano-Hayashi construction for ABA. In Section 4.5 we provide *semi-symbolic* algorithms for the computation of pre and post for symbolic alternating automata, both on finite and infinite words. These algorithms make use of ROBDD existential quantification to avoid the alphabet enumeration issue. In Section 4.6 we describe two classes of algorithms for the conversion of ROBDD-encoded closed sets back into antichains. One type of algorithm recursively traverses the ROBDD to compute the antichain, while the other encodes a strict variant of the simulation ordering over two copies of the propositions, and uses existential quantification. In Section 4.7 we briefly discuss how our new algorithms for LTL satisfiability can be adapted to perform LTL model checking on symbolic Kripke structures. In Section 4.8 we report on our experimental evaluations, both for satisfiability and model checking. Finally, we provide some discussions in Section 4.9.

4.2 Symbolic Alternating Automata

We now define *symbolic alternating automata*, a syntactic variant of the traditional alternating automata defined in Chapter 2 (see Definition 2.3.4). For any set of Boolean propositions X, we denote by $\mathsf{Lit}(X) \stackrel{\mathsf{def}}{=} X \cup \{\neg x \mid x \in X\}$ the set of *literals* over X. Symbolic alternating automata differ from their traditional counterpart in two ways: first, the alphabet Σ is replaced by a set of Boolean propositions $\mathbb{P}$, and second, the transition function $\delta : Q \times \Sigma \mapsto \mathsf{BL}^+(Q)$ is replaced by a function of the kind $\delta : Q \mapsto \mathsf{BL}^+(Q \cup \mathsf{Lit}(\mathbb{P}))$.

Definition 4.2.1 (Symbolic Alternating Automaton)**.** *A* symbolic alternating automaton *is a tuple* $\langle Q, \mathbb{P}, q_0, \delta, F \rangle$ *where:*

- $Q = \{q_1, \ldots, q_n\}$ *is a finite* set of states*;*
- $\mathbb{P} = \{s_1, \ldots, s_k\}$ *is a finite set of* Boolean propositions*;*
- $q_0 \in Q$ *is the* initial state*;*
- $\delta : Q \mapsto \mathsf{BL}^+(Q \cup \mathsf{Lit}(\mathbb{P}))$ *is a* symbolic alternating transition function*;*
- $F \subseteq Q$ *is a set of* final states.

For the sake of completeness, we redefine the notion of runs for symbolic alternating automata. Note that this definition is almost identical to Definition 2.3.5.

Definition 4.2.2 (Run of a Symbolic Alternating Automaton)**.** *Let* $A = \langle Q, \mathbb{P}, q_0, \delta, F \rangle$ *by a symbolic alternating automaton and let* $w \in (2^{\mathbb{P}})^* \cup (2^{\mathbb{P}})^\omega$ *be a word. A* run *of* A *over* w *is directed acyclic graph* $G = \langle V, \rightarrow \rangle$ *where:*

- $V \subseteq Q \times \mathbb{N}$ *is a finite or infinite set of vertices;*
- G *is rooted by* $\langle q_0, 0 \rangle$*;*
- $\rightarrow \subseteq V \times V$ *is such that:*
  1. $\langle q, i \rangle \rightarrow \langle q', j \rangle$ *implies* $j = i + 1$*;*
  2. $\forall \langle q, i \rangle \in V, i < |w| : (\{q' \mid \langle q, i \rangle \rightarrow \langle q', i+1 \rangle\} \cup w[i]) \models \delta(q)$.

The notion of *language* for symbolic alternating automata is equivalent to its definition for traditional alternating automata (see Definition 2.3.6). Likewise, we define symbolic alternating *finite* automata (sAFA, for short)

and symbolic alternating *Büchi* automata (sABA, for short) as symbolic alternating automata respectively interpreted over finite or infinite words.

Finally, we redefine the translations from alternating to non-deterministic automata. As expected, sAFA are translated into NFA using a variant of the subset construction for AFA, and sABA are translated into NBA with a variant of the Miyano-Hayashi construction. These definitions are almost identical to Definitions 2.3.8 and 2.3.11 and are provided for the sake of completeness.

Definition 4.2.3 (Subset Construction for sAFA)**.** *Let* $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$ *be a* sAFA*. The* subset construction *of* A *is the* NFA $\mathsf{SC}(A) = \langle Q', \Sigma, I, \rightarrow, F'\rangle$ *with:*

- $Q' = 2^Q$;
- $\Sigma = 2^{\mathbb{P}}$;
- $I = \{c \subseteq Q \mid q_0 \in c\}$;
- $\rightarrow = \left\{\langle c_1, v, c_2\rangle \;\middle|\; c_2 \cup v \models \bigwedge_{q\in c_1} \delta(q)\right\}$;
- $F' = 2^F$.

Definition 4.2.4 (MH Construction for sABA)**.** *Let* $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$ *be a* sABA*. To alleviate the notations, we introduce the following shorthand:* $x \overset{v}{\rightsquigarrow} y \overset{\mathsf{def}}{=} y \cup v \models \bigwedge_{q\in x} \delta(q)$*. The* Miyano Hayashi construction *of* A *is the* NBA $\mathsf{MH}(A) = \langle Q', \Sigma, I, \rightarrow, F'\rangle$ *with:*

- $Q' = 2^Q \times 2^Q$;
- $\Sigma = 2^{\mathbb{P}}$;
- $I = \{(\{q_0\}, \varnothing)\}$;
- $\rightarrow = \begin{array}{l} \left\{\langle (s_1, \varnothing), v, (s_2, s_2 \setminus F)\rangle \;\middle|\; s_1 \overset{v}{\rightsquigarrow} s_2\right\} \cup \\ \left\{\langle (s_1, o_1 \neq \varnothing), v, (s_2, o_2 \setminus F)\rangle \;\middle|\; s_1 \overset{v}{\rightsquigarrow} s_2,\ o_1 \overset{v}{\rightsquigarrow} o_2,\ o_2 \subseteq s_2\right\}; \end{array}$
- $F' = 2^Q \times \{\varnothing\}$.

4.3 LTL & Symbolic Alternating Automata

In this section, we briefly recall how LTL formulas can be efficiently translated into symbolic alternating automata (see [Var95, Tau03] for more de-

tails). In order to have clean and concise translation algorithms, we shall assume that LTL formulas are in *negation normal form*, which simply requires that all negations in the formula immediately precede a proposition. To properly define the negation normal form we need two extra LTL modalities: $\varphi \,\tilde{\mathsf{U}}\, \psi \stackrel{\mathsf{def}}{=} \neg(\neg\varphi \,\mathsf{U}\, \neg\psi)$ and $\tilde{\mathsf{X}}\,\varphi \stackrel{\mathsf{def}}{=} \neg(\mathsf{X}\,\neg\varphi)$.

Definition 4.3.1 (Negation Normal Form of LTL Formulas)**.** *An* LTL *formula is in negation normal form (*NNF*, for short) if and only if every negation operator it contains immediately precedes a proposition. It is easy to see that every* LTL *formula can be modified to become* NNF *by pushing each negation inward, using the following equivalence rules:*

$$\begin{array}{rlrl} \neg(\varphi \,\mathsf{U}\, \psi) & = \neg\varphi \,\tilde{\mathsf{U}}\, \neg\psi & \neg(\varphi \vee \psi) & = \neg\varphi \wedge \neg\psi \\ \neg(\varphi \,\tilde{\mathsf{U}}\, \psi) & = \neg\varphi \,\mathsf{U}\, \neg\psi & \neg(\varphi \wedge \psi) & = \neg\varphi \vee \neg\psi \\ \neg(\mathsf{X}\,\varphi) & = \tilde{\mathsf{X}}\,\neg\varphi & \neg\neg\varphi & = \varphi \\ \neg(\tilde{\mathsf{X}}\,\varphi) & = \mathsf{X}\,\neg\varphi & & \end{array}$$

By the equivalence rules above, we also have that $\mathsf{NNF}(\neg\mathsf{G}\,\varphi) = \mathsf{F}\,\neg\mathsf{NNF}(\varphi)$ *and* $\mathsf{NNF}(\neg\mathsf{F}\,\varphi) = \mathsf{G}\,\neg\mathsf{NNF}(\varphi)$.

It is important to note that, for the infinite-word semantics, the formulas $\mathsf{X}\,\varphi$ and $\tilde{\mathsf{X}}\,\varphi$ are equivalent. Indeed, from Definition 2.4.4 we know that $w \in [\![\mathsf{X}\,\varphi]\!]$ iff $|w| > 1$ and $w[1\ldots] \in [\![\varphi]\!]$, and $|w| > 1$ is always satisfied for infinite words. For the finite-word semantics however, the formulas $\mathsf{X}\,\varphi$ and $\tilde{\mathsf{X}}\,\varphi$ are different, since $\mathsf{X}\,\varphi$ mandates that the next position in the word satisfies φ, while $\tilde{\mathsf{X}}\,\varphi$ allows the word to end at the current position. For instance, $\tilde{\mathsf{X}}\,\mathsf{false}$ is satisfied only when the current position is the last letter of the word, while $\mathsf{X}\,\mathsf{true}$ is satisfied only for words of length at least 2.

Example 4.3.2 (Transformation of LTL to Negation Normal Form)**.**

$$\mathsf{NNF}\big(\mathsf{G}\,\big(\mathsf{F}\,p \vee \mathsf{F}\,(q \wedge \mathsf{X}\,r)\big) \Rightarrow \mathsf{G}\,p\big) = \mathsf{F}\,\big(\mathsf{G}\,\neg p \wedge \mathsf{G}\,(\neg q \vee \tilde{\mathsf{X}}\,\neg r)\big) \vee \mathsf{G}\,p$$

Theorem 4.3.3 (Language-Equivalent Alternating Automata for LTL [Var95])**.** *Let* $\mathbb{P}$ *be a finite set of Boolean propositions. For every* LTL *formula* $\varphi \in \mathsf{LTL}(\mathbb{P})$ *there exists a* sAFA A^{f} *and a* sABA A^{b} *such that* $L(A^{\mathsf{f}}) = [\![\varphi]\!] \cap (2^{\mathbb{P}})^*$ *and* $L(A^{\mathsf{b}}) = [\![\varphi]\!] \cap (2^{\mathbb{P}})^\omega$*. Moreover, both* A^{f} *and* A^{b} *have a size that is linear in the length of the formula* φ*.*

4.3.1 Translation of LTL to sABA

The translation of LTL interpreted over infinite words to sABA is relatively straightforward. The set of states of the generated sABA is the set of *subformulas* (also called the *closure*) of the input formula. In the sequel, we denote the state corresponding to a formula φ simply by $\boxed{\varphi}$. The initial state is the state that corresponds to the input formula. The transition function δ of the sABA is built for each state $\boxed{\varphi}$ by considering the root of the syntax tree of φ. For instance, the outgoing transitions of a state $\boxed{p \,\mathsf{U}\, \mathsf{X}\, q}$ is $\delta(\boxed{p \,\mathsf{U}\, \mathsf{X}\, q}) = \boxed{q} \vee (p \wedge \boxed{p \,\mathsf{U}\, \mathsf{X}\, q})$. In the resulting sABA, every state is accepting except the states of the form $\varphi \,\mathsf{U}\, \psi$, in order to prevent those states from postponing the satisfaction of their right operand indefinitely.

Definition 4.3.4 (Translation from LTL to sABA)**.** *Let $\mathbb{P}$ be a finite set of Boolean propositions and let $\rho \in \mathsf{LTL}(\mathbb{P})$ be an LTL formula in NNF. The translation of ρ into a language-equivalent symbolic alternating Büchi automaton, denoted $\mathsf{LTL2sABA}(\rho) = \langle Q, \mathbb{P}, q_0, \delta, F \rangle$, is defined as follows:*

- $Q = \left\{ \boxed{\psi} \mid \psi \text{ is a subformula of } \rho \right\}$;
- $q_0 = \boxed{\rho}$;
- $F = Q \setminus \left\{ \boxed{\varphi \,\mathsf{U}\, \psi} \;\middle|\; \boxed{\varphi \,\mathsf{U}\, \psi} \in Q \right\}$;
- *For each state $q \in Q$ the formula $\delta(q)$ satisfies the following:*

$$
\begin{array}{rclrcl}
\delta(\boxed{\mathsf{true}}) & = & \mathsf{true} & \delta(\boxed{p}) & = & p \\
\delta(\boxed{\mathsf{false}}) & = & \mathsf{false} & \delta(\boxed{\neg p}) & = & \neg p \\
\delta(\boxed{\varphi \,\mathsf{U}\, \psi}) & = & \delta(\boxed{\psi}) \vee \left(\delta(\boxed{\varphi}) \wedge \boxed{\varphi \,\mathsf{U}\, \psi}\right) & \delta(\boxed{\varphi \vee \psi}) & = & \delta(\boxed{\varphi}) \vee \delta(\boxed{\psi}) \\
\delta(\boxed{\varphi \,\tilde{\mathsf{U}}\, \psi}) & = & \delta(\boxed{\psi}) \wedge \left(\delta(\boxed{\varphi}) \vee \boxed{\varphi \,\tilde{\mathsf{U}}\, \psi}\right) & \delta(\boxed{\varphi \wedge \psi}) & = & \delta(\boxed{\varphi}) \wedge \delta(\boxed{\psi}) \\
\delta(\boxed{\mathsf{X}\, \varphi}) & = & \boxed{\varphi} & \delta(\boxed{\tilde{\mathsf{X}}\, \varphi}) & = & \boxed{\varphi}
\end{array}
$$

Example 4.3.5 (Example of LTL $\rightarrow$ sABA Translation)**.** *Let ρ be the negation normal form of $\mathsf{G}\,(r \Rightarrow \mathsf{X}\,(\neg r \,\mathsf{U}\, g))$, that is $\rho = \mathsf{false}\,\tilde{\mathsf{U}}\,(\neg r \vee \mathsf{X}\,(\neg r \,\mathsf{U}\, g))$. The automaton of Figure 4.1 depicts $\mathsf{LTL2sABA}(\rho)$. Note that most states are unreachable from q_0 and should be removed in practical implementations.*

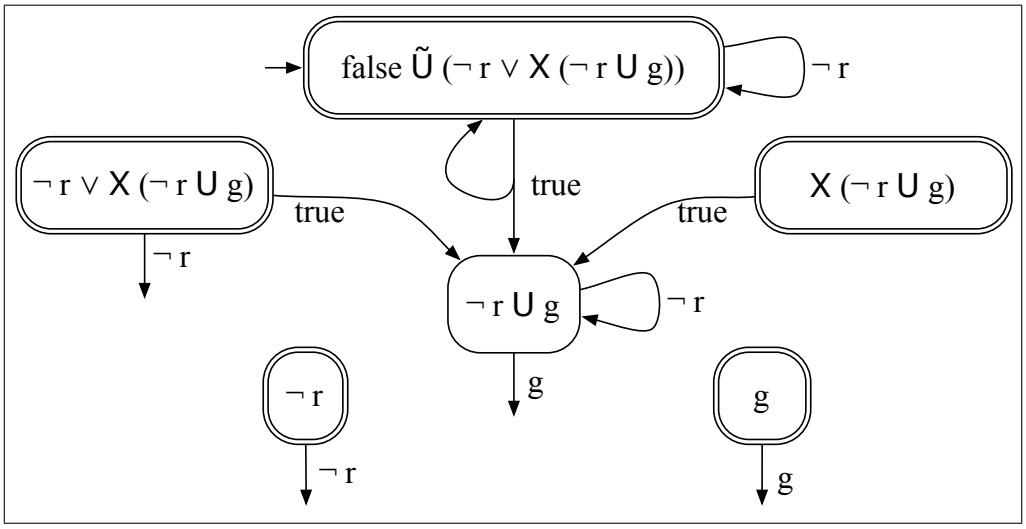


Figure 4.1: The sABA obtained by translating the formula $\mathsf{G}(r \Rightarrow \mathsf{X}(\neg r \,\mathsf{U}\, g))$. Unattached arrow tips indicate transitions which only constrain propositions.

Lemma 4.3.6 (Correctness of LTL2sABA)**.** *For every set of propositions* $\mathbb{P}$*, for every formula* $\rho \in \mathsf{LTL}(\mathbb{P})$ *we have that* $L(\mathsf{LTL2sABA}(\rho)) = [\![\rho]\!] \cap (2^{\mathbb{P}})^\omega$*.*

Proof. Let $A_\rho = \mathsf{LTL2sABA}(\rho) = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$, and for every state $q \in Q$, let $A_\rho^q = \langle Q, \mathbb{P}, q, \delta, F\rangle$. In the following, we abbreviate each $L(A_\rho^q)$ simply by $L(q)$. Let us show by induction over the structure of ρ that for every subformula φ of ρ we have that $L(\boxed{\varphi}) = [\![\varphi]\!]$. It is easy to see that this is true for the base cases such that $\varphi \in \{\mathsf{true}, \mathsf{false}, p, \neg p\}$.

For each subformula of the form $\varphi \vee \psi$ or $\varphi \wedge \psi$ we know by the induction hypothesis that $L(\boxed{\varphi}) = [\![\varphi]\!]$ and $L(\boxed{\psi}) = [\![\psi]\!]$. Clearly we have that $L(\boxed{\varphi \vee \psi}) = L(\boxed{\varphi}) \cup L(\boxed{\psi})$ and similarly that $L(\boxed{\varphi \wedge \psi}) = L(\boxed{\varphi}) \cap L(\boxed{\psi})$. Therefore we have that $L(\boxed{\varphi \vee \psi}) = [\![\varphi \vee \psi]\!]$ and $L(\boxed{\varphi \wedge \psi}) = [\![\varphi \wedge \psi]\!]$.

For subformulas of the form $\varphi \,\mathsf{U}\, \psi$ we use the well-know LTL equivalence $\varphi \,\mathsf{U}\, \psi \Leftrightarrow \psi \vee (\varphi \wedge \mathsf{X}(\varphi \,\mathsf{U}\, \psi))$. Indeed, from Definition 2.4.4 we have that $w \in [\![\varphi \,\mathsf{U}\, \psi]\!]$ if and only if:

$$\exists\, i < |w| : w[i \ldots] \in [\![\psi]\!] \wedge \forall\, j < i : w[j \ldots] \in [\![\varphi]\!]$$

We can rewrite this easily as follows (note that $w[0 \ldots] = w$ for any non-

empty word w):

$$w \in [\![\psi]\!] \vee \Big(w \in [\![\varphi]\!] \wedge \exists\, 0 < i < |w| : w[i \ldots] \in [\![\psi]\!] \wedge \forall\, j < i : w[j \ldots] \in [\![\varphi]\!]\Big)$$

It is easy to see that this expression is equivalent to $[\![\psi \vee (\varphi \wedge \mathsf{X}(\varphi \,\mathsf{U}\, \psi))]\!]$. We know by the induction hypothesis that $L(\boxed{\varphi}) = [\![\varphi]\!]$ and $L(\boxed{\psi}) = [\![\psi]\!]$. Let $w \in [\![\varphi \,\mathsf{U}\, \psi]\!]$, which implies that there exists some $i \in \mathbb{N}$ such that $w[i \ldots] \in [\![\psi]\!]$ and for all $j < i$ we have $w[j \ldots] \in [\![\varphi]\!]$. To build an accepting run of $\langle Q, \mathbb{P}, \boxed{\varphi \,\mathsf{U}\, \psi}, \delta, F\rangle$, it is easy to see that it is possible to simply "loop" over the state $\boxed{\varphi \,\mathsf{U}\, \psi}$ exactly $i - 1$ times (or zero times if $i = 0$), satisfying the constraint $\delta(\boxed{\varphi})$ each time, and then following $\delta(\boxed{\psi})$. In the reverse direction, let $w \in L(\langle Q, \mathbb{P}, \boxed{\varphi \,\mathsf{U}\, \psi}, \delta, F\rangle)$. Since $\boxed{\varphi \,\mathsf{U}\, \psi}$ is a non accepting state, an accepting run cannot "loop" indefinitely over it. This implies that there exists at least one position i in w such that $w[i \ldots] \in L(\boxed{\psi})$ and that for every position $j < i$ we have that $w[j \ldots] \in L(\boxed{\varphi})$, which implies that $w \in [\![\varphi \,\mathsf{U}\, \psi]\!]$ thanks to the induction hypothesis.

The case of subformulas of the form $\varphi \,\tilde{\mathsf{U}}\, \psi$ is dual to the previous one and can be shown similarly. Finally, the case of subformulas of the form $\mathsf{X}\,\varphi$ or $\tilde{\mathsf{X}}\,\varphi$ is trivial under the induction hypothesis.

□

4.3.2 Translation of LTL to sAFA

The translation from LTL to sAFA is a little bit more involved that its counterpart for sABA for essentially two reasons. First, computing the set F of accepting states is more difficult, since it amounts to decide for each state $\boxed{\varphi}$ whether $\epsilon \in [\![\varphi]\!]$ or not. Second, the modalities X and $\tilde{\mathsf{X}}$ have a distinct semantics on finite words, which has to be taken into account.

To solve the latter issue, the translation to sAFA adds two artificial states $\boxed{\epsilon}$ and $\boxed{\tilde{\epsilon}}$ such that $\boxed{\epsilon}$ is accepting and $\delta(\boxed{\epsilon}) = \mathsf{false}$, while $\boxed{\tilde{\epsilon}}$ is non-accepting and $\delta(\boxed{\tilde{\epsilon}}) = \mathsf{true}$. Thus we have that the language of the state $\boxed{\epsilon}$ is $\{\epsilon\}$, and the language of $\boxed{\tilde{\epsilon}}$ is $(2^{\mathbb{P}})^* \setminus \{\epsilon\}$. These two states are used to encode the transitions of the $\boxed{\mathsf{X}\,\varphi}$ and $\boxed{\tilde{\mathsf{X}}\,\varphi}$ states by either forbidding or allowing the word to end at the current position.

The set of final states F is defined with a least fixed point computation. First, states of the form $\boxed{\mathsf{true}}$, $\boxed{\epsilon}$, $\boxed{\varphi\,\tilde{\mathsf{U}}\,\psi}$ are initially set as accepting. Then, states of the form $\boxed{\varphi \vee \psi}$ and $\boxed{\varphi \wedge \psi}$ are added to F when one of $\boxed{\varphi}$ or $\boxed{\psi}$ belong to F, or when both $\boxed{\varphi}$ and $\boxed{\psi}$ belong to F, respectively.

Definition 4.3.7 (Translation from LTL to sAFA)**.** *Let $\mathbb{P}$ be a finite set of Boolean propositions and let $\rho \in \mathsf{LTL}(\mathbb{P})$ be an* LTL *formula in* NNF*. The translation of ρ into a language-equivalent symbolic alternating finite automaton, denoted $\mathsf{LTL2sAFA}(\rho) = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$, is defined as follows:*

- $Q = \left\{\boxed{\psi} \mid \psi \text{ is a subformula of } \rho\right\} \cup \left\{\boxed{\epsilon}, \boxed{\tilde{\epsilon}}\right\}$;
- $q_0 = \boxed{\rho}$;
- $F^0 = \left\{\boxed{\mathsf{true}}\right\} \cup \{\boxed{\epsilon}\} \cup \left\{\boxed{\varphi\,\tilde{\mathsf{U}}\,\psi} \;\middle|\; \boxed{\varphi\,\tilde{\mathsf{U}}\,\psi} \in Q\right\}$
  $F^\lambda = \lambda X : \left\{\boxed{\varphi \vee \psi} \;\middle|\; \boxed{\varphi} \in X \vee \boxed{\psi} \in X\right\} \cup \left\{\boxed{\varphi \wedge \psi} \;\middle|\; \boxed{\varphi} \in X \wedge \boxed{\psi} \in X\right\}$
  $F = \mathsf{LFP}(\lambda X : F^0 \cup F^\lambda(X))$
- *For each state $q \in Q$ the formula $\delta(q)$ satisfies the following:*

$$
\begin{array}{rclrcl}
\delta(\boxed{\mathsf{true}}) &=& \mathsf{true} & \delta(\boxed{p}) &=& p \\
\delta(\boxed{\mathsf{false}}) &=& \mathsf{false} & \delta(\boxed{\neg p}) &=& \neg p \\
\delta(\boxed{\varphi\,\mathsf{U}\,\psi}) &=& \delta(\boxed{\psi}) \vee \left(\delta(\boxed{\varphi}) \wedge \boxed{\varphi\,\mathsf{U}\,\psi}\right) & \delta(\boxed{\varphi \vee \psi}) &=& \delta(\boxed{\varphi}) \vee \delta(\boxed{\psi}) \\
\delta(\boxed{\varphi\,\tilde{\mathsf{U}}\,\psi}) &=& \delta(\boxed{\psi}) \wedge \left(\delta(\boxed{\varphi}) \vee \boxed{\varphi\,\tilde{\mathsf{U}}\,\psi}\right) & \delta(\boxed{\varphi \wedge \psi}) &=& \delta(\boxed{\varphi}) \wedge \delta(\boxed{\psi}) \\
\delta(\boxed{\mathsf{X}\,\varphi}) &=& \boxed{\varphi} \wedge \boxed{\tilde{\epsilon}} & \delta(\boxed{\tilde{\mathsf{X}}\,\varphi}) &=& \boxed{\varphi} \vee \boxed{\epsilon} \\
\delta(\boxed{\epsilon}) &=& \mathsf{false} & \delta(\boxed{\tilde{\epsilon}}) &=& \mathsf{true}
\end{array}
$$

Example 4.3.8 (Example of LTL → sAFA Translation)**.** *Let ρ be the* NNF *of $\mathsf{G}\,(a \Leftrightarrow \neg\mathsf{X}\,\mathsf{true})$, that is $\rho = \mathsf{false}\,\tilde{\mathsf{U}}\,\big((a \wedge \tilde{\mathsf{X}}\,\mathsf{false}) \vee (\neg a \wedge \mathsf{X}\,\mathsf{true})\big)$. The automaton of Figure 4.2 depicts $\mathsf{LTL2sAFA}(\rho)$. It is easy to see that the semantics of ρ is the set of words in which a is true at the last position and only there (this includes ϵ). This example illustrates the necessity of the $\boxed{\epsilon}$ and $\boxed{\tilde{\epsilon}}$ states.*

Lemma 4.3.9 (Soundness of LTL2sAFA)**.** *For every set of propositions $\mathbb{P}$, for every formula $\rho \in \mathsf{LTL}(\mathbb{P})$ we have that $L(\mathsf{LTL2sAFA}(\rho)) = [\![\rho]\!] \cap (2^{\mathbb{P}})^*$.*

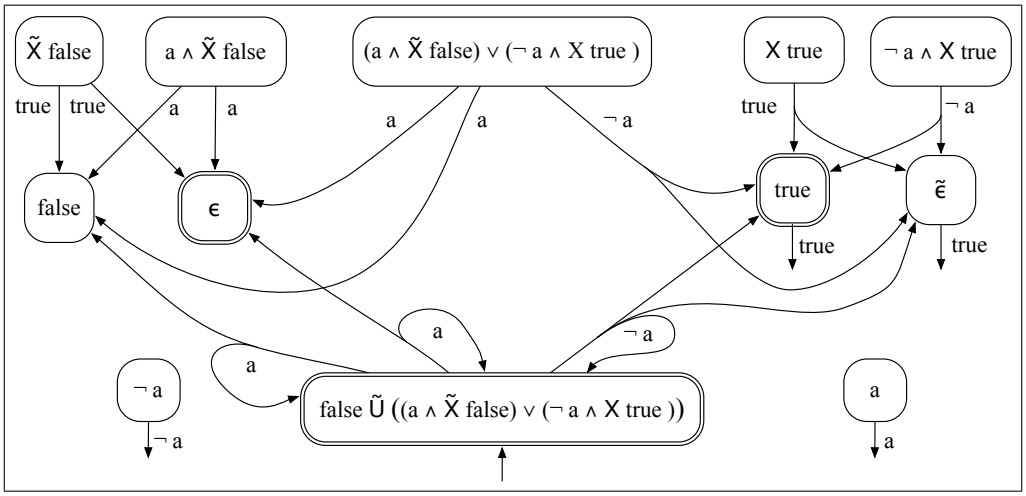


Figure 4.2: The sAFA obtained by translating the formula $\mathsf{G}\,(a \Leftrightarrow \neg \mathsf{X}\,\mathsf{true})$.

Proof. Let $A_\rho = \mathsf{LTL2sAFA}(\rho) = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$, and for every state $q \in Q$, let $A^q_\rho = \langle Q, \mathbb{P}, q, \delta, F\rangle$. Again, we abbreviate each $L(A^q_\rho)$ simply by $L(q)$. Let us show by induction over the structure of ρ that for every subformula φ of ρ we have that $L(\boxed{\varphi}) = [\![\varphi]\!]$. It is easy to see that this is true for the base cases such that $\varphi \in \{\mathsf{false}, p, \neg p\}$ since $\boxed{\mathsf{false}}$, $\boxed{p}$, and $\boxed{\neg p}$ are all non-accepting states. For the true subformulas, we have that $L(\boxed{\mathsf{true}}) = (2^{\mathbb{X}})^*$ since $\boxed{\mathsf{true}}$ is an accepting state with no constraint on its successors.

For each subformula of the form $\varphi \vee \psi$ or $\varphi \wedge \psi$ we know by the induction hypothesis that $L(\boxed{\varphi}) = [\![\varphi]\!]$ and $L(\boxed{\psi}) = [\![\psi]\!]$. Furthermore, we know that $\epsilon \in L(\boxed{\varphi \vee \psi})$ iff $\epsilon \in L(\boxed{\varphi})$ or $\epsilon \in L(\boxed{\psi})$ and $\epsilon \in L(\boxed{\varphi \wedge \psi})$ iff $\epsilon \in L(\boxed{\varphi})$ and $\epsilon \in L(\boxed{\psi})$. Therefore we have that $L(\boxed{\varphi \vee \psi}) = L(\boxed{\varphi}) \cup L(\boxed{\psi})$ and similarly that $L(\boxed{\varphi \wedge \psi}) = L(\boxed{\varphi}) \cap L(\boxed{\psi})$. Using the induction hypothesis we have that $L(\boxed{\varphi \vee \psi}) = [\![\varphi \vee \psi]\!]$ and $L(\boxed{\varphi \wedge \psi}) = [\![\varphi \wedge \psi]\!]$.

For subformulas of the form $\varphi\,\mathsf{U}\,\psi$ we use the well-know LTL equivalence $\varphi\,\mathsf{U}\,\psi \Leftrightarrow \psi \vee (\varphi \wedge \mathsf{X}\,(\varphi\,\mathsf{U}\,\psi))$. We know by the induction hypothesis that $L(\boxed{\varphi}) = [\![\varphi]\!]$ and $L(\boxed{\psi}) = [\![\psi]\!]$. Let $w \in [\![\varphi\,\mathsf{U}\,\psi]\!]$, which implies that there exists some $i < |w|$ such that $w[i\ldots] \in [\![\psi]\!]$ and for all $j < i$ we have $w[j\ldots] \in [\![\varphi]\!]$. To build an accepting run of $\langle Q, \mathbb{P}, \boxed{\varphi\,\mathsf{U}\,\psi}, \delta, F\rangle$, it is easy to see that it is possible to simply "loop" over the state $\boxed{\varphi\,\mathsf{U}\,\psi}$

exactly $i-1$ times (or zero times if $i=0$), satisfying the constraint $\delta(\boxed{\varphi})$ each time, and then following $\delta(\boxed{\psi})$. In the reverse direction, let $w \in L(\langle Q, \mathbb{P}, \boxed{\varphi \,\mathsf{U}\, \psi}, \delta, F\rangle)$. Since $\boxed{\varphi \,\mathsf{U}\, \psi}$ is a non accepting state, we know that $|w| \geq 1$ and that our accepting run takes the $\delta(\boxed{\psi})$ transition before the end of the word (otherwise it would not be accepting). This implies that there exists at least one position i in w such that $w[i\ldots] \in L(\boxed{\psi})$ and that for every position $j < i$ we have that $w[j\ldots] \in L(\boxed{\varphi})$, which implies that $w \in [\![\varphi \,\mathsf{U}\, \psi]\!]$ thanks to the induction hypothesis. Note that $\boxed{\varphi \,\mathsf{U}\, \psi}$ states must be non-accepting since $\epsilon \notin [\![\varphi \,\mathsf{U}\, \psi]\!]$ for any φ and ψ.

The case of subformulas of the form $\varphi \,\tilde{\mathsf{U}}\, \psi$ is dual to the previous one and can be shown similarly. Note that $\boxed{\varphi \,\tilde{\mathsf{U}}\, \psi}$ states must be accepting since $\epsilon \in [\![\varphi \,\tilde{\mathsf{U}}\, \psi]\!]$ for any φ and ψ.

Finally, the two cases of subformulas of the form $\mathsf{X}\,\varphi$ or $\tilde{\mathsf{X}}\,\varphi$ are handled as follows. We know that $w \in [\![\mathsf{X}\,\varphi]\!]$ iff $|w| > 1$ and $w[1\ldots] \in [\![\varphi]\!]$ and therefore that $w \in [\![\tilde{\mathsf{X}}\,\varphi]\!]$ iff $|w| \leq 1$ or $w[1\ldots] \in [\![\varphi]\!]$. Clearly, we have that $L(\boxed{\epsilon}) = \{\epsilon\}$ and $L(\boxed{\tilde{\epsilon}}) = (2^{\mathbb{P}})^* \setminus \{\epsilon\}$, thus we have that $L(\boxed{\mathsf{X}\,\varphi}) = [\![\mathsf{X}\,\varphi]\!]$ and $L(\boxed{\tilde{\mathsf{X}}\,\varphi}) = [\![\tilde{\mathsf{X}}\,\varphi]\!]$. □

4.4 ROBDD Encodings for Alternating Automata

In this section, we discuss ROBDD *encodings* for symbolic alternating automata. More specifically, we show how sets of states of sAFA or sABA can be encoded using ROBDD, and also how their δ transition function can be encoded using one ROBDD per state of the automaton.

4.4.1 ROBDD Encodings for sAFA

The ROBDD encoding for sAFA is very straightforward. Let $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$ be a sAFA such that $Q = \{q_1, \ldots q_k\}$. The ROBDD encoding for sAFA uses a set of propositions $\mathbb{X} = \{x_1, \ldots, x_k\}$ such that every $x_i \in \mathbb{X}$ represents the corresponding $q_i \in Q$ in the encoding. A set of states of $\mathsf{SC}(A)$ (that is a set of set of states of A) is therefore represented by an element in $\mathsf{ROBDD}(\mathbb{X})$.

To represent the δ of sAFA with ROBDD, we use ROBDD over the set of

propositions $\mathbb{X} \cup \mathbb{P}$. The ROBDD encoding $\delta(q)$ is denoted $\Delta(q)$ in the sequel, and is obtained by building the ROBDD corresponding to the formula $\delta(q)$ in which every occurrence of q_i is replaced with the corresponding x_i (this is denoted $\delta(q)[\forall\, q_i \in Q : q_i \leftarrow x_i]$).

Definition 4.4.1 (ROBDD encoding for SC)**.** *Let $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$ be a* sAFA *with $Q = \{q_1, \ldots, q_k\}$, and let $\mathbb{X} = \{x_1, \ldots, x_k\}$ be a set of Boolean propositions. The* ROBDD *encoding of sets of states of* $\mathsf{SC}(A)$ *is defined by two functions* $\mathsf{Enc}_{\mathsf{SC}} : 2^{2^Q} \mapsto \mathsf{ROBDD}(\mathbb{X})$ *and* $\mathsf{Dec}_{\mathsf{SC}} : \mathsf{ROBDD}(\mathbb{X}) \mapsto 2^{2^Q}$*, and the* ROBDD *encoding of δ is the function $\Delta : Q \mapsto \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{P})$ such that:*

$$\begin{aligned}
\Delta(q) &= \mathsf{BuildROBDD}(\delta(q)[\forall\, q_i \in Q : q_i \leftarrow x_i]) \\
\mathsf{Enc}_{\mathsf{SC}}(S) &= \bigvee_{s \in S} \bigwedge_{q_i \in Q} (x_i \Leftrightarrow q_i \in s) \\
\mathsf{Dec}_{\mathsf{SC}}(d) &= \{\{q_i \mid x_i \in v\} \mid v \subseteq \mathbb{X}, [\![d]\!](v) = \mathsf{true}\}
\end{aligned}$$

4.4.2 ROBDD Encodings for sABA

The ROBDD encoding for sABA is more involved than what we have previously defined for sAFA. Let $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$ be a sABA such that $Q = \{q_1, \ldots q_k\}$. The ROBDD encoding for A uses two sets of propositions $\mathbb{X} = \{x_1, \ldots, x_k\}$ and $\mathbb{Y} = \{y_1, \ldots, y_k\}$. The variables in $\mathbb{X}$ are used to constrain the "s" component of the $\langle s, o\rangle$ states of the Miyano-Hayashi construction, while the variables in $\mathbb{Y}$ are used to constrain the "o" component. A set of states of $\mathsf{MH}(A)$ (that is, a set of pairs of sets of states of A) is therefore represented by an element in $\mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$.

To represent the δ of sABA with ROBDD, we use ROBDD either over the set of propositions $\mathbb{X} \cup \mathbb{P}$, or over the set of propositions $\mathbb{Y} \cup \mathbb{P}$. The ROBDD encoding $\delta(q)$ is given by two functions denoted $\Delta_x(q) \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{P})$ and $\Delta_y(q) \in \mathsf{ROBDD}(\mathbb{Y} \cup \mathbb{P})$. These two encodings of δ are needed in the sequel to constrain the successors of the "s" and "o" components separately. Both Δ_x and Δ_y are computed in a similar fashion than Δ for sAFA.

Definition 4.4.2 (ROBDD encoding for MH)**.** *Let $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$ be a* sABA *such that $Q = \{q_1, \ldots, q_k\}$, let $\mathbb{X} = \{x_1, \ldots, x_k\}$ and $\mathbb{Y} = \{y_1, \ldots, y_k\}$*

be two sets of Boolean propositions. The ROBDD *encoding of sets of states of* $\mathsf{MH}(A)$ *is defined by two functions* $\mathsf{Enc}_{\mathsf{MH}} : 2^{(2^Q \times 2^Q)} \mapsto \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ *and* $\mathsf{Dec}_{\mathsf{MH}} : \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y}) \mapsto 2^{(2^Q \times 2^Q)}$*, and the* ROBDD *encoding of* δ *is defined by two functions* $\Delta_x : Q \mapsto \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{P})$ *and* $\Delta_y : Q \mapsto \mathsf{ROBDD}(\mathbb{Y} \cup \mathbb{P})$ *such that:*

$$\begin{aligned}
\Delta_x(q) &= \mathsf{BuildROBDD}(\delta(q)[\forall\, q_i \in Q : q_i \leftarrow x_i]) \\
\Delta_y(q) &= \mathsf{BuildROBDD}(\delta(q)[\forall\, q_i \in Q : q_i \leftarrow y_i]) \\
\mathsf{Enc}_{\mathsf{MH}}(S) &= \bigvee_{\langle s,o\rangle \in S} \bigwedge_{q_i \in Q} (x_i \Leftrightarrow q_i \in s) \wedge (y_i \Leftrightarrow q_i \in o) \\
\mathsf{Dec}_{\mathsf{MH}}(d) &= \{\langle \{q_i \mid x_i \in v\}, \{q_i \mid y_i \in v\}\rangle \mid v \subseteq \mathbb{X} \cup \mathbb{Y}, [\![d]\!](v) = \mathsf{true}\}
\end{aligned}$$

4.5 Semi-Symbolic Operators pre and post

Recall from Proposition 3.3.3 of the previous chapter that we can compute the set of predecessors or successors of an antichain X by iterating over each element of X, and each letter of the alphabet, individually. As we have argued previously, it is not scalable to enumerate the letters of symbolic alternating automata, so we shall use ROBDD *existential quantification* to perform the union over all letters symbolically.

Let $M = \langle Q, 2^{\mathbb{P}}, I, \rightarrow, F\rangle$ be a finite state machine over the symbolic alphabet $\mathbb{P}$ equipped with either a forward simulation preorder $\preceq_{\mathsf{f}}$, and/or a backward simulation preorder $\succeq_{\mathsf{b}}$, and let $\mathsf{Enc} : 2^Q \mapsto \mathsf{ROBDD}(\mathbb{X})$ and $\mathsf{Dec} : \mathsf{ROBDD}(\mathbb{X}) \mapsto 2^Q$ be appropriate encoding and decoding functions for M (that is Enc is a bijection and $\mathsf{Dec} = \mathsf{Enc}^{-1}$). We can rewrite the equalities of Proposition 3.3.3 as follows:

$$\widehat{\mathsf{pre}}(X) = \bigsqcup_{x \in X} \left\lceil \mathsf{Dec}\Big(\mathsf{Enc}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{pre}_v(\downarrow\{x\})\big)\Big) \right\rceil \qquad (\preceq_{\mathsf{f}})$$

$$\widehat{\mathsf{post}}(X) = \bigsqcup_{x \in X} \left\lfloor \mathsf{Dec}\Big(\mathsf{Enc}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{post}_v(\uparrow\{x\})\big)\Big) \right\rfloor \qquad (\succeq_{\mathsf{b}})$$

In this section, we provide *semi-symbolic* algorithms to compute both $\widehat{\mathsf{pre}}$ and $\widehat{\mathsf{post}}$ in the way shown above. Furthermore, we break the problem

into the following constituting parts:

$$\lambda\, d : \lceil \mathsf{Dec}(d) \rceil \qquad \lambda\, x : \mathsf{Enc}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{pre}_v(\downarrow\{x\})\big) \qquad (\preceq_{\mathsf{f}})$$

$$\lambda\, d : \lfloor \mathsf{Dec}(d) \rfloor \qquad \lambda\, x : \mathsf{Enc}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{post}_v(\uparrow\{x\})\big) \qquad (\succeq_{\mathsf{b}})$$

In what follows, we study efficient algorithms to solve each of the four functions above, on the FSM generated by the Subset Construction for sAFA, and the Miyano-Hayashi construction for sABA.

4.5.1 Semi-Symbolic pre and post for sAFA

We begin with the computation of $\lambda\, x : \mathsf{Enc}_{\mathsf{SC}}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{post}_v(\uparrow\{x\})\big)$ on the Subset Construction of sAFA. Recall from the previous chapter that, since $\supseteq$ is a bidirectional simulation preorder on SC, it suffices to compute the function $\lambda\, x : \mathsf{Enc}_{\mathsf{SC}}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{post}_v(\{x\})\big)$ and thus discard the inner $\uparrow(\cdot)$ operation (see Lemma 3.3.8).

Algorithm 4.1: Computation of $\mathsf{Enc}_{\mathsf{SC}}(\mathsf{post}_{\mathsf{SC}}(\{s\}))$

Input: a sAFA $A = \langle Q, \mathbb{P}, q_0, \delta, F \rangle$, and a state $s \subseteq Q$ of $\mathsf{SC}(A)$
Output: $\mathsf{Enc}_{\mathsf{SC}}(\mathsf{post}_{\mathsf{SC}}(\{s\}))$

```
1 begin PostSC(s)
2 |   return ∃ℙ : ⋀_{q∈s} Δ(q)
3 end
```

The function PostSC, depicted in Algorithm 4.1, performs the computation $\lambda s : \mathsf{Enc}_{\mathsf{SC}}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{post}_v(\{s\})\big)$ on the Subset Construction of sAFA. It simply takes the conjunction of the ROBDD $\Delta(q_i)$ for each q_i found in s, and quantifies the result existentially over $\mathbb{P}$.

Lemma 4.5.1 (Soundness of PostSC)**.** *For any* sAFA $A = \langle Q, \mathbb{P}, q_0, \delta, F \rangle$*, for any state* $s \subseteq Q$ *of* $\mathsf{SC}(A)$ *we have that* $\mathsf{PostSC}(s) = \mathsf{Enc}_{\mathsf{SC}}(\mathsf{post}_{\mathsf{SC}}(\{s\}))$.

Proof. From the definition of $\mathsf{SC}(A)$ we have that:

$$\begin{aligned}\mathsf{post}(\{s\}) &= \left\{ s' \subseteq Q \;\middle|\; \exists\, v \subseteq \mathbb{P} : s' \cup v \models \bigwedge_{q \in s} \delta(q) \right\} \\ &= \bigcup_{v \subseteq \mathbb{P}} \left\{ s' \subseteq Q \;\middle|\; s' \cup v \models \bigwedge_{q \in s} \delta(q) \right\} \\ &= \mathsf{Dec}_{\mathsf{SC}}\big(\exists\, \mathbb{P} : \bigwedge_{q \in s} \Delta(q)\big)\end{aligned}$$

□

We proceed with the computation of $\lambda s' : \mathsf{Enc}_{\mathsf{SC}}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{pre}_v(\downarrow\{s'\})\big)$ on the Subset Construction of sAFA. Again, we make use of the fact that $\subseteq$ is a bidirectional simulation to avoid the $\downarrow(\cdot)$ computation (see Lemma 3.3.7).

Algorithm 4.2: Computation of $\mathsf{Enc}_{\mathsf{SC}}(\mathsf{pre}(\{s'\}))$

Input: a sAFA $A = \langle Q, \mathbb{P}, q_0, \delta, F \rangle$, and a state $s' \subseteq Q$ of $\mathsf{SC}(A)$
Output: $\mathsf{Enc}_{\mathsf{SC}}(\mathsf{pre}_{\mathsf{SC}}(\{s'\}))$

```
1 begin PreSC(s')
2 |   return ∃ℙ : ⋀_{q_i∈Q} ( x_i ⇒ ∃𝕏 : ( Δ(q_i) ∧ ⋀_{q_j∈Q} (x_j ⇔ q_j ∈ s') ) )
3 end
```

The function PreSC, depicted in Algorithm 4.2, performs the computation $\lambda s' : \mathsf{Enc}_{\mathsf{SC}}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{pre}_v(\{s'\})\big)$ on the Subset Construction of sAFA. This algorithm first computes, for each $q_i \in Q$, the set of valuations $v \subseteq \mathbb{P}$ such that $s' \cup v \models \delta(q_i)$ (let us denote this set by $\Phi(q_i)$). Then, it returns the ROBDD equal to $\exists\, \mathbb{P} : \bigwedge_{q_i \in Q} \big(x_i \Rightarrow \Phi(q_i)\big)$. This means that PreSC(s′) is equal to the ROBDD encoding of the set of states s for which there exists a valuation $v \subseteq \mathbb{P}$ such that for every $q_i \in Q$, either $q_i \notin s$, or $s' \cup v \models \delta(q_i)$. Thanks to Lemma 3.5.2, we know that this set corresponds exactly to $\bigcup_{v \subseteq \mathbb{P}} \mathsf{pre}_v(\{s'\})$.

Lemma 4.5.2 (Soundness of PreSC)**.** *For any* sAFA $A = \langle Q, \mathbb{P}, q_0, \delta, F \rangle$*, for any state* $s' \subseteq Q$ *of* $\mathsf{SC}(A)$ *we have that* $\mathsf{PreSC}(s') = \mathsf{Enc}_{\mathsf{SC}}(\mathsf{pre}_{\mathsf{SC}}(\{s'\}))$.

Proof. We proceed step-by-step as follows:

$$
\begin{aligned}
&\exists\,\mathbb{P} : \bigwedge_{q_i \in Q} \Big(x_i \Rightarrow \exists\,\mathbb{X} : \big(\Delta(q_i) \wedge \bigwedge_{q_j \in Q} (x_j \Leftrightarrow q_j \in s')\big)\Big) \\
&= \bigcup_{v \subseteq \mathbb{P}} \bigcap_{q_i \in Q} \{ s' \subseteq Q \;\mid\; q_i \notin s' \vee v \in \{ v' \;\mid\; \exists\, s'' \subseteq Q : v' \cup s'' \models \delta(q_i) \wedge s'' = s' \} \} \\
&= \bigcup_{v \subseteq \mathbb{P}} \bigcap_{q_i \in Q} \{ s' \subseteq Q \;\mid\; q_i \notin s' \vee v \cup s' \models \delta(q_i) \} \\
&= \bigcup_{v \subseteq \mathbb{P}} \downarrow \{ \{ q_i \in Q \;\mid\; v \cup s' \models \delta(q_i) \} \} \\
&= \bigcup_{v \subseteq \mathbb{P}} \mathsf{pre}_v(\{s'\}) \\
&= \mathsf{pre}(\{s'\})
\end{aligned}
$$

□

4.5.2 Semi-Symbolic pre and post for sABA

We now turn to the computation of $\lambda\,\langle s, o\rangle : \mathsf{Enc}_{\mathsf{MH}}\big(\bigcup_{v\subseteq\mathbb{P}} \mathsf{post}_v(\uparrow\{\langle s, o\rangle\})\big)$ on the Miyano-Hayashi construction of sABA. Similarly to the finite-word case, we use Lemma 3.3.8 to compute $\lambda\,\langle s, o\rangle : \mathsf{Enc}_{\mathsf{MH}}\big(\bigcup_{v\subseteq\mathbb{P}} \mathsf{post}_v(\{\langle s, o\rangle\})\big)$ instead.

The function PostMH, depicted in Algorithm 4.3, performs the computation $\lambda\,\langle s, o\rangle : \mathsf{Enc}_{\mathsf{MH}}\big(\bigcup_{v\subseteq\mathbb{P}} \mathsf{post}_v(\{\langle s, o\rangle\})\big)$ on the Miyano-Hayashi construction of sABA. Remember from Definition 3.5.8 that for every $\langle s, o\rangle$ such that $o \neq \varnothing$ we have that:

$$
\begin{aligned}
&\mathsf{post}_{\mathsf{MH}}(\{\langle s, \varnothing\rangle\}) = \{\langle s', s' \setminus F\rangle \;\mid\; \exists\, v \subseteq \mathbb{P} : s \overset{v}{\rightsquigarrow} s'\} \\
&\mathsf{post}_{\mathsf{MH}}(\{\langle s,\, o\,\rangle\}) = \{\langle s', o'' \setminus F\rangle \;\mid\; \exists\, o'' \subseteq Q, v \subseteq \mathbb{P} : \big(o'' \subseteq s' \wedge s \overset{v}{\rightsquigarrow} s' \wedge o \overset{v}{\rightsquigarrow} o''\big)\}
\end{aligned}
$$

The function PostMH distinguishes two cases. If $o = \varnothing$, then it computes to set of successors s' of s with the ROBDD $\bigwedge_{q_i \in s} \Delta_x(q_i)$ (denoted $\Delta_x(s)$ in the algorithm), and constrains the variable in $\mathbb{Y}$ such that $o' = s' \setminus F$. The case of $o \neq \varnothing$ is a bit more involved, and requires to compute the successors s' of s and o'' of o separately, and then removing the accepting states from o'' to obtain o'. This latter part is achieved by using a *primed copy* $\mathbb{Y}' =$

Algorithm 4.3: Computation of $\mathsf{Enc}_{\mathsf{MH}}(\mathsf{post}(\{\langle s, o\rangle\}))$

Input: a sABA $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$, and $\langle s, o\rangle \in 2^Q \times 2^Q$ with $o \subseteq s \setminus F$

Notation: $\forall\, s \subseteq Q : \Delta_x(s) \stackrel{\mathsf{def}}{=} \bigwedge_{q\in s} \Delta_x(q)$ and $\Delta_{y'}(s) \stackrel{\mathsf{def}}{=} \bigwedge_{q\in s} \Delta_{y'}(q)$

Output: $\mathsf{Enc}_{\mathsf{MH}}(\mathsf{post}_{\mathsf{MH}}(\{\langle s, o\rangle\}))$

```
1 begin PostMH(⟨s, o⟩)
2     if o ≠ ∅ then
3         d ← ∃ 𝕐′ : ( ∃ ℙ : ( ⋀_{q_i∈Q}(y′_i ⇒ x_i) ∧ Δ_x(s) ∧ Δ_{y′}(o) )
                       ( ∧ ⋀_{q_i∈F} ¬y_i ∧ ⋀_{q_i∈F̄} (y_i ⇔ y′_i) ) ) ;
4     else
5         d ← ∃ ℙ : ⋀_{q_i∈F} ¬y_i ∧ Δ_x(s) ∧ ⋀_{q∈F̄} (y_i ⇔ x_i) ;
6     return d ;
7 end
```

$\{y'_1, \ldots, y'_k\}$ of the propositions in $\mathbb{Y}$. Note also that Algorithm 4.3 uses the encoding $\Delta_{y'}$ of δ, which is simply equivalent to Δ_y with every occurrence of $y_i \in \mathbb{Y}$ replaced by its primed equivalent in $\mathbb{Y}'$.

Lemma 4.5.3 (Soundness of PostMH)**.** *For any* sABA $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$*, for any state* $\langle s, o\rangle \in 2^{(2^Q \times 2^Q)}$ *of* $\mathsf{MH}(A)$ *we have that* $\mathsf{PostMH}(\langle s, o\rangle) = \mathsf{Enc}_{\mathsf{MH}}(\mathsf{post}_{\mathsf{MH}}(\{\langle s, o\rangle\}))$.

Proof. We begin with the easier case where $o = \varnothing$:

$$
\begin{aligned}
&\mathsf{Dec}_{\mathsf{MH}}\big(\exists\, \mathbb{P} : \bigwedge_{q_i\in F} \neg y_i \wedge \Delta_x(s) \wedge \bigwedge_{q\in\overline{F}} (y_i \Leftrightarrow x_i)\big) \\
&= \{\langle s', o'\rangle \;\mid\; \exists\, v \subseteq \mathbb{P} : \big(o' \cap F = \varnothing\big) \wedge \big(s' \cup v \models \bigwedge_{q\in s} \delta(q)\big) \wedge \big(s' \setminus F = o' \setminus F\big)\} \\
&= \{\langle s', o'\rangle \;\mid\; \exists\, v \subseteq \mathbb{P} : \big(s' \cup v \models \bigwedge_{q\in s} \delta(q)\big) \wedge \big(o' = s' \setminus F\big)\} \\
&= \{\langle s', s' \setminus F\rangle \;\mid\; \exists\, v \subseteq \mathbb{P} : s' \cup v \models \bigwedge_{q\in s} \delta(q)\} \\
&= \mathsf{post}_{\mathsf{MH}}(\{\langle s, \varnothing\rangle\})
\end{aligned}
$$

Finally, we turn to the case where $o \neq \varnothing$.

$$
\begin{aligned}
&\mathsf{Dec}_{\mathsf{MH}}\left(\exists\,\mathbb{Y}' : \left(\begin{array}{l} \exists\,\mathbb{P} : \left(\bigwedge_{q_i\in Q}(y'_i \Rightarrow x_i) \wedge \Delta_x(s) \wedge \Delta_{y'}(o)\right) \\ \wedge \bigwedge_{q_i\in F} \neg y_i \wedge \bigwedge_{q_i\in \overline{F}} (y_i \Leftrightarrow y'_i) \end{array}\right)\right) \\
&= \{\langle s', o'\rangle \;\mid\; \exists\, o'' \subseteq Q : \left(\begin{array}{l} \exists\, v \subseteq \mathbb{P} : \left(o'' \subseteq s' \wedge s \overset{v}{\rightsquigarrow} s' \wedge o \overset{v}{\rightsquigarrow} o''\right) \\ \wedge\, o' \cap F = \varnothing \wedge o'' \setminus F = o' \setminus F \end{array}\right)\} \\
&= \{\langle s', o'\rangle \;\mid\; \exists\, o'' \subseteq Q : \left(\begin{array}{l} \exists\, v \subseteq \mathbb{P} : \left(o'' \subseteq s' \wedge s \overset{v}{\rightsquigarrow} s' \wedge o \overset{v}{\rightsquigarrow} o''\right) \\ \wedge\, o' = o'' \setminus F \end{array}\right)\} \\
&= \{\langle s', o'' \setminus F\rangle \;\mid\; \exists\, o'' \subseteq Q, v \subseteq \mathbb{P} : \left(o'' \subseteq s' \wedge s \overset{v}{\rightsquigarrow} s' \wedge o \overset{v}{\rightsquigarrow} o''\right)\} \\
&= \mathsf{post}_{\mathsf{MH}}(\{\langle s, o\rangle\})
\end{aligned}
$$

□

Finally, our last semi-symbolic pre operator performs the computation $\lambda\,\langle s', o'\rangle : \mathsf{Enc}_{\mathsf{MH}}\big(\bigcup_{v\subseteq\mathbb{P}} \mathsf{pre}_v(\downarrow\{\langle s', o'\rangle\})\big)$ on the Miyano-Hayashi construction of symbolic alternating Büchi automata.

Algorithm 4.4: Computation of $\mathsf{Enc}_{\mathsf{MH}}(\mathsf{pre}_{\mathsf{MH}}(\downarrow\{\langle s', o'\rangle\}))$

Input: a sABA $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$ and $\langle s', o'\rangle \in 2^Q \times 2^Q$ with $o \subseteq s \setminus F$

Output: $\mathsf{Enc}_{\mathsf{MH}}(\mathsf{pre}_{\mathsf{MH}}(\downarrow\{\langle s', o'\rangle\}))$ $\quad(\preceq_{\mathsf{MH}})$

```
1  begin PreMH(⟨s', o'⟩)
2      for q_i ∈ Q do
3          Ψ(q_i) ← ∃ 𝕏 : Δ_x(q_i) ∧ ⋀_{q_j∈Q}(x_j ⇔ q_j ∈ s') ;
4          Φ(q_i) ← ∃ 𝕐 : Δ_y(q_i) ∧ ⋀_{q_j∈Q}(y_j ⇔ q_j ∈ o' ∪ (s' ∩ F)) ;
5      d_1 ← ⋀_{q_i∈Q} (x_i ⇒ Φ(q_i)) ∧ ¬y_i ;
6      d_2 ← ⋀_{q_i∈Q} (x_i ⇒ Ψ(q_i)) ∧ (y_i ⇒ Φ(q_i)) ;
7      d_3 ← ⋁_{q_i∈Q} Φ(q_i) ;
8      d_4 ← ⋁_{q_i∈Q} y_i ;
9      return ∃ ℙ : d_1 ∨ (d_2 ∧ d_3 ∧ d_4) ;
10 end
```

The function PreMH, depicted in Algorithm 4.4, performs the computation $\lambda \langle s', o' \rangle : \mathsf{Enc}_{\mathsf{MH}}\big(\bigcup_{v \subseteq \mathbb{P}} \mathsf{pre}_v(\downarrow\{\langle s', o' \rangle\})\big)$ on the Miyano-Hayashi construction of sABA.

Lemma 4.5.4 (Soundness of PreMH)**.** *For any* sABA $A = \langle Q, \mathbb{P}, q_0, \delta, F \rangle$*, for any state* $\langle s', o' \rangle \in 2^{(2^Q \times 2^Q)}$ *of* $\mathsf{MH}(A)$ *we have that* $\mathsf{PreMH}(\langle s', o' \rangle) = \mathsf{Enc}_{\mathsf{MH}}(\mathsf{pre}_{\mathsf{MH}}(\downarrow\{\langle s', o' \rangle\}))$.

Proof. First, it is easy to see that for every $q \in Q$ we have that:

$$\mathsf{Sat}(\Psi(q)) = \{v \subseteq \mathbb{P} \mid s' \cup v \models \delta(q)\}$$
$$\mathsf{Sat}(\Phi(q)) = \{v \subseteq \mathbb{P} \mid o' \cup (s' \cap F) \cup v \models \delta(q)\}$$

From here, we can proceed with the remainder of the algorithm. To alleviate the notations, we let $\alpha(v) \stackrel{\mathsf{def}}{=} \{q \in Q \mid v \models \Psi(q)\}$ and $\beta(v) \stackrel{\mathsf{def}}{=} \{q \in Q \mid v \models \Phi(q)\}$.

$$\begin{aligned}
&\exists\, \mathbb{P} : d_1 \vee (d_2 \wedge d_3) \\
&= \bigcup_{v \subseteq \mathbb{P}} \left\{ \langle s, o \rangle \;\middle|\; \begin{array}{l} \big(\forall\, q \in Q : (q \notin s \vee v \models \Phi(q)) \wedge o = \varnothing\big) \vee \\ \big(\forall\, q \in Q : (q \notin s \vee v \models \Psi(q)) \wedge (q \notin o \vee v \models \Phi(q)) \wedge \\ \quad \exists\, q' \in Q : v \models \Phi(q') \wedge o \neq \varnothing\big) \end{array} \right\} \\
&= \bigcup_{v \subseteq \mathbb{P}} \{\langle o, \varnothing \rangle \mid o \subseteq \beta(v)\} \cup \{\langle s, o \neq \varnothing \rangle \mid s \subseteq \alpha(v) \wedge o \subseteq \beta(v) \wedge \beta(v) \neq \varnothing\} \\
&= \bigcup_{v \subseteq \mathbb{P}} \downarrow\{\langle \beta(v), \varnothing \rangle\} \cup \big(\text{ if } \beta(v) \neq \varnothing \text{ then } \downarrow\{\langle \alpha(v), \beta(v) \rangle\} \text{ else } \varnothing\big) && (\preceq_{\mathsf{MH}}) \\
&= \downarrow \bigcup_{v \subseteq \mathbb{P}} \text{ if } \beta(v) \neq \varnothing \text{ then } \{\langle \alpha(v), \beta(v) \rangle\} \text{ else } \{\langle \varnothing, \varnothing \rangle\} && (\preceq_{\mathsf{MH}}) \\
&= \downarrow \bigcup_{v \subseteq \mathbb{P}} \lceil \mathsf{pre}_v^{\mathsf{MH}}(\downarrow\{\langle s', o' \rangle\}) \rceil && (\preceq_{\mathsf{MH}}) \\
&= \mathsf{pre}_{\mathsf{MH}}(\downarrow\{\langle s', o' \rangle\}) && (\preceq_{\mathsf{MH}})
\end{aligned}$$

□

4.6 Converting ROBDD to Antichains

In this section, we provide algorithms to translate ROBDD-encoded upward- or downward-closed sets, with respect to either $\subseteq$ or $\preceq_{\mathsf{MH}}$, into antichains.

For each of the four possible translations, we provide two distinct algorithms, so this section contains eight algorithms in total.

The first class of algorithms recursively traverse the ROBDD and compute the resulting antichain iteratively by starting from the ROBDD leaves and proceeding upwards. The second class of algorithms encode the simulation preorder (either $\subseteq$ or $\preceq_{\mathsf{MH}}$) over two copies of the variables and use existential quantification to compute an ROBDD which contains only the maximal or minimal elements.

4.6.1 Recursive Conversion of $\subseteq$- or $\supseteq$-Closed Sets

In order to convert ROBDD-encoded $\subseteq$-downward-closed sets into $\subseteq$-maximal antichains, or $\supseteq$-upward-closed sets into $\supseteq$-minimal antichains, we make use of two intrinsic properties of ROBDD-encoded $\subseteq$-closed sets. The two cases of $\supseteq$-upward-closed sets and $\subseteq$-downward-closed sets are duals of each other, so we begin with the translation of $\supseteq$-upward-closed sets into $\supseteq$-minimal antichains, and proceed to the dual translation afterward.

First, we show that the root node of an ROBDD-encoded $\supseteq$-upward-closed set enjoys the following useful *root property*: if $d \in \mathsf{ROBDD}(\mathbb{X})$ encodes an $\supseteq$-upward-closed set, then $[\![d]\!] \stackrel{\mathsf{def}}{=} [\![\big(\mathsf{lo}(d) \wedge \neg\mathsf{prop}(d)\big) \vee \big(\mathsf{hi}(d) \wedge \mathsf{prop}(d)\big)]\!]$ is such that $[\![d]\!] = [\![\mathsf{lo}(d) \vee \big(\mathsf{hi}(d) \wedge \mathsf{prop}(d)\big)]\!]$.

Lemma 4.6.1 (Root property of $\supseteq$-upward-closed ROBDD)**.**
For any $d \in \mathsf{ROBDD}(\mathbb{X})$, such that $d \notin \{\mathsf{true}, \mathsf{false}\}$ and $\mathsf{Dec}_{\mathsf{SC}}(d)$ is $\supseteq$-upward-closed we have that:

$$[\![d]\!] = [\![\mathsf{lo}(d) \vee \big(\mathsf{hi}(d) \wedge \mathsf{prop}(d)\big)]\!]$$

Proof. We must show that, under the Lemma's hypotheses, we have that:

$$\underbrace{[\![\big(\mathsf{lo}(d) \wedge \neg\mathsf{prop}(d)\big) \vee \big(\mathsf{hi}(d) \wedge \mathsf{prop}(d)\big)]\!]}_{\alpha} = \underbrace{[\![\mathsf{lo}(d) \vee \big(\mathsf{hi}(d) \wedge \mathsf{prop}(d)\big)]\!]}_{\beta}$$

Since $\mathsf{Dec}_{\mathsf{SC}}(d)$ is $\supseteq$-upward-closed we know that:

$$\forall\, v \subseteq \mathbb{X} : \text{if } v \setminus \{\mathsf{prop}(d)\} \models d \text{ then } v \cup \{\mathsf{prop}(d)\} \models d$$

Since neither $\mathsf{lo}(d)$ nor $\mathsf{hi}(d)$ depend on $\mathsf{prop}(d)$ we have that:

$$\forall\, v \subseteq \mathbb{X} : v \setminus \{\mathsf{prop}(d)\} \models d \;\text{ iff }\; v \models \mathsf{lo}(d)$$
$$\forall\, v \subseteq \mathbb{X} : v \cup \{\mathsf{prop}(d)\} \models d \;\text{ iff }\; v \models \mathsf{hi}(d)$$

From the statements above, we can conclude the following:

$$\forall\, v \subseteq \mathbb{X} : \text{if } v \models \mathsf{lo}(d) \text{ then } v \models \mathsf{hi}(d)$$

We now prove that $\alpha = \beta$. We clearly have that $\forall\, v \subseteq \mathbb{X}$: if $v \models \alpha$ then $v \models \beta$. For the other direction, we distinguish two cases. First, if $v \models \beta$ and $\mathsf{prop}(d) \in v$, then either $v \models \mathsf{lo}(d)$ or $v \models \mathsf{hi}(d)$. Since both imply that $v \models \mathsf{hi}(d)$ we have that $v \models \alpha$. Second, if $v \models \beta$ and $\mathsf{prop}(d) \notin v$ then $v \not\models \mathsf{hi}(d) \wedge \mathsf{prop}(d)$, which implies that $v \models \mathsf{lo}(d) \wedge \neg\mathsf{prop}(d)$ and therefore $v \models \alpha$. □

Now we show that ROBDD-encoded $\supseteq$-upward-closed sets enjoy the following useful *inductive property*: every node of an ROBDD-encoded $\supseteq$-upward-closed is also an ROBDD-encoded $\supseteq$-upward-closed set.

Lemma 4.6.2 (Inductive property of $\supseteq$-upward-closed ROBDD)**.**
For any $d \in \mathsf{ROBDD}(\mathbb{X})$ we have that $\mathsf{Dec}_{\mathsf{SC}}(d)$ is $\supseteq$-upward-closed if and only if for each node $n \in \mathsf{nodes}(d)$ we have that $\mathsf{Dec}_{\mathsf{SC}}(n)$ is $\supseteq$-upward-closed.

Proof. The "if" part of the Lemma is easy, so we prove the "only if" direction. By contradiction, let us suppose there exists $n \in \mathsf{nodes}(d)$, $v \subseteq \mathbb{X}$ and $x_i \in \mathbb{X}$ such that $v \models n$ and $v \cup \{x_i\} \not\models n$. From this supposition, let us show that there exists $v' \subseteq \mathbb{X}$ such that $v' \models d$ and $v' \cup \{x_i\} \not\models d$, thus contradicting the hypothesis that $\mathsf{Dec}_{\mathsf{SC}}(d)$ is $\supseteq$-upward-closed.

Since $n \in \mathsf{nodes}(d)$, we know that there exists $v_n \subseteq \mathbb{X}$ such that, by "reading" v_n from d by following low and high children appropriately, n is eventually reached. Let $v' = (v_n \setminus \mathsf{Dep}(n)) \cup (v \cap \mathsf{Dep}(n))$. Since $x_i \in \mathsf{Dep}(n)$ and since ROBDD are deterministic we have that, by "reading" either v' or $v' \cup \{x_i\}$ from d by following low and high children appropriately, n is eventually reached. Therefore, since $v' \models n$ and $v' \cup \{x_i\} \not\models n$, we have that $v' \models d$ and $v' \cup \{x_i\} \not\models d$. □

Algorithm 4.5: Recursive computation of $\lfloor \mathsf{Dec}_{\mathsf{SC}}(d) \rfloor$

Input: a BDD $d \in \mathsf{ROBDD}(\mathbb{X})$ such that $\mathsf{Dec}_{\mathsf{SC}}(d)$ is $\supseteq$-upward-closed

Output: the antichain $\lfloor \mathsf{Dec}_{\mathsf{SC}}(d) \rfloor$

```
1 begin MinDecSCrec(d)
2     if d = true then return {∅} ;
3     if d = false then return ∅ ;
4     let x_i = prop(d) ;
5     L ← MinDecSCrec(lo(d)) ;
6     H ← MinDecSCrec(hi(d)) ⊓ {{q_i}} ;
7     return L ⊔ H ;
8 end
```

Equipped with our root property and inductive property of ROBDD-encoded $\supseteq$-upward-closed sets, we can develop a recursive algorithm to compute the translation to $\supseteq$-minimal antichains (see Algorithm 4.5). The base cases $d = \mathsf{true}$ and $d = \mathsf{false}$ are first handled by returning the antichains $\{\varnothing\}$ or $\varnothing$, respectively (lines 2-3). Thanks to Lemma 4.6.1 and Lemma 4.6.2 the recursive case is easy; we know that $[\![d]\!] = [\![\mathsf{lo}(d) \vee \big(\mathsf{hi}(d) \wedge \mathsf{prop}(d)\big)]\!]$, so we have that $\mathsf{Dec}_{\mathsf{SC}}(d) = \mathsf{Dec}_{\mathsf{SC}}(\mathsf{lo}(d)) \cup \big(\mathsf{Dec}_{\mathsf{SC}}(\mathsf{hi}(d)) \cap \mathsf{Dec}_{\mathsf{SC}}(\mathsf{prop}(d))\big)$. Since we are interested in the set of minimal elements we have that:

$$\begin{aligned} &\lfloor \mathsf{Dec}_{\mathsf{SC}}(\mathsf{lo}(d)) \cup \big(\mathsf{Dec}_{\mathsf{SC}}(\mathsf{hi}(d)) \cap \mathsf{Dec}_{\mathsf{SC}}(\mathsf{prop}(d))\big) \rfloor \\ &= \lfloor \mathsf{Dec}_{\mathsf{SC}}(\mathsf{lo}(d)) \rfloor \sqcup \big(\lfloor \mathsf{Dec}_{\mathsf{SC}}(\mathsf{hi}(d)) \rfloor \sqcap \lfloor \mathsf{Dec}_{\mathsf{SC}}(\mathsf{prop}(d)) \rfloor \big) \end{aligned}$$

Both $\lfloor \mathsf{Dec}_{\mathsf{SC}}(\mathsf{lo}(d)) \rfloor$ and $\lfloor \mathsf{Dec}_{\mathsf{SC}}(\mathsf{hi}(d)) \rfloor$ are computed by a recursive call at lines 5-6, and it is easy to see that $\lfloor \mathsf{Dec}_{\mathsf{SC}}(\mathsf{prop}(d)) \rfloor = \{\{q_i\}\}$ when $\mathsf{prop}(d) = x_i$. Note that this algorithm should be implemented using memoization in order to make its best-case running time linear in the number of ROBDD nodes of d. In order to alleviate the algorithms, the memoization code has been omitted. Note also that even with memoization, the resulting antichain is exponentially larger than the input ROBDD in the worst case.

Lemma 4.6.3 (Soundness of MinDecSCrec)**.** *Let $A = \langle Q, \mathbb{P}, \delta, q_0, F \rangle$ be a* sAFA *such that $Q = \{q_1, \ldots, q_k\}$, and let $\mathbb{X} = \{x_1, \ldots, x_k\}$. For*

any $d \in \mathsf{ROBDD}(\mathbb{X})$ *such that* $\mathsf{Dec_{SC}}(d)$ *is* $\supseteq$*-upward-closed, we have that* $\mathsf{MinDecSCrec}(d) = \lfloor \mathsf{Dec_{SC}}(d) \rfloor$.

Proof. We proceed by induction on the height of the the DAG of d. The Lemma is trivially true for the two base cases $d \in \{\mathsf{true}, \mathsf{false}\}$. For the inductive case, by the induction hypothesis and by Lemma 4.6.2 we know that $L = \lfloor \mathsf{Dec_{SC}}(\mathsf{lo}(d)) \rfloor$ and $H = \lfloor \mathsf{Dec_{SC}}(\mathsf{hi}(d)) \rfloor \sqcap \{\{q_i\}\}$. Finally, by Lemma 4.6.1 we know that:

$$\begin{aligned}
\lfloor \mathsf{Dec_{SC}}(d) \rfloor &= \lfloor \mathsf{Dec_{SC}}(\mathsf{lo}(d) \vee (\mathsf{hi}(d) \wedge x_i)) \rfloor \\
&= \big\lfloor \mathsf{Dec_{SC}}(\mathsf{lo}(d)) \cup \big(\mathsf{Dec_{SC}}(\mathsf{hi}(d)) \cap \mathsf{Dec_{SC}}(x_i)\big) \big\rfloor \\
&= \lfloor \mathsf{Dec_{SC}}(\mathsf{lo}(d)) \rfloor \sqcup \big(\lfloor \mathsf{Dec_{SC}}(\mathsf{hi}(d)) \rfloor \sqcap \lfloor \mathsf{Dec_{SC}}(x_i) \rfloor \big) \\
&= L \sqcup \big(\lfloor \mathsf{Dec_{SC}}(\mathsf{hi}(d)) \rfloor \sqcap \{\{q_i\}\} \big) \\
&= L \sqcup H
\end{aligned}$$

□

In the remainder of this subsection, we provide the dual results for the conversion of $\subseteq$-downward-closed sets. Due to the strong similarity with the results above for $\supseteq$-upward-closed sets, we omit the corresponding proofs.

Lemma 4.6.4 (Root property of $\subseteq$-downward-closed ROBDD)**.** *For any* $d \in \mathsf{ROBDD}(\mathbb{X})$*, such that* $d \notin \{\mathsf{true}, \mathsf{false}\}$ *and* $\mathsf{Dec_{SC}}(d)$ *is* $\subseteq$*-downward-closed we have that:*

$$[\![d]\!] = [\![\big(\mathsf{lo}(d) \wedge \neg \mathsf{prop}(d)\big) \vee \mathsf{hi}(d)]\!]$$

Proof. Similar to the proof of Lemma 4.6.1. □

Lemma 4.6.5 (Inductive property of $\subseteq$-downward-closed ROBDD)**.** *For any* $d \in \mathsf{ROBDD}(\mathbb{X})$ *we have that* $\mathsf{Dec_{SC}}(d)$ *is* $\subseteq$*-downward-closed iff for each node* $n \in \mathsf{nodes}(d)$ *we have that* $\mathsf{Dec_{SC}}(n)$ *is* $\subseteq$*-downward-closed.*

Proof. Similar to the proof of Lemma 4.6.2. □

Lemma 4.6.6 (Soundness of MaxDecSCrec)**.** *Let* $A = \langle Q, \mathbb{P}, \delta, q_0, F \rangle$ *be a* sAFA *such that* $Q = \{q_1, \ldots, q_k\}$ *and let* $\mathbb{X} = \{x_1, \ldots, x_k\}$*. For any* $d \in \mathsf{ROBDD}(\mathbb{X})$ *such that* $\mathsf{Dec_{SC}}(d)$ *is* $\subseteq$*-downward-closed, we have that* $\mathsf{MaxDecSCrec}(d) = \lceil \mathsf{Dec_{SC}}(d) \rceil$.

Algorithm 4.6: Recursive computation of $\lceil \mathsf{Dec}_{\mathsf{SC}}(d) \rceil$

Input: $d \in \mathsf{ROBDD}(\mathbb{X})$ such that $\mathsf{Dec}_{\mathsf{SC}}(d)$ is $\subseteq$-downward-closed
Output: the antichain $\lceil \mathsf{Dec}_{\mathsf{SC}}(d) \rceil$

```
1 begin MaxDecSCrec(d)
2     if d = true then return {Q} ;
3     if d = false then return ∅ ;
4     let x_i = prop(d) ;
5     L ← MaxDecSCrec(lo(d)) ⊓ {Q \ {q_i}} ;
6     H ← MaxDecSCrec(hi(d)) ;
7     return L ⊔ H ;
8 end
```

Proof. Similar to the proof of Lemma 4.6.3, by using Lemma 4.6.5 and Lemma 4.6.4. □

4.6.2 Symbolic Conversion of $\subseteq$- or $\supseteq$-Closed Sets

Instead of recursively traversing the ROBDD, the conversion to antichains can be performed by using an encoding of the partial order over two copies of the state variables, and ROBDD existential quantification. First, the input ROBDD has its variables renamed to primed variables. Next, a strict variant of the partial order is computed so that it encodes the set of pairs of states in which the unprimed state is strictly smaller (or larger) state than the primed state with respect to the partial order. This ROBDD-encoded strict partial order is then conjuncted with the renamed input ROBDD and the result is then existentially quantified over the primed variables. The resulting ROBDD thus contains exactly the set of unprimed states such that there exists at least one strictly smaller (or larger) in the ROBDD-encoded input set. Finally, the algorithm returns the set difference (using a conjunction and a negation) of the original input ROBDD and the previously computed ROBDD. Since this algorithm uses ROBDD exclusively to perform the translation to antichains, we refer to it as *symbolic*.

The procedure depicted in Algorithm 4.7 translates ROBDD-encoded $\supseteq$-upward-closed sets to $\supseteq$-minimal antichains. It is easy to see that the

Algorithm 4.7: Symbolic computation of $\lfloor \mathsf{Dec_{SC}}(d) \rfloor$

Input: a BDD $d \in \mathsf{ROBDD}(\mathbb{X})$ such that $\mathsf{Dec_{SC}}(d)$ is $\supseteq$-upward-closed

Output: the antichain $\lfloor \mathsf{Dec_{SC}}(d) \rfloor$

```
1 begin MinDecSCsym(d)
2 |   d' ← d[∀ q_i ∈ Q : x_i ← x'_i] ;
3 |   return Dec_SC(d ∧ ¬∃ 𝕏' : d' ∧ ⋀_{q_i∈Q}(x'_i ⇒ x_i) ∧ ⋁_{q_i∈Q}(¬x'_i ∧ x_i));
4 end
```

ROBDD $\bigwedge_{q_i \in Q}(x'_i \Rightarrow x_i) \wedge \bigvee_{q_i \in Q}(\neg x'_i \wedge x_i)$ encodes the set of pairs of states $\{(s', s) \mid s' \subset s\}$ and is thus the strict version of $\subseteq$. Finally, the surrounding $\mathsf{Dec_{SC}}$ computation is carried out simply by enumerating the valuations of the computed ROBDD.

Lemma 4.6.7 (Soundness of MinDecSCsym)**.** *Let* $A = \langle Q, \mathbb{P}, \delta, q_0, F \rangle$ *be a* sAFA *with* $Q = \{q_1, \ldots, q_k\}$ *and let* $\mathbb{X} = \{x_1, \ldots, x_k\}$*. For any* $d \in \mathsf{ROBDD}(\mathbb{X})$ *such that* $\mathsf{Dec_{SC}}(d)$ *is* $\supseteq$*-upward-closed, we have* $\mathsf{MinDecSCsym}(d) = \lfloor \mathsf{Dec_{SC}}(d) \rfloor$*.*

Proof. The one-line ROBDD computation can be unfolded as follows:

$$
\begin{aligned}
&\mathsf{Dec_{SC}}\big(d \wedge \neg\exists\, \mathbb{X}' : d' \wedge \bigwedge_{q_i \in Q} (x'_i \Rightarrow x_i) \wedge \bigvee_{q_i \in Q} (\neg x'_i \wedge x_i)\big) \\
&= \mathsf{Dec_{SC}}(d) \setminus \big\{ s \;\big|\; \exists\, s' \subseteq Q : s' \in \mathsf{Dec_{SC}}\big(d\big) \wedge s' \subseteq s \wedge \exists\, q_i \in Q : q_i \notin s' \wedge q_i \in s \big\} \\
&= \mathsf{Dec_{SC}}(d) \setminus \big\{ s \;\big|\; \exists\, s' \subseteq Q : s' \in \mathsf{Dec_{SC}}\big(d\big) \wedge s' \subset s \big\} \\
&= \mathsf{Dec_{SC}}(d) \setminus \{ s \mid \exists\, s' \in \mathsf{Dec_{SC}}(d) : s' \subset s \} \\
&= \lfloor \mathsf{Dec_{SC}}(d) \rfloor \quad (\supseteq)
\end{aligned}
$$

□

The dual computation of translating $\subseteq$-downward-closed sets to $\subseteq$ -maximal antichains is performed as expected (see Algorithm 4.8), by reversing the direction of the partial order, in order to the extract the maximal elements instead of the minimal ones.

Lemma 4.6.8 (Soundness of MaxDecSCsym)**.**

Proof. Similar to the proof of Lemma 4.6.7. □

Algorithm 4.8: Symbolic computation of $\lceil \mathsf{Dec}_{\mathsf{SC}}(d) \rceil$

Input: $d \in \mathsf{ROBDD}(\mathbb{X})$ such that $\mathsf{Dec}_{\mathsf{SC}}(d)$ is $\subseteq$-downward-closed
Output: the antichain $\lceil \mathsf{Dec}_{\mathsf{SC}}(d) \rceil$

```
1 begin MaxDecSCsym(d)
2 |  d' ← d[∀ q_i ∈ Q : x_i ← x'_i] ;
3 |  return Dec_SC(d ∧ ¬∃ X' : d' ∧ ⋀_{q_i∈Q}(x_i ⇒ x'_i) ∧ ⋁_{q_i∈Q}(¬x_i ∧ x'_i));
4 end
```

4.6.3 Conversion of $\preceq_{\mathsf{MH}}$- or $\succeq_{\mathsf{MH}}$-Closed Sets

The conversion of $\succeq_{\mathsf{MH}}$-upward-closed or $\preceq_{\mathsf{MH}}$-downward-closed sets to $\succeq_{\mathsf{MH}}$-minimal or $\preceq_{\mathsf{MH}}$-maximal antichains is not as straightforward as the case for subset inclusion. The problem is that ROBDD-encoded $\succeq_{\mathsf{MH}}$-upward-closed or $\preceq_{\mathsf{MH}}$-downward-closed sets do not enjoy the required inductive property: their descendant nodes do not always themselves encode $\succeq_{\mathsf{MH}}$-upward-closed or $\preceq_{\mathsf{MH}}$-downward-closed sets. Indeed, the requirement that $\langle s, o\rangle \preceq_{\mathsf{MH}} \langle s', o'\rangle$ only if $o = \varnothing \Leftrightarrow o' = \varnothing$ is not "local" to any proposition in $\mathbb{Y}$.

To circumvent this obstacle, we use the *double inclusion* partial order, denoted $\langle s, o\rangle \subseteq\!\subseteq \langle s', o'\rangle \overset{\text{def}}{\Leftrightarrow} s \subseteq s'$ and $o \subseteq o'$. It is easy to see that $\subseteq\!\subseteq$-downward-closed and $\supseteq\!\supseteq$-upward-closed sets enjoy both the root and inductive properties defined earlier for (single) subset inclusion. Therefore, it remains to show how to compute the set of $\succeq_{\mathsf{MH}}$-minimal or $\preceq_{\mathsf{MH}}$-maximal elements of an ROBDD by a reduction to $\supseteq\!\supseteq$-maximization or $\subseteq\!\subseteq$-minimization. This reduction is relatively simple and amounts to deal with accepting ($o = \varnothing$) and non-accepting ($o \neq \varnothing$) states separately, as show by the following two lemmas.

Lemma 4.6.9. *Let $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$ be a* sABA *with $Q = \{q_1, \ldots, q_k\}$ encoded over $\mathbb{X} = \{x_1, \ldots, x_k\}$ and $\mathbb{Y} = \{y_1, \ldots, y_k\}$. For every* ROBDD *$d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ such that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\preceq_{\mathsf{MH}}$-downward-closed we have that $\mathsf{Dec}_{\mathsf{MH}}(d \wedge \bigwedge_{q_i \in Q} \neg y_i)$ and $\mathsf{Dec}_{\mathsf{MH}}(d \vee \bigwedge_{q_i \in Q} \neg y_i)$ are $\subseteq\!\subseteq$-downward-*

closed. Furthermore, we have that:

$$\lceil \mathsf{Dec}_{\mathsf{MH}}(d) \rceil = X \sqcup (Y \sqcap \{\langle Q, Q \rangle\}) \qquad (\preceq_{\mathsf{MH}})$$

$$\textit{with } X = \left\lceil \mathsf{Dec}_{\mathsf{MH}}(d \wedge \bigwedge_{q_i \in Q} \neg y_i) \right\rceil \qquad (\subseteq\subseteq)$$

$$Y = \left\lceil \mathsf{Dec}_{\mathsf{MH}}(d \vee \bigwedge_{q_i \in Q} \neg y_i) \right\rceil \qquad (\subseteq\subseteq)$$

Proof. We first show that $\mathsf{Dec}_{\mathsf{MH}}(d \wedge \bigwedge_{q_i \in Q} \neg y_i)$ and $\mathsf{Dec}_{\mathsf{MH}}(d \vee \bigwedge_{q_i \in Q} \neg y_i)$ are $\subseteq\subseteq$-downward-closed sets. It is easy to see that $\mathsf{Dec}_{\mathsf{MH}}(d \wedge \bigwedge_{q_i \in Q} \neg y_i) = \{\langle s, \varnothing \rangle \mid \langle s, o \rangle \in \mathsf{Dec}_{\mathsf{MH}}(d)\}$, and since $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\preceq_{\mathsf{MH}}$-downward-closed, we can conclude that $\mathsf{Dec}_{\mathsf{MH}}(d \wedge \bigwedge_{q_i \in Q} \neg y_i)$ is $\subseteq\subseteq$-downward-closed. Next, we have that $\mathsf{Dec}_{\mathsf{MH}}(d \vee \bigwedge_{q_i \in Q} \neg y_i) = \mathsf{Dec}_{\mathsf{MH}}(d) \cup \{\langle s, \varnothing \rangle \mid s \subseteq Q\}$. Since $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\preceq_{\mathsf{MH}}$-downward-closed, it is easy to see that the set $\mathsf{Dec}_{\mathsf{MH}}(d) \cup \{\langle s, \varnothing \rangle \mid \langle s, o \rangle \in \mathsf{Dec}_{\mathsf{MH}}(d)\}$ is $\subseteq\subseteq$-downward-closed. Clearly, $\{\langle s, \varnothing \rangle \mid s \subseteq Q\}$ is $\subseteq\subseteq$-downward-closed, and since $\mathsf{Dec}_{\mathsf{MH}}(d) \cup \{\langle s, \varnothing \rangle \mid \langle s, o \rangle \in \mathsf{Dec}_{\mathsf{MH}}(d)\} \cup \{\langle s, \varnothing \rangle \mid s \subseteq Q\} = \mathsf{Dec}_{\mathsf{MH}}(d) \cup \{\langle s, \varnothing \rangle \mid s \subseteq Q\} = \mathsf{Dec}_{\mathsf{MH}}(d \vee \bigvee_{q_i \in Q} \neg y_i)$ we can conclude that $\mathsf{Dec}_{\mathsf{MH}}(d \vee \bigvee_{q_i \in Q} \neg y_i)$ is $\subseteq\subseteq$-downward-closed.

We now show that $\lceil \mathsf{Dec}_{\mathsf{MH}}(d) \rceil = X \sqcup (Y \sqcap \{\langle Q, Q \rangle\})$. First, we have that:

$$\begin{aligned} X &= \left\lceil \mathsf{Dec}_{\mathsf{MH}}(d \wedge \bigwedge_{q_i \in Q} \neg y_i) \right\rceil && (\subseteq\subseteq) \\ &= \lceil \{\langle s, \varnothing \rangle \mid \langle s, \varnothing \rangle \in \mathsf{Dec}_{\mathsf{MH}}(d)\} \rceil && (\subseteq\subseteq) \\ &= \lceil \{\langle s, \varnothing \rangle \mid \langle s, \varnothing \rangle \in \mathsf{Dec}_{\mathsf{MH}}(d)\} \rceil && (\preceq_{\mathsf{MH}}) \end{aligned}$$

Second, we have that:

$$\begin{aligned} Y &= \left\lceil \mathsf{Dec}_{\mathsf{MH}}(d \vee \bigwedge_{q_i \in Q} \neg y_i) \right\rceil && (\subseteq\subseteq) \\ &= \lceil \{\langle s, o \rangle \mid \langle s, o \rangle \in \mathsf{Dec}_{\mathsf{MH}}(d) \vee o = \varnothing\} \rceil && (\subseteq\subseteq) \\ &= \lceil \{\langle s, o \rangle \mid \langle s, o \rangle \in \mathsf{Dec}_{\mathsf{MH}}(d) \wedge o \neq \varnothing\} \cup \{\langle s, \varnothing \rangle \mid s \subseteq Q\} \rceil && (\subseteq\subseteq) \\ &= \lceil \{\langle s, o \rangle \mid \langle s, o \rangle \in \mathsf{Dec}_{\mathsf{MH}}(d) \wedge o \neq \varnothing\} \rceil \sqcup \{\langle Q, \varnothing \rangle\} && (\subseteq\subseteq) \\ &= \lceil \{\langle s, o \rangle \mid \langle s, o \rangle \in \mathsf{Dec}_{\mathsf{MH}}(d) \wedge o \neq \varnothing\} \rceil \sqcup \{\langle Q, \varnothing \rangle\} && (\preceq_{\mathsf{MH}}) \end{aligned}$$

By substituting X and Y by the values computed above we obtain:

$$\begin{aligned}
& X \sqcup (Y \sqcap \{\langle Q, Q\rangle\}) && (\preceq_{\mathsf{MH}}) \\
= {} & \lceil \{\langle s, \varnothing\rangle \mid \langle s, \varnothing\rangle \in \mathsf{Dec}_{\mathsf{MH}}(d)\} \rceil \\
& \sqcup \lceil \{\langle s, o \neq \varnothing\rangle \mid \langle s, o\rangle \in \mathsf{Dec}_{\mathsf{MH}}(d)\} \rceil && (\preceq_{\mathsf{MH}}) \\
= {} & \lceil \mathsf{Dec}_{\mathsf{MH}}(d) \rceil && (\preceq_{\mathsf{MH}})
\end{aligned}$$

□

Lemma 4.6.10. *Let $A = \langle Q, \mathbb{P}, q_0, \delta, F\rangle$ be a* sABA *with $Q = \{q_1, \ldots, q_k\}$ encoded over $\mathbb{X} = \{x_1, \ldots, x_k\}$ and $\mathbb{Y} = \{y_1, \ldots, y_k\}$. For every* ROBDD *$d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ such that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\succeq_{\mathsf{MH}}$-upward-closed we have that $\mathsf{Dec}_{\mathsf{MH}}(d \wedge \bigvee_{q_i \in Q} y_i)$ and $\mathsf{Dec}_{\mathsf{MH}}(d \vee \bigvee_{q_i \in Q} y_i)$ are both $\supseteq\supseteq$-upward-closed. Furthermore, we have that:*

$$\begin{aligned}
\lfloor \mathsf{Dec}_{\mathsf{MH}}(d) \rfloor &= X \sqcup (Y \sqcap \{\langle \varnothing, \varnothing\rangle\}) && (\succeq_{\mathsf{MH}}) \\
\text{with } X &= \left\lfloor \mathsf{Dec}_{\mathsf{MH}}\Big(d \wedge \bigvee_{q_i \in Q} y_i\Big) \right\rfloor && (\supseteq\supseteq) \\
Y &= \left\lfloor \mathsf{Dec}_{\mathsf{MH}}\Big(d \vee \bigvee_{q_i \in Q} y_i\Big) \right\rfloor && (\supseteq\supseteq)
\end{aligned}$$

Proof. Similar to the proof of Lemma 4.6.9. □

In the remainder of this section, we provide the recursive and symbolic algorithms for translating $\preceq_{\mathsf{MH}}$-closed sets into antichains. The soundness proofs of these algorithms have been omitted, as all are simple adaptations of the previous algorithms given in this chapter

Lemma 4.6.11 (Root property of $\supseteq\supseteq$-upward-closed ROBDD)**.**
For any $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$, such that $d \notin \{\mathsf{true}, \mathsf{false}\}$ and $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\supseteq\supseteq$-upward-closed we have that:

$$[\![d]\!] = [\![\mathsf{lo}(d) \vee \big(\mathsf{hi}(d) \wedge \mathsf{prop}(d)\big)]\!]$$

Lemma 4.6.12 (Root property of $\subseteq\subseteq$-downward-closed ROBDD)**.**
For any $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$, such that $d \notin \{\mathsf{true}, \mathsf{false}\}$ and $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\subseteq\subseteq$-downward-closed we have that:

$$[\![d]\!] = [\![\big(\mathsf{lo}(d) \wedge \neg\mathsf{prop}(d)\big) \vee \mathsf{hi}(d)]\!]$$

Lemma 4.6.13 (Inductive property of $\supseteq\supseteq$-upward-closed ROBDD)**.** *For any $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ we have that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\supseteq\supseteq$-upward-closed iff for each node $n \in \mathsf{nodes}(d)$ we have that $\mathsf{Dec}_{\mathsf{MH}}(n)$ is $\supseteq\supseteq$-upward-closed.*

Lemma 4.6.14 (Inductive property of $\subseteq\subseteq$-downward-closed ROBDD)**.** *For any $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ we have that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\subseteq\subseteq$-downward-closed iff for each node $n \in \mathsf{nodes}(d)$ we have that $\mathsf{Dec}_{\mathsf{MH}}(n)$ is $\subseteq\subseteq$-downward-closed.*

Algorithm 4.9: Recursive computation of $\lfloor \mathsf{Dec}_{\mathsf{MH}}(d) \rfloor$

Input: $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ such that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\supseteq\supseteq$-upward-closed

Output: the antichain $\lfloor \mathsf{Dec}_{\mathsf{MH}}(d) \rfloor \in \mathsf{Antichains}[2^Q, \supseteq\supseteq]$

```
1  begin MinDecMHrec(d)
2      if d = true then return {⟨∅, ∅⟩} ;
3      if d = false then return ∅ ;
4      if prop(d) ∈ 𝕏 then
5          let x_i = prop(d) ;
6          ⟨s, o⟩ ← ⟨{q_i}, ∅⟩ ;
7      else
8          let y_i = prop(d) ;
9          ⟨s, o⟩ ← ⟨∅, {q_i}⟩ ;
10     L ← MinDecMHrec(lo(d)) ;
11     H ← MinDecMHrec(hi(d)) ⊓ {⟨s, o⟩} ;
12     return L ⊔ H ;
13 end
```

Lemma 4.6.15 (Soundness of MinDecMHrec)**.** *Let $A = \langle Q, \mathbb{P}, \delta, q_0, F \rangle$ be a sABA with $Q = \{q_1, \ldots, q_k\}$ and let $\mathbb{X} = \{x_1, \ldots, x_k\}$, $\mathbb{Y} = \{y_1, \ldots, y_k\}$. For any $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ such that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\supseteq\supseteq$-upward-closed, we have $\mathsf{MinDecMHrec}(d) = \lfloor \mathsf{Dec}_{\mathsf{MH}}(d) \rfloor$ $(\supseteq\supseteq)$.*

Lemma 4.6.16 (Soundness of MaxDecMHrec)**.** *Let $A = \langle Q, \mathbb{P}, \delta, q_0, F \rangle$ be a sABA with $Q = \{q_1, \ldots, q_k\}$ and let $\mathbb{X} = \{x_1, \ldots, x_k\}$, $\mathbb{Y} = \{y_1, \ldots, y_k\}$.*

Algorithm 4.10: Symbolic computation of $\lfloor \mathsf{Dec}_{\mathsf{MH}}(d) \rfloor$

Input: $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ such that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\supseteq\supseteq$-upward-closed

Output: the antichain $\lfloor \mathsf{Dec}_{\mathsf{MH}}(d) \rfloor \in \mathsf{Antichains}[2^Q, \supseteq\supseteq]$

```
1 begin MinDecMHsym(d)
2 |   d' ← d[∀ q_i ∈ Q : x_i ← x'_i ∧ y_i ← y'_i] ;
3 |   o ← ⋀_{q_i∈Q}(x_i ⇒ x'_i) ∧ (y_i ⇒ y'_i) ∧ ⋁_{q_i∈Q}(¬x_i ∧ x'_i) ∨ (¬y_i ∧ y'_i) ;
4 |   return Dec_SC(d ∧ ¬∃ 𝕏' ∪ 𝕐' : d' ∧ o);
5 end
```

For any $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ such that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\subseteq\subseteq$-downward-closed, we have $\mathsf{MaxDecMHrec}(d) = \lceil \mathsf{Dec}_{\mathsf{MH}}(d) \rceil$ $(\supseteq\supseteq)$.

Lemma 4.6.17 (Soundness of MinDecMHsym)**.** *Let $A = \langle Q, \mathbb{P}, \delta, q_0, F \rangle$ be a sABA with $Q = \{q_1, \ldots, q_k\}$ and let $\mathbb{X} = \{x_1, \ldots, x_k\}$, $\mathbb{Y} = \{y_1, \ldots, y_k\}$. For any $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ such that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\supseteq\supseteq$-upward-closed, we have $\mathsf{MinDecMHsym}(d) = \lfloor \mathsf{Dec}_{\mathsf{MH}}(d) \rfloor$ $(\supseteq\supseteq)$.*

Lemma 4.6.18 (Soundness of MaxDecMHsym)**.** *Let $A = \langle Q, \mathbb{P}, \delta, q_0, F \rangle$ be a sABA with $Q = \{q_1, \ldots, q_k\}$ and let $\mathbb{X} = \{x_1, \ldots, x_k\}$, $\mathbb{Y} = \{y_1, \ldots, y_k\}$. For any $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ such that $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\subseteq\subseteq$-downward-closed, we have $\mathsf{MaxDecMHsym}(d) = \lceil \mathsf{Dec}_{\mathsf{MH}}(d) \rceil$ $(\supseteq\supseteq)$.*

4.7 LTL Model Checking With Antichains

In this section, we briefly discuss how our semi-symbolic algorithms for sABA emptiness defined in this chapter can be adapted to the case of LTL model checking. Given a propositional Kripke structure $\mathcal{K} = \langle Q, \mathbb{P}, I, \mathcal{L}, \rightarrow \rangle$ and an LTL formula $\varphi \in \mathsf{LTL}(\mathbb{P})$, the model checking problem $\mathcal{K} \models \varphi$ reduces to the emptiness question $L(\mathcal{K}) \cap L(A_{\neg\varphi}) = \varnothing$ where $A_{\neg\varphi} = \mathsf{LTL2sABA}(\neg\varphi)$.

In order to decide $L(\mathcal{K}) \cap L(A_{\neg\varphi}) = \varnothing$, we explore the state space of the finite state machine corresponding to the *synchronous product* of the Kripke structure $\mathcal{K}$ and the non-deterministic Büchi automaton $\mathsf{MH}(A)$. Let $\mathcal{K} = \langle Q^{\mathcal{K}}, \mathbb{P}, I^{\mathcal{K}}, \mathcal{L}^{\mathcal{K}}, \rightarrow_{\mathcal{K}} \rangle$ be a propositional Kripke structure and let A

Algorithm 4.11: Recursive computation of $\lceil \mathsf{Dec}_{\mathsf{MH}}(d) \rceil$

Input: $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ where $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\subseteq\subseteq$-downward-closed
Output: the antichain $\lceil \mathsf{Dec}_{\mathsf{MH}}(d) \rceil \in \mathsf{Antichains}[2^Q, \subseteq\subseteq]$

```
1  begin MaxDecMHrec(d)
2      if d = true then return {⟨Q, Q⟩} ;
3      if d = false then return ∅ ;
4      if prop(d) ∈ 𝕏 then
5          let x_i = prop(d) ;
6          ⟨s, o⟩ ← ⟨Q \ {q_i}, ∅⟩ ;
7      else
8          let y_i = prop(d) ;
9          ⟨s, o⟩ ← ⟨∅, Q \ {q_i}⟩ ;
10     L ← MaxDecMHrec(lo(d)) ⊓ {⟨s, o⟩} ;
11     H ← MaxDecMHrec(hi(d)) ;
12     return L ⊔ H ;
13 end
```

be a sABA such that $\mathsf{MH}(A) = \langle Q^{\mathsf{MH}}, 2^{\mathbb{P}}, I^{\mathsf{MH}}, \rightarrow_{\mathsf{MH}}, F^{\mathsf{MH}} \rangle$. The synchronous product of $\mathcal{K}$ and $\mathsf{MH}(A)$, denoted $\mathcal{K} \otimes \mathsf{MH}(A)$, is the finite state machine $\langle Q^{\mathcal{K}} \times Q^{\mathsf{MH}}, 2^{\mathbb{P}}, I^{\mathcal{K}} \times I^{\mathsf{MH}}, \rightarrow, Q^{\mathcal{K}} \times F^{\mathsf{MH}} \rangle$ such that:

$$(q, \langle s, o \rangle) \xrightarrow{v} (q', \langle s', o' \rangle) \text{ iff } q \rightarrow_{\mathcal{K}} q' \text{ and } \langle s, o \rangle \rightarrow^{v}_{\mathsf{MH}} \langle s', o' \rangle \text{ and } \mathcal{L}^{\mathcal{K}}(q) = v$$

Let $(q, \langle s, o \rangle) \preceq_{\mathsf{KMH}} (q', \langle s', o' \rangle)$ iff $q = q'$ and $\langle s, o \rangle \preceq_{\mathsf{MH}} \langle s', o' \rangle$. It is easy to see that $\preceq_{\mathsf{KMH}}$ and $\succeq_{\mathsf{KMH}}$ are respectively forward and backward simulation partial orders on the product $\mathcal{K} \otimes \mathsf{MH}(A)$ finite state machine. If the Kripke structure $\mathcal{K}$ is given explicitly, (e.g., by an adjacency matrix) an LTL model checking algorithm can be obtained by a simple adaptation of the LTL satisfiability algorithms given previously. However, due to the state-space explosion problem usually encountered in model checking, this is not a realistic assumption.

In the sequel, we assume that Kripke structures are *not* given explicitly, but are encoded using ROBDD over a set of Boolean propositions $\mathbb{V} = \{z_1, \ldots, z_m\}$ such that $\mathbb{P} \subseteq \mathbb{V}$ with $\mathcal{L}$ being the projection from $2^{\mathbb{V}}$ to $2^{\mathbb{P}}$.

Algorithm 4.12: Symbolic computation of $\lceil \mathsf{Dec}_{\mathsf{MH}}(d) \rceil$

Input: $d \in \mathsf{ROBDD}(\mathbb{X} \cup \mathbb{Y})$ where $\mathsf{Dec}_{\mathsf{MH}}(d)$ is $\subseteq\subseteq$-downward-closed
Output: the antichain $\lceil \mathsf{Dec}_{\mathsf{MH}}(d) \rceil \in \mathsf{Antichains}[2^Q, \subseteq\subseteq]$

```
1 begin MaxDecMHsym(d)
2     d' ← d[∀ q_i ∈ Q : x_i ← x'_i ∧ y_i ← y'_i] ;
3     o ← ⋀_{q_i∈Q}(x'_i ⇒ x_i) ∧ (y'_i ⇒ y_i) ∧ ⋁_{q_i∈Q}(¬x'_i ∧ x_i) ∨ (¬y'_i ∧ y_i) ;
4     return Dec_SC(d ∧ ¬∃ 𝕏' ∪ 𝕐' : d' ∧ o);
5 end
```

We also assume that the transition relation $\rightarrow_{\mathcal{K}}$ is encoded by an ROBDD $T^{\mathcal{K}} \in \mathsf{ROBDD}(\mathbb{V} \cup \mathbb{V}')$ such that $(q, q') \in [\![T^{\mathcal{K}}]\!]$ iff $q \rightarrow_{\mathcal{K}} q'$.

Under the hypothesis that the huge size of $Q = 2^{\mathbb{V}}$ is the main obstacle, we consider a "semi-symbolic" representation of antichains, as a set of pairs $(B, \langle s, o \rangle)$ where B is a ROBDD over $\mathbb{V}$. A pair $(B, \langle s, o \rangle)$ represents the set $\{(q, \langle s, o \rangle) \mid q \in [\![B]\!]\}$. For any $\preceq_{\mathsf{MH}}$-antichain X, its semi-symbolic representation is defined as:

$$R(X) \stackrel{\mathsf{def}}{=} \{(B, \langle s, o \rangle) \mid \exists\, (q, \langle s, o \rangle) \in X : [\![B]\!] = \{q \mid (q, \langle s, o \rangle) \in X\}\}$$

It is easy to see that, for any $\preceq_{\mathsf{MH}}$-antichain X, we have that its semi-symbolic representation $R(X)$ is such that for every $(B_1, \langle s_1, o_1 \rangle) \in X$, and $(B_2, \langle s_2, o_2 \rangle) \in X$ we have that if $\langle s_1, o_1 \rangle \prec_{\mathsf{MH}} \langle s_2, o_2 \rangle$ then $[\![B_1]\!] \cap [\![B_2]\!] = \varnothing$. This property relates to the Property Driven Partitioning of Vardi et al [STV05], as it quotients the Kripke state variables with respect to the Miyano-Hayashi states.

Based on the observations above, adapting our LTL satisfiability algorithms to the model checking case is straightforward: one must simply alter the pre or post computations in order to synchronize the exploration of the Miyano-Hayashi NBA with the ROBDD-encoded Kripke transition relation.

4.8 Experimental Evaluation

In this section, we provide experimental evidence that the combination of antichains and ROBDD for LTL satisfiability and model checking is promis-

ing. To this end, we have developed the ALASKA tool [DDMR08a], which implements the algorithms described in this chapter. ALASKA is developed in Python, and uses the PyCuDD[1] binding library to the CuDD ROBDD package [Som98]. ALASKA implements the reachability and reachability antichain-based fixed points defined in Chapter 3, both in backward and forward versions (see Definition 3.4.3, Definition 3.4.6, Definition 3.4.9, and Definition 3.4.12). For finite-word LTL satisfiability, our tool translates the formula into a sAFA and checks for emptiness, by using either the $\widehat{\mathsf{B}}^*$ or the $\widehat{\mathsf{F}}^*$ algorithm on the subset construction state space. For infinite-word LTL satisfiability, ALASKA translates the formula into a sABA and checks for emptiness, by using either the $\widehat{\mathsf{BB}}^*$ or the $\widehat{\mathsf{FF}}^*$ algorithm on the Miyano-Hayashi state space. The ALASKA tool currently only implements the *symbolic* translations from ROBDD to antichains, not the recursive ones (the latter are recent developments).

According to the extensive survey of Vardi and Rozier [RV07] NuSMV is the most efficient (freely available) tool for LTL satisfiability. We therefore compare the performance of ALASKA with NuSMV. As mentioned previously, satisfiability checking with NuSMV is done simply by model checking the negation of the formula against a universal Kripke structure. In all our experiments, we used NuSMV 2.4 with the default options.[2] No variable reordering techniques were activated in either tool.

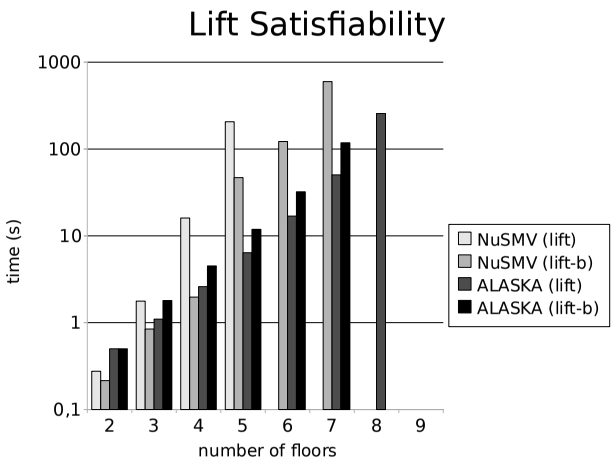
4.8.1 LTL Satisfiability Benchmarks

We have compared ALASKA and NuSMV on four families of (infinite-word) LTL formulas. All the experiments were performed on a single Intel Xeon CPU at 3.0 GHz, with 4 GB of RAM, using a timeout of 10 min and a maximum memory usage limit of 2.5 GB (all experiments either succeeded or timed out before exceeding the memory limit). In all our experiments, the $\widehat{\mathsf{BB}}^*$ algorithm performed considerably worse than the $\widehat{\mathsf{FF}}^*$ algorithm, so we report results only for the latter.

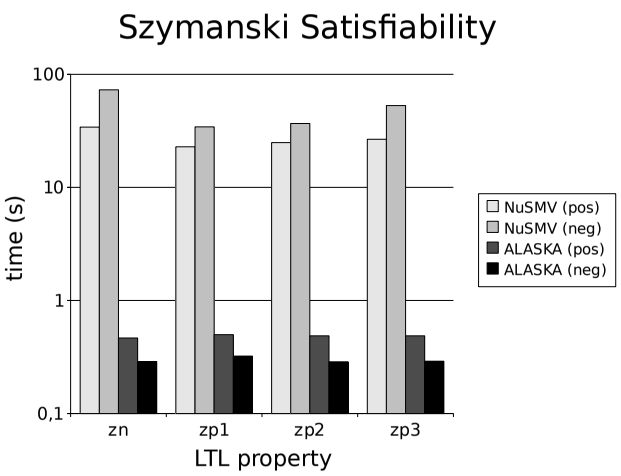
The first family is a parametric specification of a lift system with n

[1] http://www.ece.ucsb.edu/bears/pycudd.html

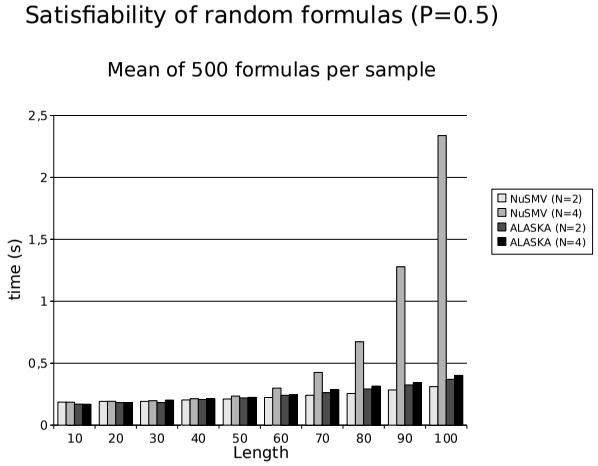
[2] The options are numerous, check the NuSMV user manual for full details.



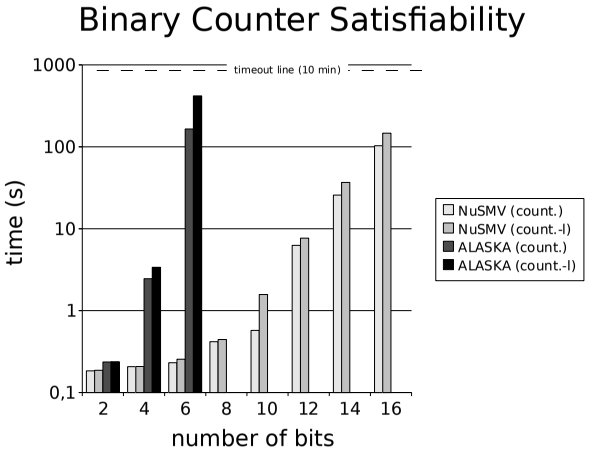
(a)



(b)

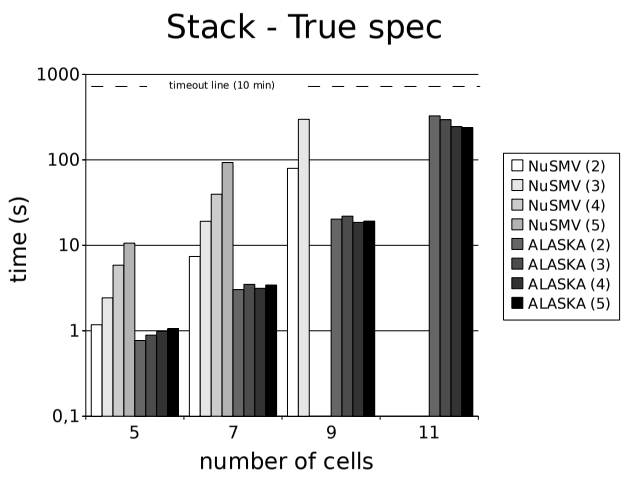


(c)

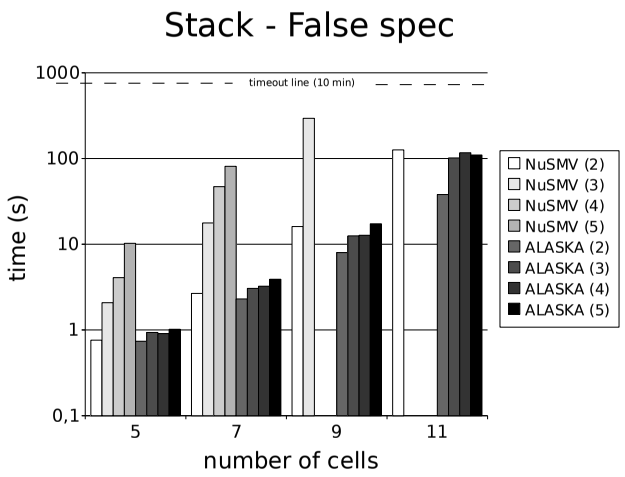


(d)

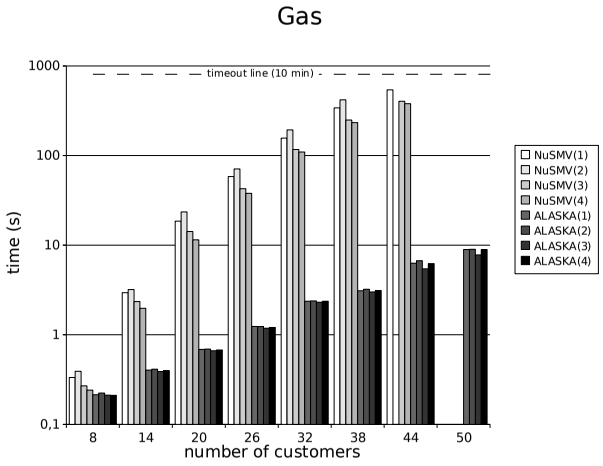
Figure 4.3: Experimental results for LTL satisfiability.



(a)



(b)



(c)

Time and Memory for the Bakery Algorithm				
	Mutual Exclusion		Fairness	
	NuSMV	ALASKA	NuSMV	ALASKA
2 proc.	0.22s 10.7MB	0.55s 60.3MB	10.66s 35.4MB	3.47s 60.3MB
3 proc.	359.12s 656.7MB	8.81s 87.3MB	28740.17s 495.0MB	730.60s 201.3MB
4 proc.	> 1000s out of Mem.	$630,7$s $579,01$MB	N/A N/A	N/A N/A

(d)

Figure 4.4: Experimental results for LTL model checking.

floors that we have taken from Harding's thesis [Har05]. Two encodings are used: one ("lift") that uses a linear number of variables per floor, and another ("lift-b") which uses a number of variables that is logarithmic in the number of floors (resulting in larger formulas). As seen in figure 4.3(a), our algorithm scales better than NuSMV, especially for the "lift" formulas. For more than 7 floors (a formula with 91 temporal operators and 17 distinct Boolean propositions), NuSMV is more than 60 times slower than our tool.

The second family of formulas was referenced in [STV05] as examples of difficult LTL to NBA translation and describes liveness properties for the Szymanski mutual exclusion protocol and for a variant of this protocol due to Pnueli. We have run both our prototype and NuSMV on these four formulas (pos) and their negation (neg). Again, our tool shows better performance (by factors of 50 and higher), as reported in figure 4.3(b).

The third family we used is a random model described in [DGV99] and also used in [RV07]. Random LTL formulas are generated according to the following parameters: the length of the formula (L), the number of propositions (N) each with equal probability of occurrence, and the probability (P) of choosing a temporal operator ($\mathcal{U}$ or $\mathcal{R}$). As in [RV07], we fix $P = 0.5$ and compare execution times for $L \in \{10, 20, \ldots, 100\}$ and for both $N = 2$ and $N = 4$. As indicated by figure 4.3(c), our algorithm copes much better with the joint increase in formula length and number of propositions[3]. For $L = 100$, going from $N = 2$ to $N = 4$ multiplies the time needed by NuSMV by 7, while our prototype only exhibits an 8% increase in execution time.

Finally, the last set of formulas (also taken in [RV07]) describes how a binary counter, parametrized by its length, is incremented. Two ways of formalizing the increment are considered ("count", "count-l"). Those formulas are quite particular as they all define a unique model: for $n = 2$, the model is $(00 \cdot 01 \cdot 10 \cdot 11)^\omega$. In this benchmark, the classical fully-symbolic algorithm behaves much better than our antichain algorithm. This is not surprising for two reasons. First, the efficiency of our antichain-based algorithms comes from the ability to identify prefixes of runs in the sABA which can be ignored because they impose more constraints than others on the future. As there is only one future allowed by the formula,

[3]We report the mean execution times; the standard deviation is similar for both tools.

the locations of the NBA defined by the Miyano-Hayashi construction are incomparable for the $\preceq_{\mathsf{MH}}$ simulation relation, causing very long antichains and poor performances. This can be considered as a pathological and maybe not very interesting case.

4.8.2 LTL Model Checking Benchmarks

The implementation of the LTL model checking algorithms in ALASKA is very similar to their counterpart for satisfiability. The majority of the implementation effort of model checking in ALASKA concerns the necessary interfacing with NuSMV in order to extract the ROBDD*s* that NuSMV produces after reading the program given in the NuSMVs input language. This approach has two advantages. First, we can effectively run our algorithm on any available SMV model, making direct comparisons with NuSMV straightforward. Second, we are guaranteed to use *exactly* the same ROBDD*s* for the Kripke structure (with the same ordering on variables) than NuSMV, making direct comparisons meaningful. Like for satisfiability, the $\widehat{\mathsf{BB}}^*$ algorithm performed (much) worse on all our examples than $\widehat{\mathsf{FF}}^*$, so we report results for the forward algorithm.

Concerning the use of NuSMV, all our experiments were performed in a similar way than for satisfiability, namely using NuSMV 2.4 without any option except "-dcx" which disables the creation of counter-examples. By default, NuSMV implements the following version of the LTL symbolic algorithm: it precomputes the reachable states of the Kripke structure and then evaluates a backward fixed point expression (the Emerson-Lei algorithm [EL86]) for checking emptiness of the product of the structure and the NBA of the formula (encoded with ROBDD*s*). Guiding the backward iterations with reachable states usually improves execution times dramatically. It also makes the comparison with our algorithm fair as it also only visits reachable states.

We have compared ALASKA with NuSMV on three families of scalable NuSMV models. The first family describes a gas station with an operator, one pump, and n customers (n is a parameter) waiting in line to use the pump. The operator manages the customer queue and activates or

deactivates the pump. This resource-access protocol was used in [STV05] as an LTL model-checking benchmark. We have used the same four LTL formulas as in [STV05]. The running times for n between 8 and 50 are given in Fig. 4.4(c). The difference in scalability is striking. For instance, for $n = 38$ NuSMV needs several minutes (between 233 and 418 seconds depending on the property), while our algorithm completes in just over 3 seconds for all properties. NuSMV is not able to verify models with 50 customers within 10 minutes while our algorithm handles them in less than 10 seconds.

The second family of models also comes from [STV05] and represents a stack, on which push, pop, empty and freeze operations can be performed. Each cell of the stack can hold a value from the set {1,2} and a freeze operation allows to permanently freeze the stack, after which the model runs a pointer along the stack from top to bottom repeatedly. At each step of this infinite loop, a "call" predicate indicates the value currently pointed.[4] As we needed a scalable set of formulas for at least one model to compare the scalability of our algorithm with NuSMV, we have provided and used our own specifications for this model. These specifications enforce that if the sequence of push operations "$12 \ldots n$" is performed and not followed by any pop until the freeze operation, then the subsequence of call operations "$n \ldots 21$" appears infinitely often. Figure 4.4(a) and Figure 4.4(b) show the execution times for stack lengths between 5 and 11 (*number of cells* in the figure) and push sequences of length 2 to 5. Figure 4.4(a) shows the performance of the tools for model checking the formulas as presented in [STV05] (*true spec*), while Figure 4.4(b) reports on the performance obtained when a bug is purposefully inserted in the LTL formula (*false spec*). This benchmark indicates that the ALASKA tool can outperform NuSMV on both negative and positive instances of the LTL model checking problem.

Finally, the last family of models that we consider is a finite state version of the Lamport's bakery mutex protocol [Lam74]. This protocol is

[4]For example, if the stack contains, from bottom to top, {1,2} then after the freeze operation, the model will behave like this : call2, call1, call2, call1, ...

interesting because it imposes fairness among all processes and again it is parametric in the number n of participating processes. Our model is large and grows very rapidly with the number of processes. For 2 processes, it uses 42 Boolean variables and requires BDDs with a total of 7750 nodes to encode the model, for 4 processes, it uses 91 variables and BDDs with more than 20 million nodes. Again, our algorithm scales much better than the classical fully symbolic algorithm. For 3 processes, we are able to verify the fairness requirement in 730.6 seconds while NuSMV needs 28740.17s. Also, our algorithm requires much less memory than NuSMV, see Table 4.4(d) for the details.

4.9 Discussion

In this chapter, we have proposed new antichain-based algorithms for the LTL satisfiability and (symbolic) model checking problems. These new algorithms result from non-trivial adaptations of the basic antichain framework described in the previous chapter, and combine the use of antichains with ROBDD-based techniques. We have implemented our algorithms into a prototype tool called ALASKA and showed that, on various meaningful benchmarks, it compares favorably with the state-of-the-art tool NuSMV.

Several lines of future works can be envisioned. First, we should try to exploit the fact that the alternating automata generated by the translation from LTL are *very weak* [Roh97] (they are also called *linear weak* or *1-weak*). Like LTL, very weak alternating automata on infinite words are strictly less expressive than classical alternating automata, so there is hope that our algorithms could be improved for these automata. Note that it is also an advantage of our approach: our algorithms can be readily used to model check full ω-regular specifications, similarly to the work in [DR07].

Second, it is likely that our algorithms would benefit from the more recent antichain approaches presented in [DR10] and [ACH+10]. This should also be combined with state-of-the-art minimization techniques for LTL formulas and alternating automata (see [Tau03] for a survey). The ALASKA implementation is currently lacking in these domains.

Also, it is likely that the combination of antichains and ROBDD would

appropriately fit a number of other automata-theoretic problems, and it would be useful to generalize the mapping relationships and translation algorithms between antichains and ROBDD to arbitrary finite domains and arbitrary simulations.

Chapter 5

Fixed Point-Guided Abstraction Refinement

The antichain approach to alternating automata emptiness can encounter scalability issues when the number of states of the automaton becomes too large. In this chapter, we develop an original abstraction framework for alternating automata, designed to keep antichains from growing prohibitively.

5.1 Introduction

As discussed in the preliminary chapter of this thesis, the reachability and repeated reachability problems for finite-state systems reduce to the evaluation of fixed points of monotone functions. Because of the state space explosion problem, the domain on which such fixed points are computed is often huge, making the computation of the fixed point unfeasible. The reduction of reachability to fixed point evaluation generalizes also to *infinite-state* systems, making the direct fixed point evaluations usually non computable.

Cousot and Cousot have developed in the late 1970's an *abstract interpretation* framework for evaluating fixed points over so-called *abstract domains* [CC77]. The idea is to find a mapping of the concrete state space on which the fixed point is computed, to a more tractable state space. The formalization of this mapping is given by means of *Galois connections*.

The process of devising good abstractions by hand is difficult. If the ab-

stract domain is too coarse, the abstract fixed point will not contain enough information to be useful. If the abstraction is too precise, then the fixed point evaluation may be too hard to evaluate. In recent years, much effort has been devoted to the development of so-called *abstraction-refinement* algorithms, which attempt to find suitable abstractions automatically with successive refinements of the abstract domain.

Current methods for handling very large alternating automata suffer from the state space explosion problem. In practice, this blowup manifests itself in fully-symbolic algorithms by either excessive memory usage (when the number of ROBDD nodes exceeds RAM capacity), or by very long variable-reordering times. In the case of antichain-based algorithms, very large alternating automata often lead to excessively long antichains.

Contributions To mitigate the state-space explosion problem, and apply our antichain-based algorithms to larger instances of alternating automata, we instantiate a generic abstract-refinement method that has been proposed in [CGR07] and further developed in [Gan07] and [GRB08]. This abstract-refinement method does not use counter-examples to refine inconclusive abstractions contrary to most of the methods presented and implemented in the literature (see for example [CGJ+03]). Instead, our algorithm uses the entire information computed by the abstract analysis and combines it with information obtained by one application of a concrete state space operator. The algorithm presented in [CGR07] is a generic solution that does not lead directly to efficient implementations. In particular, as shown in [Gan07], in order to obtain an efficient implementation of this algorithm, we need to define a family of abstract domains on which abstract analysis can be effectively computed, as well as practical operators to refine the elements of this family of abstract domains. In this work, we use the set of *partitions* of the states of an alternating automaton to define the family of abstract domains. Those abstract domains and their refinement operators can be used both in *forward* and *backward* algorithms for checking emptiness of alternating automata.

To show the practical interest of these new algorithms, we have implemented them into Alaska. We illustrate the efficiency of our new algorithms on examples of alternating automata constructed from LTL specifi-

cations interpreted over finite words. With the help of these examples, we show that our algorithms are able to concentrate the analysis on the important parts of the state-space and abstract away the less interesting parts *automatically*. This allows us to treat much larger instances than with the concrete forward or backward algorithms.

Structure of the chapter In Section 5.2, we develop an efficient abstraction method for AFA, and we show how the state space of the subset construction of those abstract AFA is linked to the state space of concrete AFA by a Galois connection. We also show in this section that the evaluation of pre and post on abstract AFA are the *best approximations* of the corresponding pre and post computations on concrete AFA. In Section 5.3, we present our abstract forward and backward algorithms with refinement. In Section 5.4, we report on experiments that illustrate the efficiency of our algorithms. Finally, we draw some conclusions and evaluate future directions in Section 5.5.

5.2 Abstraction of Alternating Automata

This section introduces our abstraction framework for AFA and is structured as follows. In Section 5.2.2 we formally define a pair of concrete and abstract domains, along with a pair of abstraction and concretization functions and show that they form a Galois connection. We also show that this Galois connection has useful algebraic properties with respect to $\subseteq$-closed sets, which we need for subsequent proofs. Computing abstractions and concretizations is costly, so Section 5.2.3 defines a *syntactic abstraction* for AFA, computable in linear time, which saves the need of explicitly evaluating abstractions and concretizations. Finally, Section 5.2.4 provides precision and representability results of our Galois connection; essentially, for any set of states X, we show that there exists a unique coarsest abstract domain, which is easily computable, and such that when X is abstracted and then concretized, the resulting set is still X. In other words, we can easily compute the coarsest abstract domain to represent any set of states exactly.

5.2.1 The Lattice of Partitions

The heart of our abstraction scheme is to *partition* the set of states Q of an AFA in order to build a smaller (hopefully more tractable) automaton. We now recall the notion of partitions and some of their properties. Let $\mathcal{P}$ be a partition of $Q = \{q_1, \ldots, q_n\}$ into k *classes* (called *blocks* in the sequel) $\mathcal{P} = \{b_1, \ldots, b_k\}$. Partitions are classically ordered as follows: $\mathcal{P}_1 \preceq \mathcal{P}_2$ (read $\mathcal{P}_1$ *refines* $\mathcal{P}_2$) iff $\forall\, b_1 \in \mathcal{P}_1, \exists\, b_2 \in \mathcal{P}_2 : b_1 \subseteq b_2$. It is well known (see [BS81]) that the set of partitions together with $\preceq$ form a complete lattice where $\{\{q_1\}, \ldots, \{q_n\}\}$ is the $\preceq$-minimal element, $\{\{q_1, \ldots, q_n\}\}$ is the $\preceq$-maximal element and the greatest lower bound of two partitions $\mathcal{P}_1$ and $\mathcal{P}_2$, noted $\mathcal{P}_1 \curlywedge \mathcal{P}_2$, is the partition given by $\{b \neq \varnothing \mid \exists\, b_1 \in \mathcal{P}_1, \exists\, b_2 \in \mathcal{P}_2 \colon b = b_1 \cap b_2\}$. The least upper bound of two partitions $\mathcal{P}_1$ and $\mathcal{P}_2$, noted $\mathcal{P}_1 \curlyvee \mathcal{P}_2$, is the finest partition such that given $b \in \mathcal{P}_1 \cup \mathcal{P}_2$, for all $q_i \neq q_j$ with $q_i \in b$ and $q_j \in b$ we have: $\exists\, b' \in \mathcal{P}_1 \curlyvee \mathcal{P}_2 \colon q_i \in b'$ and $q_j \in b'$. Note that by the above definitions we have that $\mathcal{P}_1 \preceq \mathcal{P}_2$ iff there exists a partition $\mathcal{P}$ such that $\mathcal{P}_1 \curlyvee \mathcal{P} = \mathcal{P}_2$. In the sequel, we use $\mathcal{P}$ as a function such that $\mathcal{P}(q)$ simply returns the block b to which q belongs in $\mathcal{P}$.

Example 5.2.1. *Given the set* $S = \{a, b, c\}$ *and two partitions* $A_1 = \{\{a, b\}, \{c\}\}$ *and* $A_2 = \{\{a, c\}, \{b\}\}$. *We have that* $A_1 \curlywedge A_2 = \{\{a\}, \{b\}, \{c\}\}$, $A_1 \curlyvee A_2 = \{\{a, b, c\}\}$, *and* $A_2(a) = \{a, c\}$.

5.2.2 Abstract domain

Given an AFA with set of states Q, our algorithm will use a family of abstract domains defined by the set of partitions of Q. The concrete domain is the complete lattice $\langle 2^{2^Q}, \subseteq \rangle$ (that is, the set of states of $\mathsf{SC}(A)$, see Definition 2.3.8), and each partition $\mathcal{P}$ defines the abstract domain $\langle 2^{2^\mathcal{P}}, \subseteq \rangle$. We refer to elements of 2^Q as *concrete* NFA *states* and elements of $2^\mathcal{P}$ as *abstract* NFA *states*[1]. An abstract state is thus a set of blocks of the partition $\mathcal{P}$ and it represents all the concrete states which can be constructed by choosing at least one AFA state from each block. To capture this representation role of abstract states, we define the following predicate:

[1] In the sequel, NFA states are simply referred to as "states".

Definition 5.2.2. *The predicate* $\mathsf{Covers} : 2^{\mathcal{P}} \times 2^{Q} \to \{\mathsf{true}, \mathsf{false}\}$ *is defined as follows:* $\mathsf{Covers}(a, c)$ *iff* $a = \{\mathcal{P}(q) \mid q \in c\}$.

Note that for every concrete state c there exists *exactly one* abstract state a such that $\mathsf{Covers}(a, c)$. Similarly, we have that for every abstract state a there exists *at least one* concrete state c such that $\mathsf{Covers}(a, c)$.

Example 5.2.3. *Let* $Q = \{1, \ldots, 5\}$, $\mathcal{P} = \{b_1 : \{1\}, b_{2,3} : \{2, 3\}, b_{4,5} : \{4, 5\}\}$. *We have* $\mathsf{Covers}(\{b_1, b_{4,5}\}, \{1, 3\}) = \mathsf{false}$, $\mathsf{Covers}(\{b_1, b_{4,5}\}, \{1, 4\}) = \mathsf{true}$, *and* $\mathsf{Covers}(\{b_1, b_{4,5}\}, \{1\}) = \mathsf{false}$.

To make proper use of the theory of abstract interpretation, we define an *abstraction function* and a *concretization function*, and show, in Lemma 5.2.5, that they form a *Galois connection* between the concrete domain and each of our abstract domains.

Definition 5.2.4. *Let* $\mathcal{P}$ *be a partition of the set* Q. *We define the functions* $\alpha_{\mathcal{P}} : 2^{2^{Q}} \to 2^{2^{\mathcal{P}}}$ *and* $\gamma_{\mathcal{P}} : 2^{2^{\mathcal{P}}} \to 2^{2^{Q}}$ *as follows:*

$$\alpha_{\mathcal{P}}(X) = \{a \mid \exists\, c \in X : \mathsf{Covers}(a, c)\}$$
$$\gamma_{\mathcal{P}}(X) = \{c \mid \exists\, a \in X : \mathsf{Covers}(a, c)\}$$

Lemma 5.2.5. *For any finite set* Q, *and for any partition* $\mathcal{P}$ *of* Q, *we have that* $(\alpha_{\mathcal{P}}, \langle 2^{2^{Q}}, \subseteq\rangle, \langle 2^{2^{\mathcal{P}}}, \subseteq\rangle, \gamma_{\mathcal{P}})$ *forms a Galois insertion.*

Proof. By Definition 2.1.8, we must first show that $\alpha(X) \subseteq Y$ iff $X \subseteq \gamma(Y)$:

$$\begin{aligned}
&\alpha(X) \subseteq Y \\
&\text{iff } \{a \mid \exists\, c \in X : \mathsf{Covers}(a, c)\} \subseteq Y \\
&\text{iff } \forall\, c \in X : \exists\, a \in Y : \mathsf{Covers}(a, c) \\
&\text{iff } X \subseteq \{c \mid \exists\, a \in Y : \mathsf{Covers}(a, c)\} \\
&\text{iff } X \subseteq \gamma(Y)
\end{aligned}$$

To show that it is a Galois insertion, we prove that $\{a\} \subseteq \alpha \circ \gamma(a)$.

$$\begin{aligned}
\alpha \circ \gamma(a) &= \alpha\big(\{c \mid \mathsf{Covers}(a, c)\}\big) \\
&= \{a' \mid \exists\, c : \mathsf{Covers}(a, c) \wedge \mathsf{Covers}(a', c)\} \\
&\supseteq \{a\}
\end{aligned}$$

□

Remark 5.2.6. *In the sequel, we alleviate the notations by omitting* (*i*) *the* $\mathcal{P}$ *subscript of* α *and* γ *when the partition is clear from the context, and* (*ii*) *the extra brackets when* $\alpha(\cdot)$, $\gamma(\cdot)$, $\uparrow\cdot$ *and* $\downarrow\cdot$ *are applied to singleton sets. Additionally, we define* $\mu_{\mathcal{P}} \stackrel{\mathsf{def}}{=} \gamma_{\mathcal{P}} \circ \alpha_{\mathcal{P}}$.

Example 5.2.7. *Let* $Q = \{1, \ldots, 5\}$, $\mathcal{P} = \{b_1 : \{1\}, b_{2,3} : \{2,3\}, b_{4,5} : \{4,5\}\}$. *The* $\supseteq$*-upward-closed set* $\uparrow\{\{1,2\},\{1,3\},\{1,4\}\}$ *is abstracted in the Galois connection by* $\alpha(\uparrow\{\{1,2\},\{1,3\},\{1,4\}\}) = \uparrow\{\{b_1, b_{2,3}\}, \{b_1, b_{4,5}\}\}$. *We can see that this set is not represented exactly by the abstraction, since we have* $\gamma \circ \alpha(\uparrow\{\{1,2\},\{1,3\},\{1,4\}\})) = \uparrow\{\{1,2\},\{1,3\},\{1,4\},\{1,5\}\}$.

Lemma 5.2.8 and Lemma 5.2.9 below provide a couple of algebraic properties of α and γ which we will need later to prove precision results of our abstract operators.

Lemma 5.2.8. *Let* $\mathcal{P}$ *be a partition of* Q, $c \in 2^Q$ *and* $a \in 2^{\mathcal{P}}$. *We have that* $\alpha(\uparrow c) = \uparrow\alpha(c)$, $\alpha(\downarrow c) = \downarrow\alpha(c)$, $\gamma(\uparrow a) = \uparrow\gamma(a)$, *and* $\gamma(\downarrow a) = \downarrow\gamma(a)$.

Proof. For the first equality, let $c \in 2^Q$, we show that $\uparrow\alpha(c) = \alpha(\uparrow c)$. We conclude from the definition of α that $\alpha(c) = \{a \mid \mathsf{Covers}(a, c)\}$, hence that $\alpha(c) = \{a \mid a = \{\mathcal{P}(q) \mid q \in c\}\}$ by definition of Covers, and finally that $\alpha(c) = \{\mathcal{P}(q) \mid q \in c\}$.

$$\begin{aligned} \alpha(\uparrow c) &= \{\alpha(c') \mid c \subseteq c'\} \\ &= \{\{\mathcal{P}(q) \mid q \in c'\} \mid c \subseteq c'\} \end{aligned} \tag{5.1}$$

We now prove each inclusion in turn.

$\uparrow\alpha(c) \subseteq \alpha(\uparrow c)$
iff $\forall\, a' \colon \alpha(c) \subseteq a' \Rightarrow a' \in \alpha(\uparrow c)$
iff $\forall\, a' \colon \{\mathcal{P}(q) \mid q \in c\} \subseteq a' \Rightarrow a' \in \{\{\mathcal{P}(q) \mid q \in c'\} \mid c \subseteq c'\}$ by (5.1)
iff $\forall\, a' \colon \{\mathcal{P}(q) \mid q \in c\} \subseteq a' \Rightarrow \exists\, c' \colon c \subseteq c' \wedge a' = \{\mathcal{P}(q) \mid q \in c'\}$
iff true (set $c' = c \cup \{q \in b \mid b \in a'\}$)

For the reverse direction, we have

$\alpha(\uparrow c) \subseteq \uparrow \alpha(c)$
iff $\forall\, c' \colon c \subseteq c' \Rightarrow \alpha(c') \in \uparrow \alpha(c)$
iff $\forall\, c' \colon c \subseteq c' \Rightarrow \{\mathcal{P}(q) \mid q \in c'\} \in \uparrow \{\mathcal{P}(q) \mid q \in c\}$ by $\alpha(c)$ above
iff true

For the second equality, let $a \in 2^{\mathcal{P}}$, we show that $\uparrow \gamma(a) = \gamma(\uparrow a)$.

$$\begin{aligned}
\gamma(\uparrow a) &= \{c \mid \exists\, a' \colon a \subseteq a' \wedge \mathsf{Covers}(a', c)\} && \text{definition of } \gamma \\
&= \{c \mid \exists\, a' \colon a \subseteq a' \wedge a' = \{\mathcal{P}(q) \mid q \in c\}\} && \text{definition of } \mathsf{Covers} \\
&= \{c \mid a \subseteq \{\mathcal{P}(q) \mid q \in c\}\} && \text{elim. } a'
\end{aligned}$$

If c is such that $a \subseteq \{\mathcal{P}(q) \mid q \in c\}$, so is every $c \subseteq c'$

$$\begin{aligned}
&= \uparrow \{c \mid a \subseteq \{\mathcal{P}(q) \mid q \in c\}\} \\
&= \uparrow \{c \mid a = \{\mathcal{P}(q) \mid q \in c\}\} && \text{keep the minimum} \\
&= \uparrow \{c \mid a = \mathsf{Covers}(a, c)\} && \text{definition of } \mathsf{Covers} \\
&= \uparrow \gamma(a) && \text{definition of } \gamma
\end{aligned}$$

The remaining cases are proved similarly.

□

Lemma 5.2.9. *Let $X, Y \subseteq 2^{\mathcal{Q}}$ such that both are either upward- or downward-closed. We have that $\alpha(X \cup Y) = \alpha(X) \cup \alpha(Y)$, $\alpha(X \cap Y) = \alpha(X) \cap \alpha(Y)$. Moreover, we have $\gamma(X \cap Y) = \gamma(X) \cap \gamma(Y)$, and $\gamma(X \cup Y) = \gamma(X) \cup \gamma(Y)$ for $X, Y \subseteq 2^{\mathcal{P}}$.*

Proof. The equality $\alpha(X \cup Y) = \alpha(X) \cup \alpha(Y)$ follows from the fact that (α, γ) is a Galois connection. Let us prove each direction of $\alpha(X \cap Y) = \alpha(X) \cap \alpha(Y)$ in turn. The inclusion $\subseteq$ follows trivially by the (α, γ) Galois connection. For the other direction, we have

$a \in \alpha(X) \cap \alpha(Y)$
iff $\exists\, c \in X \colon \mathsf{Covers}(a, c) \wedge \exists\, c' \in Y \colon \mathsf{Covers}(a, c')$ definition of α
iff $\exists\, c \in X \colon a = \{\mathcal{P}(q) \mid q \in c\} \wedge \exists\, c' \in Y \colon a = \{\mathcal{P}(q) \mid q \in c'\}$ definition of Covers
only if $\exists\, c \in X\, \exists\, c' \in Y \colon a = \{\mathcal{P}(q) \mid q \in c \cup c'\}$ $(*)$

For each $c \in X, c' \in Y$, let c'' denote $c \cup c'$. We have $c'' \in X \cap Y$ since $X = \uparrow X$ and $Y = \uparrow Y$, so we find that

$$\begin{aligned}
&\exists\, c'' \in X \cap Y \colon a = \{\mathcal{P}(q) \mid q \in c''\} && \text{equivalent to } (*)\\
&\text{iff } \exists\, c'' \in X \cap Y \colon \mathsf{Covers}(a, c'') && \text{definition of } \mathsf{Covers}\\
&\text{iff } a \in \alpha(X \cap Y)
\end{aligned}$$

The equality $\gamma(X \cap Y) = \gamma(X) \cap \gamma(Y)$ is a property of Galois connections. Finally,

$$\begin{aligned}
\gamma(X \cup Y) &= \{c \mid \exists\, a \in X \cup Y \colon \mathsf{Covers}(a, c)\} && \text{definition of } \gamma\\
&= \{c \mid \exists\, a \in X \colon \mathsf{Covers}(a, c)\} \cup\\
&\quad \{c \mid \exists\, a \in Y \colon \mathsf{Covers}(a, c)\}\\
&= \gamma(X) \cup \gamma(Y) && \text{definition of } \gamma
\end{aligned}$$

□

Corollary 5.2.10. *Let $\mathcal{P}$ be a partition of Q, $X \subseteq 2^Q$ and $Y \subseteq 2^{\mathcal{P}}$. We have $\alpha(\uparrow X) = \uparrow \alpha(X)$, $\alpha(\downarrow X) = \downarrow \alpha(X)$, $\gamma(\uparrow Y) = \uparrow \gamma(Y)$, and $\gamma(Y) = \downarrow \gamma(Y)$.*

5.2.3 Efficient abstract analysis

In the sequel, we will need to evaluate fixed point expressions over the abstract domain. Following Theorem 2.1.10, we have that:

$$\alpha(\mathsf{LFP}[2^{2^Q}, \subseteq](\lambda\, X : \mathsf{post}(X) \cup I)) \subseteq \mathsf{LFP}[2^{2^{\mathcal{P}}}, \subseteq](\lambda\, X : \alpha(\mathsf{post}(\gamma(X)) \cup I))$$
$$\alpha(\mathsf{LFP}[2^{2^Q}, \subseteq](\lambda\, X : \mathsf{pre}(X) \cup F)) \subseteq \mathsf{LFP}[2^{2^{\mathcal{P}}}, \subseteq](\lambda\, X : \alpha(\mathsf{pre}(\gamma(X)) \cup F))$$

For performance reasons, we want to avoid as many explicit evaluations of γ and α as possible, and above all we want to avoid the evaluation of pre or post on the concrete domain. To achieve this goal, we proceed as follows. Given a partition $\mathcal{P}$ of the set of states of an alternating automaton, we use a *syntactic transformation* Abs which builds an *abstract* AFA which overapproximates the behavior of the original automaton. Later we will show that the pre and post predicate transformers can be directly evaluated

on this abstract automaton to obtain the same result as the $\alpha \circ \mathsf{pre} \circ \gamma$ and $\alpha \circ \mathsf{post} \circ \gamma$ computations on the original automaton. Doing so results in avoiding almost all explicit evaluations of α and γ. To express this syntactic transformation, we define *syntactic variants* of the abstraction and concretization functions.

Definition 5.2.11. *Let $\mathcal{P}$ be a partition of the set Q. We define the following* syntactic *abstraction and concretization functions over positive Boolean formulas.*

$$\begin{aligned} \hat{\alpha} : \mathsf{BL}^+(Q) &\to \mathsf{BL}^+(\mathcal{P}) & \hat{\gamma} : \mathsf{BL}^+(\mathcal{P}) &\to \mathsf{BL}^+(Q) \\ \hat{\alpha}(l) &= \mathcal{P}(q) & \hat{\gamma}(b) &= \textstyle\bigvee_{q \in b} q \\ \hat{\alpha}(\varphi_1 \vee \varphi_2) &= \hat{\alpha}(\varphi_1) \vee \hat{\alpha}(\varphi_2) & \hat{\gamma}(\varphi_1 \vee \varphi_2) &= \hat{\gamma}(\varphi_1) \vee \hat{\gamma}(\varphi_2) \\ \hat{\alpha}(\varphi_1 \wedge \varphi_2) &= \hat{\alpha}(\varphi_1) \wedge \hat{\alpha}(\varphi_2) & \hat{\gamma}(\varphi_1 \wedge \varphi_2) &= \hat{\gamma}(\varphi_1) \wedge \hat{\gamma}(\varphi_2) \end{aligned}$$

Lemma 5.2.12. *For every $\varphi \in \mathsf{BL}^+(Q)$, we have that $[\![\hat{\alpha}(\varphi)]\!] = \alpha([\![\varphi]\!])$, and for every $\varphi \in \mathsf{BL}^+(\mathcal{P})$, we have that $[\![\hat{\gamma}(\varphi)]\!] = \gamma([\![\varphi]\!])$.*

Proof. We start by $[\![\hat{\alpha}(\varphi)]\!] = \alpha([\![\varphi]\!])$, and proceed by induction over the structure of φ. For the base case, Lemma 5.2.8 and the definition of $[\![\cdot]\!]$ show that $\alpha([\![q]\!]) = \alpha(\uparrow\{\{q\}\}) = \uparrow\alpha(\{\{q\}\}) = \uparrow\{\{\mathcal{P}(q)\}\} = [\![\mathcal{P}(q)]\!] = [\![\hat{\alpha}(q)]\!]$. The inductive cases are immediate by Lemma 5.2.9. We follow with $[\![\hat{\gamma}(\varphi)]\!] = \gamma([\![\varphi]\!])$ and proceed again by induction on the structure of φ. For the base case, again Lemma 5.2.8 and definition of $[\![\cdot]\!]$ show that $\gamma([\![b]\!]) = \gamma(\uparrow\{\{b\}\}) = \uparrow\gamma(\{\{b\}\}) = \left\uparrow \bigcup_{q \in b} \{\{q\}\} = [\![\bigvee_{q \in b} q]\!] = [\![\hat{\gamma}(b)]\!]$. Finally, the inductive cases follow from Lemma 5.2.9. □

Definition 5.2.13. *Let $A = \langle Q, \Sigma, q_0, \delta, F\rangle$ be an* AFA*, and let $\mathcal{P}$ a partition of Q. We define $\mathsf{Abs}(A, \mathcal{P}) = \langle Q^{\mathsf{Abs}}, \Sigma, b_0, \delta^{\mathsf{Abs}}, F^{\mathsf{Abs}}\rangle$ to be an* AFA *where: $Q^{\mathsf{Abs}} = \mathcal{P}$, $b_0 = \mathcal{P}(q_0)$, $\delta^{\mathsf{Abs}}(b, \sigma) = \hat{\alpha}(\bigvee_{q \in b} \delta(q, \sigma))$, and $F^{\mathsf{Abs}} = \{b \in \mathcal{P} \mid b \cap F \neq \varnothing\}$.*

Lemma 5.2.14. *For every* AFA *$A = \langle Q, \Sigma, q_0, \delta, F\rangle$, and for every partition $\mathcal{P}$ of Q we have that $L(A) \subseteq L(\mathsf{Abs}(A, \mathcal{P}))$.*

Proof. We begin by showing that every state $c \in 2^Q$ of $\mathsf{SC}(A)$ is simulated by its corresponding abstract state $a \in 2^{Q^{\mathsf{Abs}}}$ of $\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))$. Recall from Definition 2.3.8 that $\mathsf{SC}(A)$ is such that $c_1 \xrightarrow{\sigma} c_2$ iff $c_2 \models \bigwedge_{q \in c_1} \delta(q, \sigma)$. Similarly, $\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))$ is such that $a_1 \xrightarrow{\sigma} a_2$ iff $a_2 \models \bigwedge_{b \in a_1} \delta^{\mathsf{Abs}}(b, \sigma)$. Let $\alpha(\{c_1\}) = \{a_1\}$ and $\alpha(\{c_2\}) = \{a_2\}$. We have the following equivalences: $a_1 \xrightarrow{\sigma} a_2$ iff $a_2 \in [\![\bigwedge_{b \in a_1} \delta^{\mathsf{Abs}}(b, \sigma)]\!]$ iff $a_2 \in [\![\bigwedge_{b \in a_1} \hat{\alpha}(\bigvee_{q \in b} \delta(q, \sigma))]\!]$ (by Definition 5.2.13) iff $a_2 \in \alpha([\![\bigwedge_{b \in a_1} \bigvee_{q \in b} \delta(q, \sigma)]\!])$ (by Lemma 5.2.12 and Definition 5.2.11). It is easy to see that we have $a_1 \xrightarrow{\sigma} a_2$ because $c_2 \in [\![\bigwedge_{b \in a_1} \bigvee_{q \in b} \delta(q, \sigma)]\!]$ holds since $c_2 \in [\![\bigwedge_{q \in c_1} \delta(q, \sigma)]\!]$. Now let us show that for every accepting run $c_0, c_1, \ldots, c_n$ in $\mathsf{SC}(A)$, we have that the corresponding run $a_0, a_1, \ldots, a_n$ in $\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))$ is also accepting. We know that $q_0 \in c_0$ and we can easily show that $b_0 \in a_0$ since we know that $\mathcal{P}(q_0) \in a_0$ and $\mathcal{P}(q_0) = b_0$. Finally, we must show that if $c_n \in 2^F$, then $a_n \in 2^{F^{\mathsf{Abs}}}$. This is easy to see because each block in Q^{Abs} is accepting as soon as it contains at least one accepting AFA state; thus every accepting NFA state in $\mathsf{SC}(A)$ is abstracted by an accepting NFA state in $\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))$. □

We already know from Lemma 3.3.5 that the post_σ state space operator distributes over union. We now show that, in the case of $\subseteq$-upward-closed sets of states on the subset construction of AFA, post_σ also distributes over set intersection.

Lemma 5.2.15. *Let $A = \langle Q, \Sigma, q_0, \delta, F \rangle$ be an AFA. For any $\sigma \in \Sigma$, and for any $\subseteq$-upward-closed sets $X, Y \subseteq 2^Q$, we have that $\mathsf{post}_\sigma[\mathsf{SC}(A)](X \cap Y) = \mathsf{post}_\sigma[\mathsf{SC}(A)](X) \cap \mathsf{post}_\sigma[\mathsf{SC}(A)](Y)$.*

Proof. We prove each direction in turn. The inclusion "$\supseteq$" follows easily from monotonicity of post_σ. For the case $\mathsf{post}_\sigma[A](X \cap Y) \subseteq \mathsf{post}_\sigma[A](X) \cap \mathsf{post}_\sigma[A](Y)$, we prove that:

$$\forall\, x \in X\ \forall\, y \in Y : c' \in \mathsf{post}_\sigma[A](x) \cap \mathsf{post}_\sigma[A](y) \rightarrow c' \in \mathsf{post}_\sigma[A](\uparrow x \cap \uparrow y)\ .$$

We first observe that $x \cup y \in \uparrow x \cap \uparrow y \subseteq X \cap Y$. Then,

$$\begin{aligned}\mathsf{post}_\sigma[A](x \cup y) &= \textstyle\bigwedge_{q \in x \cup y} \delta(q, \sigma) \\ &= \textstyle\bigwedge_{q \in x} \delta(q, \sigma) \wedge \bigwedge_{q \in y} \delta(q, \sigma) \\ &= [\![\textstyle\bigwedge_{q \in x} \delta(q, \sigma)]\!] \cap [\![\bigwedge_{q \in y} \delta(q, \sigma)]\!] \\ &= \mathsf{post}_\sigma[A](x) \cap \mathsf{post}_\sigma[A](y)\end{aligned}$$

We conclude from $c' \in \mathsf{post}_\sigma[A](x) \cap \mathsf{post}_\sigma[A](y)$ that $c' \in \mathsf{post}_\sigma[A](x \cup y)$, and finally that $c' \in \mathsf{post}_\sigma[A](\uparrow x \cap \uparrow y)$ since $x \cup y \in \uparrow x \cap \uparrow y$. □

The fact that post_σ distributes over union and intersection allows us to evaluate $\mathsf{post}_\sigma[\mathsf{SC}(A)]([\![\varphi]\!])$ with $\varphi \in \mathsf{BL}^+(Q)$ in a natural fashion.

Lemma 5.2.16. *For every formula $\varphi \in \mathsf{BL}^+(Q)$, $\mathsf{post}_\sigma[\mathsf{SC}(A)]([\![\varphi]\!]) = [\![\varphi']\!]$ where φ' is the formula φ where each occurrence of every variable $q \in Q$ is replaced by $\delta(q, \sigma)$.*

Proof. The result is a consequence of the definition of SC, Lemma 3.3.5, Lemma 5.2.15, and the fact that $[\![\varphi_1 \vee \varphi_2]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!]$ and $[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \cap [\![\varphi_2]\!]$. □

The following Theorem is one of the main results of this chapter and its proof makes use of most of the results stated previously. We will show that for any AFA A with set of states Q and for any partition $\mathcal{P}$ of Q, $\mathsf{post}[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))]$ and $\mathsf{pre}[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))]$ are the best abstract counterparts for $\mathcal{P}$ of $\mathsf{post}[\mathsf{SC}(A)]$ and $\mathsf{pre}[\mathsf{SC}(A)]$, respectively. This will be crucial for the practical efficiency of our algorithm, as this enables us to avoid expensive computations of α and γ. Moreover, the $\mathsf{Abs}(A, \mathcal{P})$ computation can be done in linear time and is easy to implement.

Theorem 5.2.17. *For every AFA A, and for every partition $\mathcal{P}$ of the set of states of A we have that:*

$$\alpha \circ \mathsf{post}[\mathsf{SC}(A)] \circ \gamma = \mathsf{post}[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))]$$
$$\alpha \circ \mathsf{pre}[\mathsf{SC}(A)] \circ \gamma = \mathsf{pre}[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))]$$

Proof. We prove each equality in turn.

For $\alpha \circ \mathsf{post}[\mathsf{SC}(A)] \circ \gamma = \mathsf{post}[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))]$, we will show the following:

$$a \in (\alpha \circ \mathsf{post}[\mathsf{SC}(A)] \circ \gamma)(X) \text{ iff } a \in \mathsf{post}[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))](X) \ .$$

$a \in \mathsf{post}[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))](X)$

iff $a \in \bigcup_{\sigma \in \Sigma} \mathsf{post}_\sigma[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))](X)$ — definition of post

iff $a \in \bigcup_{\sigma \in \Sigma} \left\{ a_2 \;\middle|\; \exists\, a_1 \in X : a_1 \xrightarrow{\sigma} a_2 \right\}$ — definition of post_σ

iff $a \in \bigcup_{\sigma \in \Sigma} \left\{ a_2 \;\middle|\; \exists\, a_1 \in X : a_2 \in [\![\bigwedge_{b \in a_1} \delta^{\mathsf{Abs}}(b, \sigma)]\!] \right\}$ — definition of SC

iff $a \in \bigcup_{\sigma \in \Sigma} \left\{ a_2 \;\middle|\; \exists\, a_1 \in X : a_2 \in [\![\bigwedge_{b \in a_1} \hat{\alpha}(\bigvee_{q \in b} \delta(q, \sigma))]\!] \right\}$ — definition of δ^{Abs}

iff $a \in \bigcup_{\sigma \in \Sigma} \left\{ a_2 \;\middle|\; \exists\, a_1 \in X : a_2 \in [\![\hat{\alpha}(\bigwedge_{b \in a_1} \bigvee_{q \in b} \delta(q, \sigma))]\!] \right\}$ — Definition 5.2.11

iff $a \in \bigcup_{\sigma \in \Sigma} \left\{ a_2 \;\middle|\; \exists\, a_1 \in X : a_2 \in \alpha([\![\bigwedge_{b \in a_1} \bigvee_{q \in b} \delta(q, \sigma)]\!]) \right\}$ — Lemma 5.2.12

iff $a \in \bigcup_{\sigma \in \Sigma} \left\{ a_2 \;\middle|\; \exists\, a_1 \in X : a_2 \in \alpha \circ \mathsf{post}_\sigma[\mathsf{SC}(A)]([\![\bigwedge_{b \in a_1} \bigvee_{q \in b} q]\!]) \right\}$ — Lemma 5.2.16

iff $a \in \bigcup_{\sigma \in \Sigma} \left\{ a_2 \;\middle|\; \exists\, a_1 \in X : a_2 \in \alpha \circ \mathsf{post}_\sigma[\mathsf{SC}(A)]([\![\hat{\gamma}(\bigwedge_{b \in a_1} b)]\!]) \right\}$ — Definition 5.2.11

iff $a \in \bigcup_{\sigma \in \Sigma} \left\{ a_2 \;\middle|\; \exists\, a_1 \in X : a_2 \in \alpha \circ \mathsf{post}_\sigma[\mathsf{SC}(A)] \circ \gamma(\uparrow \{a_1\}) \right\}$ — Lemma 5.2.12

iff $a \in \bigcup_{\sigma \in \Sigma} \{ a_2 \mid a_2 \in \alpha \circ \mathsf{post}_\sigma[\mathsf{SC}(A)] \circ \gamma(\uparrow X) \}$

iff $a \in \bigcup_{\sigma \in \Sigma} \{ a_2 \mid a_2 \in \alpha \circ \mathsf{post}_\sigma[\mathsf{SC}(A)] \circ \uparrow \circ \gamma(X) \}$ — Corollary 5.2.10

iff $a \in \bigcup_{\sigma \in \Sigma} \alpha \circ \mathsf{post}_\sigma[\mathsf{SC}(A)] \circ \gamma(X)$ — Lemma 3.3.8

iff $a \in \alpha \circ \mathsf{post}[\mathsf{SC}(A)] \circ \gamma(X)$ — definition of post

For $\alpha \circ \mathsf{pre}[\mathsf{SC}(A)] \circ \gamma = \mathsf{pre}[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))]$, we will show the following:

$$\exists\, \sigma \in \Sigma \colon a \in (\alpha \circ \mathsf{pre}_\sigma[\mathsf{SC}(A)] \circ \gamma)(X) \text{ iff } a \in \mathsf{pre}_\sigma[\mathsf{SC}(\mathsf{Abs}(A, \mathcal{P}))](X) \ .$$

$a \in \mathsf{pre}_\sigma[\mathsf{SC}(\mathsf{Abs}(A,\mathcal{P}))](X)$

iff $a \in \left\{ a_1 \;\middle|\; \exists\, a_2 \in X\colon a_1 \xrightarrow{\sigma} a_2 \right\}$ definition of pre_σ

iff $a \in \left\{ a_1 \;\middle|\; \exists\, a_2 \in X\colon a_2 \in [\![\bigwedge_{b\in a_1} \delta^{\mathsf{Abs}}(b,\sigma)]\!] \right\}$ definition of SC

iff $a \in \left\{ a_1 \;\middle|\; \exists\, a_2 \in X\colon a_2 \in [\![\bigwedge_{b\in a_1} \hat{\alpha}(\bigvee_{q\in b} \delta(q,\sigma))]\!] \right\}$ definition of δ^{Abs}

iff $a \in \left\{ a_1 \;\middle|\; \exists\, a_2 \in X\colon a_2 \in [\![\hat{\alpha}(\bigwedge_{b\in a_1} \bigvee_{q\in b} \delta(q,\sigma))]\!] \right\}$ Definition 5.2.11

iff $a \in \left\{ a_1 \;\middle|\; \exists\, a_2 \in X\colon a_2 \in \alpha([\![\bigwedge_{b\in a_1} \bigvee_{q\in b} \delta(q,\sigma)]\!]) \right\}$ Lemma 5.2.12

iff $a \in \left\{ a_1 \;\middle|\; \exists\, a_2 \in X\, \exists\, c\colon a_2 = \alpha(c) \wedge c \in [\![\bigwedge_{b\in a_1} \bigvee_{q\in b} \delta(q,\sigma)]\!] \right\}$

iff $a \in \left\{ a_1 \;\middle|\; \exists\, a_2 \in X\, \exists\, c,c'\colon a_2 = \alpha(c) \wedge c' \in \gamma(a_1) \wedge c \in [\![\bigwedge_{q\in c'} \delta(q,\sigma)]\!] \right\}$ definition of γ

iff $a \in \left\{ a_1 \;\middle|\; \exists\, a_2 \in X\, \exists\, c,c'\colon a_2 = \alpha(c) \wedge c' \in \gamma(a_1) \wedge c \xrightarrow{\sigma} c' \right\}$ definition of SC

iff $a \in \left\{ a_1 \;\middle|\; \exists\, a_2 \in X\, \exists\, c,c'\colon c \in \gamma(a_2) \wedge \alpha(c') = a_1 \wedge c \xrightarrow{\sigma} c' \right\}$ definition of α, γ

iff $a \in \{ a_1 \mid \exists\, a_2 \in X\, \exists\, c,c'\colon c \in \gamma(a_2) \wedge \alpha(c') = a_1 \wedge c' \in \mathsf{pre}_\sigma[\mathsf{SC}(A)](c) \}$ definition of pre

iff $a \in \{ a_1 \mid a_1 \in \alpha \circ \mathsf{pre}_\sigma[\mathsf{SC}(A)] \circ \gamma(X) \}$

iff $a \in \alpha \circ \mathsf{pre}_\sigma[\mathsf{SC}(A)] \circ \gamma(X)$

□

5.2.4 Precision of the abstract domain

We now present some results about precision and representability in our family of abstract domains. In particular, for the automatic refinement of abstract domains, we will need an effective way of computing the *coarsest partition* which can represent an upward- or downward-closed set of states without loss of precision. In the sequel, when a set of concrete states X is exactly representable in the abstract domain $2^{2^{\mathcal{P}}}$, we shall simply say that $\mathcal{P}$ can exactly represent X.

Recall from Remark 5.2.6 that $\mu(X) \stackrel{\mathsf{def}}{=} \gamma(\alpha(X))$.

Definition 5.2.18. *Let $c, c' \in 2^Q$ and $\mathcal{P}$ a partition of Q. We write $c \equiv_{\mathcal{P}} c'$ iff $\{\mathcal{P}(q) \mid q \in c\} = \{\mathcal{P}(q) \mid q \in c'\}$.*

Lemma 5.2.19. *Let $X \subseteq 2^Q$, $\mathcal{P}$ a partition of Q, and $c \in X$. The following equality holds:* $\mu_{\mathcal{P}}(X) = \{c' \mid \exists\, c \in X \colon c \equiv_{\mathcal{P}} c'\}$.

Proof. $\boxed{\supseteq}$ Let $c \in X$, $c \equiv_{\mathcal{P}} c'$, and a be the unique abstract state such that $\mathsf{Covers}(a, c)$. From the definition of $\equiv_{\mathcal{P}}$, it is clear that we also have $\mathsf{Covers}(a, c')$. Thus we have $c' \in \gamma(\{a\})$ with $a \in \alpha(\{c\})$ and $c \in X$ which, by monotonicity of α and γ, implies that $c' \in \mu_{\mathcal{P}}(X)$. $\boxed{\subseteq}$ Let $c' \in \mu_{\mathcal{P}}(X)$ and a be the unique abstract state such that $\mathsf{Covers}(a, c')$. Since $c' \in \gamma(\alpha(X))$, we know that $a \in \alpha(X)$. By definition of α, we know that there exists $c \in X$ with $\mathsf{Covers}(a, c)$, which implies that $c \equiv_{\mathcal{P}} c'$. □

The following lemma states that if a set is representable exactly by two partitions, then it is representable exactly by their least upper bound. This will immediately imply that, for each set X, there exists a unique coarsest partition which represents exactly X.

Lemma 5.2.20. *Let $X \subseteq 2^Q$, $\mathcal{P}_1, \mathcal{P}_2$ two partitions of Q and $\mathcal{P} = \mathcal{P}_1 \curlyvee \mathcal{P}_2$. If $\mu_{\mathcal{P}_1}(X) = X$ and $\mu_{\mathcal{P}_2}(X) = X$, then $\mu_{\mathcal{P}}(X) = X$.*

Proof. By hypothesis, we have that $\mathcal{P} = \mathcal{P}_1 \curlyvee \mathcal{P}_2$. By properties of the least upper bound of two partitions (see [BS81]), we have that for every $q, q' \in Q$: $\{q\} \equiv_{\mathcal{P}} \{q'\}$ iff

$$\exists\, q_0, \ldots, q_n \colon (q = q_0) \wedge (q' = q_n) \wedge \{q_0\} \equiv_1 \{q_1\} \equiv_2 \{q_2\} \cdots \{q_{n-1}\} \equiv_n \{q_n\}$$

where $\equiv_i \in \{\equiv_{\mathcal{P}_1}, \equiv_{\mathcal{P}_2}\}$. Now we generalize the above results from Q to 2^Q:

$$\begin{aligned}
& c \equiv_{\mathcal{P}} c' \\
& \text{iff } \{\mathcal{P}(q) \mid q \in c\} = \{\mathcal{P}(q') \mid q' \in c'\} && \text{definition of } \equiv_{\mathcal{P}} \\
& \text{iff } \forall\, q \in c\ \exists\, q' \in c' \colon \mathcal{P}(q) = \mathcal{P}(q') \\
& \quad \wedge \forall q' \in c'\ \exists\, q \in c \colon \mathcal{P}(q) = \mathcal{P}(q') \\
& \text{iff } \forall\, q \in c\ \exists\, q' \in c' \colon \{q\} \equiv_{\mathcal{P}} \{q'\} \\
& \quad \wedge \forall q' \in c'\ \exists\, q \in c \colon \{q\} \equiv_{\mathcal{P}} \{q'\} && \text{definition of } \equiv_{\mathcal{P}}
\end{aligned}$$

Note that $c \equiv_{\mathcal{P}} c'$ is equivalent to $\alpha_{\mathcal{P}}(c) = \alpha_{\mathcal{P}}(c')$. Hence, by using the above result, we obtain that for every $c, c' \in 2^Q$:

$$c \equiv_{\mathcal{P}} c' \text{ iff } \exists\, c_0, \ldots, c_n \colon (c = c_0) \wedge (c' = c_n) \wedge c_0 \equiv_1 c_1 \equiv_2 c_2 \cdots c_{n-1} \equiv_n c_n$$

where $\equiv_i \in \{\equiv_{\mathcal{P}_1}, \equiv_{\mathcal{P}_2}\}$. We now prove that $\mu_{\mathcal{P}_2}(X) \subseteq X$ and $\mu_{\mathcal{P}_1}(X) \subseteq X$ implies $\mu_{\mathcal{P}}(X) \subseteq X$. Note that the reverse inclusions follow directly from properties of Galois insertions. From $\mu_{\mathcal{P}_1}(X) \subseteq X$ and $\mu_{\mathcal{P}_2}(X) \subseteq X$, we conclude that $\forall\, c \in X\, \forall\, c' \colon (c \equiv_{\mathcal{P}_1} c') \Rightarrow c' \in X$ and $\forall\, c \in X\, \forall\, c' \colon (c \equiv_{\mathcal{P}_2} c') \Rightarrow c' \in X$, respectively, hence that $\forall\, c \in X\, \forall\, c' \colon (c \equiv_{\mathcal{P}} c') \Rightarrow c' \in X$ by the above, and finally that $\mu_{\mathcal{P}}(X) \subseteq X$.

□

As the lattice of partitions is complete, we have the following corollary.

Corollary 5.2.21. *For all $X \subseteq 2^Q$, $\mathcal{P} = \curlyvee\{\mathcal{P}' \mid \mu_{\mathcal{P}'}(X) = X\}$ is the coarsest partition such that $\mu_{\mathcal{P}}(X) = X$.*

For upward- and downward-closed sets, we have an efficient way to compute this coarsest partition. We start with upward-closed sets. To obtain an algorithm, we use the notion of *neighbor list*. The neighbor list of an AFA state q with respect to an upward-closed set X, denoted $\mathcal{N}_X(q)$, is the set of states containing q in $\lfloor X \rfloor$, from which q has been removed.

Definition 5.2.22. *Let $X \subseteq 2^Q$ be an upward-closed set. The* neighbor list *of an* AFA *state $q \in Q$ w.r.t. X is the set $\mathcal{N}_X(q) = \{c \backslash \{q\} \mid c \in \lfloor X \rfloor, q \in c\}$.*

The following lemma states that if two AFA states share the same neighbor lists with respect to an upward-closed set X, then they can be put in the same partition block and preserve the representability of X. Conversely, X cannot be represented exactly by any partition which puts into the same block two AFA states that have different neighbor lists.

Lemma 5.2.23. *Let $\mathcal{P}$ be a partition of Q and X be an upward-closed set, $\mu_{\mathcal{P}}(X) = X$ iff $\forall\, q, q' \in Q \colon \mathcal{P}(q) = \mathcal{P}(q') \Rightarrow \mathcal{N}_X(q) = \mathcal{N}_X(q')$.*

Proof. $\boxed{\Rightarrow}$ By contradiction, we have that $\mu_{\mathcal{P}}(X) = X$, and for some q, q', we have $\mathcal{P}(q) = \mathcal{P}(q')$ but $\mathcal{N}_X(q) \neq \mathcal{N}_X(q')$. Without loss of generality, we consider that $s \in \mathcal{N}_X(q)$ and $s \notin \mathcal{N}_X(q')$. By the definition of $\mathcal{N}_X$, we know that $s \cup \{q\} \in \lfloor X \rfloor$, $s \cup \{q'\} \notin \lfloor X \rfloor$. By Lemma 5.2.19 and as $q \equiv_{\mathcal{P}} q'$, we also know that $(s \cup \{q'\}) \equiv_{\mathcal{P}} (s \cup \{q\})$, and so $s \cup \{q'\} \in \mu_{\mathcal{P}}(X)$. If $s \cup \{q'\} \notin X$, then $X \neq \mu_{\mathcal{P}}(X)$ and we have a contradiction. Otherwise, because $s \cup \{q'\} \in X$ but $s \cup \{q'\} \notin \lfloor X \rfloor$, there must exist a smaller state

$c \subset s \cup \{q'\}$ with $c \in \lfloor X \rfloor$. Two cases remain, both of which lead to a contradiction. First, if $q' \notin c$, then $c \subseteq s$, but then $s \cup \{q\}$ could not have appeared in $\lfloor X \rfloor$. Finally, if $q' \in c$, then $c = s' \cup \{q'\}$ with $s' \subset s$. As $q \equiv_{\mathcal{P}} q'$, we know that $s' \cup \{q\} \in \mu_{\mathcal{P}}(X)$, but $s' \cup \{q\}$ cannot be in X because $s \cup \{q\} \in \lfloor X \rfloor$.

$\boxed{\leftarrow}$ First, let us consider the particular case of partition $\mathcal{P}_{q,q'}$ where only q and q' share a block, i.e. $\mathcal{P}_{q,q'} = \{\{q, q'\}\} \cup \{\{q''\} \mid q'' \notin \{q, q'\}\}$. Assume that $\mathcal{N}_X(q) = \mathcal{N}_X(q')$. We will show that X is representable exactly in that partition. By definition of abstraction, and concretization functions, and their algebraic properties, $\mu_{\mathcal{P}_{q,q'}}(X) = \uparrow \cup_{x \in \lfloor X \rfloor} \mu_{\mathcal{P}_{q,q'}}(x)$. Let us consider $\mu_{\mathcal{P}_{q,q'}}(x)$. There are two cases, either (i) $x \cap \{q, q'\} = \varnothing$, then clearly $\mu_{\mathcal{P}_{q,q'}}(\{x\}) = \{\{x\}\}$, or (ii) $x \cap \{q, q'\} \neq \varnothing$, then there are a priori three cases to consider: (a) $q \in x, q' \in x$, (b) $q \in x, q' \notin x$, (c) $q \notin x, q' \in x$. Case (a) is not possible as $\mathcal{N}_X(q) = \mathcal{N}_X(q')$, and so q and q' cannot appear together in a set in $\lfloor X \rfloor$. Case (b), $\mu_{\mathcal{P}_{q,q'}}(\{x\}) = \{(x \setminus \{q\}) \cup \{q'\}, x\}$, and let us show that $(x \setminus \{q\}) \cup \{q'\}$ belongs to $\lfloor X \rfloor$. Assume it is not the case, so $x \setminus \{q\} \notin \mathcal{N}_X(q')$, but as $x \setminus \{q\} \in \mathcal{N}_X(q)$, we obtain a contradiction with our hypothesis that $\mathcal{N}_X(q) = \mathcal{N}_X(q')$. Case (c) is symmetrical to case (b). And so, we have established that X is representable exactly in $\mathcal{P}_{q,q'}$. Let us now turn to the general case. Let $\mathcal{P}$ be a partition such that $\forall q, q' \in Q \colon \mathcal{P}(q) = \mathcal{P}(q') \Rightarrow \mathcal{N}_X(q) = \mathcal{N}_X(q')$. One can easily check that $\mathcal{P} = \curlyvee_{\{q,q' \mid \mathcal{P}(q) = \mathcal{P}(q')\}} \mathcal{P}_{q,q'}$, because the least upper-bound on partitions has the effect of merging blocks which overlap by at least one element. By the previous result and by Lemma 5.2.20, we have that X is representable exactly in $\mathcal{P}$. □

Lemmas 5.2.20 and 5.2.23 yield a natural algorithm for computing the *coarsest partition which can represent an upward-closed set X exactly.* In fact, computing the neighbor list w.r.t. X for each element of Q suffices to compute the coarsest partition which can represent X exactly.

Corollary 5.2.24. *Let $X \subseteq 2^Q$ be an upward-closed set. Then the partition $\mathcal{P}$ induced by the equivalence relation $q \sim q'$ iff $\mathcal{N}_X(q) = \mathcal{N}_X(q')$ is the coarsest partition such that $\mu_{\mathcal{P}}(X) = X$.*

Example 5.2.25. *Let $Q = \{1, \ldots, 7\}$ be a set of* AFA *states, and let X be an upward-closed set such that $\lfloor X \rfloor = \{\{1, 2, 4\}, \{1, 3, 4\}, \{2, 4, 5\}, \{3, 4, 5\}\}$.*

We have that $\mathcal{N}_X(1) = \mathcal{N}_X(5) = \{\{2,4\},\{3,4\}\}$, $\mathcal{N}_X(2) = \mathcal{N}_X(3) = \{\{1,4\},\{4,5\}\}$, *and* $\mathcal{N}_X(6) = \mathcal{N}_X(7) = \{\}$. *The coarsest partition which can represent* X *exactly is the partition* $\mathcal{P} = \{b_{1,5} : \{1,5\}, b_{2,3} : \{2,3\}, b_4 : \{4\}, b_{6,7} : \{6,7\}\}$. *The representation of* X *by* $\mathcal{P}$ *is such that* $\lfloor \alpha_{\mathcal{P}}(X) \rfloor = \{\{b_{1,5}, b_{2,3}, b_4\}\}$.

The representability of downward-closed sets follows immediately from Lemma 5.2.26. In practice, we simply compute the coarsest partition for the complementary upward-closed set.

Lemma 5.2.26. *Let* $X \subseteq 2^Q$, *and let* $\mathcal{P}$ *be a partition of* Q. *We have that* $\mu_{\mathcal{P}}(X) = X$ *iff* $\mu_{\mathcal{P}}(\overline{X}) = \overline{X}$.

Proof. Because for every $Y \subseteq 2^Q$, we have $\mu_{\mathcal{P}}(Y) \supseteq Y$ (by property of Galois connections, see Lemma 2.1.9) it is enough to show that for every $X \subseteq 2^Q$, $\mu_{\mathcal{P}}(X) \subseteq X$ implies $\mu_{\mathcal{P}}(\overline{X}) \subseteq \overline{X}$. We conclude from Lemma 5.2.19 that $\mu_{\mathcal{P}}(X) \subseteq X$ iff

$$\forall\, c \in X\, \forall\, c' \colon c \equiv_{\mathcal{P}} c' \Rightarrow c' \in X \ . \tag{5.2}$$

On the other hand, $\mu_{\mathcal{P}}(\overline{X}) \subseteq \overline{X}$ is equivalent to $\mu_{\mathcal{P}}(\overline{X}) \cap X = \varnothing$. Assuming that this equality does not hold, we find that there is $c \notin X$ and $c' \in X$ such that $c \equiv_{\mathcal{P}} c'$ by definition of $\mu_{\mathcal{P}}$, which contradicts (5.2), hence the result. □

5.3 Abstraction Refinement Algorithm

This section presents two fixed point-guided abstraction refinement algorithms for AFA. These algorithms share several ideas with the generic algorithm presented in [CGR07] but they are formally different because they do not strictly refine the abstract domain at each refinement step, but always use the most abstract domain instead.

We concentrate here on explanations related to the abstract forward algorithm. The abstract backward algorithm is the dual of this algorithm, hence proofs are similar to the forward case. We first give an informal presentation of the ideas underlying the algorithm and then we give proofs for its soundness and completeness.

Algorithm 5.1: The *abstract forward* algorithm.

Input: $A = \langle Q, \Sigma, q_0, \delta, F \rangle$
Output: true iff $L(A) = \varnothing$

```
1  begin ForwardFGAR(A)
2      P_0 ← {F, Q \ F} ;
3      Z_0 ← 2^F ;
4      for i in 0, 1, 2, ... do
5          if [[q_0]] ⊈ Z_i then
6              return false ;
7          A_i^Abs ← Abs(A, P_i) ;
8          let A_i^Abs = ⟨Q^Abs, Σ, b_0, δ^Abs, F^Abs⟩ ;
9          R_i ← LFP(λ x : ([[b_0]] ∩ α(Z_i)) ∪ (post[SC(A_i^Abs)](x) ∩ α_{P_i}(Z_i)))
           ;
10         if post[SC(A_i^Abs)](R_i) ⊆ α_{P_i}(Z_i) then
11             return true ;
12         Z_{i+1} ← γ_{P_i}(R_i) ∩ cpre[A](γ_{P_i}(R_i)) ;
13         P_{i+1} ← ϒ {P | μ_P(Z_{i+1}) = Z_{i+1}} ;
14 end
```

5.3.1 Abstract Forward Algorithm

The abstract forward algorithm is depicted in Algorithm 5.1. The most important information computed in the abstract forward algorithm is Z_i, which is an over-approximation of the set of reachable states which cannot reach an accepting state in i steps or less. In other words, all the states outside Z_i are either unreachable, or can lead to an accepting state in i steps or less (or both). Our algorithm always uses the coarsest partition $\mathcal{P}_i$ that allows Z_i to be represented exactly in the abstract domain $\langle 2^{2^{\mathcal{P}_i}}, \subseteq \rangle$. The algorithm begins by initializing Z_0 with the set of non-accepting states and by initializing $\mathcal{P}_0$ accordingly (lines 2 and 3). The main loop proceeds as follows. First, we check whether an initial state (i.e., a state containing q_0) is no longer in Z_i in which case we know that it can lead to an accepting state in i steps or less (as it is obviously reachable) and we conclude that

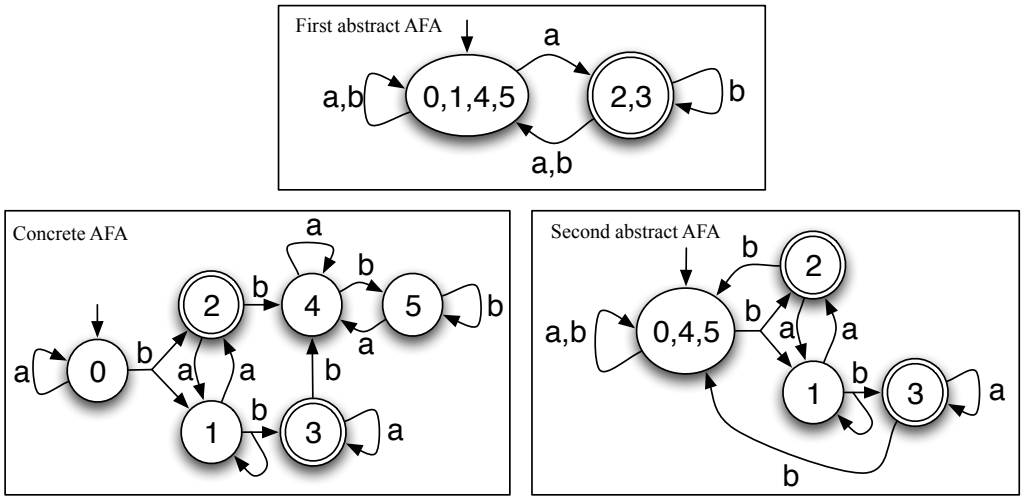


Figure 5.1: Illustration of the *abstract forward* algorithm. One refinement step suffices to show emptiness.

the automaton is non-empty (line 5). Note that, for Z_0, this amounts to checking for the existence of an initial accepting state. Next, we compute the abstract reachable states R_i which are within Z_i, which is done by applying the Abs transformation using $\mathcal{P}_i$ (lines 7-8), and by computing an abstract forward fixed point (line 9). If R_i does not contain a state which can leave Z_i, we know (as we will formally prove later in this section) that the automaton is empty (line 10). If this second test failed, we *refine* the information contained in Z_i by removing all the states which can leave R_i in one step, as we know that these states are either surely unreachable or can lead to an accepting state in $i+1$ steps or less. Finally, the current abstract domain is changed to be able to represent the new Z_i exactly (line 13), using the neighbor list algorithm of Corollary 5.2.24. It is important to note that this refinement operation is not the traditional refinement used in counter-example guided abstraction refinement. Note also that our algorithm does not necessarily choose a new abstract domain that is strictly more precise than the previous one as in [CGR07]. Instead, the algorithm uses the most abstract domain possible at all times.

An illustration of the abstract forward algorithm is depicted at Fig-

ure 5.1. The algorithm begins by computing $\mathcal{P}_0 = \{[0,1,4,5],[2,3]\}$ and $Z_0 = \uparrow\{\{0\},\{1\},\{4\},\{5\}\}$. Since $\uparrow\{\{0\}\} \subseteq Z_0$ the algorithm does not return false. Next, the algorithm computes $A_0^{\mathsf{Abs}} = \mathsf{Abs}(A, \mathcal{P}_0)$, which is depicted by the topmost automaton of Figure 5.1. The abstract computation of R_0 is performed as follows:

$$\begin{aligned}
\alpha_{\mathcal{P}_0}(Z_0) &= \uparrow\{\{[0,1,4,5]\}\} \\
[\![b_0]\!] \cap \alpha_{\mathcal{P}_0}(Z_0) &= \uparrow\{\{[0,1,4,5]\}\} \\
\mathsf{post}(\uparrow\{\{[0,1,4,5]\}\}) &= \uparrow\{\{[0,1,4,5]\},\{[2,3]\}\} \\
\mathsf{post}(\uparrow\{\{[0,1,4,5]\}\}) \cap \alpha_{\mathcal{P}_0}(Z_0) &= \uparrow\{\{[0,1,4,5]\}\}
\end{aligned}$$

The result of the fixed point computation is therefore $R_0 = \uparrow\{\{[0,1,4,5]\}\}$. Since $\mathsf{post}(R_0) \not\subseteq \alpha_{\mathcal{P}_0}(Z_0)$ the algorithm does not return true. Next, the algorithm computes $\mathsf{cpre}(\gamma(R_0)) = \uparrow\{\{0\},\{2\},\{4\},\{5\}\}$ which means that:

$$Z_1 = \uparrow\{\{0\},\{1\},\{4\},\{5\}\} \cap \uparrow\{\{0\},\{2\},\{4\},\{5\}\} = \uparrow\{\{0\},\{1,2\},\{4\},\{5\}\}$$

In other words, we have just removed the state $\{1\}$ from Z_0 to obtain Z_1, since it is the only NFA state in Z_0 which can reach an accepting state in 1 step (by reading a and going to $\{2\}$). To finish the first iteration, the partition $\mathcal{P}_1$ is computed with the neighbor list algorithm of Corollary 5.2.24, as follows:

$$\begin{aligned}
\mathcal{N}_{\lfloor Z_1 \rfloor}(0) &= \{\varnothing\} & \mathcal{N}_{\lfloor Z_1 \rfloor}(1) &= \{2\} \\
\mathcal{N}_{\lfloor Z_1 \rfloor}(2) &= \{1\} & \mathcal{N}_{\lfloor Z_1 \rfloor}(3) &= \{\varnothing\} \\
\mathcal{N}_{\lfloor Z_1 \rfloor}(4) &= \{\varnothing\} & \mathcal{N}_{\lfloor Z_1 \rfloor}(5) &= \varnothing
\end{aligned}$$

Therefore we have that $\mathcal{P}_1 = \{[0,4,5],[1],[2],[3]\}$. In the second iteration, we still have that $\uparrow\{\{0\}\} \subseteq Z_1$, so the algorithm does not return false. Next, the algorithm computes $A_1^{\mathsf{Abs}} = \mathsf{Abs}(A, \mathcal{P}_1)$, which is depicted by the rightmost automaton of Figure 5.1. The abstract fixed point computation (left to the reader) leads to $R_1 = \uparrow\{\{[0,4,5]\},\{[1],[2]\}\}$. Finally, the algorithm discovers that $\mathsf{post}(\uparrow\{\{[0,4,5]\},\{[1],[2]\}\}) \subseteq \alpha_{\mathcal{P}_1}(Z_1)$ and therefore returns true (that is, the AFA is empty).

The soundness and completeness of Algorithm 5.1 rely on the properties formalized in the following lemma.

Lemma 5.3.1. *Let* $\mathsf{Reach} = \mathsf{LFP}\big(\lambda X : [\![q_0]\!] \cup \mathsf{post}[\mathsf{SC}(A)](X)\big)$ *be the reachable states of* A*, let* $\mathsf{Bad}^k = \cup_{j=0}^{k} \mathsf{pre}^j[\mathsf{SC}(A)](2^F)$ *be the states that can reach an accepting state in* k *steps or less. The following four properties hold:*

1. $\forall i \geq 0$: $\mu_{\mathcal{P}_i}(Z_i) = Z_i$*, i.e.* Z_i *is representable exactly in* $\mathcal{P}_i$*;*
2. $\forall i \geq 0$: $Z_{i+1} \subseteq Z_i$*, i.e. the sets* Z_i *are decreasing;*
3. $\forall i \geq 0$: $\mathsf{Reach} \setminus \mathsf{Bad}^i \subseteq Z_i$*, i.e.* Z_i *over-approximates the reachable states that cannot reach an accepting state in* i *steps or less;*
4. $\forall i \geq 0$ *if* $Z_i = Z_{i+1}$*, then* $\mathsf{post}[\mathsf{SC}(A_i^{\mathsf{Abs}})](R_i) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$.

Proof. We prove each point in turn. Point 1 is straightforward as $\mathcal{P}_0$ is chosen in line 1 to be able to represent Z_0 exactly, and $\mathcal{P}_{i+1}$ is chosen in line 13 to be able to represent Z_{i+1} exactly. Point 2 follows directly from the fact that $R_i \subseteq \alpha_{\mathcal{P}_i}(Z_i)$, Z_i is representable exactly in $\mathcal{P}_i$ by the previous point, and the definition of Z_{i+1} in line 12. Point 3 is established by induction. The property is clearly true for Z_0. Let us establish it for Z_{i+1} using the induction hypothesis that it is true for Z_i. By soundness of the theory of abstract interpretation, we know that in line 7 we compute a set R_i which over-approximates the set $\mathsf{Reach} \setminus \mathsf{Bad}^i$. In line 12, we remove states that can leave this set in one step, so Z_{i+1} is an over-approximation of the reachable states that cannot reach an accepting state in $i+1$ steps or less, i.e. $\mathsf{Reach} \setminus \mathsf{Bad}^{i+1} \subseteq Z_{i+1}$, which concludes the proof. Point 4 is established as follows. If $Z_i = Z_{i+1}$, then clearly $\mathsf{post}[\mathsf{SC}(A)](\gamma_{\mathcal{P}_i}(R_i)) \subseteq \gamma_{\mathcal{P}_i}(R_i)$ as no state can leave $\gamma_{\mathcal{P}_i}(R_i)$ in one step (from line 12). Then $\gamma_{\mathcal{P}_i}(R_i) \subseteq Z_i$ shows that $\mathsf{post}[\mathsf{SC}(A)](\gamma_{\mathcal{P}_i}(R_i)) \subseteq Z_i$. Finally, we conclude from monotonicity of $\alpha_{\mathcal{P}_i}$ (itself a consequence of the Galois connection, see Lemma 5.2.5) that $\alpha_{\mathcal{P}_i}(\mathsf{post}[\mathsf{SC}(A)](\gamma_{\mathcal{P}_i}(R_i))) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$. This is equivalent to $\mathsf{post}[\mathsf{SC}(A_i^{\mathsf{Abs}})](R_i) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$ by Theorem 5.2.17. □

We now establish the soundness and completeness of Algorithm 5.1.

Theorem 5.3.2. *The abstract forward algorithm with refinement is sound and complete to decide the emptiness of* AFA*.*

Proof. Let A be the AFA on which the algorithm is executed. First, let us show that the algorithm is sound. Assume that the algorithm returns

true. In this case, the test of line 8 evaluates to true which implies that $\mathsf{post}[\mathsf{SC}(A_i^{\mathsf{Abs}})](R_i) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$, hence $\alpha_{\mathcal{P}_i} \circ \mathsf{post}[\mathsf{SC}(A)] \circ \gamma_{\mathcal{P}_i}(R_i) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$ by Theorem 5.2.17 and then that $\mathsf{post}[\mathsf{SC}(A)](\gamma_{\mathcal{P}_i}(R_i)) \subseteq \gamma_{\mathcal{P}_i} \circ \alpha_{\mathcal{P}_i}(Z_i)$ by Galois connection and finally that $\mathsf{post}[\mathsf{SC}(A)](\gamma_{\mathcal{P}_i}(R_i)) \subseteq Z_i$ by Point 1 of Lemma 5.3.1. Because $\gamma_{\mathcal{P}_i}(R_i)$ is an over-approximation of the concrete reachable states and as $\mathsf{post}[\mathsf{SC}(A)](\gamma_{\mathcal{P}_i}(R_i)) \subseteq Z_i \subseteq \overline{2^F}$, we know that all the accepting states are unreachable, hence $L(A) = \varnothing$. Now, assume that the algorithm returns false. Then $[\![q_0]\!] \not\subseteq Z_i$ which means that q_0 is able to reach an accepting state in i steps or less. Since q_0 is obviously reachable, we can conclude that $L(A) \neq \varnothing$. To prove the completeness of the algorithm, we only need to establish its termination. Suppose it does not terminate, point 2 and 4 of Lemma 5.3.1 show that the chain of Z_i is strictly descending, namely $Z_{i+1} \subset Z_i$. So, if the algorithm does not terminate, it means that we have an infinite strictly decreasing chain of values in 2^Q which contradicts that Q is a finite set. Hence the algorithm terminates. □

5.3.2 Abstract Backward Algorithm

Let us turn to the backward algorithm which is the dual of the forward algorithm. Lemma 5.3.3, which is the dual of Lemma 5.3.1, is the central argument to prove termination and correctness of the algorithm. We state the result without providing a proof which can be obtained easily by dualizing the proof of Lemma 5.3.1.

Lemma 5.3.3. *Let* $\mathsf{Success} = \mathsf{LFP}\big(\lambda\, X : 2^F \cup \mathsf{pre}[\mathsf{SC}(A)](X)\big)$ *be the states that reach a state of* 2^F*, let* $\mathsf{Reach}^k = \cup_{j=0}^{k}\ \mathsf{post}^j[\mathsf{SC}(A)]([\![q_0]\!])$ *be the states reachable in* k *steps or less. The following four properties hold:*

1. $\forall i \geq 0 \colon \mu_{\mathcal{P}_i}(Z_i) = Z_i$*, i.e.* Z_i *is representable exactly in* $\mathcal{P}_i$*;*
2. $\forall i \geq 0 \colon Z_{i+1} \subseteq Z_i$*, i.e. the sets* Z_i *are decreasing;*
3. $\forall i \geq 0 \colon \mathsf{Success} \setminus \mathsf{Reach}^i \subseteq Z_i$*, i.e.* Z_i *over-approximates the states that can reach an accepting state and that are unreachable from* $[\![q_0]\!]$ *in* i *steps or less;*
4. $\forall i \geq 0$ *if* $Z_i = Z_{i+1}$*, then* $\mathsf{pre}[\mathsf{SC}(A_i^{\mathsf{Abs}})](R_i) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$.

Let us now turn to the soundness and completeness of Algorithm 5.2.

Algorithm 5.2: The *abstract backward* algorithm.

Input: $A = \langle Q, \Sigma, q_0, \delta, F \rangle$
Output: true iff $L(A) = \varnothing$

```
1  begin BackwardFGAR(A)
2      P_0 ← {{q_0}, Q \ {q_0}} ;
3      Z_0 ← ⟦q_0⟧‾ ;
4      for i in 0, 1, 2, ... do
5          if 2^F ⊈ Z_i then
6              return false ;
7          A_i^Abs ← Abs(A, P_i) ;
8          let A_i^Abs = ⟨Q^Abs, Σ, b_0, δ^Abs, F^Abs⟩ ;
9          R_i ← LFP(λ x : (2^F^Abs ∩ α(Z_i)) ∪ (pre[SC(A_i^Abs)](x) ∩ α_{P_i}(Z_i)))
           ;
10         if pre[SC(A_i^Abs)](R_i) ⊆ α_{P_i}(Z_i) then
11             return true ;
12         Z_{i+1} ← γ_{P_i}(R_i) ∩ cpost[A](γ_{P_i}(R_i)) ;
13         P_{i+1} ← ϒ {P | μ_P(Z_{i+1}) = Z_{i+1}} ;
14 end
```

Theorem 5.3.4. *The abstract backward algorithm with refinement is sound and complete to decide the emptiness of* AFA.

Proof. Let A be the AFA on which the algorithm is executed. First, let us show that the algorithm is sound. Assume that the algorithm returns true. In this case, the test of line 8 evaluates to true which implies that $\mathsf{pre}[\mathsf{SC}(A_i^{\mathsf{Abs}})](R_i) \subseteq \alpha_{\mathcal{P}_i}(Z_i)$ and so $\mathsf{pre}[\mathsf{SC}(A)](\gamma_{\mathcal{P}_i}(R_i)) \subseteq Z_i$ by Point 1 of Lemma 5.3.3 and Theorem 5.2.17. Because $\gamma_{\mathcal{P}_i}(R_i)$ is an over-approximation of the concrete states that reach an accepting state and as $\mathsf{pre}[\mathsf{SC}(A)](\gamma_{\mathcal{P}_i}(R_i)) \subseteq Z_i \subseteq \overline{[\![q_0]\!]}$, we know that all the states that can reach an accepting state are unreachable from $[\![q_0]\!]$. Hence $L(A) = \varnothing$. Now, assume that the algorithm returns false. Then $2^F \not\subseteq Z_i$ which means that a reachable state is able to reach an accepting state in i steps or less. Hence we conclude that $L(A) \neq \varnothing$. To prove the completeness of the algorithm, we only need to

establish its termination. This part of the proof is exactly the same as for the abstract forward algorithm. □

5.4 Experimental Evaluation

In this section, we evaluate the practical performance of our techniques by applying them to the satisfiability problem of LTL over finite words.

We have run our algorithms on three series of benchmarks. Each benchmark is composed of a pair of LTL formulas $\langle \psi, \varphi \rangle$ interpreted on finite words, and for which we want to know whether φ is a logical consequence of ψ, i.e. whether $\psi \models \varphi$ holds. To solve this problem, we translate the formula $\psi \wedge \neg\varphi$ into an AFA and check that the language of the AFA is empty. As we will see, our ψ formulas are constructed as large conjunctions of constraints and model the behavior of finite-state systems, while the φ formulas model properties of those systems. We defined properties with varying degrees of *locality*. Intuitively, a property φ is local when only a small number of subformulas of ψ are needed to establish $\psi \models \varphi$. This is not a formal notion but it will be clear from the examples. We will show in this section that our abstract algorithms are able to automatically identify subformulas which are not needed to establish the property. We only report results where $\psi \models \varphi$ holds; these are arguably the most difficult instances, as the entire fixed point must be computed. We now present each benchmark in turn.

Benchmark 1 The first benchmark takes 2 parameters $n > 0$ and $0 < k \leq n$: $\mathsf{Bench1}(n, k) = \langle \psi \equiv \bigwedge_{i=0}^{n-1} \mathsf{G}\,(p_i \Rightarrow (\mathsf{F}\,(\neg p_i) \wedge \mathsf{F}\,(p_{i+1}))), \varphi \equiv \mathsf{F}\,p_0 \Rightarrow \mathsf{F}\,p_k \rangle$. Clearly we have that $\psi \models \varphi$ holds for all values of k and also that the subformulas of ψ for $i > k$ are not needed to establish $\psi \models \varphi$.

Benchmark 2 This second benchmark is used to demonstrate how our algorithms can automatically detect less obvious versions of locality than for Bench1. It uses 2 parameters k and n with $0 < k \leq n$ and is built using the following recursive nesting definition: $\mathsf{Sub}(n, 1) = \mathsf{F}\,p_n$; for odd values

of $k > 1$ $\mathsf{Sub}(n,k) = \mathsf{F}\,(p_n \wedge \mathsf{X}\,(\mathsf{Sub}(n,k-1)))$; and for even values of $k > 1$ $\mathsf{Sub}(n,k) = \mathsf{F}\,(\neg p_n \wedge \mathsf{X}\,(\mathsf{Sub}(n,k-1)))$. We have : $\mathsf{Bench2}(n,k) = \langle \psi \equiv \bigwedge_{i=0}^{n-1} \mathsf{G}\,(p_i \Rightarrow \mathsf{Sub}(i+1,k)), \varphi \equiv \mathsf{F}\,p_0 \Rightarrow \mathsf{F}\,p_n \rangle$. It is relatively easy to see that $\psi \models \varphi$ holds for any value of k, and that for odd values of k, the nested subformulas beyond the first level are not needed to establish the property.

Benchmark 3 This third and final benchmark aims at demonstrating the usefulness of our abstraction algorithms in a more realistic setting. We specified the behavior of a lift with n floors with a parametric LTL formula. For n floors, $\mathsf{Prop} = \{f_1, \ldots, f_n, b_1, \ldots, b_n, \mathit{open}\}$. The f_i propositions represent the current floor. Only one of the f_i's can be true at any time, which is initially f_1. The b_i propositions represent the state (lit or unlit) of the call-buttons of each floor and there is only one button per floor. The additional *open* proposition is true when the doors of the lift are open. The constraints on the dynamics of this system are as follows : (i) initially the lift is at the first floor and the doors are open, (ii) the lift must close its doors when changing floors, (iii) the lift must go through floors in the correct order, (iv) when a button is lit, the lift eventually reaches the corresponding floor and opens its doors, and finally (v) when the lift reaches a floor, the corresponding button becomes unlit. Let n be the number of floors. We apply our algorithms to check two properties which depend on a parameter k with $1 < k \leq n$, namely $\mathsf{Spec1}(k) = \mathsf{G}\,((f_1 \wedge b_k) \rightarrow (\neg f_k\,\mathsf{U}\,f_{k-1}))$, and $\mathsf{Spec2}(k) = \mathsf{G}\,((f_1 \wedge b_k \wedge b_{k-1}) \rightarrow (b_k\,\mathsf{U}\,\neg b_{k-1}))$.

Experimental results All the results of our experiments are found in Fig. 6.9, and were performed on a quad-core 3,2 GHz Intel CPU with 12 GB of memory. We only report results for the concrete forward and abstract backward algorithms which were the fastest (by a large factor) in all our experiments. The columns of the table are as follows. ATC is the size of the largest antichain encountered, *iters* is the number of iterations of the fixed point in the concrete case and the maximal number of iterations of all the abstract fixed points in the abstract case, ATC^{α} and ATC^{γ} are respectively the sizes of the largest abstract and concrete antichains encountered, *steps* is the number of execution of the refinement steps and $|\mathcal{P}|$ is the maximum number of blocks in the partitions.

Benchmark 1. The partition sizes of the first benchmark illustrate how our algorithm exploits the locality of the property to abstract away the irrelevant parts of the system. For local properties, i.e. for small values of k, $|\mathcal{P}|$ is small compared to $|Q|$ meaning that the algorithm automatically ignores many subformulas which are irrelevant to the property. For larger values of k, the overhead induced by the successive abstraction computations becomes larger, but it becomes less important as the system grows.

Benchmark 2. On the second benchmark, our abstract algorithm largely outperforms the concrete algorithm. Notice how for $k \geq 3$ the partition sizes do not continue to grow (it also holds for values of k beyond 5). This means that contrary to the concrete algorithm, the abstract algorithm does not get trapped in the intricate nesting of the F modalities (which are not necessary to prove the property) and abstracts it completely with a constant number of partition blocks. The speed improvement is important.

Benchmark 3. On this final benchmark, the abstract algorithm outperforms the concrete algorithm when the locality of the property spans less than 5 floors. Beyond that value, the abstract algorithm starts to take longer than the concrete version. From the ATC column, the antichain sizes remain constant in the concrete algorithm, when the number of floors increases. This indicates that the difficulty of this benchmark comes mainly from the exponential size of the alphabet rather than the state-space itself. As our algorithms only abstract the states and not the alphabet, these results are not surprising.

5.5 Discussion

We have proposed in this work two new abstract algorithms with refinement for the language emptiness problem of AFA. Our algorithm is based on an abstraction-refinement scheme inspired from [CGR07], which is different from the usual refinement techniques based on counter-example elimination [CGJ+03]. Our algorithm also builds on the successful framework of antichains presented in the two preceding chapters. We have demonstrated

with a set of benchmarks, that our algorithm is able to find coarse abstractions for complex automata constructed from large LTL formulas. For a large number of instances of those benchmarks, the abstract algorithms outperform by several orders of magnitude the concrete algorithms. We believe that this clearly shows the interest of these algorithms and their potential future developments.

Several lines of future works can be envisioned. First, we should try to design a version of our algorithms where refinements are based on counter-examples and compare the relative performance of the two methods. Second, we have developed our technique for automata on finite words. We need to develop more theory to be able to apply our ideas to automata on infinite words. The fixed points involved in deciding emptiness for the infinite word case are more complicated and our theory must be extended to handle this case. Finally, it would be interesting to enrich our abstraction framework to deal with very large alphabets, possibly by partitioning the set of alphabet symbols.

					concrete forward			*abstract backward*					
	n	k	$\|Q\|$	$\|\mathbb{P}\|$	time	ATC	iters	time	ATC^{α}	ATC^{γ}	iters	steps	$\|\mathcal{P}\|$
Bench1	11	5	50	12	0,10	6	3	0,23	55	2	5	3	27
	15	5	66	16	1,60	6	3	0,56	55	2	5	3	31
	19	5	82	20	76,62	6	3	8,64	55	2	5	3	35
	11	7	50	12	0,13	8	3	0,87	201	2	5	3	31
	15	7	66	16	2,04	8	3	1,21	201	2	5	3	35
	19	7	82	20	95,79	8	3	9,99	201	2	5	3	39
	11	9	50	12	0,16	10	3	12,60	779	2	5	3	35
	15	9	66	16	2,69	10	3	13,42	779	2	5	3	39
	19	9	82	20	125,85	10	3	46,47	779	2	5	3	43
Bench2	7	1	19	8	0,06	8	2	0,10	11	2	4	3	14
	10	1	25	11	0,06	10	2	0,10	14	2	4	3	17
	13	1	31	14	0,08	14	2	0,12	17	2	4	3	20
	7	3	33	8	0,78	201	14	0,13	11	2	4	3	26
	10	3	45	11	802,17	4339	20	0,30	14	2	4	3	35
	13	3	57	14	> 1000	-	-	1,26	17	2	4	3	44
	7	5	47	8	88,15	2122	26	0,14	11	2	4	3	26
	10	5	65	11	> 1000	-	-	0,37	14	2	4	3	35
	13	5	83	14	> 1000	-	-	1,47	17	2	4	3	44
Lift : Spec1	8	3	84	17	0,30	10	17	0,51	23	40	7	4	21
	12	3	116	25	17,45	10	25	1,63	23	40	7	4	21
	16	3	148	33	498,65	10	33	26,65	23	40	7	4	21
	8	4	84	17	0,26	10	17	1,29	37	72	10	6	24
	12	4	116	25	17,81	10	25	5,02	37	72	10	6	24
	16	4	148	33	555,44	10	33	78,75	37	72	10	6	24
	8	5	84	17	0,32	10	17	3,70	42	141	12	8	27
	12	5	116	25	20,24	10	25	47,45	42	141	12	8	27
	16	5	148	33	543,27	10	33	> 1000	-	-	-	-	-
Lift : Spec2	8	3	84	17	0,46	10	17	1,18	58	72	8	4	22
	12	3	116	25	17,98	10	25	3,64	58	72	8	4	22
	16	3	148	33	557,75	10	33	48,90	58	72	8	4	22
	8	4	84	17	0,29	10	17	3,04	124	126	11	6	25
	12	4	116	25	19,29	10	25	10,63	124	126	11	6	25
	16	4	148	33	576,56	10	33	128,40	124	126	11	6	25
	8	5	84	17	0,31	10	17	15,88	131	266	14	8	28
	12	5	116	25	19,47	10	25	283,90	131	266	14	8	28
	16	5	148	33	568,83	10	33	> 1000	-	-	-	-	-

Figure 5.2: Experimental results. Times are in seconds.

Chapter 6

Lattice-Valued Binary Decision Diagrams

In this chapter, we introduce a new data structure, called *lattice-valued binary decision diagram* (or LVBDD for short), for the compact representation and manipulation of functions of the form $\theta : 2^{\mathbb{P}} \mapsto \mathcal{L}$, where $\mathbb{P}$ is a finite set of Boolean propositions and $\mathcal{L}$ is a finite and distributive lattice. LVBDD are a natural generalization of *multi-terminal* ROBDD which exploit the structure of the underlying lattice to achieve more compact representations. We introduce two canonical forms for LVBDD and present algorithms to symbolically compute their conjunction, disjunction and projection. We argue in this work that LVBDD are particularly well-suited for the encoding of the transition function of symbolic alternating automata. Indeed, such transition functions can be represented by functions of the form $2^{\mathbb{P}} \mapsto \mathcal{L}$, where $\mathcal{L}$ is the lattice of *upward-closed sets* of sets of states. To support our claims, we have implemented an LVBDD C++ template library, and adapted the LTL satisfiability algorithms described in Chapter 4 to use LVBDD. Our experimental evaluations on a number of LTL satisfiability benchmarks show that LVBDD can outperform ROBDD by several orders of magnitude.

6.1 Introduction

Efficient symbolic data structures are the cornerstone of efficient implementations of symbolic model checking algorithms. Tools like SMV [McM99] and NuSMV [CCG+02] have been applied with success to industrial-strength verification problems. These tools exploit ROBDD [Bry86], which is the reference data structure that has been designed to compactly encode and manipulate Boolean functions. ROBDD are often regarded as the *basic toolbox* of verification, and are indeed a *very general symbolic data structure*.

The dramatic success of ROBDD in the field of computer aided verification has prompted researchers to introduce several variants of ROBDD [iM93, AH97] or to extend them to represent other kinds of functions [DC03, FMY97]. For instance, *multi-terminal binary decision diagrams* [FMY97] (MTBDD) and *algebraic decision diagrams* [BFG+97] (ADD) have been successfully applied to the representation of functions of the form $2^{\mathbb{P}} \mapsto D$, where D is an *arbitrary domain*. Both MTBDD and ADD are variants of ROBDD which allow for more than two terminal nodes (in the worst case, there can be as many terminal nodes as the number of values in D).

In this work, we introduce a new data structure, called *lattice-valued binary decision diagrams* (or LVBDD for short) to efficiently handle *lattice-valued Boolean functions*, which are functions of the form $\theta : 2^{\mathbb{P}} \mapsto \mathcal{L}$, where $\mathbb{P}$ is a finite set of Boolean propositions and $\mathcal{L}$ is a *finite distributive lattice*. Syntactically, LVBDD are ordered binary decision diagrams with one or more terminal nodes, with an additional lattice value labeling each node. We define two semantics for LVBDD; the *meet-semantics* (respectively *join-semantics*) associates, to each valuation $v \in 2^{\mathbb{P}}$, the *meet* (respectively *join*) of the lattice values encountered along the path corresponding to v.

Our primary motivation for the development of LVBDD is the efficient encoding of the transition function of symbolic alternating automata. Let us illustrate the syntax and semantics of LVBDD in that context. For instance, consider a sAFA $A = \langle Q, \mathbb{P}, q_0, \delta, F \rangle$ with $Q = \{q_1, q_2, q_3\}$, $\mathbb{P} = \{p_1, p_2, p_3\}$, and such that $\delta(q_1) = (p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge (p_3 \vee q_2 \vee q_3)$. Two examples of meet-semantics LVBDD representing $\delta(q_1)$ are depicted in Figure 6.1. Both LVBDD represent a function $\theta : 2^{\mathbb{P}} \mapsto \mathsf{UCS}[2^Q, \subseteq]$, that is a mapping from the set of valuations over $\mathbb{P}$ to the set of *upward-closed sets of sets of states*

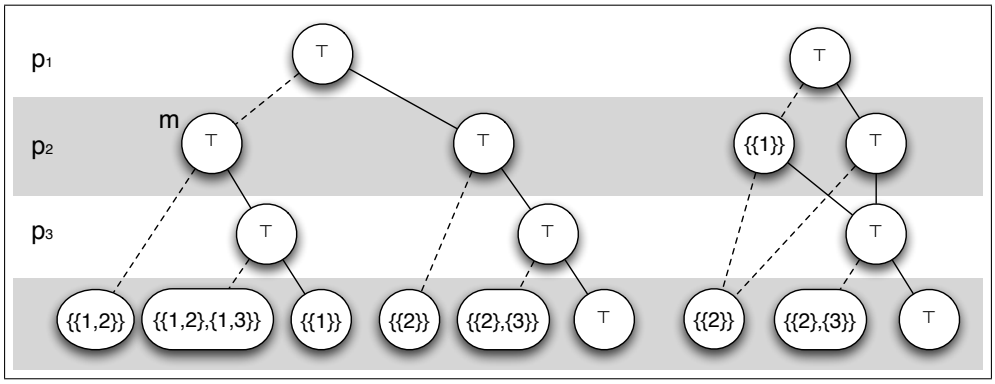


Figure 6.1: Two examples of LVBDD, respectively in UNF (left) and SNF (right). Both represent the same formula $(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge (p_3 \vee q_2 \vee q_3)$. To simplify the figure, upward-closed sets are represented by their minimal antichain, and the propositions q_1, q_2, q_3 are abbreviated 1,2,3.

of the sAFA. The truth table of the function θ is depicted in Figure 6.2.

Let us illustrate the semantics of LVBDD on the examples of Figure 6.1. Note that $\top = \uparrow\{\{\}\}$ is the largest element in the lattice of upward-closed sets. Consider the valuation $p_1 = \mathsf{false}, p_2 = \mathsf{true}, p_3 = \mathsf{false}$. If we take the meet of the values encountered on the corresponding path in the left LVBDD we obtain $\bigsqcap \{\top, \top, \top, \uparrow\{\{q_1, q_2\}, \{q_1, q_3\}\}\} = \uparrow\{\{q_1, q_2\}, \{q_1, q_3\}\}$. For the same valuation, if we take the meet of the values encountered in the right LVBDD we obtain $\bigsqcap \{\top, \uparrow\{\{q_1\}\}, \top, \uparrow\{\{q_2\}, \{q_3\}\}\} = \uparrow\{\{q_1, q_2\}, \{q_1, q_3\}\}$. This is not a coincidence, as the two LVBDD have the same semantics, i.e. they represent the same function. The LVBDD on the left of Figure 6.1 is in *unshared normal form* (UNF, for short), while the LVBDD on the right is in *shared normal form* (SNF, for short). The UNF has the property that all non-terminal nodes are labeled with $\top$, the greatest element in the lattice, and can thus be ignored ($\top \sqcap x = x$, for any x). As a consequence, the number of terminal nodes of an LVBDD in SNF is always equal to the number of values in the *image of the function it represents*. Therefore, LVBDD in UNF are unsuitable for the representation of functions with very large (typically exponential) images. To handle the representation of lattice-valued functions with large images, we have developed the shared normal form.

p_1	p_2	p_3	$\theta(v)$
0	0	0	$\uparrow\{\{q_1, q_2\}\}$
0	0	1	$\uparrow\{\{q_1, q_2\}\}$
0	1	0	$\uparrow\{\{q_1, q_2\}, \{q_1, q_3\}\}$
0	1	1	$\uparrow\{\{q_1\}\}$
1	0	0	$\uparrow\{\{q_2\}\}$
1	0	1	$\uparrow\{\{q_2\}\}$
1	1	0	$\uparrow\{\{q_2, q_3\}\}$
1	1	1	$\uparrow\{\{\}\}$

Figure 6.2: Truth table of the function $\theta = [\![(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge (p_3 \vee q_2 \vee q_3)]\!]$.

Let us give some intuitions on the SNF, using the examples of Figure 6.1. Consider the node m on the left LVBDD of the figure. If we examine the terminal nodes of the corresponding subtree, we conclude that they all impose that the state q_1 be present. We can *factor out* this constraint, and label the node m with the value $\uparrow\{\{q_1\}\}$. Note that this is the *strongest* constraint (the smallest set) that we can put on m, since q_1 is the only sAFA state that is shared among all lattice values in m's subtree. Now that we have changed the label of m, some labels of its subtree have become *redundant*, since the presence of q_1 is already enforced by m. We can relax these constraints without changing the semantics of the LVBDD, by replacing $\uparrow\{\{q_1, q_2\}\}$ by $\uparrow\{\{q_2\}\}$, $\uparrow\{\{q_1, q_2\}, \{q_1, q_3\}\}$ by $\uparrow\{\{q_2\}, \{q_3\}\}$, and so on. Notice that this relaxation corresponds to choosing the *weakest* possible constraint (the largest set). If we proceed in this way systematically, and if we merge isomorphic subgraphs, we obtain the LVBDD on the right, which is in shared normal form.

Contributions In this work, we formally define the two normal forms for LVBDD (SNF and UNF), and provide algorithms for computing their meet, join and projection. All of these algorithms are accompanied with correctness proofs, that allow the use for LVBDD over any (finite and distributive) lattice. We also study the worst-case complexity of these algorithms, and provide some theoretical compactness properties of LVBDD.

We have implemented our LVBDD data structure in the form of a C++

template library, and we have used this library to experimentally evaluate the performance of LVBDD in the context of LTL satisfiability over finite-words. Again, we reduce the satisfiability problem to the emptiness of a sAFA, by using the antichain algorithm described in Chapter 4. Our experiments reveal that LVBDD are particularly well-suited for the encoding of the transition functions of symbolic alternating automata, and can outperform ROBDD by several orders of magnitude in that context.

Related works While LVBDD represent functions of the form $\theta : 2^{\mathsf{P}} \mapsto \mathcal{L}$, other structures to represent similar but different kinds of functions have been studied in the literature. ROBDD [Bry86], and some variants like *zero-suppressed binary decision diagrams* [iM93] and *Boolean expression diagrams* [AH97], encode purely Boolean functions $\theta : 2^{\mathsf{P}} \mapsto \{0, 1\}$. On the other hand, *multi-terminal binary decision diagrams* [FMY97] and *algebraic decision diagrams* [BFG+97] represent functions of the form $2^{\mathbb{P}} \mapsto D$, but do not exploit the structure of D when it is a lattice. Another related data structure is *edge-shifted decision diagrams* [DC03] represent functions of the form $\mathcal{L}^{\mathbb{P}} \mapsto \mathcal{L}$ and have been applied to the field of *multi-valued model checking* [CEP01]. Finally, *lattice automata* [KL07] represent functions of the form $\Sigma^* \mapsto \mathcal{L}$, where Σ is a finite alphabet and $\mathcal{L}$ is a lattice. In some sense, ROBDD are to NFA what LVBDD are to lattice automata.

Structure of the chapter In Section 6.2, we formally define lattice-valued Boolean functions and its accompanying logic, along with some specific notations. In Section 6.3, we show that the transition function of symbolic alternating automata can be naturally defined using LVBF. This serves as the primary motivation for introducing LVBDD. In Section 6.4, we formally define the syntax and semantics of LVBDD. In Section 6.5, we introduce two normal forms for LVBDD, the *unshared normal form* (UNF) and the *shared normal form* (SNF). In Section 6.6, we define and prove a number of algebraic properties of the *relative pseudo-complement* on finite and distributive lattices. These properties are needed for the efficient manipulation of LVBDD in SNF. In Section 6.7, we describe the algorithms to compute the join, meet and projection of LVBDD, both for the shared and unshared normal forms. In Section 6.8, we discuss the worst-case complex-

ity of LVBDD algorithms. In Section 6.9 we present experimental evidence that LVBDD-based algorithms can outperform state-of-the-art tools in the context of finite-word LTL satisfiability. Finally, we offer some discussion in Section 6.10.

6.2 Lattice-Valued Boolean Functions

In the same way that ROBDD encode *Boolean functions* of the form $\theta : 2^{\mathbb{P}} \mapsto \mathbb{B}$, LVBDD encode *lattice-valued Boolean functions* (LVBF, for short) which are the functions of the form $\theta : 2^{\mathbb{P}} \mapsto \mathcal{L}$, where $\mathcal{L}$ is a lattice. In what follows, we denote by $\mathsf{LVBF}(\mathbb{P}, \mathcal{L})$ the set of LVBF of type $\theta : 2^{\mathbb{P}} \mapsto \mathcal{L}$. Since $\mathbb{B}$ is a lattice, LVBF generalize traditional Boolean functions in a natural fashion. Similarly, we can generalize propositional Boolean logic to a *lattice-valued* Boolean logic, which mixes Boolean constraints with lattice values.

Definition 6.2.1 (Lattice-Valued Boolean Logic – Syntax)**.** *Let $\mathbb{P}$ be a finite set of Boolean propositions, and let $\mathcal{L}$ be a lattice. The set of* lattice-valued Boolean formulas *is defined recursively with the following grammar:*

$$\varphi ::= \mathsf{false} \mid \mathsf{true} \mid p \mid \neg p \mid d \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \qquad \text{where } p \in \mathbb{P} \text{ and } d \in \mathcal{L}$$

We denote the set of lattice-valued formulas over $\mathbb{P}$ and $\mathcal{L}$ by $\mathsf{LVBL}(\mathbb{P}, \mathcal{L})$.

As expected, we define the semantics of LVBL formulas with LVBF. The semantics definition is very natural; we simply map the disjunction and conjunction to join and meet, respectively. In order to give an appropriate semantics to true and false, we restrict our semantics definition to *bounded* lattices.

Definition 6.2.2 (Lattice-Valued Boolean Logic – Semantics)**.** *Let $\mathbb{P}$ be a finite set of Boolean propositions, let $\mathcal{L}$ be a bounded lattice, and let $\varphi \in \mathsf{LVBL}(\mathbb{P}, \mathcal{L})$. The* semantics *of the formula φ, denoted $[\![\varphi]\!]$, is the lattice-valued Boolean function such that, for every valuation $v \in 2^{\mathbb{P}}$:*

- $[\![\mathsf{true}]\!](v) = \top$;
- $[\![\mathsf{false}]\!](v) = \bot$;

- $[\![d]\!](v) = d$, *for each* $d \in \mathcal{L}$;
- $[\![p]\!](v) = \top$ if $p \in v$ else $\bot$, *for each* $p \in \mathbb{P}$;
- $[\![\neg p]\!](v) = \top$ if $p \notin v$ else $\bot$, *for each* $p \in \mathbb{P}$;
- $[\![\varphi_1 \vee \varphi_2]\!](v) = [\![\varphi_1]\!](v) \sqcup [\![\varphi_2]\!](v)$;
- $[\![\varphi_1 \wedge \varphi_2]\!](v) = [\![\varphi_1]\!](v) \sqcap [\![\varphi_2]\!](v)$.

Note that the semantics of LVBL formulas is such that the distributivity properties of $\wedge$ and $\vee$ are maintained when the codomain lattice is distributive.

For any LVBF $\theta\colon 2^{\mathbb{P}} \to \mathcal{L}$, with $p \in \mathbb{P}$ and $t \in \mathbb{B}$, we define $\theta|_{p=t}$ to be the function $\theta'\colon 2^{\mathbb{P}} \to \mathcal{L}$ such that $\theta'(v) = \theta(v|_{p=t})$ for every valuation $v \in 2^{\mathbb{P}}$. Also, the *dependency set* $\mathsf{Dep}(\theta) \subseteq \mathbb{P}$ of an LVBF $\theta\colon 2^{\mathbb{P}} \mapsto \mathcal{L}$ is the set of propositions over which θ depends; formally $\mathsf{Dep}(\theta) = \{p \in \mathbb{P} \mid \theta|_{p=\mathsf{false}} \neq \theta|_{p=\mathsf{true}}\}$. The *existential quantification* of θ is $\exists\mathbb{P} \cdot \theta \stackrel{\mathsf{def}}{=} \mathsf{LUB}(\{\theta(v) \mid v \in 2^{\mathbb{P}}\})$. Similarly, the *universal quantification* of θ is $\forall\mathbb{P} \cdot \theta \stackrel{\mathsf{def}}{=} \mathsf{GLB}(\{\theta(v) \mid v \in 2^{\mathbb{P}}\})$.

Remark 6.2.3. *Let* $\theta_1, \theta_2 \in \mathsf{LVBF}(\mathbb{P}, \mathcal{L})$ *for some* $\mathbb{P}$ *and* $\mathcal{L}$*, and let* $p \in \mathbb{P}$ *and* $d \in \mathcal{L}$*. To simplify the notations, we often omit the semantics* $[\![\cdot]\!]$ *brackets around atoms such as* d*,* p *and* $\neg p$*. Also, we define the shorthands* $\theta_1 \sqcap \theta_2 \stackrel{\mathsf{def}}{=} \lambda v \cdot [\![\theta_1]\!](v) \sqcap [\![\theta_2]\!](v)$ *and* $\theta_1 \sqcup \theta_2 \stackrel{\mathsf{def}}{=} \lambda v \cdot [\![\theta_1]\!](v) \sqcup [\![\theta_2]\!](v)$*. For instance, we have that* $d \sqcap \theta = \lambda v \cdot d \sqcap [\![\theta]\!](v)$ *and* $\neg p \sqcup \theta = \lambda v \cdot [\![\theta]\!](v)$ if $p \notin v$ else $\bot$*, and so on.*

6.3 LVBF and Alternating Automata

An efficient data structure for manipulating LVBF is highly desirable, and is immediately useful for analyzing alternating automata. Indeed, the transition function of an sAFA or an sABA maps each state of the automaton to a Boolean formula in the set $\mathsf{BL}^+(\mathsf{Lit}(\mathbb{P}) \cup Q)$ where $\mathbb{P}$ is the symbolic alphabet, and Q is the set of states. As discussed in Chapter 4, these formulas can be encoded using ROBDD by encoding them into functions of the form $f : 2^{\mathbb{P} \cup Q} \mapsto \mathbb{B}$, assigning one ROBDD variable to each location and each atomic proposition.

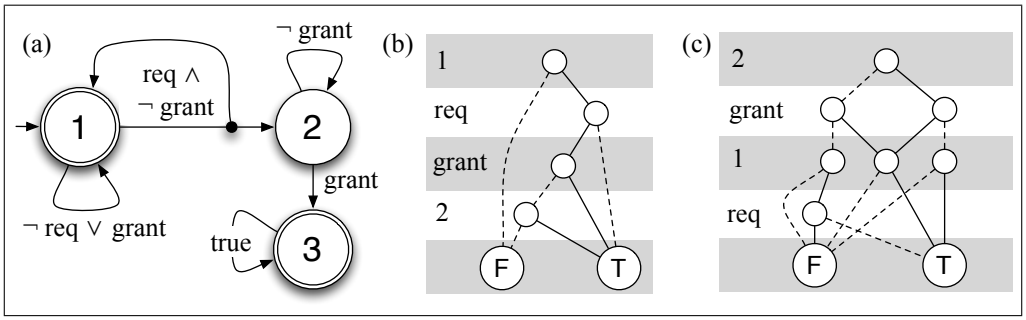


Figure 6.3: A symbolic-alphabet alternating automaton (a) and the ROBDD encodings of $\delta(1)$ with two variable orderings (b and c).

Example 6.3.1 (ROBDD encoding of sAFA transitions)**.** *Figure 6.3 depicts a symbolic automaton equivalent to the* LTL *formula* $\mathsf{G}(\text{req} \Rightarrow \mathsf{F}\,\text{grant})$*, along with two* ROBDD *which encode* $\delta(1)$ *for two different variable orderings.*

The use of LVBF for representing symbolic alternating automata transitions is motivated by two disadvantages of the traditional ROBDD encoding. First, as ROBDD require a *total ordering* between variables, one must find an adequate *interleaving* of the alphabet variables and the location variables, which in practice may be hard to find. Second, since location variables are constrained only positively (without negations), there are more efficient ways of representing those specific constraints, such as antichains or covering sharing trees [DRB04]. The use of LVBF allows for an elegant *decoupling* of the fully-Boolean constraints from the positive Boolean constraints in the encoding. Instead of representing alternating automata transitions with ROBDD that encode functions of the form $f : 2^{\mathbb{P}\cup Q} \mapsto \mathbb{B}$, we propose to use LVBDD to encode functions of the form $f : 2^{\mathbb{P}} \mapsto \mathsf{UCS}[2^Q, \subseteq]$, thus mapping each valuation of the alphabet propositions to an $\subseteq$-upward-closed set of subsets of Q. Since $\mathsf{UCS}[2^Q, \subseteq]$ is a lattice, these functions are in fact LVBF. Thanks to Theorem 2.1.4, we know that the lattice of upward-closed subsets of a finite set is a finite and distributive lattice (since the set of subsets of a finite set is clearly a join-semilattice).

Since ADD and MTBDD encode functions of the form $f : 2^{\mathbb{P}} \mapsto D$ where D is an arbitrary domain, one could readily use MTBDD or ADD to

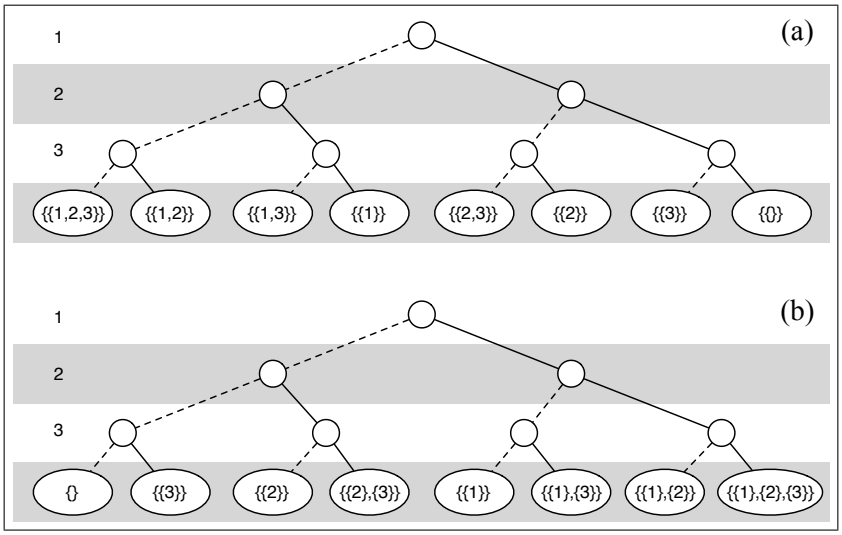


Figure 6.4: Multi-terminal ROBDD of Example 6.3.2. The top (a) MTBDD represents $[\![\varphi]\!]$, the bottom (b) MTBDD represents $[\![\psi]\!]$. To alleviate the figure, upward-closed sets are represented by their minimal antichain.

encode alternating automata transitions with functions of the form $f : 2^{\mathbb{P}} \mapsto \mathsf{UCS}[2^Q, \subseteq]$. However, this encoding would be highly unpractical because it would yield MTBDD or ADD with exponentially many (in $|\mathbb{P}|$) leaf nodes in a lot of cases. This is best illustrated with a simple example.

Example 6.3.2 (Exponential ADD, MTBDD when encoding LVBF)**.** *Let $S = \{s_1, \ldots, s_k\}$ be a finite set, let $\mathcal{U} = \mathsf{UCS}[2^S, \subseteq]$, and let $\mathbb{P} = \{p_1, \ldots, p_k\}$ be a finite set of Boolean propositions. We consider two lattice-valued Boolean formulas $\varphi, \psi \in \mathsf{LVBL}(\mathbb{P}, \mathcal{U})$ which are respectively in disjunctive and conjunctive normal form:*

$$\varphi = \bigwedge_{i=1}^{k} (p_i \vee \uparrow\{\{s_i\}\}) \qquad \psi = \bigvee_{i=1}^{k} (p_i \wedge \uparrow\{\{s_i\}\})$$

The semantics of φ and ψ is such that $[\![\varphi]\!](v) = \uparrow\{\{s_i\} \mid p_i \notin v\}$ and that $[\![\psi]\!](v) = \mathsf{LUB}(\{\uparrow\{\{s_i\}\} \mid p_i \in v\})$. It is easy to see that for any $v, v' \in 2^{\mathbb{P}}$:

$$v = v' \text{ iff } [\![\varphi]\!](v) = [\![\varphi]\!](v') \text{ iff } [\![\psi]\!](v) = [\![\psi]\!](v')$$

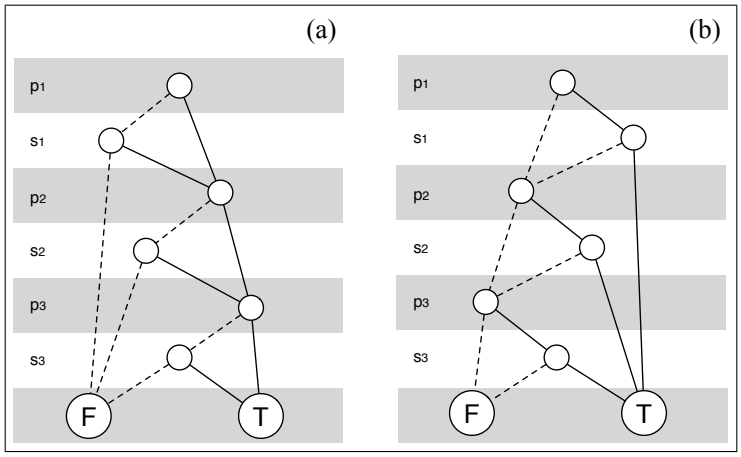


Figure 6.5: ROBDD encoding of the LVBF of Example 6.3.2. The left (a) ROBDD represents $[\![\varphi]\!]$, the right (b) ROBDD represents $[\![\psi]\!]$.

Therefore both $[\![\varphi]\!]$ *and* $[\![\psi]\!]$ *have an image of size* $|2^{\mathbb{P}}|$*, and thus cannot be represented with* ADD *or* MTBDD *in a compact way. An illustration of the* MTBDD *encoding of* $[\![\varphi]\!]$ *and* $[\![\psi]\!]$ *for* $k = 3$ *is depicted in Figure 6.4.*

The LVBF of Example 6.3.2 show that MTBDD and ADD are not practical data structures for the representations of functions whose image is exponentially large. On the other hand, these LVBF can be efficiently represented with ROBDD, with the formulas $\bigwedge_{i=1}^{k}(p_i \vee s_i)$ and $\bigvee_{i=1}^{k}(p_i \wedge s_i)$, and by ordering the variables in alternating order: $p_1, s_1, p_2, s_2, \ldots, p_k, s_k$, as illustrated in Figure 6.5. However, this ROBDD encoding suffers from the two disadvantages that we have mentioned previously: (i) it is extremely sensitive to the interleaving of domain and codomain variables, and (ii) it does not allow to represent codomain values with ad-hoc data structures.

6.4 LVBDD – Syntax & Semantics

Syntactically, an LVBDD over the propositions $\mathbb{P}$ and lattice $\mathcal{L}$ is an ordered binary decision diagram over the variables in $\mathbb{P}$, whose nodes are labeled with lattice values taken from $\mathcal{L}$. This simple syntax has the advantage of

decoupling the representation of the constraints over $\mathbb{P}$ and the constraints over $\mathcal{L}$.

Definition 6.4.1 (LVBDD – Syntax)**.** *Given a finite set of Boolean propositions* $\mathbb{P} = \{p_1, \ldots, p_k\}$ *and a bounded lattice* $\mathcal{L}$*, an* LVBDD n *over* $\mathbb{P}$ *and* $\mathcal{L}$ *is: (i) either a* terminal LVBDD $\langle \mathsf{index}(n), \mathsf{val}(n) \rangle$ *where* $\mathsf{index}(n) = k+1$ *and* $\mathsf{val}(n) \in \mathcal{L}$*; or (ii) a* non-terminal LVBDD $\langle \mathsf{index}(n), \mathsf{val}(n), \mathsf{lo}(n), \mathsf{hi}(n) \rangle$*, where* $1 \leq \mathsf{index}(n) \leq k$*,* $\mathsf{val}(n) \in \mathcal{L}$ *and* $\mathsf{lo}(n)$ *and* $\mathsf{hi}(n)$ *are (terminal or non-terminal)* LVBDD *such that* $\mathsf{index}(\mathsf{hi}(n)) > \mathsf{index}(n)$ *and* $\mathsf{index}(\mathsf{lo}(n)) > \mathsf{index}(n)$*. The set of* LVBDD *over* $\mathbb{P}$ *and* $\mathcal{L}$ *is denoted by* $\mathsf{LVBDD}(\mathbb{P}, \mathcal{L})$*.*

Similarly to ROBDD, for every $n \in \mathsf{LVBDD}(\{p_1, \ldots, p_k\}, \mathcal{L})$ with $\mathsf{index}(n) = i$, we denote by $\mathsf{prop}(n)$ the proposition p_i. Likewise, we refer to LVBDD also as *"*LVBDD *node"*, or simply *"node"*. For any non-terminal node n, we call $\mathsf{hi}(n)$ (resp. $\mathsf{lo}(n)$) the *high-child* (*low-child*) of n. Finally, the set $\mathsf{nodes}(n)$ of an LVBDD n is defined recursively as follows. If n is terminal, then $\mathsf{nodes}(n) = \{n\}$. Otherwise $\mathsf{nodes}(n) = \{n\} \cup \mathsf{nodes}(\mathsf{lo}(n)) \cup \mathsf{nodes}(\mathsf{hi}(n))$. The *number of nodes* of an LVBDD is denoted by $|n|$.

Similarly to ROBDD, an LVBDD semantics must associate a lattice value to each valuation in $2^{\mathbb{P}}$. There are two natural definitions for LVBDD semantics: the first maps each valuation v to the *meet* of the values encountered along the path corresponding to v; the second maps each valuation to the *join* of those lattice values.

Definition 6.4.2 (LVBDD – Meet / Join Semantics)**.** *Let* $\otimes \in \{\sqcap, \sqcup\}$*. For any set of Boolean propositions* $\mathbb{P}$ *and bounded lattice* $\mathcal{L}$*, the* meet-semantics *(respectively* join-semantics*) of an* LVBDD *over* $\mathbb{P}$ *and* $\mathcal{L}$ *is defined by the unary function* $[\![\cdot]\!]_{\otimes} : \mathsf{LVBDD}(\mathsf{P}, \mathcal{L}) \mapsto \mathsf{LVBF}(\mathsf{P}, \mathcal{L})$ *such that for any* $n \in \mathsf{LVBDD}(\mathsf{P}, \mathcal{L})$*, with* $\mathsf{index}(n) = i$*:*

$$[\![n]\!]_{\otimes} = \begin{cases} \mathsf{val}(n) & \text{if } n \text{ is terminal} \\ \mathsf{val}(n) \otimes \Big((\neg p_i \sqcap [\![\mathsf{lo}(n)]\!]) \sqcup (p_i \sqcap [\![\mathsf{hi}(n)]\!]) \Big) & \text{otherwise} \end{cases}$$

The meet- and join-semantics of LVBDD are illustrated in Figure 6.6, which depicts a meet-semantics LVBDD encoding for the φ LVBL formula of Example 6.3.2, and a join-semantics LVBDD for the ψ formula.

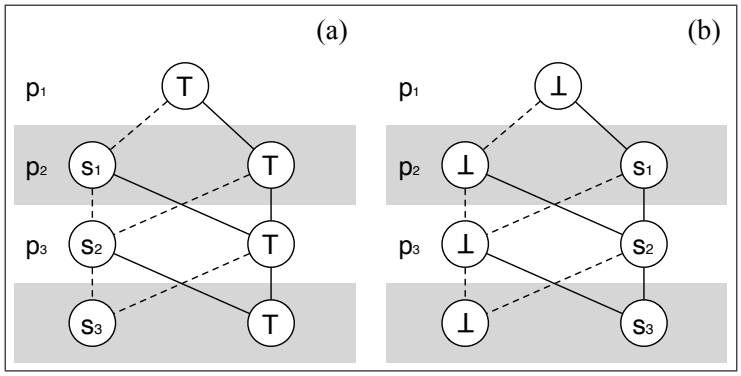


Figure 6.6: LVBDD encodings of the LVBF of Example 6.3.2 for $k = 3$. The left (a) LVBDD represents $[\![\varphi]\!]$ using the *meet-semantics*, the right (b) ROBDD represents $[\![\psi]\!]$ using the *join-semantics*. To alleviate the figure, each lattice value $\uparrow\{\{s_i\}\}$ is abbreviated by s_i.

An interesting property of the meet- and join-semantics of LVBDD is that they are *incomparable* in terms of compactness. In other words, there exists families of LVBF for which the smallest meet-semantics LVBDD is exponentially larger than the smallest join-semantics LVBDD, and vice versa. In fact, these families of LVBF are exactly those used previously in Example 6.3.2.

Theorem 6.4.3 (Compactness Incomparability of LVBDD Semantics)**.** *Let* $\mathbb{S} = \{s_1, \ldots, s_k\}$ *be a finite set, and let* $\mathbb{P} = \{p_1, \ldots, p_k\}$ *be a finite set of Boolean propositions. For any* $k \in \mathbb{N}_0$*, let* $\varphi^k = \bigwedge_{i=1}^{k}(p_i \vee \uparrow\{\{s_i\}\})$ *and* $\psi^k = \bigvee_{i=1}^{k}(p_i \wedge \uparrow\{\{s_i\}\})$ *be two formulas in* LVBL $\big(\mathbb{P}, \mathsf{UCS}[2^{\mathbb{S}}, \subseteq]\big)$*. For any* $n \in \mathsf{LVBDD}\big(\mathbb{P}, \mathsf{UCS}[2^{\mathbb{S}}, \subseteq]\big)$*:*

- (A) *if* $[\![n]\!]_{\sqcup} = [\![\varphi^k]\!]$ *then* $|n| \geq 2^k$*;*
- (B) *if* $[\![n]\!]_{\sqcap} = [\![\psi^k]\!]$ *then* $|n| \geq 2^k$*.*

Also, for any $k \in \mathbb{N}_0$*, we have that:*

- (C) *there exists an* LVBDD n *such that* $[\![n]\!]_{\sqcap} = [\![\varphi^k]\!]$ *and* $|n| = 2k + 1$*;*
- (D) *there exists an* LVBDD n *such that* $[\![n]\!]_{\sqcup} = [\![\psi^k]\!]$ *and* $|n| = 2k + 1$*.*

Proof. Let us first prove (A) that whenever $[\![n]\!]_{\sqcup} = [\![\varphi^k]\!]$ then $|n| \geq 2^k$. Observe that $\mathsf{img}([\![\varphi^k]\!]) = \big\{\uparrow\{S\} \mid S \subseteq \mathbb{S}\big\}$. Let us define, for any complete

lattice $\mathcal{L}$, a *join-generator* of a set $X \subseteq \mathcal{L}$ to be a set $G \subseteq \mathcal{L}$ such that for any $x \in X$ there exists $Y \subseteq G$ such that $\mathsf{LUB}(Y) = x$. It is easy to see that if the smallest join-generator of $\mathsf{img}(\llbracket \varphi^k \rrbracket)$ has at least 2^k elements, for every k, then (A) must be true. This is easy to show because $\mathsf{img}(\llbracket \varphi^k \rrbracket)$ is composed exclusively of join-irreducible elements, so any join-generator for $\mathsf{img}(\llbracket \varphi^k \rrbracket)$ contains at least each element of $\mathsf{img}(\llbracket \varphi^k \rrbracket)$. Therefore the smallest join-generator for $\mathsf{img}(\llbracket \varphi^k \rrbracket)$ has exactly 2^k elements, thereby proving the (A) statement.

Statement (B) is shown by first observing that the image $\mathsf{img}(\llbracket \psi^k \rrbracket) = \left\{ \uparrow \{\{s_i\} \mid s_i \in S\} \mid S \subseteq \mathbb{S} \right\}$ is composed exclusively of meet-irreducible elements and has cardinality 2^k. Therefore, the smallest *meet-generator* for $\mathsf{img}(\llbracket \psi^k \rrbracket)$ has cardinality 2^k, which is sufficient to prove (B).

Finally, to show (C) and (D), it suffices to generalize the two LVBDD of Figure 6.6 to arbitrary $k \in \mathbb{N}_0$. It is easy to see that these have a size of exactly $2k + 1$ nodes. □

The theorem above exposes a fundamental *compactness limit* of LVBDD and their meet- and join-semantics. Meet-semantics (respectively join-semantics) LVBDD will always have at least as many nodes as the number of meet-irreducible (respectively join-irreducible) elements in the *image* of the represented function.

Theorem 6.4.3 shows that both the meet-semantics and join-semantics of LVBDD are equally interesting (in theory, at least). However, in order to avoid duplicating every property or algorithm, the remainder of this chapter focuses solely on *meet-semantics* LVBDD. From here on, for any LVBDD n, the expression $\llbracket n \rrbracket$ is equivalent to $\llbracket n \rrbracket_{\sqcap}$. All of what follows in this chapter should readily apply to join-semantics LVBDD as well, by using the inherent symmetry of join and meet.

6.5 Unshared and Shared Normal Forms

One of the reasons that ROBDD are so successful in practical implementations is the fact that it is a *canonical data structure* (up to variable ordering). This canonicity property provides a number of advantages.

- **constant-time equality test** Canonicity permits an efficient equality test that amounts to checking for equality between pointers.

- **running time savings** ROBDD can be hashed by their root node pointer, allowing for efficient *caching* and *dynamic programming*.

- **memory usage savings** Because each ROBDD node is stored only once in memory, even across multiple ROBDD, a large number of Boolean functions can be represented in memory by using only a fraction of the number of required nodes.

In order to provide LVBDD with the same advantages in terms of canonicity than ROBDD, we develop *normal forms* for LVBDD which ensure that each LVBF has exactly one LVBDD representation, up to isomorphism. In what follows, we define two normal forms for LVBDD, the *unshared normal form* and the *shared normal form*.

6.5.1 Unshared Normal Form

The first normal form we define is the *unshared normal form* (UNF, for short), which simply requires that every non-terminal node be labeled with the $\top$ element of the lattice. This makes UNF LVBDD very similar to ADD or MTBDD.

To formalize the UNF, we need to define a notion of LVBDD isomorphism. Moreover, similarly to ROBDD, we say that an LVBDD is *reduced* iff it contains no distinct isomorphic subgraphs.

Definition 6.5.1 (LVBDD isomorphism)**.** *Let* $n_1, n_2 \in \mathsf{LVBDD}(\mathsf{P}, \mathcal{L})$*. We say that* n_1 *and* n_2 *are* isomorphic*, denoted by* $n_1 \equiv n_2$*, iff either* (i) n_1 *and* n_2 *are both terminal and* $\mathsf{val}(n_1) = \mathsf{val}(n_2)$*, or* (ii) n_1 *and* n_2 *are both non-terminal and* $\mathsf{val}(n_1) = \mathsf{val}(n_2)$*,* $\mathsf{index}(n_1) = \mathsf{index}(n_2)$*,* $\mathsf{lo}(n_1) \equiv \mathsf{lo}(n_2)$ *and* $\mathsf{hi}(n_1) \equiv \mathsf{hi}(n_2)$*. An* LVBDD n *is* reduced *iff* (i) *for all* $n \in \mathsf{nodes}(n)$*: either* n *is terminal or* (i) $\mathsf{lo}(n) \neq \mathsf{hi}(n)$ *and* (ii) *for all* $n_1, n_2 \in \mathsf{nodes}(n)$*:* $n_1 \equiv n_2$ *implies* $n_1 = n_2$*.*

It is easy to see that there exist LVBF θ for which one can find at least two different *reduced* LVBDD n_1 and n_2 such that $[\![n_1]\!] = [\![n_2]\!] = \theta$. However, if we require that every non-terminal node be labeled with $\top$, and require that all isomorphic subgraphs be merged, then we obtain a canonical form.

Definition 6.5.2 (Unshared Normal Form for LVBDD)**.** *Let $\mathbb{P}$ be a set of Boolean propositions and $\mathcal{L}$ be a bounded lattice. For any $\theta \in \mathsf{LVBF}(\mathsf{P}, \mathcal{L})$, $\mathsf{UNF}(\theta)$ is the* reduced LVBDD *defined recursively as follows. If $\mathsf{Dep}(\theta) = \varnothing$, then $\theta(v) = d$ for some $d \in \mathcal{L}$. In that case, $\mathsf{UNF}(\theta)$ is the terminal* LVBDD *$\langle k+1, d\rangle$. Otherwise, let p_i be the proposition of lowest index in $\mathsf{Dep}(\theta)$. Then, $\mathsf{UNF}(\theta)$ is the non-terminal* LVBDD *$\langle i, \top, \mathsf{UNF}(\theta|_{p_i=\mathsf{false}}), \mathsf{UNF}(\theta|_{p_i=\mathsf{true}})\rangle$. We denote by $\mathsf{UNF}(\mathbb{P}, \mathcal{L})$ the set of* LVBDD *in unshared normal form over the set of Boolean propositions $\mathbb{P}$ and lattice $\mathcal{L}$, or formally, we have that $\mathsf{UNF}(\mathbb{P}, \mathcal{L}) = \{n \in \mathsf{LVBDD}(\mathbb{P}, \mathcal{L}) \mid \mathsf{UNF}([\![n]\!]) = n\}$.*

Since the above definition for UNF is completely constructive (it describes a deterministic algorithm), it is obvious that $\mathsf{UNF}(\cdot)$ and $[\![\cdot]\!]$ are bijections between the sets $\mathsf{LVBF}(\mathbb{P}, \mathcal{L})$ and $\mathsf{UNF}(\mathbb{P}, \mathcal{L})$.

6.5.2 Shared Normal Form

It is immediate that the UNF is exponentially less compact (in the worst case) than unrestricted LVBDD. Like ADD or MTBDD, the number of leaf nodes of an LVBDD in UNF is exactly equal to the size of the image of the represented function. The LVBDD of Figure 6.6 clearly show that LVBDD can be exponentially more compact than UNF when using appropriate lattice-valued labels on non-terminal nodes. We therefore introduce a *shared normal form*, whose purpose is to exploit the fact that codomain values are taken from a *structured set*, namely a lattice. By using the structure of the codomain lattice, we can hope for more compact representations of LVBF than can be achieved with ADD, MTBDD, or LVBDD in UNF.

As discussed in the introduction, the shared normal form (SNF for short) of LVBDD is defined using the lattice-theoretic *relative pseudo-complement* operation (RPC for short).

Definition 6.5.3 (Relative Pseudo-complement)**.** *Let $\langle \mathcal{L}, \preceq \rangle$ be a lattice. For any x, y in $\mathcal{L}$ we consider the set $\{z \mid z \sqcap x \preceq y\}$. If this set has a* unique *maximal element, we call this element the* pseudo-complement of x relative to y *and denote it by $x \to y$. Otherwise $x \to y$ is undefined.*

The definition above is somewhat unpractical because it leaves the possibility of $x \to y$ to be undefined. However, we show that for any *finite* and *distributive* lattice, the relative pseudo-complement exists for any pair x, y.

Lemma 6.5.4 (Existence of RPC on Finite Distributive Lattices)**.** *Let $\langle \mathcal{L}, \preceq \rangle$ be a finite distributive lattice. For any x, y in $\mathcal{L}$, the set $\{z \mid z \sqcap x \preceq y\}$ has a unique maximal element.*

Proof. Let us show that for any $z_1, z_2 \in \{z \mid z \sqcap x \preceq y\}$ we have that $z_1 \sqcup z_2 \in \{z \mid z \sqcap x \preceq y\}$. We know that $(z_1 \sqcap x) \sqcup (z_2 \sqcap x) = (z_1 \sqcup z_2) \sqcap x$ by distributivity. Since y is an upper bound of both $(z_1 \sqcap x)$ and $(z_2 \sqcap x)$ it is also an upper-bound of $(z_1 \sqcap x) \sqcup (z_2 \sqcap x)$ and therefore $(z_1 \sqcup z_2) \sqcap x \preceq y$, thus $z_1 \sqcup z_2 \in \{z \mid z \sqcap x \preceq y\}$. By inductively applying this property, since $\{z \mid z \sqcap x \preceq y\}$ is a finite set we know that $\mathsf{LUB}(\{z \mid z \sqcap x \preceq y\}) \in \{z \mid z \sqcap x \preceq y\}$ which implies that $\{z \mid z \sqcap x \preceq y\}$ has a unique maximal element. □

In the context of LVBDD, the key property that we need with the relative pseudo-complement is the fact that $x \to y$ is such that $(x \to y) \sqcap x = y$, and that it is the greatest such element. We show that for finite distributive lattices, this is the case whenever $x \succeq y$.

Corollary 6.5.5. *For any finite distributive lattice $\langle \mathcal{L}, \preceq \rangle$ for any $x, y \in \mathcal{L}$, if $x \succeq y$ then $(x \to y) \sqcap x = y$. Moreover, for every $z \in \mathcal{L}$ such that $z \sqcap x = y$ we have that $z \preceq (x \to y)$.*

We extend the notion of relative pseudo-complement to LVBF as follows.

Definition 6.5.6 (RPC on LVBF)**.** *Let $\theta \in \mathsf{LVBF}(\mathbb{P}, \mathcal{L})$ and let $x \in \mathcal{L}$; if $x \to \theta(v)$ is defined for all $v \in 2^{\mathbb{P}}$, then $x \to \theta$ is defined as $\lambda v \cdot x \to \theta(v)$. Otherwise, $x \to \theta$ is undefined.*

We are now able to properly define the shared normal form for LVBDD. As for the UNF, we provide a constructive definition that naturally describes a deterministic algorithm which builds an LVBDD for any LVBF. Note that our definition of SNF is somewhat more restrictive than UNF in that it requires the codomain lattice to be finite and distributive rather than bounded.

Definition 6.5.7 (Shared Normal Form for LVBDD)**.** *Let $\mathbb{P}$ be a set of Boolean propositions and $\mathcal{L}$ be a finite and distributive lattice. For any $\theta \in \mathsf{LVBF}(\mathsf{P}, \mathcal{L})$, $\mathsf{SNF}(\theta)$ is the* reduced LVBDD *defined recursively as follows. If $\mathsf{Dep}(\theta) = \varnothing$, then $\theta(v) = d$ for some $d \in \mathcal{L}$. In that case, $\mathsf{SNF}(\theta)$ is the terminal* LVBDD *$\langle k+1, d\rangle$. Otherwise, let p_i be the proposition of lowest index in $\mathsf{Dep}(\theta)$, and let $d = \exists\mathbb{P} \cdot \theta$. Then, $\mathsf{SNF}(\theta)$ is the non-terminal* LVBDD *$\langle i, d, \mathsf{SNF}(d \to (\theta|_{p_i=\mathsf{false}})), \mathsf{SNF}((d \to (\theta|_{p_i=\mathsf{true}}))\rangle$. We denote by $\mathsf{SNF}(\mathbb{P}, \mathcal{L})$ the set of* LVBDD *in shared normal form over the set of propositions $\mathbb{P}$ and lattice $\mathcal{L}$, or formally, $\mathsf{SNF}(\mathbb{P}, \mathcal{L}) = \{n \in \mathsf{LVBDD}(\mathbb{P}, \mathcal{L}) \mid \mathsf{SNF}([\![n]\!]) = n\}$.*

A natural consequence of the shared normal form definition is that every non-terminal node has a lattice label that is smaller than the join of the labels of its children. This property will be useful in the sequel.

Lemma 6.5.8. *For every non-terminal* LVBDD *$n \in \mathsf{SNF}(\mathbb{P}, \mathcal{L})$ we have that:*

$$\mathsf{val}(n) \preceq \mathsf{val}(\mathsf{lo}(n)) \sqcup \mathsf{val}(\mathsf{hi}(n))$$

Proof. From Definition 6.5.7, we know that $\mathsf{val}(n) = \exists\mathbb{P} : [\![n]\!]$. By Definition 6.4.2, and by the distributivity of $\mathcal{L}$ we have that $\exists\mathbb{P} : [\![n]\!] = \mathsf{val}(n) \sqcap \big((\exists\mathbb{P} : [\![\mathsf{lo}(n)]\!]) \sqcup (\exists\mathbb{P} : [\![\mathsf{lo}(n)]\!])\big)$. Since $\mathsf{val}(\mathsf{lo}(v)) = \exists\mathbb{P} : [\![\mathsf{lo}(n)]\!]$ and $\mathsf{val}(\mathsf{hi}(v)) = \exists\mathbb{P} : [\![\mathsf{hi}(n)]\!]$, we have that $\mathsf{val}(n) = \mathsf{val}(n) \sqcap \big(\mathsf{val}(\mathsf{lo}(n)) \sqcup \mathsf{val}(\mathsf{hi}(n))\big)$, so $\mathsf{val}(n) \preceq \mathsf{val}(\mathsf{lo}(n)) \sqcup \mathsf{val}(\mathsf{hi}(n))$. □

As can be seen from Definition 6.5.7, the RPC operator is integral to the definition of the shared normal form. In order to develop efficient algorithms for the SNF, we need to investigate a number of *algebraic properties* of the RPC operator on finite distributive lattices (FDL, for short). This is the purpose of the next section.

6.6 Algebraic properties of RPC on FDL

To efficiently manipulate LVBDD in SNF, we make use of the following algebraic properties of the relative pseudo-complement on finite and distributive lattices.

Proposition 6.6.1. *For any* FDL $\langle \mathcal{L}, \preceq \rangle$*, and set of Boolean propositions* $\mathbb{P}$*, and for any* x*,* y *in* $\mathcal{L}$*, for any* $\theta \in \mathsf{LVBF}(\mathbb{P}, \mathcal{L})$*:*

$$x \to (x \sqcap \theta) = \theta \quad \textit{implies} \quad (x \sqcap y) \to (x \sqcap y \sqcap \theta) = y \to (y \sqcap \theta) \tag{6.1}$$

$$x \to (x \sqcap \theta) = \theta \quad \textit{implies} \quad y \to (x \sqcap \theta) = (y \to x) \sqcap \theta \quad \textit{if} \quad y \succeq x \tag{6.2}$$

$$x \to (x \sqcap \theta) = \theta \quad \textit{implies} \quad y \to (y \sqcap \theta) = \theta \quad \textit{if} \quad y \succeq x \tag{6.3}$$

$$x \to (y \to (x \sqcap y \sqcap \theta)) \quad \textit{equals} \quad (x \sqcap y) \to (x \sqcap y \sqcap \theta) \tag{6.4}$$

In this section, we provide a proof for each of the algebraic properties of the proposition above. To mitigate the difficulty of providing proofs that take into account every possible finite distributive lattice, we use Birkhoff's representation theorem for finite and distributive lattices (see Theorem 2.1.3).

6.6.1 The Birkhoff Antichain Lattice

Birkhoff's theorem states that any FDL is isomorphic to the lattice of upward-closed sets of its meet-irreducible elements. Since every upward-closed set can be canonically represented using the antichain of its minimal elements, we can reformulate Birkhoff's theorem in terms of antichains.

Corollary 6.6.2 (Antichains Representation of FDL)**.** *Any finite distributive lattice* $\langle \mathcal{L}, \preceq \rangle$ *is isomorphic to the lattice* antichains of its meet-irreducible elements*, or formally, the lattice* $\langle \mathsf{Antichains}[\mathsf{MIR}[\mathcal{L}, \preceq], \preceq], \sqsubseteq \rangle$*, such that* $A_1 \sqsubseteq A_2$ *iff* $\uparrow A_1 \subseteq \uparrow A_2$*. We call this lattice the* Birkhoff antichain lattice *of* $\langle \mathcal{L}, \preceq \rangle$*.*

Proof. This is a direct consequence of Theorem 2.1.3. □

The isomorphism between a finite and distributive lattice and its Birkhoff antichain lattice is the composition of the classical Birkhoff isomorphism of Theorem 2.1.3 with the $\lfloor \cdot \rfloor$ operation.

Lemma 6.6.3 (Birkhoff Antichain Lattice Isomorphism)**.** *Let $\langle \mathcal{L}, \preceq \rangle$ be a finite and distributive lattice, and let $\langle \mathcal{B}, \sqsubseteq \rangle$ be the corresponding Birkhoff antichain lattice. The following function $\sigma : \mathcal{L} \mapsto \mathcal{B}$ is a lattice isomorphism:*

$$\begin{aligned} \sigma(x) &= \lfloor \{ y \in \mathsf{MIR}[\mathcal{L}, \preceq] \mid y \succeq x \} \rfloor \\ \sigma^{-1}(X) &= \mathsf{GLB}[\mathcal{L}, \preceq](X) \end{aligned}$$

The meet and join operations, as well as the partial order, are easily defined on the Birkhoff antichain lattice, as shown by the following lemma.

Lemma 6.6.4 (Operations on the Birkhoff Antichain Lattice)**.** *Let $\langle \mathcal{L}, \preceq \rangle$ be a finite and distributive lattice, and let $\langle \mathcal{B}, \sqsubseteq \rangle$ be the corresponding Birkhoff antichain lattice. For every $X, Y \in \mathcal{B}$ we have that:*

$$\begin{aligned} X \sqcap Y &= \lfloor X \cup Y \rfloor \\ X \sqsubseteq Y &\text{ iff } \forall y \in Y \; \exists x \in X \colon x \preceq y \end{aligned}$$

6.6.2 RPC on the Birkhoff Antichain Lattice

The key property of the lattice of the Birkhoff antichain lattice, is the fact that the RPC operation (between comparable elements) amounts to a simple *set difference* on that lattice.

Lemma 6.6.5. *Let $\langle \mathcal{L}, \preceq \rangle$ be a finite and distributive lattice, let $\langle \mathcal{B}, \sqsubseteq \rangle$ be the corresponding Birkhoff antichain lattice, and let $\sigma : \mathcal{L} \mapsto \mathcal{B}$ be the isomorphism between them. For every $x, y \in \mathcal{L}$ such that $x \succeq y$ we have that:*

$$\sigma(x \to y) = \sigma(y) \setminus \sigma(x)$$

Proof. Since σ is a bijection, we can show equivalently that $x \to y = \sigma^{-1}(\sigma(y) \setminus \sigma(x))$. By definition of $\to$ we have to show that $\sigma^{-1}(\sigma(y) \setminus \sigma(x))$ is the $\preceq$-maximal element of $\{z \mid z \sqcap x \preceq y\}$ (which exists and is unique by Lemma 6.5.4). For that purpose, we first show that $\sigma^{-1}(\sigma(y) \setminus \sigma(x))$ belongs to $\{z \mid z \sqcap x \preceq y\}$, i.e. that $\sigma^{-1}(\sigma(y) \setminus \sigma(x)) \sqcap x \preceq y$. This is obtained as

follows:

$$\begin{array}{lll} & \sigma^{-1}(\sigma(y) \setminus \sigma(x)) \sqcap x & \\ = & \sigma^{-1}(\sigma(y) \setminus \sigma(x) \sqcap \sigma(x)) & \text{Corollary 6.6.2} \\ = & \sigma^{-1}(\lfloor \sigma(y) \setminus \sigma(x) \cup \sigma(x) \rfloor) & \text{Lemma 6.6.4} \\ = & \sigma^{-1}(\lfloor \sigma(y) \cup \sigma(x) \rfloor) & \\ = & \sigma^{-1}(\sigma(y) \sqcap \sigma(x)) & \text{Lemma 6.6.4} \\ = & y \sqcap x & \text{Corollary 6.6.2} \\ \preceq & y & \end{array}$$

Then, we show that $\sigma^{-1}(\sigma(y) \setminus \sigma(x))$ is the maximal element of $\{z \mid z \sqcap x \preceq y\}$, i.e. that, for all $z \in \mathcal{L}$ such that $z \sqcap x = y$ we have that $z \preceq \sigma^{-1}(\sigma(y) \setminus \sigma(x))$. We proceed by contradiction. Assume that there is $z \in \mathcal{L}$ such that $z \sqcap x = y$ and $z \not\preceq \sigma^{-1}(\sigma(y) \setminus \sigma(x))$. Thus, z is such that:

$$\begin{array}{lll} & z \sqcap x = y \text{ and } z \not\preceq \sigma^{-1}(\sigma(y) \setminus \sigma(x))\big) & \\ \Leftrightarrow & \sigma(z) \sqcap \sigma(x) = \sigma(y) \text{ and } \sigma(z) \not\sqsubseteq \sigma(y) \setminus \sigma(x) & (a) \\ \Leftrightarrow & \lfloor \sigma(z) \cup \sigma(x) \rfloor = \sigma(y) \text{ and } \exists e \in \sigma(y) \setminus \sigma(x) : \forall e' \in \sigma(z) : e \not\preceq e' & (b) \\ \Rightarrow & \lfloor \sigma(z) \cup \sigma(x) \rfloor = \sigma(y) \text{ and } \exists e \in \sigma(y) \setminus \sigma(x) : e \notin \sigma(z) & \\ \Rightarrow & \sigma(z) \cup \sigma(x) \supseteq \sigma(y) \text{ and } \sigma(y) \not\subseteq \sigma(z) \cup \sigma(x) & (c) \end{array}$$

Point (a) is shown by Corollary 6.6.2, (b) by Lemma 6.6.4, and (c) by definition of $\lfloor \cdot \rfloor$ and $\subseteq$. The last line is clearly a contradiction, and such a z cannot exist. □

6.6.3 Proofs of RPC's Algebraic Properties

Equipped with our definition of the relative pseudo-complement over the Birkhoff antichain lattice, we provide a proof for each of the four algebraic properties of Proposition 6.6.1.

Lemma 6.6.6. *For every* FDL $\langle \mathcal{L}, \preceq \rangle$*, for all* x*,* y*,* z *in* $\mathcal{L}$*,* if $x \to (x \sqcap y) = y$ then $(x \sqcap z) \to (x \sqcap y \sqcap z) = z \to (y \sqcap z)$.

Proof. Let $\langle \mathcal{B}, \sqsubseteq \rangle$ be the Birkhoff antichain lattice of $\mathcal{L}$, let x, y and z be three elements from $\mathcal{L}$, and let $\sigma : \mathcal{L} \mapsto \mathcal{B}$ be the corresponding lattice isomorphism. Throughout the proof, we will denote, $\sigma(x)$, $\sigma(y)$ and $\sigma(z)$

respectively by $\hat{x}$, $\hat{y}$ and $\hat{z}$, in order to alleviate the notations. Let us first rephrase the statement of the Lemma by exploiting the bijection σ:

$$\begin{array}{lll} & \text{if} \quad x \to (x \sqcap y) = y & \\ & \text{then} \quad (x \sqcap z) \to (x \sqcap y \sqcap z) = z \to (y \sqcap z) & \\ \Leftrightarrow & \text{if} \quad \sigma(x \to (x \sqcap y)) = \hat{y} & \\ & \text{then} \quad \sigma((x \sqcap z) \to (x \sqcap y \sqcap z)) = \sigma(z \to (y \sqcap z)) & \text{Lemma 6.6.3} \\ \Leftrightarrow & \text{if} \quad \sigma(x \sqcap y) \setminus \hat{x} = \hat{y} & \\ & \text{then} \quad \sigma(x \sqcap y \sqcap z) \setminus \sigma(x \sqcap z) = \sigma(y \sqcap z) \setminus \hat{z} & \text{Lemma 6.6.5} \\ \Leftrightarrow & \text{if} \quad \big(\hat{x} \sqcap \hat{y}\big) \setminus \hat{x} = \hat{y} & \\ & \text{then} \quad \big(\hat{x} \sqcap \hat{y} \sqcap \hat{z}\big) \setminus \big(\hat{x} \sqcap \hat{z}\big) = \big(\hat{y} \sqcap \hat{z}\big) \setminus \hat{z} & \text{Lemma 6.6.3} \\ \Leftrightarrow & \text{if} \quad \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{x} = \hat{y} & \\ & \text{then} \quad \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus \lfloor \hat{x} \cup \hat{z} \rfloor = \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z} & \text{Lemma 6.6.4} \end{array}$$

Now let us prove that this last line holds. Recall that, by definition, $\hat{x}$, $\hat{y}$ and $\hat{z}$ are antichains of elements of $\mathsf{MIR}[\mathcal{L}, \preceq]$. First, observe that the hypothesis $\lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{x} = \hat{y}$ implies that:

$$\forall e_y \in \hat{y} : \nexists e_x \in \hat{x} : e_x \preceq e_y. \tag{6.5}$$

Indeed, if there were $e_y \in \hat{y}$ and $e_x \in \hat{x}$ with $e_x \preceq e_y$, then $e_y \notin \lfloor \hat{x} \cup \hat{y} \rfloor$, and thus, $e_y \in \hat{y}$ but $e_y \notin \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{x}$, which contradicts $\lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{x} = \hat{y}$. Under the hypothesis (6.5), let us now prove that (i) $\lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus \lfloor \hat{x} \cup \hat{z} \rfloor \supseteq \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z}$ and that (ii) $\lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus \lfloor \hat{x} \cup \hat{z} \rfloor \subseteq \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z}$.

For point (i), we consider $e \in \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z}$. Observe that $\lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z} \subseteq \lfloor \hat{y} \cup \hat{z} \rfloor \subseteq \lfloor \hat{y} \rfloor \cup \lfloor \hat{z} \rfloor$. Thus, either $e \in \lfloor \hat{y} \rfloor$ or $e \in \lfloor \hat{z} \rfloor$. However, if $e \in \lfloor \hat{z} \rfloor$, we have $e \in \hat{z}$, and it is thus not possible to have $e \in \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z}$. We conclude that $e \in \lfloor \hat{y} \rfloor$. Moreover, there is no $e_z \in \hat{z}$ such that $e_z \preceq e$, otherwise e would not belong to $\lfloor \hat{y} \cup \hat{z} \rfloor$. Since $e \in \hat{y}$, there is also no $e_x \in \hat{x}$ such that $e_x \preceq e$, by (6.5). Thus, $e \in \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor$, but $e \notin \lfloor \hat{x} \cup \hat{z} \rfloor$. Hence, $e \in \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus \lfloor \hat{x} \cup \hat{z} \rfloor$. Since this is valid for all $e \in \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z}$ we conclude that $\lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus \lfloor \hat{x} \cup \hat{z} \rfloor \supseteq \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z}$.

For point (ii), we consider $e \in \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus \lfloor \hat{x} \cup \hat{z} \rfloor$. By the same reasoning as above, we conclude that $e \in \lfloor \hat{y} \rfloor$ and that there is no $e_z \in \hat{z}$ such that

$e_z \preceq e$. Thus, $e \in \lfloor \hat{y} \cup \hat{z} \rfloor$, but $e \notin \hat{z}$. Hence, $e \in \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z}$. Since this is valid for all $e \in \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus \lfloor \hat{x} \cup \hat{z} \rfloor$ we conclude that $\lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus \lfloor \hat{x} \cup \hat{z} \rfloor \subseteq \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z}$. □

Corollary 6.6.7. *For any set of Boolean propositions* $\mathbb{P}$*, for any* FDL $\langle \mathcal{L}, \preceq \rangle$*, for any* x*,* y *in* $\mathcal{L}$*, for any* $\theta \in \mathsf{LVBF}(\mathbb{P}, \mathcal{L})$*:* if $x \to (x \sqcap \theta) = \theta$ then $(x \sqcap y) \to (x \sqcap \theta \sqcap y) = y \to (\theta \sqcap y)$.

Lemma 6.6.8. *For all* FDL $\langle \mathcal{L}, \preceq \rangle$*, for all* x*,* y*,* z *in* L *with* $z \succeq x$*:* if $x \to (x \sqcap y) = y$ then $z \to (x \sqcap y) = (z \to x) \sqcap y$.

Proof. Let $\langle \mathcal{B}, \sqsubseteq \rangle$ be the Birkhoff antichain lattice of $\langle \mathcal{L}, \preceq \rangle$, let x, y and z be three elements from $\mathcal{L}$, and let $\sigma : \mathcal{L} \mapsto \mathcal{B}$ be the corresponding lattice isomorphism. Throughout the proof, we will denote, $\sigma(x)$, $\sigma(y)$ and $\sigma(z)$ respectively by $\hat{x}$, $\hat{y}$ and $\hat{z}$, in order to alleviate the notations. Let us first rephrase the statement of the Lemma by exploiting the bijection σ:

$$
\begin{array}{lrll}
 & \text{if} & x \to (x \sqcap y) = y & \\
 & \text{then} & z \to (x \sqcap y) = (z \to x) \sqcap y & \\
\Leftrightarrow & \text{if} & \sigma(x \to (x \sqcap y)) = \hat{y} & \\
 & \text{then} & \sigma(z \to (x \sqcap y)) = \sigma((z \to x) \sqcap y) & \text{Lemma 6.6.3} \\
\Leftrightarrow & \text{if} & \sigma(x \sqcap y) \setminus \hat{x} = \hat{y} & \\
 & \text{then} & \sigma(x \sqcap y) \setminus \hat{z} = \sigma((z \to x) \sqcap y) & \text{Lemma 6.6.5} \\
\Leftrightarrow & \text{if} & (\hat{x} \sqcap \hat{y}) \setminus \hat{x} = \hat{y} & \\
 & \text{then} & (\hat{x} \sqcap \hat{y}) \setminus \hat{z} = \sigma(z \to x) \sqcap \hat{y} & \text{Corollary 6.6.2} \\
\Leftrightarrow & \text{if} & (\hat{x} \sqcap \hat{y}) \setminus \hat{x} = \hat{y} & \\
 & \text{then} & (\hat{x} \sqcap \hat{y}) \setminus \hat{z} = (\hat{x} \setminus \hat{z}) \sqcap \hat{y} & \text{Lemma 6.6.5} \\
\Leftrightarrow & \text{if} & \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{x} = \hat{y} & \\
 & \text{then} & \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{z} = \lfloor (\hat{x} \setminus \hat{z}) \cup \hat{y} \rfloor & \text{Lemma 6.6.4}
\end{array}
$$

Now let us prove that this last line holds. This part of the proof is very similar to the end of the proof of Lemma 6.6.6. Recall that, by definition, $\hat{x}$, $\hat{y}$ and $\hat{z}$ are antichains of elements of $\mathsf{MIR}[\mathcal{L}, \preceq]$. First, observe that, as in the proof of Lemma 6.6.6, the hypothesis $\lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{x} = \hat{y}$ implies that:

$$\forall e_y \in \hat{y} : \nexists\, e_x \in \hat{x} : e_x \preceq e_y \tag{6.6}$$

Moreover, we know that $x \preceq z$. Hence, by Corollary 6.6.2, $\hat{x} \sqsubseteq \hat{z}$. Thus, by definition of $\sqsubseteq$, this means that for all $e_z \in \hat{z}$, there is $e_x \in \hat{x}$ such that $e_x \preceq e_z$. As a consequence, for all $e_y \in \hat{y}$, there is no $e_z \in \hat{z}$ such that $e_z \preceq e_y$. Indeed, if there were $e_z \in \hat{z}$ and $e_y \in \hat{y}$ such that $e_z \preceq e_y$, we know that there is $e_x \in \hat{x}$ with $e_x \preceq e_z$, and thus $e_x \preceq e_y$, which is not possible by (6.6). We thus conclude that:

$$\forall e_y \in \hat{y} \;\; : \;\; \left(\nexists e_x \in \hat{x} : e_x \preceq e_y \text{ and } \nexists e_z \in \hat{z} : e_z \preceq e_y\right) \tag{6.7}$$

Under the hypothesis (6.7), let us now prove that (i) $\lfloor \hat{x} \cup \hat{y} \rfloor \backslash \hat{z} \supseteq \lfloor (\hat{x} \setminus \hat{z}) \cup \hat{y} \rfloor$ and $(ii) \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{z} \subseteq \lfloor (\hat{x} \setminus \hat{z}) \cup \hat{y} \rfloor$.

For point (i), we consider $e \in \lfloor (\hat{x} \setminus \hat{z}) \cup \hat{y} \rfloor$. We have two possibilities. Either (a) $e \in \lfloor \hat{x} \setminus \hat{z} \rfloor$ and there is no $e_y \in \hat{y}$ with $e_y \preceq e$, or (b) $e \in \lfloor \hat{y} \rfloor$. In the first case (a), it is clear that $e \in \lfloor \hat{x} \rfloor$. Since there is no $e_y \in \hat{y}$ with $e_y \preceq e$, e is in $\lfloor \hat{x} \cup \hat{y} \rfloor$ too. Moreover, $e \in \lfloor \hat{x} \setminus \hat{z} \rfloor \subseteq \hat{x} \setminus \hat{z}$ implies that $e \notin \hat{z}$. We conclude thus that $e \in \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{z}$. In the former case (b), we can invoke (6.7), since $e \in \lfloor \hat{y} \rfloor$ implies that $e \in \hat{y}$. From $e \in \lfloor \hat{y} \rfloor$ and the fact that there is no $e_x \in \hat{x}$ such that $e_x \preceq e$, we obtain $e \in \lfloor \hat{x} \cup \hat{y} \rfloor$. From the fact that there is no $e_z \in \hat{z}$ such that $e_z \preceq e$ we obtain $e \notin \hat{z}$. Thus, $e \in \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{z}$. Since this is valid for all $e \in \lfloor (\hat{x} \setminus \hat{z}) \cup \hat{y} \rfloor$ we conclude that $\lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{z} \succeq \lfloor (\hat{x} \setminus \hat{z}) \cup \hat{y} \rfloor$.

For point (ii), we consider $e \in \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{z}$. Clearly, $e \notin \hat{z}$, and either (a) $e \in \lfloor \hat{x} \rfloor$ and there is no $e_y \in \hat{y}$ such that $e_y \preceq e$, or (b) $e \in \lfloor \hat{y} \rfloor$ and there is no $e_x \in \hat{x}$ such that $e_x \preceq e$. In the former case (a), $e \in \hat{x}$ but $e \notin \hat{z}$. Hence, $e \in \hat{x} \setminus \hat{z}$. Moreover, since there is no $e_y \in \hat{y}$ with $e_y \preceq e$, we conclude that $e \in \lfloor (\hat{x} \setminus \hat{z}) \cup \hat{y} \rfloor$. In the latter case (b), $e \in \hat{y}$, and, since there is no $e_x \in \hat{x}$ with $e_x \sqsubseteq e$, we have, in particular, that there is no $e' \in \hat{x} \setminus \hat{z}$ such that $e' \preceq e$. Thus, we obtain: $e \in \lfloor \hat{y} \cup (\hat{x} \setminus \hat{z}) \rfloor$. Since this is valid for all $e \in \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{z}$ we conclude that $\lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{z} \subseteq \lfloor (\hat{x} \setminus \hat{z}) \cup \hat{y} \rfloor$. □

Corollary 6.6.9. *For any set of Boolean propositions $\mathbb{P}$, for any* FDL $\langle \mathcal{L}, \preceq \rangle$*, for any x, y in $\mathcal{L}$ with $y \succeq x$, for any $\theta \in$* LVBF $(\mathbb{P}, \mathcal{L})$*:* if $x \to (x \sqcap \theta) = \theta$ then $y \to (x \sqcap \theta) = (y \to x) \sqcap \theta$.

Lemma 6.6.10. *For any* FDL $\langle \mathcal{L}, \preceq \rangle$*, for any x, y, z in $\mathcal{L}$ with $z \succeq x$:* if $x \to (x \sqcap y) = y$ then $z \to (z \sqcap y) = y$.

Proof. Let $\langle \mathcal{B}, \sqsubseteq \rangle$ be the Birkhoff antichain lattice of $\langle \mathcal{L}, \preceq \rangle$, let x, y and z be three elements from $\mathcal{L}$, and let $\sigma : \mathcal{L} \mapsto \mathcal{B}$ be the corresponding lattice isomorphism. Throughout the proof, we will denote, $\sigma(x)$, $\sigma(y)$ and $\sigma(z)$ respectively by $\hat{x}$, $\hat{y}$ and $\hat{z}$, in order to alleviate the notations. Let us first rephrase the statement of the Lemma by exploiting the bijection σ:

$$
\begin{aligned}
& \text{if } x \to (x \sqcap y) = y \text{ then } z \to (z \sqcap y) = y \\
\Leftrightarrow\ & \text{if } \sigma(x \to (x \sqcap y)) = \hat{y} \text{ then } \sigma(z \to (z \sqcap y)) = \hat{y} && \text{Lemma 6.6.3} \\
\Leftrightarrow\ & \text{if } \sigma(x \sqcap y) \setminus \hat{x} = \hat{y} \text{ then } \sigma(z \sqcap y) \setminus \hat{z} = \hat{y} && \text{Lemma 6.6.5} \\
\Leftrightarrow\ & \text{if } (\hat{x} \sqcap \hat{y}) \setminus \hat{x} = \hat{y} \text{ then } (\hat{z} \sqcap \hat{y}) \setminus \hat{z} = \hat{y} && \text{Corollary 6.6.2} \\
\Leftrightarrow\ & \text{if } \lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{x} = \hat{y} \text{ then } \lfloor \hat{z} \cup \hat{y} \rfloor \setminus \hat{z} = \hat{y} && \text{Lemma 6.6.4}
\end{aligned}
$$

Now let us prove that this last line holds. Recall that, by definition, $\hat{x}$, $\hat{y}$ and $\hat{z}$ are antichains of elements of $\mathsf{MIR}[\mathcal{L}, \preceq]$. First, observe that the hypothesis $\lfloor \hat{x} \cup \hat{y} \rfloor \setminus \hat{x} = \hat{y}$ together with the fact that $z \succeq x$ imply that:

$$\forall e_y \in \hat{y} : \left(\nexists e_x \in \hat{x} : e_x \preceq e_y \text{ and } \nexists e_z \in \hat{z} : e_z \preceq e_y \right) \tag{6.8}$$

as established in the proof of Lemma 6.6.8. Under the hypothesis (6.8), let us now prove that (i) $\lfloor \hat{z} \cup \hat{y} \rfloor \setminus \hat{z} \subseteq \hat{y}$ and that (ii) $\lfloor \hat{z} \cup \hat{y} \rfloor \setminus \hat{z} \supseteq \hat{y}$.

For point (i), we consider $e \in \lfloor \hat{z} \cup \hat{y} \rfloor \setminus \hat{z}$. Since $\lfloor \hat{z} \cup \hat{y} \rfloor \setminus \hat{z} \subseteq \lfloor \hat{z} \cup \hat{y} \rfloor \subseteq \lfloor \hat{z} \rfloor \cup \lfloor \hat{y} \rfloor$, e is either in $\lfloor \hat{y} \rfloor$ or in $\lfloor \hat{z} \rfloor$. However, since $e \in \lfloor \hat{z} \cup \hat{y} \rfloor \setminus \hat{z}$, e cannot by in $\hat{z}$, neither in $\lfloor \hat{z} \rfloor \subseteq \hat{z}$. Thus, $e \in \lfloor \hat{y} \rfloor$. Since $\lfloor \hat{y} \rfloor \subseteq \hat{y}$, we have $e \in \hat{y}$. Since this is valid for all $e \in \lfloor \hat{z} \cup \hat{y} \rfloor \setminus \hat{z}$, we conclude that $\lfloor \hat{z} \cup \hat{y} \rfloor \setminus \hat{z} \subseteq \hat{y}$.

For point (ii), we consider $e \in \hat{y}$. Since $\hat{y}$ is an antichain, by definition, we have $\hat{y} = \lfloor \hat{y} \rfloor$, and thus $e \in \lfloor \hat{y} \rfloor$. Since there is no $e_z \in \hat{z}$ s.t. $e_z \sqsubseteq e$, by (6.8), we obtain that $e \in \lfloor \hat{y} \cup \hat{z} \rfloor$, and that $e \notin \hat{z}$. As a consequence $e \in \lfloor \hat{y} \cup \hat{z} \rfloor \setminus \hat{z}$. Since this is valid for all $e \in \hat{y}$, we conclude that $\lfloor \hat{z} \cup \hat{y} \rfloor \setminus \hat{z} \supseteq \hat{y}$. □

Corollary 6.6.11. *For any set of Boolean propositions $\mathbb{P}$, for any* FDL *$\langle \mathcal{L}, \preceq \rangle$, for any x, y in $\mathcal{L}$ with $y \succeq x$, for all $\theta \in \mathsf{LVBF}(\mathsf{P}, \mathcal{L})$:* if $x \to (x \sqcap \theta) = \theta$ then $y \to (y \sqcap \theta) = \theta$.

Lemma 6.6.12. *For any* FDL $\langle \mathcal{L}, \preceq \rangle$*, for any* x*,* y*,* z *in* $\mathcal{L}$*, we have that:* $x \to (y \to (x \sqcap y \sqcap z)) = (x \sqcap y) \to (x \sqcap y \sqcap z)$.

Proof. Let $\langle \mathcal{B}, \sqsubseteq \rangle$ be the Birkhoff antichain lattice of $\langle \mathcal{L}, \preceq \rangle$, let x, y and z be three elements from $\mathcal{L}$, and let $\sigma : \mathcal{L} \mapsto \mathcal{B}$ be the corresponding lattice isomorphism. Throughout the proof, we will denote, $\sigma(x)$, $\sigma(y)$ and $\sigma(z)$ respectively by $\hat{x}$, $\hat{y}$ and $\hat{z}$, in order to alleviate the notations. Let us first rephrase the statement of the Lemma by exploiting the bijection σ:

$$
\begin{aligned}
& x \to (y \to (x \sqcap y \sqcap z)) = (x \sqcap y) \to (x \sqcap y \sqcap z) \\
\Leftrightarrow\ & \sigma(x \to (y \to (x \sqcap y \sqcap z))) = \sigma((x \sqcap y) \to (x \sqcap y \sqcap z)) && \text{Lemma 6.6.3} \\
\Leftrightarrow\ & \sigma(x \sqcap y \sqcap z) \setminus (\hat{y} \cup \hat{x}) = \sigma(x \sqcap y \sqcap z) \setminus \sigma(x \sqcap y) && \text{Lemma 6.6.5} \\
\Leftrightarrow\ & (\hat{x} \sqcap \hat{y} \sqcap \hat{z}) \setminus (\hat{y} \cup \hat{x}) = (\hat{x} \sqcap \hat{y} \sqcap \hat{z}) \setminus (\hat{x} \sqcap \hat{y}) && \text{Corollary 6.6.2} \\
\Leftrightarrow\ & \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus (\hat{y} \cup \hat{x}) = \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor \setminus \lfloor \hat{x} \cup \hat{y} \rfloor && \text{Lemma 6.6.4} \\
\Leftrightarrow\ & ((\hat{x} \cup \hat{y}) \setminus \lfloor \hat{y} \cup \hat{x} \rfloor) \cap \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor = \varnothing
\end{aligned}
$$

Note that last line is obtain from the previous line by using the fact that $\lfloor \hat{x} \cup \hat{y} \rfloor \subseteq (\hat{x} \cup \hat{y})$. We now show that the last line holds. First, we reformulate it as follows:

$$\forall e \in \mathcal{L}\colon \text{ if } e \in (\hat{y} \cup \hat{x}) \text{ and } e \notin \lfloor \hat{x} \cup \hat{y} \rfloor \text{ then } e \notin \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor$$

Clearly, if $e \in (\hat{y} \cup \hat{x})$ and $e \notin \lfloor \hat{x} \cup \hat{y} \rfloor$ then $\exists e' \in \lfloor \hat{x} \cup \hat{y} \rfloor$ such that $e \succ e'$. If $e' \in \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor$ then clearly $e \notin \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor$. If $e' \notin \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor$ then $\exists e'' \in \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor$ such that $e \succ e' \succ e''$, which also implies that $e \notin \lfloor \hat{x} \cup \hat{y} \cup \hat{z} \rfloor$. □

Corollary 6.6.13. *For any set of Boolean propositions* $\mathbb{P}$*, for any* FDL $\langle \mathcal{L}, \preceq \rangle$*, for any* x*,* y *in* $\mathcal{L}$ *with* $y \succeq x$*, for all* $\theta \in \mathsf{LVBF}(\mathsf{P}, \mathcal{L})$*:* $x \to (y \to (x \sqcap y \sqcap \theta)) = (x \sqcap y) \to (x \sqcap y \sqcap \theta)$.

6.7 LVBDD Manipulation Algorithms

In this section, we discuss *symbolic* algorithms to manipulate LVBF *via* their LVBDD representation. Throughout this section, we assume that

we manipulate LVBDD ranging over a totally ordered set of propositions $\mathbb{P} = \{p_1, \ldots, p_k\}$ and the finite and distributive lattice $\langle \mathcal{L}, \preceq \rangle$. We present algorithms to compute the join, the meet, and the existential or universal quantification of *all* the Boolean variables in $\mathbb{P}$, for LVBDD in both shared and unshared normal forms.

6.7.1 Memory Management and Memoization

The creation of LVBDD nodes in memory is carried out by function MK: calling $\mathsf{MK}(i, d, \ell, h)$ returns the LVBDD node $\langle i, d, \ell, h \rangle$. As in most BDD packages (see for instance [LN]), our implementation exploits caching techniques to ensure that each unique LVBDD is stored only once, even across multiple diagrams. The implementation maintains a global cache that maps each tuple $\langle i, d, \ell, h \rangle$ to a memory address storing the corresponding LVBDD node (if it exists). A call to $\mathsf{MK}(i, d, \ell, h)$ first queries the cache and allocates fresh memory space for $\langle i, d, \ell, h \rangle$ in the case of a cache miss. Thus, MK guarantees that two isomorphic LVBDD always occupy the same memory address, but does not guarantee that any particular normal form is enforced. We assume that cache queries and updates take $\mathcal{O}(1)$ which is what is observed in practice when using a good hash map. We implemented a simple reference counting scheme to automatically free unreferenced nodes from the cache.

We use the standard *memoization* technique in graph traversal algorithms. Each algorithm has access to its own pair of memo! and memo? functions; $\mathsf{memo!}(key, value)$ stores a computed value and associates it to a key; $\mathsf{memo?}(key)$ returns the previously stored value, or nil if none was found. Both memo! and memo? are assumed to run in $\mathcal{O}(1)$.

6.7.2 Relative Pseudo-complement of LVBDD in SNF

In order to efficiently manipulate LVBDD in shared normal form, we need an efficient way of computing the RPC operation on LVBDD. Considering the definition of the RPC over LVBF: $d \to \theta \stackrel{\mathsf{def}}{=} \lambda v \cdot (d \to \theta(v))$, one might assume that computing the RPC over LVBDD is non-trivial. Remarkably however, *computing the* RPC *of an* LVBDD *in* SNF *only amounts to computing the*

Algorithm 6.1: Relative pseudo-complementation for LVBDD in SNF

```
1 begin PseudoCompSNF(n, d)
2     if index(n) = k + 1 then
3         return MK(k + 1, d → val(n), nil, nil) ;
4     else
5         d' ← (d → val(n)) ⊓ (val(lo(n)) ⊔ val(hi(n))) ;
6         return MK(index(n), d', lo(n), hi(n)) ;
7 end
```

RPC *of its root label*, and is therefore computable in *constant time* with respect to the number of nodes of the LVBDD. Proving this requires the use of the algebraic properties of the RPC, particularly Proposition 6.6.1 Point 6.2 which transforms an RPC operation of an LVBF into an RPC of a constant value.

The procedure PseudoCompSNF(n, d) (see Algorithm 6.1) takes an LVBDD $n \in \mathsf{LVBF}(\mathbb{P}, \mathcal{L})$ and a lattice value $d \succeq \mathsf{val}(n)$, and computes the LVBDD $n' = \mathsf{SNF}(d \to [\![n]\!])$.

Lemma 6.7.1 (Soundness of PseudoCompSNF)**.** *For all* $n \in \mathsf{SNF}(\mathbb{P}, \mathcal{L})$ *and* $d \in \mathcal{L}$ *with* $d \succeq \mathsf{val}(n)$ *we have that* PseudoCompSNF$(n, d) = \mathsf{SNF}(d \to [\![n]\!])$.

Proof. If n is terminal, the result is immediate. Assume that n is non-terminal and let $i = \mathsf{index}(n)$. Clearly, the algorithm returns a non-terminal node $n' = \langle i, (d \to \mathsf{val}(n)) \sqcap (\mathsf{val}(\mathsf{lo}(n)) \sqcup \mathsf{val}(\mathsf{hi}(n))), \mathsf{lo}(n), \mathsf{hi}(n) \rangle$.

We start by showing that $[\![n']\!] = d \to [\![n]\!]$. From Definition 6.4.2 we have that $d \to [\![n]\!] = d \to \big(\mathsf{val}(n) \sqcap (\neg p_i \sqcap [\![\mathsf{lo}(n)]\!]) \sqcup (p_i \sqcap [\![\mathsf{hi}(n)]\!])\big)$. Let $v \in 2^{\mathbb{P}}$, and recall that $d \to \theta \stackrel{\mathsf{def}}{=} \lambda v \cdot (d \to \theta(v))$; we see that for any valuation v:

$$(d \to [\![n]\!])(v) = \begin{cases} d \to \big(\mathsf{val}(n) \sqcap [\![\mathsf{lo}(n)]\!](v)\big) & \text{if } \; p_i \notin v \\ d \to \big(\mathsf{val}(n) \sqcap [\![\mathsf{hi}(n)]\!](v)\big) & \text{if } \; p_i \in v \end{cases}$$

From Definition 6.5.7 we know that $[\![\mathsf{lo}(n)]\!] = \mathsf{val}(n) \to ([\![n]\!]|_{p_i=\mathsf{false}})$. It is easy to see that $[\![n]\!]|_{p_i=\mathsf{false}} = \mathsf{val}(n) \sqcap [\![\mathsf{lo}(n)]\!]$, thus we have that $[\![\mathsf{lo}(n)]\!] = \mathsf{val}(n) \to (\mathsf{val}(n) \sqcap [\![\mathsf{lo}(n)]\!])$. With similar reasoning we obtain: $[\![\mathsf{hi}(n)]\!] =$

$\mathsf{val}(n) \to (\mathsf{val}(n) \sqcap [\![\mathsf{hi}(n)]\!])$. By using Proposition 6.6.1, point (6.2) and the decomposition of $(d \to [\![n]\!])(v)$ above, we now have, for any valuation v:

$$(d \to [\![n]\!])(v) = \begin{cases} (d \to \mathsf{val}(n)) \sqcap [\![\mathsf{lo}(n)]\!](v) & \text{if } \ p_i \notin v \\ (d \to \mathsf{val}(n)) \sqcap [\![\mathsf{hi}(n)]\!](v) & \text{if } \ p_i \in v \end{cases}$$

We thus have that: $d \to [\![n]\!] = \big(d \to \mathsf{val}(n)\big) \sqcap \big((\neg p_i \sqcap [\![\mathsf{lo}(n)]\!]) \sqcup (p_i \sqcap [\![\mathsf{hi}(n)]\!])\big)$. It is now easy to see that $d \to [\![n]\!] = d' \sqcap \big((\neg p_i \sqcap [\![\mathsf{lo}(n)]\!]) \sqcup (p_i \sqcap [\![\mathsf{hi}(n)]\!])\big) = [\![n']\!]$, since $(\mathsf{val}(\mathsf{lo}(n)) \sqcup \mathsf{val}(\mathsf{hi}(n))) \sqcap [\![\mathsf{lo}(n)]\!] = [\![\mathsf{lo}(n)]\!]$ and likewise for $[\![\mathsf{hi}(n)]\!]$.

We have shown that n' has the correct semantics so it remains to prove that n' is in shared normal form. We begin by showing that n' has the correct label, i.e. we need to show that $\exists \mathbb{P} : [\![n']\!] = \mathsf{val}(n') = d'$. We know that $[\![n']\!] = d \to [\![n]\!]$ so by reusing the results above we have:

$$\begin{aligned} & \exists\, \mathbb{P} : [\![n']\!] \\ = \ & \exists\, \mathbb{P} : (d \to \mathsf{val}(n)) \sqcap \big((\neg p_i \sqcap [\![\mathsf{lo}(n)]\!]) \sqcup (p_i \sqcap [\![\mathsf{hi}(n)]\!])\big) \\ = \ & (d \to \mathsf{val}(n)) \sqcap \big(\mathsf{val}(\mathsf{lo}(n)) \sqcup \mathsf{val}(\mathsf{hi}(n))\big) \\ = \ & d' \end{aligned}$$

It remains to show that n' has the correct low- and high-child, i.e. $[\![\mathsf{lo}(n)]\!] = \mathsf{val}(n') \to ([\![n']\!]|_{p_i=\mathsf{false}})$ and $[\![\mathsf{hi}(n)]\!] = \mathsf{val}(n') \to ([\![n']\!]|_{p_i=\mathsf{true}})$. We know that $[\![n']\!]|_{p_i=\mathsf{false}} = d' \sqcap [\![\mathsf{lo}(n)]\!]$, and since $\mathsf{val}(n') = d'$ we have that $\mathsf{val}(n') \to ([\![n']\!]|_{p_i=\mathsf{false}}) = d' \to (d' \sqcap [\![\mathsf{lo}(n)]\!])$. By definition of $\to$ and by Lemma 6.5.8, it is easy to see that $\mathsf{val}(n) \preceq d'$. We have already shown that $[\![\mathsf{lo}(n)]\!] = \mathsf{val}(n) \to (\mathsf{val}(n) \sqcap [\![\mathsf{lo}(n)]\!])$, thus by using Proposition 6.6.1, point (6.3), we have that $[\![\mathsf{lo}(n)]\!] = d' \to (d' \sqcap [\![\mathsf{lo}(n)]\!]) = d' \to ([\![n']\!]|_{p_i=\mathsf{false}})$. Since $\mathsf{lo}(n)$ is assumed in shared normal form, we have proved that it is the correct low-child of n'; by similar reasoning we can show that $\mathsf{hi}(n)$ is the correct high-child of n'.

Finally, we show that p_i is the lowest-index variable in $\mathsf{Dep}(d \to [\![n]\!])$. We know by hypothesis that p_i is the lowest-index variable in $\mathsf{Dep}([\![n]\!])$, because n is in shared normal form. Hence, it is sufficient to show that $p_i \in \mathsf{Dep}(d \to [\![n]\!])$. We have shown above that $[\![\mathsf{lo}(n)]\!] = d' \to ([\![n']\!]|_{p_i=\mathsf{false}})$ and that $[\![\mathsf{hi}(n)]\!] = d' \to ([\![n']\!]|_{p_i=\mathsf{true}})$, with $\mathsf{lo}(n) \neq \mathsf{hi}(n)$, by hypothesis

Algorithm 6.2: Meet of a LVBDD in SNF with a lattice value

```
1  begin ConstMeetSNF(n, d)
2      n' ← memo?(⟨n, d⟩) ;
3      if n' ≠ nil then
4          return n' ;
5      else if val(n) ⪯ d then
6          n' ← n ;
7      else if val(n) ⊓ d = ⊥ then
8          n' ← MK(k + 1, ⊥, nil, nil) ;
9      else if index(n) = k + 1 then
10         n' ← MK(k + 1, val(n) ⊓ d, nil, nil) ;
11     else
12         ℓ ← ConstMeetSNF(lo(n), d) ;
13         h ← ConstMeetSNF(hi(n), d) ;
14         if ℓ = h then
15             n' ← MK(index(ℓ), val(ℓ) ⊓ val(n), lo(ℓ), hi(ℓ)) ;
16         else
17             ℓ' ← PseudoCompSNF(ℓ, d) ;
18             h' ← PseudoCompSNF(h, d) ;
19             n' ← MK(index(n), val(n) ⊓ d, ℓ', h') ;
20     memo!(⟨n, d⟩, n') ;
21     return n' ;
22 end
```

that n is in $\mathsf{SNF}(\mathbb{P}, \mathcal{L})$. Since $\mathsf{lo}(n)$ and $\mathsf{hi}(n)$ are in $\mathsf{SNF}(\mathbb{P}, \mathcal{L})$, we have: $d' \to (\llbracket n' \rrbracket|_{p_i=\mathsf{false}}) \neq d' \to (\llbracket n' \rrbracket|_{p_i=\mathsf{true}})$ Hence, we deduce that $\llbracket n' \rrbracket|_{p_i=\mathsf{false}} \neq \llbracket n' \rrbracket|_{p_i=\mathsf{true}}$, and thus $p_i \in \mathsf{Dep}(d \to \llbracket n \rrbracket)$. □

6.7.3 Meet of an LVBDD in SNF With a Constant

The procedure $\mathsf{ConstMeetSNF}(n, d)$ (Algorithm 6.2), computes the LVBDD in shared normal form n' such that $n' = \llbracket n \rrbracket \sqcap d$, for any lattice value $d \in \mathcal{L}$. ConstMeetSNF recursively traverses the graph (the two recursive calls at

lines 12 and 13 return the new subgraphs ℓ and h), and uses PseudoCompSNF on ℓ and h to obtain the correct low- and high-child of n'. Thanks to the distributivity of the lattice, computing $\mathsf{val}(n')$ is easy since $\mathsf{val}(n') = \exists\, \mathbb{P} : (\llbracket n \rrbracket \sqcap d) = (\exists\, \mathbb{P} : \llbracket n \rrbracket) \sqcap d = \mathsf{val}(n) \sqcap d$.

Lemma 6.7.2 (Soundness of ConstMeetSNF)**.** *For every* $n \in \mathsf{SNF}(\mathbb{P}, \mathcal{L})$*, for every* $d \in \mathcal{L}$ *we have that* $\mathsf{ConstMeetSNF}(n, d) = \mathsf{SNF}(\llbracket n \rrbracket \sqcap d)$*.*

Proof. Since Algorithm 6.2 is recursive, the proof is by induction on the structure of the LVBDD n. We thus show that the LVBDD returned by Algorithm 6.2 is $\mathsf{SNF}(\llbracket n \rrbracket \sqcap d)$.

Base cases The first base case at line 5 occurs when $\mathsf{val}(n) \preceq d$. From Definition 6.5.7 we have that $\mathsf{val}(n) = \exists \mathbb{P} : \llbracket \mathsf{val}(n) \rrbracket$, thus $\llbracket \mathsf{val}(n) \rrbracket \sqcap d = \llbracket \mathsf{val}(n) \rrbracket$, thus $\mathsf{SNF}(\llbracket n \rrbracket \sqcap d) = n$. The second base case at line 7 occurs when $\mathsf{val}(n) \sqcap d = \bot$, which implies that $\llbracket n \rrbracket \sqcap d = \bot$ and $\mathsf{SNF}(\llbracket n \rrbracket \sqcap d) = \langle k+1, \bot, \mathsf{nil}, \mathsf{nil} \rangle$. The final base case of line 9 occurs when n is a terminal node, thus clearly $\mathsf{SNF}(\llbracket n \rrbracket \sqcap d) = \langle k+1, \mathsf{val}(n) \sqcap d, \mathsf{nil}, \mathsf{nil} \rangle$.

Inductive cases Let $i = \mathsf{index}(n)$; since we know that n is a non-terminal node we have that $\llbracket n \rrbracket = \mathsf{val}(n) \sqcap \big((\neg p_i \sqcap \llbracket \mathsf{lo}(n) \rrbracket) \sqcup (p_i \sqcap \llbracket \mathsf{hi}(n) \rrbracket)\big)$, thus $\llbracket n \rrbracket \sqcap d = \mathsf{val}(n) \sqcap \big((\neg p_i \sqcap \llbracket \mathsf{lo}(n) \rrbracket \sqcap d) \sqcup (p_i \sqcap \llbracket \mathsf{hi}(n) \rrbracket \sqcap d)\big)$. By the induction hypothesis, we know that that $\ell = \mathsf{SNF}(\llbracket \mathsf{lo}(n) \rrbracket \sqcap d)$ and $h = \mathsf{SNF}(\llbracket \mathsf{hi}(n)) \rrbracket \sqcap d)$, which are computed at lines 12 and 13, respectively.

Subcase 1 : $\ell = h$

In the case where $\ell = h$, the algorithm computes, at line 15, the LVBDD $n' = \langle \mathsf{index}(\ell), \mathsf{val}(\ell) \sqcap \mathsf{val}(n), \mathsf{lo}(\ell), \mathsf{hi}(\ell) \rangle$. Let us show that n' has the correct semantics i.e., $\llbracket n' \rrbracket = \llbracket n \rrbracket \sqcap d$. By Definition 6.4.2, and because $\llbracket \mathsf{lo}(n) \rrbracket \sqcap d = \llbracket \mathsf{hi}(n) \rrbracket \sqcap d$ and $\llbracket \mathsf{lo}(n) \rrbracket \sqcap d = \llbracket \ell \rrbracket$ we have that:

$$
\begin{aligned}
& \llbracket n \rrbracket \sqcap d \\
= \;& \mathsf{val}(n) \sqcap (\neg p_i \sqcap \llbracket \mathsf{lo}(n) \rrbracket \sqcap d) \sqcup (p_i \sqcap \llbracket \mathsf{hi}(n) \rrbracket \sqcap d)) \\
= \;& \mathsf{val}(n) \sqcap \llbracket \mathsf{lo}(n) \rrbracket \sqcap d \\
= \;& \mathsf{val}(n) \sqcap \llbracket \ell \rrbracket
\end{aligned}
$$

Clearly we also have that $[\![n']\!] = \mathsf{val}(n) \sqcap [\![\ell]\!]$ hence $[\![n']\!] = [\![n]\!] \sqcap d$.

Next, we need to ensure that n' has the appropriate label i.e., $\mathsf{val}(n') = \exists\mathbb{P} : [\![n']\!]$. This is easy because $\exists\mathbb{P} : [\![n']\!] = \exists\mathbb{P} : ([\![\ell]\!] \sqcap \mathsf{val}(n)) = (\exists\mathbb{P} : [\![\ell]\!]) \sqcap \mathsf{val}(n) = \mathsf{val}(\ell) \sqcap \mathsf{val}(n) = \mathsf{val}(n')$.

Let $j = \mathsf{index}(n') = \mathsf{index}(\ell)$; we now need to prove that n' has the appropriate low- and high-child. If ℓ is a terminal node this easy since n' must then be terminal also, which means that both $\mathsf{lo}(n')$ and $\mathsf{hi}(n')$ must be nil and $\mathsf{lo}(\ell), \mathsf{hi}(\ell)$ are nil if ℓ is terminal. We thus assume that ℓ is nonterminal and show that $[\![\mathsf{lo}(n')]\!] = \mathsf{val}(n') \to ([\![n']\!]|_{p_j=\mathsf{false}})$ and $[\![\mathsf{hi}(n')]\!] = \mathsf{val}(n') \to ([\![n']\!]|_{p_j=\mathsf{true}})$. We show this for $\mathsf{lo}(n')$. By substituting $\mathsf{lo}(n')$ and $\mathsf{val}(n')$ by their assigned values of line 15, we see that we must show the following equivalent expression:

$$[\![\mathsf{lo}(\ell)]\!] = (\mathsf{val}(\ell) \sqcap \mathsf{val}(n)) \to (\mathsf{val}(\ell) \sqcap \mathsf{val}(n) \sqcap [\![\mathsf{lo}(\ell)]\!])$$

Moreover, we can easily see that $\mathsf{val}(\ell) = \mathsf{val}(\mathsf{lo}(n)) \sqcap d$ and thus obtain:

$$[\![\mathsf{lo}(\ell)]\!] = \big(\mathsf{val}(\mathsf{lo}(n)) \sqcap d \sqcap \mathsf{val}(n)\big) \to \big(\mathsf{val}(\mathsf{lo}(n)) \sqcap d \sqcap \mathsf{val}(n) \sqcap [\![\mathsf{lo}(\ell)]\!]\big) \quad (*)$$

By unfolding Definition 6.5.7 twice we can see that:

$$[\![\mathsf{lo}(\mathsf{lo}(n))]\!] = \mathsf{val}(\mathsf{lo}(n)) \to \big(\mathsf{val}(n) \to (\mathsf{val}(\mathsf{lo}(n)) \sqcap \mathsf{val}(n) \sqcap [\![\mathsf{lo}(\mathsf{lo}(n))]\!])\big)$$

which by application of Proposition 6.6.1, point (6.4), can be rewritten as:

$$[\![\mathsf{lo}(\mathsf{lo}(n))]\!] = \big(\mathsf{val}(\mathsf{lo}(n)) \sqcap \mathsf{val}(n)\big) \to \big(\mathsf{val}(\mathsf{lo}(n)) \sqcap \mathsf{val}(n) \sqcap [\![\mathsf{lo}(\mathsf{lo}(n))]\!]\big) \quad (**)$$

We now make the following assumption (justified later for the sake of clarity):

$$d \to (d \sqcap [\![\mathsf{lo}(\mathsf{lo}(n))]\!]) = [\![\mathsf{lo}(\ell)]\!] \qquad (***)$$

By Proposition 6.6.1, point (6.1), using $(**)$ and $(***)$, we obtain:

$$[\![\mathsf{lo}(\ell)]\!] = (\mathsf{val}(\mathsf{lo}(n)) \sqcap d \sqcap \mathsf{val}(n)) \to (\mathsf{val}(\mathsf{lo}(n)) \sqcap d \sqcap \mathsf{val}(n) \sqcap [\![\mathsf{lo}(\mathsf{lo}(n))]\!])$$

Finally, by using $(***)$ again, we can substitute $d \sqcap [\![\mathsf{lo}(\mathsf{lo}(n))]\!]$ by $d \sqcap [\![\mathsf{lo}(\ell)]\!]$ in the previous expression and thus show that $(*)$ is true. To complete

our argument, we now prove that the assumption $(***)$ is correct. By Definition 6.5.7, and the fact that $\mathsf{val}(\ell) = \mathsf{val}(\mathsf{lo}(n)) \sqcap d$, we can see that:

$$[\![\mathsf{lo}(\ell)]\!] = (\mathsf{val}(\mathsf{lo}(n)) \sqcap d) \rightarrow \big(\mathsf{val}(\mathsf{lo}(n)) \sqcap d \sqcap [\![\ell]\!]|_{p_j=\mathsf{false}}\big)$$

Using the induction hypothesis, we know that $[\![\ell]\!] = [\![\mathsf{lo}(n)]\!] \sqcap d$, thus:

$$\begin{aligned}[\![\mathsf{lo}(\ell)]\!] &= (\mathsf{val}(\mathsf{lo}(n)) \sqcap d) \rightarrow \big(\mathsf{val}(\mathsf{lo}(n)) \sqcap d \sqcap ([\![\mathsf{lo}(n)]\!] \sqcap d)|_{p_j=\mathsf{false}}\big) \\ &= (\mathsf{val}(\mathsf{lo}(n)) \sqcap d) \rightarrow \big(\mathsf{val}(\mathsf{lo}(n)) \sqcap d \sqcap [\![\mathsf{lo}(\mathsf{lo}(n))]\!]\big)\end{aligned}$$

Moreover, by Definition 6.5.7 (this time applied to $\mathsf{lo}(\mathsf{lo}(n))$):

$$[\![\mathsf{lo}(\mathsf{lo}(n))]\!] = \mathsf{val}(\mathsf{lo}(n)) \rightarrow \big(\mathsf{val}(\mathsf{lo}(n)) \sqcap [\![\mathsf{lo}(\mathsf{lo}(n))]\!]\big)$$

Finally, $(***)$ is obtained by the two previous expressions and Proposition 6.6.1, point (6.1).

Subcase 2 : $\ell \neq h$

If $\ell \neq h$ the algorithm computes ℓ' and h' at lines 17-18, which by soundness of Algorithm 6.1 are such that $\ell' = \mathsf{SNF}(d \rightarrow [\![\ell]\!])$ and $h' = \mathsf{SNF}(d \rightarrow [\![h]\!])$. We begin by showing that n' has the correct semantics, i.e. $[\![n']\!] = [\![n]\!] \sqcap d$. Indeed, $[\![n']\!] = d \sqcap \mathsf{val}(n) \sqcap \big((\neg p_i \sqcap [\![\ell']\!]) \sqcup (p_i \sqcap [\![h']\!])\big) = d \sqcap \mathsf{val}(n) \sqcap \big((\neg p_i \sqcap [\![\mathsf{lo}(n)]\!]) \sqcup (p_i \sqcap [\![\mathsf{hi}(n)]\!])\big) = d \sqcap [\![n]\!]$, since by definition of $\rightarrow$ we know that $[\![\ell']\!] \sqcap d = [\![\mathsf{lo}(n)]\!] \sqcap d$ and $[\![h']\!] \sqcap d = [\![\mathsf{hi}(n)]\!] \sqcap d$.

We now show that $n' = \mathsf{SNF}([\![n]\!] \sqcap d)$, which we know is equivalent to $n' = \mathsf{SNF}([\![n']\!])$. We begin by showing that $\mathsf{val}(n') = \exists \mathbb{P} : [\![n']\!]$; this is easy because $\exists \mathbb{P} : [\![n']\!] = \exists \mathbb{P} : ([\![n]\!] \sqcap d) = (\exists \mathbb{P} : [\![n]\!]) \sqcap d = \mathsf{val}(n) \sqcap d$, by distributivity.

Next, we must show that n' has the correct low- and high-child, i.e. $\mathsf{lo}(n') = \mathsf{SNF}(\mathsf{val}(n') \rightarrow [\![n']\!]|_{p_i=\mathsf{false}})$ and $\mathsf{hi}(n') = \mathsf{SNF}(\mathsf{val}(n') \rightarrow [\![n']\!]|_{p_i=\mathsf{true}})$. We show this for $\mathsf{lo}(n')$; from Definition 6.4.2 we know that $[\![n']\!]|_{p_i=\mathsf{false}} = \mathsf{val}(n') \sqcap [\![\ell']\!]$. Since $\mathsf{val}(n') = \mathsf{val}(n) \sqcap d$ we have that:

$$\mathsf{val}(n') \rightarrow [\![n']\!]|_{p_i=\mathsf{false}} = (\mathsf{val}(n) \sqcap d) \rightarrow (\mathsf{val}(n) \sqcap d \sqcap [\![\ell']\!])$$

Since $[\![\ell']\!] \sqcap d = [\![\mathsf{lo}(n)]\!] \sqcap d$ we can see that:

$$\mathsf{val}(n') \rightarrow [\![n']\!]|_{p_i=\mathsf{false}} = (\mathsf{val}(n) \sqcap d) \rightarrow (\mathsf{val}(n) \sqcap d \sqcap [\![\mathsf{lo}(n)]\!])$$

We also know from Definition 6.5.7 that $\mathsf{lo}(n) = \mathsf{SNF}(\mathsf{val}(n) \to [\![n]\!]|_{p_i=\mathsf{false}})$, thus $[\![\mathsf{lo}(n)]\!] = \mathsf{val}(n) \to (\mathsf{val}(n) \sqcap [\![\mathsf{lo}(n)]\!])$. By Proposition 6.6.1, point (6.1), we obtain that:

$$\mathsf{val}(n') \to [\![n']\!]|_{p_i=\mathsf{false}} = d \to (d \sqcap [\![\mathsf{lo}(n)]\!]) = d \to [\![\ell]\!] = [\![\ell']\!]$$

By soundness of Algorithm 6.1, $\ell' = \mathsf{SNF}(d \to [\![\ell]\!])$, thus $\ell' = \mathsf{SNF}(\mathsf{val}(n') \to [\![n']\!]|_{p_i=\mathsf{false}})$. Similar reasoning shows that $h' = \mathsf{SNF}(\mathsf{val}(n') \to [\![n']\!]|_{p_i=\mathsf{true}})$.

Finally, we must show that n' has the correct index i.e., that p_i is the proposition of lowest index in $\mathsf{Dep}([\![n']\!])$. Clearly $\mathsf{Dep}([\![n']\!]) \subseteq \mathsf{Dep}([\![n]\!])$ so either $p_i \notin \mathsf{Dep}([\![n']\!])$ or p_i is the proposition of lowest index in $\mathsf{Dep}([\![n']\!])$, since we know that p_i is the proposition of lowest index in $\mathsf{Dep}([\![n]\!])$. Let us show that $p_i \in \mathsf{Dep}([\![n']\!])$ or equivalently that $[\![n']\!]|_{p_i=\mathsf{false}} \neq [\![n']\!]|_{p_i=\mathsf{true}}$. By Definition 6.4.2 we know that $[\![n']\!]|_{p_i=\mathsf{false}} = \mathsf{val}(n) \sqcap d \sqcap [\![\ell']\!]$ and $[\![n']\!]|_{p_i=\mathsf{true}} = \mathsf{val}(n) \sqcap d \sqcap [\![h']\!]$. We also know that $d \sqcap [\![\ell']\!] = [\![\ell]\!]$ and that $d \sqcap [\![h']\!] = [\![h]\!]$, by definition of $\to$ and soundness of Algorithm 6.1. Furthermore, we know that $[\![\ell]\!] = [\![\mathsf{lo}(n)]\!] \sqcap d$ and $[\![h]\!] = [\![\mathsf{hi}(n)]\!] \sqcap d$, by soundness of Algorithm 6.2. We must therefore show that $d \sqcap \mathsf{val}(n) \sqcap [\![\mathsf{lo}(n)]\!] \neq d \sqcap \mathsf{val}(n) \sqcap [\![\mathsf{hi}(n)]\!]$. We already know that $d \sqcap [\![\mathsf{lo}(n)]\!] \neq d \sqcap [\![\mathsf{hi}(n)]\!]$ since $\ell' \neq h'$, which implies that $d \to (d \sqcap [\![\mathsf{lo}(n)]\!]) \neq d \to (d \sqcap [\![\mathsf{hi}(n)]\!])$. From Definition 6.5.7, we know that $[\![\mathsf{lo}(n)]\!] = \mathsf{val}(n) \to (\mathsf{val}(n) \sqcap [\![\mathsf{lo}(n)]\!])$ and $[\![\mathsf{hi}(n)]\!] = \mathsf{val}(n) \to (\mathsf{val}(n) \sqcap [\![\mathsf{hi}(n)]\!])$; by Proposition 6.6.1, point (6.1) we thus obtain:

$$d \to (d \sqcap [\![\mathsf{lo}(n)]\!]) = (\mathsf{val}(n) \sqcap d) \to (\mathsf{val}(n) \sqcap d \sqcap [\![\mathsf{lo}(n)]\!])$$
$$d \to (d \sqcap [\![\mathsf{hi}(n)]\!]) = (\mathsf{val}(n) \sqcap d) \to (\mathsf{val}(n) \sqcap d \sqcap [\![\mathsf{hi}(n)]\!])$$

We can now see that $\mathsf{val}(n) \sqcap d \sqcap [\![\mathsf{lo}(n)]\!] \neq \mathsf{val}(n) \sqcap d \sqcap [\![\mathsf{hi}(n)]\!]$. □

6.7.4 Meet of two LVBDD in SNF

The procedure $\mathsf{MeetSNF}(n_1, n_2)$ (Algorithm 6.3) takes two LVBDD $n_1, n_2 \in \mathsf{SNF}(\mathbb{P}, \mathcal{L})$ and computes the LVBDD in shared normal form n' such that $[\![n']\!] = [\![n_1]\!] \sqcap [\![n_2]\!]$. In the case where either n_1 or n_2 are terminal nodes, the algorithm simply calls ConstMeetSNF. Otherwise, subsequent recursive calls distinguish between two cases:

Algorithm 6.3: Meet of two LVBDD in SNF

```
1  begin MeetSNF(n_1, n_2)
2      n' ← memo?(⟨n_1, n_2⟩) ;
3      if n' ≠ nil then
4          return n' ;
5      else if index(n_1) = k + 1 then
6          n' ← ConstMeetSNF(n_2, val(n_1));
7      else if index(n_2) = k + 1 then
8          n' ← ConstMeetSNF(n_1, val(n_2));
9      else
10         if index(n_1) = index(n_2) then
11             ℓ ← MeetSNF(lo(n_1), lo(n_2)) ;
12             h ← MeetSNF(hi(n_1), hi(n_2)) ;
13             d ← val(n_1) ⊓ val(n_2) ;
14         else
15             if index(n_1) > index(n_2) then
16                 swap(n_1, n_2) ;
17             ℓ ← MeetSNF(lo(n_1), n_2) ;
18             h ← MeetSNF(hi(n_1), n_2) ;
19             d ← val(n_1) ;
20         ℓ' ← ConstMeetSNF(ℓ, d) ;
21         h' ← ConstMeetSNF(h, d) ;
22         if ℓ' = h' then
23             n' ← ℓ' ;
24         else
25             d' ← val(ℓ') ⊔ val(h') ;
26             ℓ'' ← PseudoCompSNF(ℓ', d') ;
27             h'' ← PseudoCompSNF(h', d') ;
28             n' ← MK(index(n_1), d', ℓ'', h'') ;
29     memo!(⟨n_1, n_2⟩, n') ;
30     memo!(⟨n_2, n_1⟩, n') ;
31     return n' ;
32 end
```

- If $\mathsf{index}(n_1) = \mathsf{index}(n_2) = i$, the algorithm recursively computes $\ell = [\![\mathsf{lo}(n_1)]\!] \sqcap [\![\mathsf{lo}(n_2)]\!]$ and $h = [\![\mathsf{hi}(n_1)]\!] \sqcap [\![\mathsf{hi}(n_2)]\!]$, along with the value $d = \mathsf{val}(n_1) \sqcap \mathsf{val}(n_2)$. Then, it computes $\ell' = d \sqcap [\![\ell]\!]$ and $h' = d \sqcap [\![h]\!]$. If $\ell' = h'$ then $[\![\ell']\!] = [\![h']\!] = [\![n_1]\!] \sqcap [\![n_2]\!]$ so the algorithm returns ℓ'. Otherwise, the algorithm computes the value $d' = \mathsf{val}(\ell') \sqcup \mathsf{val}(h')$ and returns the node $n' = \langle i, d', d' \to [\![\ell']\!], d' \to [\![h']\!] \rangle$.
- Otherwise, the algorithm assumes that $\mathsf{index}(n_1) < \mathsf{index}(n_2)$ (the reverse case is handled by swapping n_1 and n_2) and recursively computes $\ell = [\![\mathsf{lo}(n_1)]\!] \sqcap [\![n_2]\!]$ and $h = [\![\mathsf{hi}(n_1)]\!] \sqcap [\![n_2]\!]$ and sets the value $d = \mathsf{val}(n_1)$. From there on, it proceeds like in the previous case.

Example 6.7.3 (Computation of MeetSNF)**.** *Figure 6.7 illustrates the computation of the* MeetSNF *algorithm. In the figure, the lattice values* $\mathsf{A}, \mathsf{B}, \mathsf{X}, \mathsf{Y}$ *are all pair-wise incomparable. To clarify the figure, the node* $\langle k{+}1, \top, \mathsf{nil}, \mathsf{nil} \rangle$ *is duplicated when appropriate.*

(a) *The input* LVBDD n_1, n_2 *are such that* $[\![n_1]\!] = A \sqcap (p_1 \sqcup B)$ *and* $[\![n_2]\!] = X \sqcap (p_2 \sqcup Y)$*. Therefore* $[\![n']\!] = [\![n_1]\!] \sqcap [\![n_2]\!] = A \sqcap X \sqcap (p_1 \sqcup B) \sqcap (p_2 \sqcup Y)$.

(b) *Since* $\mathsf{index}(n_1) < \mathsf{index}(n_2)$*, the algorithm recursively computes* $\ell = \mathsf{SNF}([\![\mathsf{lo}(n_1)]\!] \sqcap [\![n_2]\!]) = \mathsf{ConstMeetSNF}(n_2, \mathsf{B})$ *and* $h = \mathsf{SNF}([\![\mathsf{hi}(n_1)]\!] \sqcap [\![n_2]\!]) = \mathsf{ConstMeetSNF}(n_2, \top) = n_2$.

(c) *Then, the algorithm computes* $\ell' = \mathsf{val}(n_1) \sqcap [\![\ell]\!] = \mathsf{ConstMeetSNF}(\ell, \mathsf{A})$ *and* $h' = \mathsf{val}(n_1) \sqcap [\![h]\!] = \mathsf{ConstMeetSNF}(h, \mathsf{A})$.

(d) *The value* d' *is set to* $\mathsf{val}(\ell') \sqcup \mathsf{val}(h') = (\mathsf{A} \sqcap \mathsf{B} \sqcap \mathsf{X}) \sqcup (\mathsf{A} \sqcap \mathsf{X}) = \mathsf{A} \sqcap \mathsf{X}$. *Finally, the algorithm computes the* LVBDD $\ell'' = \mathsf{SNF}(d' \to [\![\ell']\!]) = \mathsf{PseudoCompSNF}(\ell', d')$ *and the* LVBDD $h'' = \mathsf{SNF}(d' \to [\![h']\!]) = \mathsf{PseudoCompSNF}(h', d')$ *and returns the result* LVBDD $n' = \langle 1, d', \ell'', h'' \rangle$.

Lemma 6.7.4 (Soundness of MeetSNF)**.** *For every* $n_1, n_2 \in \mathsf{SNF}(\mathbb{P}, \mathcal{L})$ *we have that* $\mathsf{MeetSNF}(n_1, n_2) = \mathsf{SNF}([\![n_1]\!] \sqcap [\![n_2]\!])$.

Proof. Since Algorithm 6.3 is recursive, the proof is by induction on the structure of the LVBDD n_1 and n_2. Throughout the proof we shall let $\theta = [\![n_1]\!] \sqcap [\![n_2]\!]$; we must now show that the LVBDD n' returned by Algorithm 6.3 is such that $n' = \mathsf{SNF}(\theta)$.

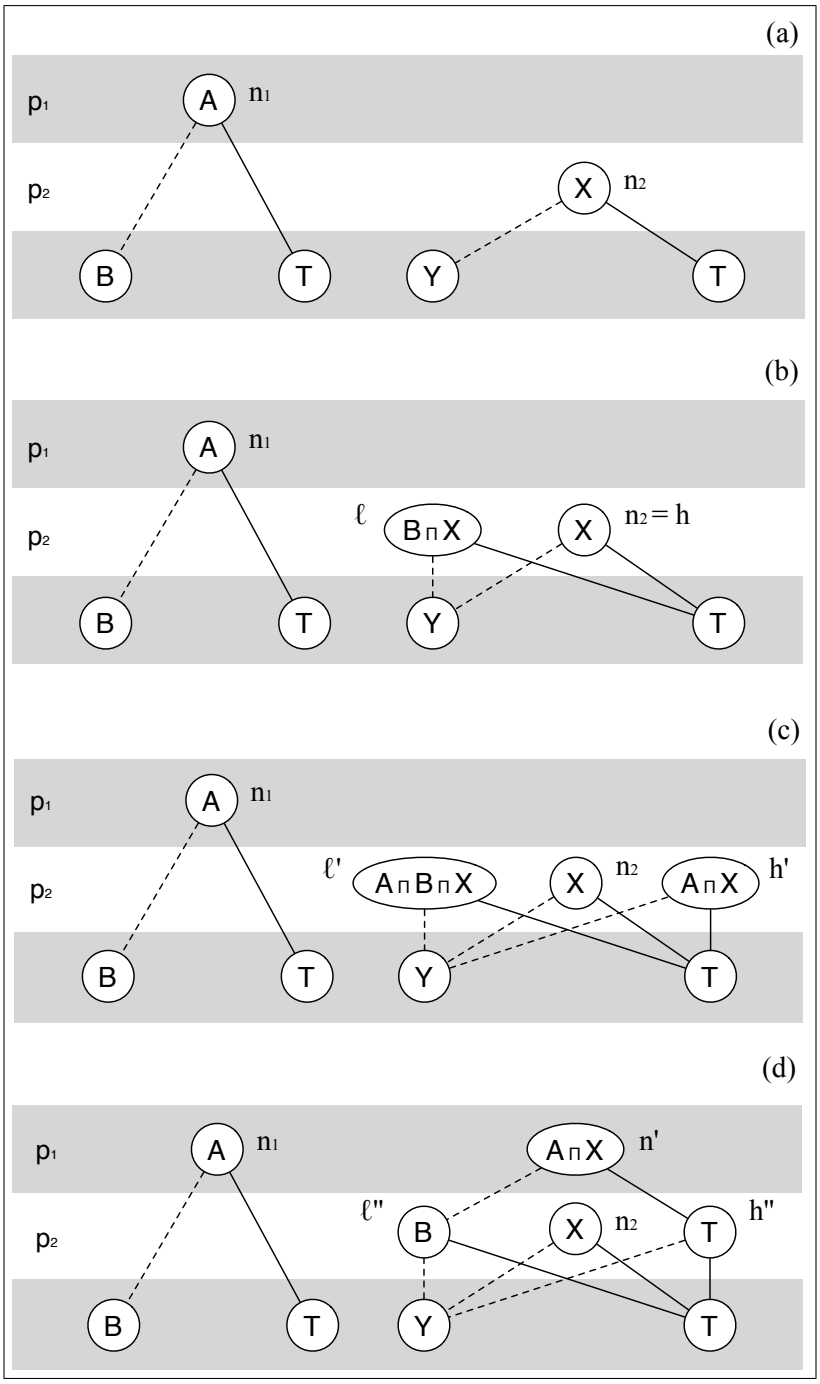


Figure 6.7: Example of the computation of $n' = \mathsf{MeetSNF}(n_1, n_2)$.

Base cases Lines 5 and 7 detect whether n_1 or n_2 are terminal nodes. In either case, we can use the procedure $\mathsf{ConstMeetSNF}$ of Algorithm 6.2 to compute n'.

Inductive case For the inductive cases, we know that neither n_1 nor n_2 are terminal nodes. We consider the following cases:

1. If $\mathsf{index}(n_1) = \mathsf{index}(n_2) = i$, we see from Definition 6.4.2 that:

   $$\theta = \mathsf{val}(n_1) \sqcap \mathsf{val}(n_2) \sqcap \big((\neg p_i \sqcap [\![\mathsf{lo}(n_1)]\!] \sqcap [\![\mathsf{lo}(n_2)]\!]) \sqcup (p_i \sqcap [\![\mathsf{hi}(n_1)]\!] \sqcap [\![\mathsf{hi}(n_2)]\!])\big)$$

   By the induction hypothesis we have that $\ell = \mathsf{SNF}([\![\mathsf{lo}(n_1)]\!] \sqcap [\![\mathsf{lo}(n_2)]\!])$ and $h = \mathsf{SNF}([\![\mathsf{hi}(n_1)]\!] \sqcap [\![\mathsf{hi}(n_2)]\!])$, which are computed lines 11-12. Furthermore, by soundness of Algorithm 6.2 we have that $\ell' = \mathsf{SNF}([\![\ell]\!] \sqcap \mathsf{val}(n_1) \sqcap \mathsf{val}(n_2))$ and $h' = \mathsf{SNF}([\![h]\!] \sqcap \mathsf{val}(n_1) \sqcap \mathsf{val}(n_2))$, which are computed at lines 20-21. From here on, we must again consider two cases:

   (a) If $\ell' = h'$ then we claim that $n' = \mathsf{SNF}(\theta)$, for n' computed at line 23. By soundness of Algorithm 6.2 we already know that $\ell' = \mathsf{SNF}([\![\ell']\!])$ so we just need to show that $[\![\ell']\!] = \theta$. Indeed, if $\ell' = h'$, then $\mathsf{val}(n_1) \sqcap \mathsf{val}(n_2) \sqcap [\![\mathsf{lo}(n_1)]\!] \sqcap [\![\mathsf{lo}(n_2)]\!] = \mathsf{val}(n_1) \sqcap \mathsf{val}(n_2) \sqcap [\![\mathsf{hi}(n_1)]\!] \sqcap [\![\mathsf{hi}(n_2)]\!]$, so we have:

   $$\theta = \mathsf{val}(n_1) \sqcap \mathsf{val}(n_2) \sqcap [\![\mathsf{lo}(n_1)]\!] \sqcap [\![\mathsf{lo}(n_2)]\!]$$

   which we know is equal to $[\![\ell']\!]$.

   (b) If $\ell' \neq h'$ then the algorithm proceeds to line 25 and computes ℓ'' and h'' which by soundness of Algorithm 6.1 are such that:

   $$\begin{aligned} \ell'' &= \mathsf{SNF}\big((\mathsf{val}(\ell') \sqcup \mathsf{val}(h')) \to [\![\ell']\!]\big) \\ h'' &= \mathsf{SNF}\big((\mathsf{val}(\ell') \sqcup \mathsf{val}(h')) \to [\![h']\!]\big) \end{aligned}$$

   Let us now show that $[\![n']\!] = \theta$, for n' computed at line 28.
   By Definition 6.4.2 we have:

   $$\begin{aligned} & [\![n']\!] \\ = \;& d' \sqcap \big((\neg p_i \sqcap [\![\ell'']\!]) \sqcup (p_i \sqcap [\![h'']\!])\big) \\ = \;& (\neg p_i \sqcap [\![\ell'']\!] \sqcap d') \sqcup (p_i \sqcap [\![h'']\!] \sqcap d') \end{aligned}$$

By definition of $\rightarrow$ we know that $[\![\ell'']\!] \sqcap d' = [\![\ell']\!]$ and $[\![h'']\!] \sqcap d' = [\![h']\!]$. Furthermore, we have seen earlier that $[\![\ell']\!] = \mathsf{val}(n_1) \sqcap \mathsf{val}(n_2) \sqcap [\![\ell]\!]$ and $[\![h']\!] = \mathsf{val}(n_1) \sqcap \mathsf{val}(n_2) \sqcap [\![h]\!]$. By substitution and distributivity we have:

$$[\![n']\!] = \mathsf{val}(n_1) \sqcap \mathsf{val}(n_2) \sqcap \big((\neg p_i \sqcap [\![\ell]\!]) \sqcup (p_i \sqcap [\![h]\!])\big) = \theta$$

Let us now show that n' is in shared normal form, i.e. $n' = \mathsf{SNF}([\![n']\!])$. We first show that $\mathsf{val}(n') = \exists \mathbb{P} : [\![n']\!]$. By Definition 6.4.2 we have:

$$\exists \mathbb{P} : [\![n']\!] = \exists \mathbb{P} : \theta = d \sqcap (\exists \mathbb{P} : [\![\ell]\!] \sqcup \exists \mathbb{P} : [\![h]\!])$$

Since we know that $[\![\ell]\!] \sqcap d = [\![\ell']\!]$ and $[\![h]\!] \sqcap d = [\![h']\!]$ we see that:

$$\exists \mathbb{P} : [\![n']\!] = \exists \mathbb{P} : [\![\ell']\!] \sqcup \exists \mathbb{P} : [\![h']\!] = d' = \mathsf{val}(n')$$

Next, we must show that ℓ'' and h'' are the correct low- and high-child of n', i.e. $\ell'' = \mathsf{SNF}(\mathsf{val}(n') \rightarrow [\![n']\!]|_{p_i=0})$ and $h'' = \mathsf{SNF}(\mathsf{val}(n') \rightarrow [\![n']\!]|_{p_i=1})$. We show this for ℓ''. From Definition 6.4.2 we have that $[\![n']\!]|_{p_i=0} = d' \sqcap [\![\ell'']\!]$, and since $\mathsf{val}(n') = d'$ we must show that $\ell'' = \mathsf{SNF}(d' \rightarrow d' \sqcap [\![\ell'']\!])$. We already know that $d' \sqcap [\![\ell'']\!] = [\![\ell']\!]$ and that $\ell' = \mathsf{SNF}(d' \rightarrow [\![\ell']\!])$ so we have shown that $\ell'' = \mathsf{SNF}(\mathsf{val}(n') \rightarrow [\![n']\!]|_{p_i=0})$. Similar reasoning shows that $h'' = \mathsf{SNF}(\mathsf{val}(n') \rightarrow [\![n']\!]|_{p_i=1})$.

Finally, we must show that n' has the correct index i.e., that p_i is the proposition of lowest index such that $p_i \in \mathsf{Dep}(\theta)$. Since $i = \mathsf{index}(n_1) = \mathsf{index}(n_2)$, p_i is the proposition of lowest index in both $\mathsf{Dep}([\![n_1]\!])$ and $\mathsf{Dep}([\![n_2]\!])$. Clearly we have that $\mathsf{Dep}(\theta) \subseteq \mathsf{Dep}([\![n_1]\!]) \cup \mathsf{Dep}([\![n_2]\!])$, so either $p_i \notin \mathsf{Dep}(\theta)$ or p_i is the proposition of lowest index in $\mathsf{Dep}(\theta)$. Let us show that $p_i \in \theta$, which is equivalent to $\theta_{p_i=0} \neq \theta_{p_i=1}$. We need to show that $d' \sqcap [\![\ell'']\!] \neq d' \sqcap [\![h'']\!]$, thus that $[\![\ell']\!] \neq [\![h']\!]$, which we know is true thanks to the **if** statement of line 22 and by canonicity of the shared normal form.

2. If $\mathsf{index}(n_1) \neq \mathsf{index}(n_2)$, we assume that $\mathsf{index}(n_1) < \mathsf{index}(n_2)$ which translates into the swap of line 16. By Definition 6.4.2 we have that:

$$\theta = \mathsf{val}(n_1) \sqcap \big((\neg p_i \sqcap [\![\mathsf{lo}(n_1)]\!] \sqcap [\![n_2]\!]) \sqcup (p_i \sqcap [\![\mathsf{hi}(n_1)]\!] \sqcap [\![n_2]\!])\big)$$

By the induction hypothesis we know that: $\ell = \mathsf{SNF}([\![\mathsf{lo}(n_1)]\!] \sqcap [\![n_2]\!])$ and that $h = \mathsf{SNF}([\![\mathsf{hi}(n_1)]\!] \sqcap [\![n_2]\!])$, which are computed lines 17-18. The remainder of the proof is similar to the case where $\mathsf{index}(n_1) = \mathsf{index}(n_2)$.

□

6.7.5 Join of two LVBDD in SNF

The procedure $\mathsf{ReJoinSNF}(n_1, n_2, d_1, d_2)$ (Algorithm 6.4) takes two LVBDD $n_1, n_2 \in \mathsf{UNF}(\mathbb{P}, \mathcal{L})$, along with two lattice values $d_1, d_2 \in \mathcal{L}$, and computes the LVBDD in shared normal form n' such that $[\![n']\!] = (d_1 \sqcap [\![n_1]\!]) \sqcup (d_2 \sqcap [\![n_2]\!])$. In the case where both n_1 and n_2 terminal nodes, the algorithm returns the terminal node $\langle k+1, (d_1 \sqcap \mathsf{val}(n_1)) \sqcup (d_2 \sqcap \mathsf{val}(n_2)), \mathsf{nil}, \mathsf{nil}\rangle$. Otherwise, the algorithm computes the values $d'_1 = d_1 \sqcap \mathsf{val}(n_1)$ and $d'_2 = d_2 \sqcap \mathsf{val}(n_2)$. Then, the algorithm distinguish between two cases:

- If $\mathsf{index}(n_1) = \mathsf{index}(n_2) = i$, the algorithm recursively computes $\ell = ([\![\mathsf{lo}(n_1)]\!] \sqcap d'_1) \sqcup ([\![\mathsf{lo}(n_2)]\!] \sqcap d'_2)$ and $h = ([\![\mathsf{hi}(n_1)]\!] \sqcap d'_1) \sqcup ([\![\mathsf{hi}(n_2)]\!] \sqcap d'_2)$. If $\ell = h$ then $[\![\ell]\!] = [\![h]\!] = ([\![n_1]\!] \sqcap d_1) \sqcup ([\![n_2]\!] \sqcap d_2)$ so the algorithm returns ℓ. Otherwise, the algorithm returns the node $n' = \langle i, d'_1 \sqcup d'_2, (d'_1 \sqcup d'_2) \to [\![\ell']\!], (d'_1 \sqcup d'_2) \to [\![h']\!]\rangle$.
- Otherwise, the algorithm assumes that $\mathsf{index}(n_1) < \mathsf{index}(n_2)$ (the reverse case is handled by swapping n_1 and n_2) and recursively computes $\ell = ([\![\mathsf{lo}(n_1)]\!] \sqcap d_1) \sqcup ([\![n_2]\!] \sqcap d_2)$ and $h = ([\![\mathsf{hi}(n_1)]\!] \sqcap d_1) \sqcup ([\![n_2]\!] \sqcap d_2)$. From there on, it proceeds like in the previous case.

Example 6.7.5 (Computation of JoinSNF)**.** *Figure 6.8 illustrates the computation of the* JoinSNF *algorithm. The input* LVBDD *are the same as those of the* MeetSNF *example of Figure 6.7, so the lattice values* $\mathsf{A}, \mathsf{B}, \mathsf{X}, \mathsf{Y}$ *are again all pair-wise incomparable. Again, the node* $\langle k+1, \top, \mathsf{nil}, \mathsf{nil}\rangle$ *is duplicated in the figure for the sake of clarity.*

Algorithm 6.4: Join of two LVBDD in SNF

```
1  begin ReJoinSNF(n1, n2, d1, d2)
2      n' ← memo?(⟨n1, n2, d1, d2⟩) ;
3      if n' ≠ nil then
4          return n';
5      d'1 ← d1 ⊓ val(n1) ;
6      d'2 ← d2 ⊓ val(n2) ;
7      if index(n1) = index(n2) = k + 1 then
8          n' ← MK(k + 1, d'1 ⊔ d'2, nil, nil) ;
9      else
10         if index(n1) = index(n2) then
11             ℓ ← ReJoinSNF(lo(n1), lo(n2), d'1, d'2) ;
12             h ← ReJoinSNF(hi(n1), hi(n2), d'1, d'2) ;
13         else
14             if index(n1) > index(n2) then
15                 swap(n1, n2) ;
16                 swap(d1, d2) ;
17                 swap(d'1, d'2) ;
18             ℓ ← ReJoinSNF(lo(n1), n2, d'1, d'2) ;
19             h ← ReJoinSNF(hi(n1), n2, d'1, d'2) ;
20         if ℓ = h then
21             n' ← ℓ ;
22         else
23             ℓ' ← PseudoCompSNF(ℓ, d'1 ⊔ d'2) ;
24             h' ← PseudoCompSNF(h, d'1 ⊔ d'2) ;
25             n' ← MK(index(n1), d'1 ⊔ d'2, ℓ', h') ;
26     memo!(⟨n1, n2, d1, d2⟩, n') ;
27     memo!(⟨n2, n1, d2, d1⟩, n') ;
28     return n' ;
29 end
30 begin JoinSNF(n1, n2)
31     return ReJoinSNF(n1, n2, ⊤, ⊤);
32 end
```

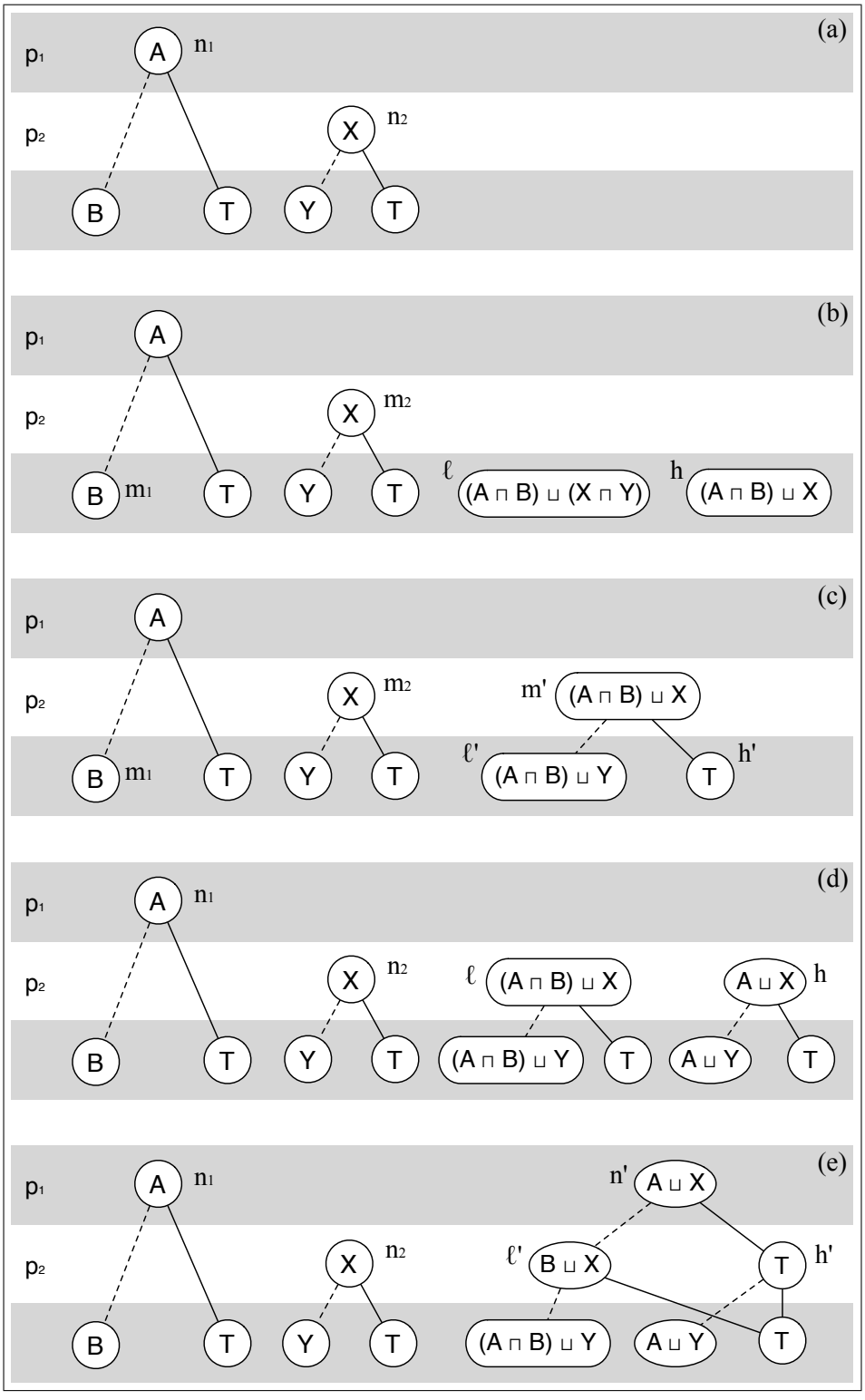


Figure 6.8: Example of the computation of $n' = \mathsf{JoinSNF}(n_1, n_2)$.

(a) *The input* LVBDD n_1, n_2 *are such that* $[\![n_1]\!] = (A \sqcap p_1) \sqcup (A \sqcap B)$ *and* $[\![n_2]\!] = (X \sqcap p_2) \sqcup (X \sqcap Y)$. *Therefore* $[\![n']\!] = [\![n_1]\!] \sqcup [\![n_2]\!] = (A \sqcap p_1) \sqcup (A \sqcap B) \sqcup (X \sqcap p_2) \sqcup (X \sqcap Y)$.

(b) *Since* $\mathsf{index}(n_1) < \mathsf{index}(n_2)$, *the algorithm makes the recursive call* $\mathsf{ReJoinSNF}(\mathsf{lo}(n_1), n_2, \mathsf{A}, \mathsf{X})$. *To clarify, within this recursive call, we let* $m_1 = \mathsf{lo}(n_1)$ *and* $m_2 = n_2$, *and compute* $m' = ([\![m_1]\!] \sqcap \mathsf{A}) \sqcup ([\![m_2]\!] \sqcap X)$. *This time, since* $\mathsf{index}(m_1) > \mathsf{index}(m_2)$, *the algorithm computes* $\ell = ([\![\mathsf{lo}(m_2)]\!] \sqcap \mathsf{X}) \sqcup ([\![m_1]\!] \sqcap \mathsf{A}) = (\mathsf{Y} \sqcap \mathsf{X}) \sqcup (\mathsf{B} \sqcap \mathsf{A})$ *and* $h = ([\![\mathsf{hi}(m_2)]\!] \sqcap \mathsf{X}) \sqcup ([\![m_1]\!] \sqcap \mathsf{A}) = \mathsf{X} \sqcup (\mathsf{B} \sqcap \mathsf{A})$.

(c) *Afterward, the algorithm computes the value* $d'_1 \sqcup d'_2 = (\mathsf{A} \sqcap \mathsf{B}) \sqcup \mathsf{X}$ *and computes* $\ell' = \big((\mathsf{A} \sqcap \mathsf{B}) \sqcup \mathsf{X}\big) \to [\![\ell]\!] = (\mathsf{A} \sqcap \mathsf{B}) \sqcup \mathsf{Y}$ *and* $h' = \big((\mathsf{A} \sqcap \mathsf{B}) \sqcup \mathsf{X}\big) \to [\![h]\!] = \top$. *This enables the construction of the* LVBDD $m' = \langle 2, (\mathsf{A} \sqcap \mathsf{B}) \sqcup \mathsf{X}, \ell', h' \rangle$.

(d) *Returning the first call to* $\mathsf{ReJoinSNF}(\mathsf{n_1}, \mathsf{n_2}, \top, \top)$, *the algorithm just finished the computation of* ℓ *(called* m' *in the previous step) and proceeds to compute* $h = ([\![\mathsf{hi}(n_1)]\!] \sqcap \mathsf{A}) \sqcup ([\![n_2]\!] \sqcap \mathsf{X}) = \mathsf{A} \sqcup [\![n_2]\!]$ *(the computation of* h *is similar to that of* ℓ*).*

(e) *Finally, the algorithm computes the value* $d'_1 \sqcup d'_2 = \mathsf{A} \sqcup \mathsf{X}$ *and computes* $\ell' = (\mathsf{A} \sqcup \mathsf{X}) \to [\![\ell]\!]$ *and* $h' = (\mathsf{A} \sqcup \mathsf{X}) \to [\![h]\!]$. *The final result* LVBDD *is thus* $n' = \langle 1, \mathsf{A} \sqcup \mathsf{X}, \ell', h' \rangle$.

Lemma 6.7.6 (Soundness of ReJoinSNF)**.** *For every* $n_1, n_2 \in \mathsf{SNF}(\mathbb{P}, \mathcal{L})$, $d_1, d_2 \in \mathcal{L}$ *we have* $\mathsf{ReJoinSNF}(n_1, n_2, d_1, d_2) = \mathsf{SNF}\big(([\![n_1]\!] \sqcap d_1) \sqcup ([\![n_2]\!] \sqcap d_2)\big)$.

Proof. Since ReJoinSNF is recursive, the proof is by induction on the structure of the LVBDD n_1 and n_2. Throughout the proof we shall let $\theta = \big(([\![n_1]\!] \sqcap d_1) \sqcup ([\![n_2]\!] \sqcap d_2)\big)$; we must now show that the LVBDD n' returned by the procedure ReJoinSNF is such that $n' = \mathsf{SNF}(\theta)$. The algorithm begins by computing the values d'_1, d'_2 at lines 5-6, which are such that:

$$d'_1 = d_1 \sqcap \mathsf{val}(n_1) = d_1 \sqcap (\exists\, \mathbb{P} : [\![n_1]\!]) = \exists\, \mathbb{P} : (d_1 \sqcap [\![n_1]\!])$$
$$d'_2 = d_2 \sqcap \mathsf{val}(n_2) = d_2 \sqcap (\exists\, \mathbb{P} : [\![n_2]\!]) = \exists\, \mathbb{P} : (d_2 \sqcap [\![n_2]\!])$$

We can therefore already conclude that $\mathsf{val}(n') = \exists\,\mathbb{P} : \theta = d'_1 \sqcup d'_2$:

$$\begin{aligned} & \exists\,\mathbb{P} : \theta \\ = \;& \exists\,\mathbb{P}\big((\llbracket n_1 \rrbracket \sqcap d_1) \sqcup (\llbracket n_2 \rrbracket \sqcap d_2)\big) \\ = \;& \exists\,\mathbb{P}(\llbracket n_1 \rrbracket \sqcap d_1) \sqcup \exists\,\mathbb{P}(\llbracket n_2 \rrbracket \sqcap d_2) \\ = \;& d'_1 \sqcup d'_2 \end{aligned}$$

Base cases Line 7 detects the case when n_1 and n_2 are both terminal nodes. In that case, it is easy to see that n' should be the terminal node $\langle k+1, d'_1 \sqcup d'_2, \mathsf{nil}, \mathsf{nil}\rangle$, as done by the algorithm at line 8.

Induction hypothesis We assume that for every pair of lattice values $x_1, x_2 \in \mathcal{L}$, and for every $m_1, m_2 \in \mathsf{SNF}(\mathbb{P}, \mathcal{L})$ with $\mathsf{min}(\mathsf{index}(m_1), \mathsf{index}(m_2)) < \mathsf{min}(\mathsf{index}(n_1), \mathsf{index}(n_2))$, we have that:

$$\mathsf{ReJoinSNF}(m_1, m_2, x_1, x_2) = \mathsf{SNF}\big((\llbracket m_1 \rrbracket \sqcap x_1) \sqcup (\llbracket m_2 \rrbracket \sqcap x_2)\big)$$

Induction step For the inductive cases, we know that at least n_1 or n_2 is a non-terminal node. We consider the following cases:

1. If $\mathsf{index}(n_1) = \mathsf{index}(n_2) = i$ then the algorithm computes the LVBDD ℓ and h at lines 11-12. By the induction hypothesis, we know ℓ and h are such that:

$$\begin{aligned} \ell &= \mathsf{SNF}\big((\llbracket \mathsf{lo}(n_1) \rrbracket \sqcap d'_1) \sqcup (\llbracket \mathsf{lo}(n_2) \rrbracket \sqcap d'_2)\big) \\ h &= \mathsf{SNF}\big((\llbracket \mathsf{hi}(n_1) \rrbracket \sqcap d'_1) \sqcup (\llbracket \mathsf{hi}(n_2) \rrbracket \sqcap d'_2)\big) \end{aligned}$$

   Using Definition 6.4.2, we can see that:

$$\begin{aligned} & \theta|_{p_i=\mathsf{false}} \\ = \;& (d_1 \sqcap \llbracket n_1 \rrbracket|_{p_i=\mathsf{false}}) \sqcup (d_2 \sqcap \llbracket n_2 \rrbracket|_{p_i=\mathsf{false}}) \\ = \;& (d_1 \sqcap \mathsf{val}(n_1) \sqcap \llbracket \mathsf{lo}(n_1) \rrbracket) \sqcup (d_2 \sqcap \mathsf{val}(n_2) \sqcap \llbracket \mathsf{lo}(n_2) \rrbracket) \\ = \;& (d'_1 \sqcap \llbracket \mathsf{lo}(n_1) \rrbracket) \sqcup (d'_2 \sqcap \llbracket \mathsf{lo}(n_2) \rrbracket) \\ = \;& \ell \end{aligned}$$

   And likewise we have that $h = \theta|_{p_i=\mathsf{true}}$. We consider two further cases.

(a) If $\ell = h$ then $\theta|_{p_i=\mathsf{false}} = \theta|_{p_i=\mathsf{true}} = \theta$, so the algorithm safely assigns n' to ℓ.

(b) If $\ell \neq h$, then the algorithm computes ℓ' and h' at lines 23-24. By soundness of PseudoCompSNF we know that $[\![\ell']\!] = \mathsf{SNF}((d'_1 \sqcup d'_2) \to [\![\ell]\!])$ and $[\![h']\!] = \mathsf{SNF}((d'_1 \sqcup d'_2) \to [\![h]\!])$. We have already shown that $\mathsf{val}(n') = d'_1 \sqcup d'_2$, and we just showed that ℓ' and h' are the correct low- and high-child of n'. Finally, the fact that $p_i \in \mathsf{Dep}([\![n']\!])$ is a consequence of the fact that $\ell' \neq h'$, implied by $\ell \neq h$, which is tested at line 20.

2. If $i = \mathsf{index}(n_1) < \mathsf{index}(n_2)$ then the algorithm computes the LVBDD ℓ and h at lines 18-19. By the induction hypothesis, we know ℓ and h are such that:

$$\begin{aligned} \ell &= \mathsf{SNF}\big(([\![\mathsf{lo}(n_1)]\!] \sqcap d'_1) \sqcup ([\![n_2]\!] \sqcap d'_2)\big) \\ h &= \mathsf{SNF}\big(([\![\mathsf{hi}(n_1)]\!] \sqcap d'_1) \sqcup ([\![n_2]\!] \sqcap d'_2)\big) \end{aligned}$$

Like before, using Definition 6.4.2, we can see that:

$$\begin{aligned} & \theta|_{p_i=\mathsf{false}} \\ = \; & (d_1 \sqcap [\![n_1]\!]|_{p_i=\mathsf{false}}) \sqcup (d_2 \sqcap [\![n_2]\!]|_{p_i=\mathsf{false}}) \\ = \; & (d_1 \sqcap \mathsf{val}(n_1) \sqcap [\![\mathsf{lo}(n_1)]\!]) \sqcup (d_2 \sqcap \mathsf{val}(n_2) \sqcap [\![n_2]\!]) \\ = \; & (d'_1 \sqcap [\![\mathsf{lo}(n_1)]\!]) \sqcup (d'_2 \sqcap [\![n_2]\!]) \\ = \; & \ell \end{aligned}$$

And likewise we have that $h = \theta|_{p_i=\mathsf{true}}$. The remainder of this part of the proof is very similar to the case where $\mathsf{index}(n_1) = \mathsf{index}(n_2)$.

3. If $i = \mathsf{index}(n_1) > \mathsf{index}(n_2)$ then the algorithm treats it like the case where $\mathsf{index}(n_1) < \mathsf{index}(n_2)$ by a series of swaps at lines 15-17.

□

Corollary 6.7.7 (Soundness of JoinSNF)**.** *For every* $n_1, n_2 \in \mathsf{SNF}(\mathbb{P}, \mathcal{L})$ *we have that* $\mathsf{JoinSNF}(n_1, n_2) = \mathsf{SNF}([\![n_1]\!] \sqcup [\![n_2]\!])$.

Proof. The result is immediate from Lemma 6.7.6. □

Algorithm 6.5: Meet or Join of two LVBDD in UNF

```
1  begin ApplyUNF(n_1, n_2, op)
2      n' ← memo?(⟨n_1, n_2, op⟩) ;
3      if n' ≠ nil then
4          return n' ;
5      else if index(n_1) = index(n_2) = k + 1 then
6          n' ← MK(k + 1, val(n_1) op val(n_2), nil, nil) ;
7      else
8          if index(n_1) = index(n_2) then
9              ℓ ← ApplyUNF(lo(n_1), lo(n_2), op) ;
10             h ← ApplyUNF(hi(n_1), hi(n_2), op) ;
11         else
12             if index(n_1) > index(n_2) then
13                 swap(n_1, n_2) ;
14             ℓ ← ApplyUNF(lo(n_1), n_2) ;
15             h ← ApplyUNF(hi(n_1), n_2) ;
16         if ℓ = h then
17             n' ← ℓ ;
18         else
19             n' ← MK(index(n_1), ⊤, ℓ, h) ;
20     memo!(⟨n_1, n_2, op⟩, n') ; memo!(⟨n_2, n_1, op⟩, n') ;
21     return n' ;
22 end
```

6.7.6 Meet or Join of two LVBDD in UNF

The computation of meet and join of two LVBDD can be performed by an algorithm that is very similar to the computation of conjunction and disjunction of two ROBDD. It amounts essentially to the computation of a synchronized product of the LVBDD and to ensure canonicity.

The procedure ApplyUNF (Algorithm 6.5) takes two LVBDD in unshared normal form n_1, n_2 along with an operation $\mathsf{op} \in \{\sqcup, \sqcap\}$, and computes the LVBDD $n' = \mathsf{UNF}([\![n_1]\!]\, \mathsf{op}\, [\![n_2]\!])$.

Lemma 6.7.8 (Soundness of ApplyUNF)**.** *For every* $n_1, n_2 \in \mathsf{UNF}(\mathbb{P}, \mathcal{L})$*, and* $\mathsf{op} \in \{\sqcap, \sqcup\}$ *we have that* $\mathsf{ApplyUNF}(n_1, n_2, \mathsf{op}) = \mathsf{UNF}([\![n_1]\!]\, \mathsf{op}\, [\![n_2]\!])$*.*

Proof. Since Algorithm 6.5 is recursive, the proof is by induction on the structure of the LVBDD n_1 and n_2. Throughout the proof, we denote by θ the function $[\![n_1]\!]\, \mathsf{op}\, [\![n_2]\!]$. We thus establish that the LVBDD returned by Algorithm 6.5 is $\mathsf{UNF}(\theta)$.

Base Case The base case is when n_1 and n_2 are terminal, i.e., $\mathsf{index}(n_1) = \mathsf{index}(n_2) = k + 1$. In this case, $\theta = d \in \mathcal{L}$, and thus $\mathsf{Dep}(\theta) = \varnothing$. Thus, $\mathsf{UNF}(\theta)$ is the terminal LVBDD $\langle k+1, d\rangle$. It is easy to see that, in this case, Algorithm 6.5 computes and returns $\mathsf{UNF}(\theta)$, since it enters the enters the **if** at line 5.

Inductive Case The induction hypothesis is that every recursive call performed by Algorithm 6.5 respects the Lemma. Under this hypothesis, let us prove that the LVBDD returned by Algorithm 6.5 is $\mathsf{UNF}(\theta)$. Let $i = \mathsf{index}(n_1)$ and $j = \mathsf{index}(n_2)$. Since n_1 and n_2 are in unshared normal form, and by definition of $[\![n_1]\!]$ and $[\![n_2]\!]$, we know that the following equalities hold:

$$\begin{array}{rclcrcl} [\![n_1]\!]|_{p_i=\mathsf{false}} & = & [\![\mathsf{lo}(n_1)]\!] & \qquad & [\![n_2]\!]|_{p_j=\mathsf{false}} & = & [\![\mathsf{lo}(n_2)]\!] \\ [\![n_1]\!]|_{p_i=\mathsf{true}} & = & [\![\mathsf{hi}(n_1)]\!] & & [\![n_2]\!]|_{p_j=\mathsf{true}} & = & [\![\mathsf{hi}(n_2)]\!] \end{array}$$

Then, we consider several cases:

1. Either $\mathsf{index}(n_1) = \mathsf{index}(n_2)$. In that case we see that θ is equal to:

$$\lambda v \cdot \begin{cases} (\llbracket n_1 \rrbracket|_{p_i=0}) \,\mathsf{op}\, (\llbracket n_2 \rrbracket|_{p_i=0}) = \llbracket \mathsf{lo}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{lo}(n_2) \rrbracket & \text{if } \; p_i \notin v \\ (\llbracket n_1 \rrbracket|_{p_i=1}) \,\mathsf{op}\, (\llbracket n_2 \rrbracket|_{p_i=1}) = \llbracket \mathsf{hi}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{hi}(n_2) \rrbracket & \text{if } \; p_i \in v \end{cases}$$

by definition of the application of $\mathsf{op} \in \{\sqcap, \sqcup\}$ on two LVBF. Moreover, we observe that, since $\mathsf{index}(n_1) = \mathsf{index}(n_2) = i$, Algorithm 6.5 enters the **if** at line 8, and thus compute the LVBDD $\ell = \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{lo}(n_2) \rrbracket)$, and $h = \mathsf{UNF}(\llbracket \mathsf{hi}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{hi}(n_2) \rrbracket)$, by the induction hypothesis. Then, we consider two further cases:

   (a) Either $p_i \in \mathsf{Dep}(\theta)$. In this case, by Definition 6.5.2, $\mathsf{UNF}(\theta)$ is the non-terminal LVBDD $\langle i, \top, \mathsf{UNF}(\theta|_{p_i=0}), \mathsf{UNF}(\theta|_{p_i=1}) \rangle$. Therefore, $\mathsf{UNF}(\theta) = \langle i, \top, \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{lo}(n_2) \rrbracket), \mathsf{UNF}(\llbracket \mathsf{hi}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{hi}(n_2) \rrbracket) \rangle$. Moreover, $\ell \neq h$ because $p_i \in \mathsf{Dep}(\theta)$, since otherwise we would have $\theta|_{p_i=\mathsf{false}} = \theta|_{p_i=\mathsf{true}}$, which contradicts $p_i \in \mathsf{Dep}(\theta)$. Thus, Algorithm 6.5 enters the **else** at line 18 and returns the LVBDD

$$\begin{array}{rl} & \langle \mathsf{index}(n_1), \top, \ell, h \rangle \\ = & \langle i, \top, \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{lo}(n_2) \rrbracket), \mathsf{UNF}(\llbracket \mathsf{hi}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{hi}(n_2) \rrbracket) \rangle \end{array}$$

   which is the expected result.

   (b) Or $p_i \notin \mathsf{Dep}(\theta)$. Since $p_i \notin \mathsf{Dep}(\theta)$, $\theta = \theta|_{p_i=\mathsf{false}} = \theta|_{p_i=\mathsf{false}}$ by definition of $\mathsf{Dep}(\theta)$. Thus, $\mathsf{UNF}(\theta) = \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{lo}(n_2) \rrbracket)$. On the other hand, since $p_i \notin \mathsf{Dep}(\theta)$, we know that $\theta|_{p_i=\mathsf{false}} = \theta|_{p_i=\mathsf{true}}$, and therefore that:

$$\llbracket \mathsf{lo}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{lo}(n_2) \rrbracket = \llbracket \mathsf{hi}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{hi}(n_2) \rrbracket$$

   Hence,

$$\ell = \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{lo}(n_2) \rrbracket) = \mathsf{UNF}(\llbracket \mathsf{hi}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{hi}(n_2) \rrbracket) = h$$

   Since $\ell = h$, Algorithm (6.9) enters the **if** at line 16 and returns

$$\ell = \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket \,\mathsf{op}\, \llbracket \mathsf{lo}(n_2) \rrbracket)$$

   which is the expected result.

2. Or $i = \mathsf{index}(n_1) < \mathsf{index}(n_2)$. In this case, we have that θ is equal to:

$$\lambda v \cdot \begin{cases} (\llbracket n_1 \rrbracket|_{p_i=\mathsf{false}})\ \mathsf{op}\ \llbracket n_2 \rrbracket = \llbracket \mathsf{lo}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket & \text{If } p_i \notin v \\ (\llbracket n_1 \rrbracket|_{p_i=\mathsf{true}})\ \mathsf{op}\ \llbracket n_2 \rrbracket = \llbracket \mathsf{hi}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket & \text{If } p_i \in v \end{cases}$$

by definition of the application of $\mathsf{op} \in \{\sqcap, \sqcup\}$ on two LVBF. Moreover, we observe that, since $i = \mathsf{index}(n_1) < \mathsf{index}(n_2)$, Algorithm 6.5 enters the **else** at line 11, and computes $\ell = \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket)$, and $h = \mathsf{UNF}(\llbracket \mathsf{hi}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket)$, by the induction hypothesis. Then, we consider two further cases:

   (a) Either $p_i \in \mathsf{Dep}(\theta)$. By similar arguments as above, we conclude that $\mathsf{UNF}(\theta)$ is the non-terminal LVBDD:

$$\langle i, \top, \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket), \mathsf{UNF}(\llbracket \mathsf{hi}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket) \rangle$$

   On the other hand, Algorithm 6.5 enters the **else** at line 18 and returns the LVBDD:

$$\begin{array}{rl} & \langle \mathsf{index}(n_1), \top, \ell, h \rangle \\ = & \langle i, \top, \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket), \mathsf{UNF}(\llbracket \mathsf{hi}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket) \rangle \end{array}$$

   which is the expected result.

   (b) Or $p_i \notin \mathsf{Dep}(\theta)$. By similar arguments as above, we conclude that $\mathsf{UNF}(\theta) = \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket)$ and that $h = \ell$. On the other hand, Algorithm 6.5 enters the **if** at line 16 and returns $\ell = \mathsf{UNF}(\llbracket \mathsf{lo}(n_1) \rrbracket\ \mathsf{op}\ \llbracket n_2 \rrbracket)$, which is the expected result.

3. Or $\mathsf{index}(n_1) > \mathsf{index}(n_2)$. This case is symmetrical to the former and is treated by a swap at line 13, which reduces it to the former case.

□

6.7.7 Existential and Universal Quantification of LVBDD

We briefly discuss how to compute, for an LVBDD $n \in \mathsf{UNF}(\mathbb{P}, \mathcal{L}) \cup \mathsf{SNF}(\mathbb{P}, \mathcal{L})$, how to compute $\exists\ \mathbb{P} \cdot \llbracket n \rrbracket$ and $\forall\ \mathbb{P} \cdot \llbracket n \rrbracket$. Existentially quantifying LVBDD in shared normal form is trivial, as by definition it is the label of its root node.

Algorithm 6.6: Bottom-up propagation for LVBDD in UNF or SNF

```
1  begin Propagate(n, op)
2      d ← memo?(⟨n, op⟩) ;
3      if d ≠ nil then
4          return d ;
5      else if index(n) = k + 1 then
6          d ← val(n) ;
7      else
8          d_1 ← Propagate(lo(n), op) ;
9          d_2 ← Propagate(hi(n), op) ;
10         d ← val(n) ⊓ (d_1 op d_2) ;
11     memo!(⟨n, op⟩, d) ;
12     return d ;
13 end
```

For all the other cases, the solution can be computed with the procedure Propagate (Algorithm 6.6), which performs a simple bottom-up propagation of the node labels.

Lemma 6.7.9. *For every* LVBDD $n \in \mathsf{UNF}(\mathbb{P}, \mathcal{L}) \cup \mathsf{SNF}(\mathbb{P}, \mathcal{L})$ *we have that* $\exists\, \mathbb{P} : [\![n]\!] = \mathsf{Propagate}(n, \sqcup)$ *and* $\forall\, \mathbb{P} : [\![n]\!] = \mathsf{Propagate}(n, \sqcap)$.

6.8 Complexity of LVBDD Algorithms

In this section, we study the general worst-case complexity of computing the meet and join of LVBDD in SNF. In the case of the UNF, it is easy to see that, thanks to memoization, the ApplyUNF procedure of Algorithm 6.5 runs in $\mathcal{O}(n^2)$ in the worst case, so we do not discuss it further. For the SNF however, it turns out that the worst-case complexity of both the meet and join operations on LVBDD in SNF is *worst-case exponential*. Indeed, we show in this section that there exists a family of LVBDD that grows linearly, and which blows up exponentially after a single meet or join operation.

Let $\mathbb{P} = \{p_1, \ldots, p_k\}$ be a finite set of Boolean propositions, let $S =$

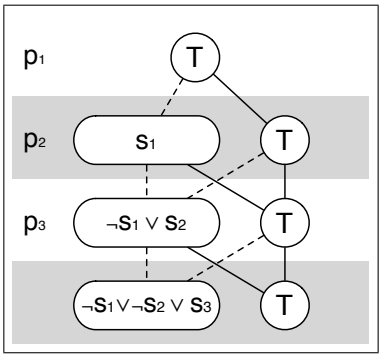


Figure 6.9: LVBDD encoding in SNF of the LVBF of Lemma 6.8.3, for $k = 3$.

$\{s_1, \ldots, s_k\}$ be a finite set, and let $\mathcal{L} = \langle \mathsf{BF}(S), \preceq \rangle$ be the (finite and distributive) *lattice of Boolean functions over* S, such that $f_1 \preceq f_2$ iff $\forall\, v \in 2^S$: if $f_1(v) = \mathsf{true}$ then $f_2(v) = \mathsf{true}$. It is easy to see that $\mathcal{L}$ satisfies the following, for every $f_1, f_2 \in \mathcal{L}$:

$$f_1 \sqcup f_2 = \lambda v \cdot f_1(v) \vee f_2(v) \qquad\qquad f_1 \sqcap f_2 = \lambda v \cdot f_1(v) \wedge f_2(v)$$

In $\mathcal{L}$, RPC corresponds to the logical implication, such that for any $f_1, f_2 \in \mathcal{L}$:

$$f_1 \to f_2 = \lambda v \cdot \neg f_1(v) \vee f_2(v)$$

We consider the parametric lattice-valued Boolean formulas $\alpha \in \mathsf{LVBL}\,(\mathbb{P}, \mathcal{L})$:

$$\alpha^k = \bigwedge_{i=1}^{k} (p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j)$$

The LVBDD in SNF representing $[\![\alpha^3]\!]$ is depicted at Figure 6.9.

Definition 6.8.1. *Let* $\theta \in \mathsf{LVBF}\,(\mathbb{P}, \mathcal{L})$ *for some* $\mathbb{P} = \{p_1, \ldots, p_k\}$ *and lattice* $\mathcal{L}$. *A* partial truth assignment *for* θ *is a pair* $\langle \tau, \phi \rangle$ *of disjoint sets* $\tau \subseteq \mathbb{P}$, $\phi \subseteq \mathbb{P}$ *such that there exists* ℓ, $1 \leq \ell \leq k$ *such that* $\tau \cup \phi = \{p_1, \ldots, p_\ell\}$. *The set* τ *represents the set of propositions assigned to* true, *while* ϕ *represents the set of propositions assigned to* false.

The following down function follows a partial truth assignment from the root of an LVBDD, down to the corresponding node, by following lo or hi edges accordingly.

Definition 6.8.2. *Let $n \in \mathsf{LVBDD}(\mathbb{P}, \mathcal{L})$ for some $\mathbb{P} = \{p_1, \ldots, p_k\}$ and lattice $\mathcal{L}$, with $\mathsf{index}(n) = i$, and let $\langle \tau, \phi \rangle$ be a partial truth assignment of $[\![n]\!]$. We define the function* down *as follows:*

$$\mathsf{down}(n, \langle \tau, \phi \rangle) = \begin{cases} n & \text{if } p_i \notin \tau \cup \phi \text{ or if } i = k+1; \\ \mathsf{down}(\mathsf{lo}(n), \langle \tau, \phi \rangle) & \text{if } p_i \in \phi; \\ \mathsf{down}(\mathsf{hi}(n), \langle \tau, \phi \rangle) & \text{if } p_i \in \tau; \end{cases}$$

The following lemma formalizes the worst-case exponential lower bound of the computation of meet or join on LVBDD in SNF. The formula α^k can be represented linearly by an LVBDD in SNF, but both $\alpha^k \wedge p_k$ and $\alpha^k \vee p_k$ make the SNF representation blow up exponentially.

Lemma 6.8.3. *For any $k \in \mathbb{N}_0$ we have:*

- (A) $|\mathsf{SNF}([\![\alpha^k]\!])| = 2k + 1$;
- (B) $|\mathsf{SNF}([\![\alpha^k \wedge p_k]\!])| \geq 2^{k-1}$;
- (C) $|\mathsf{SNF}([\![\alpha^k \vee p_k]\!])| \geq 2^{k-1}$;
- (D) $|\mathsf{SNF}([\![p_k]\!])| = 3$.

Proof. Statement (D) is a direct consequence of the definition of LVBDD. We prove each of (A), (B) and (C) in turn. In these proofs, the $[\![\cdot]\!]$ brackets are often omitted in order to alleviate the notations.

Proof of (A)

Let $n = \mathsf{SNF}([\![\alpha^k]\!])$, let $\langle \tau, \phi \rangle$ be a partial truth assignment of $[\![\alpha^k]\!]$ such that $\tau \cup \phi = \{p_1, \ldots, p_\ell\}$ and $\ell \geq 1$, and let $n' = \mathsf{down}(n, \langle \tau, \phi \rangle)$. We claim that:

$$[\![n']\!] = \bigwedge_{i=\ell+1}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \qquad \text{if } p_\ell \in \tau \qquad (6.9)$$

$$[\![n']\!] = \left(s_\ell \vee \bigvee_{j=1}^{\ell-1} \neg s_j \right) \wedge \bigwedge_{i=\ell+1}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \qquad \text{if } p_\ell \in \phi \qquad (6.10)$$

To show this, we proceed by induction. Note that $\exists\,\mathbb{P} : [\![\alpha^k]\!] = \mathsf{true}$ (simply assign every p_i to true). For $\ell = 1$ we have that (by definition of SNF):

$$\begin{aligned}
[\![\mathsf{down}(n, \langle\{p_1\}, \{\}\rangle)]\!] &= (\exists\,\mathbb{P} : [\![\alpha^k]\!]) \rightarrow ([\![\alpha^k]\!]|_{p_1=\mathsf{true}}) \\
&= [\![\alpha^k]\!]|_{p_1=\mathsf{true}} \\
&= \bigwedge_{i=2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \\
[\![\mathsf{down}(n, \langle\{\}, \{p_1\}\rangle)]\!] &= (\exists\,\mathbb{P} : [\![\alpha^k]\!]) \rightarrow ([\![\alpha^k]\!]|_{p_1=\mathsf{false}}) \\
&= [\![\alpha^k]\!]|_{p_1=\mathsf{false}} \\
&= s_1 \wedge \bigwedge_{i=2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)
\end{aligned}$$

We have thus shown that our claim is true for the base case. For the induction step, we assume that our claim above is correct for $\ell = x$, and show that it is correct for $\ell = x + 1$. We must consider the four cases, depending on whether $p_x \in \tau$ or $p_x \in \phi$ and whether p_{x+1} is added in τ or ϕ. Before handling each case, note that:

$$\begin{aligned}
&\exists\,\mathbb{P} : [\![n' = \mathsf{down}(n, \langle\tau, \phi\rangle)]\!] = \mathsf{true} && \text{if } p_x \in \tau \quad \text{(see (6.9))} \\
&\exists\,\mathbb{P} : [\![n' = \mathsf{down}(n, \langle\tau, \phi\rangle)]\!] = s_x \vee \bigvee_{j=1}^{x-1} \neg s_j && \text{if } p_x \in \phi \quad \text{(see (6.10))}
\end{aligned}$$

Case 1 : $p_x \in \tau$ and p_{x+1} added in τ

$$\begin{aligned}
[\![\mathsf{down}(n, \langle\tau \cup \{p_{x+1}\}, \phi\rangle)]\!] &= \neg\mathsf{true} \vee [\![\mathsf{down}(n, \langle\tau, \phi\rangle)]\!]|_{p_{x+1}=\mathsf{true}} \\
&= \bigwedge_{i=x+2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)
\end{aligned}$$

Case 2 : $p_x \in \phi$ and p_{x+1} added in τ

$$\begin{aligned}
[\![\mathsf{down}(n, \langle \tau \cup \{p_{x+1}\}, \phi \rangle)]\!] &= \neg \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) \vee [\![\mathsf{down}(n, \langle \tau, \phi \rangle)]\!]|_{p_{x+1}=\mathsf{true}} \\
&= \neg \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) \vee \bigwedge_{i=x+2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \\
&= \bigwedge_{i=x+2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \vee \left(\neg s_x \wedge \bigwedge_{j=1}^{x-1} s_j \right) \right) \\
&= \bigwedge_{i=x+2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)
\end{aligned}$$

To obtain the last line above, we exploit the fact that $a \vee (a \wedge b) = a$. Indeed, $\neg s_x$ is always present in the disjunction $\bigvee_{j=1}^{i-1} \neg s_j$, since $i \geq x+2$, and therefore $\neg s_x \wedge \bigwedge_{j=1}^{x-1} s_j$ can disappear.

Case 3 : $p_x \in \tau$ and p_{x+1} added in ϕ

$$\begin{aligned}
[\![\mathsf{down}(n, \langle \tau, \phi \cup \{p_{x+1}\} \rangle)]\!] &= \neg \mathsf{true} \vee [\![\mathsf{down}(n, \langle \tau, \phi \rangle)]\!]|_{p_{x+1}=\mathsf{false}} \\
&= \left(s_{x+1} \vee \bigvee_{j=1}^{x} \neg s_j \right) \wedge \bigwedge_{i=x+2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)
\end{aligned}$$

Case 4 : $p_x \in \phi$ and p_{x+1} added in ϕ

$$\begin{aligned}
[\![\mathsf{down}(n, \langle \tau, \phi \cup \{p_{x+1}\} \rangle)]\!] &= \neg \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) \vee [\![\mathsf{down}(n, \langle \tau, \phi \rangle)]\!]|_{p_{x+1}=\mathsf{false}} \\
= \neg \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) &\vee \left[\left(s_{x+1} \vee \bigvee_{j=1}^{x} \neg s_j \right) \wedge \bigwedge_{i=x+2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \right] \\
&= \left(s_{x+1} \vee \bigvee_{j=1}^{x} \neg s_j \right) \wedge \bigwedge_{i=x+2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)
\end{aligned}$$

The last line above is obtained in the same way than for Case 2.

We have now shown that our claim (6.9,6.10) is correct. What is more, (6.9,6.10) shows that the semantics of a node $\mathsf{down}(\mathsf{SNF}([\![\alpha^k]\!]), \langle \tau, \phi \rangle)$ depends *only* on the variable of highest index in $\tau \cup \phi$. Since there are two

possibilities for each of the k indices, we know that $\mathsf{SNF}([\![\alpha^k]\!])$ has exactly $2k$ descendants, and therefore that $|\mathsf{SNF}([\![\alpha^k]\!])| = 2k + 1$.

Proof of (B)

Let $n = \mathsf{SNF}([\![\alpha^k \wedge p_k]\!])$, let $\langle \tau, \phi \rangle$ be a partial truth assignment of $[\![\alpha^k \wedge p_k]\!]$ such that $\tau \cup \phi = \{p_1, \ldots, p_\ell\}$ and $\ell \geq 1$, and let $n' = \mathsf{down}(n, \langle \tau, \phi \rangle)$. We claim that:

$$[\![n']\!] = \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < \ell}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \bigwedge_{i=\ell+1}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \qquad \text{if } p_\ell \in \tau \tag{6.11}$$

$$[\![n']\!] = \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < \ell}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \left(s_\ell \vee \bigvee_{j=1}^{\ell-1} \neg s_j \right) \wedge \bigwedge_{i=\ell+1}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \qquad \text{if } p_\ell \in \phi \tag{6.12}$$

Again, we proceed by induction. Also, we have that $\exists\, \mathbb{P} : [\![\alpha^k \wedge p_k]\!] = \mathsf{true}$.

For $\ell = 1$ we have that (by definition of SNF):

$$\begin{aligned}
[\![\mathsf{down}(n, \langle\{p_1\}, \{\}\rangle)]\!] &= (\exists\, \mathbb{P} : [\![\alpha^k \wedge p_k]\!]) \rightarrow ([\![\alpha^k \wedge p_k]\!]|_{p_1=\mathsf{true}}) \\
&= [\![\alpha^k \wedge p_k]\!]|_{p_1=\mathsf{true}} \\
&= p_k \wedge \bigwedge_{i=2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \\
[\![\mathsf{down}(n, \langle\{\}, \{p_1\}\rangle)]\!] &= (\exists\, \mathbb{P} : [\![\alpha^k \wedge p_k]\!]) \rightarrow ([\![\alpha^k \wedge p_k]\!]|_{p_1=\mathsf{false}}) \\
&= [\![\alpha^k \wedge p_k]\!]|_{p_1=\mathsf{false}} \\
&= p_k \wedge s_1 \wedge \bigwedge_{i=2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)
\end{aligned}$$

We have thus shown that the claim (6.11,6.12) is true for the base case. For the induction step, we assume that our claim above is correct for $\ell = x$, and show that it is correct for $\ell = x+1$. We must again consider the four cases, depending on whether $p_x \in \tau$ or $p_x \in \phi$ and whether p_{x+1} is added in τ or ϕ. Before handling each case, note that:

$$\begin{aligned}
&\exists\, \mathbb{P} : [\![\mathsf{down}(n, \langle\tau, \phi\rangle)]\!] = \mathsf{true} && \text{if } p_x \in \tau \\
&\exists\, \mathbb{P} : [\![\mathsf{down}(n, \langle\tau, \phi\rangle)]\!] = s_x \vee \bigvee_{j=1}^{x-1} \neg s_j && \text{if } p_x \in \phi
\end{aligned}$$

Case 1 : $p_x \in \tau$ and p_{x+1} added in τ

$$\begin{aligned}
&[\![\mathsf{down}(n, \langle\tau \cup \{p_{x+1}\}, \phi\rangle)]\!] = \neg\mathsf{true} \vee [\![\mathsf{down}(n, \langle\tau, \phi\rangle)]\!]|_{p_{x+1}=\mathsf{true}} \\
&= \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < x}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \bigwedge_{i=x+2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \\
&= \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ \boldsymbol{i < x+1}}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \bigwedge_{i=x+2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)
\end{aligned}$$

We can replace $i < x$ by $i < x+1$ in the line above because $p_x \notin \phi$.

Case 2 : $p_x \in \phi$ and p_{x+1} added in τ

$$[\![\mathsf{down}(n, \langle \tau \cup \{p_{x+1}\}, \phi \rangle)]\!] = \neg \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) \vee [\![\mathsf{down}(n, \langle \tau, \phi \rangle)]\!]|_{p_{x+1}=\mathsf{true}}$$

$$= \neg \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) \vee \left[\left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < x}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) \wedge \bigwedge_{i=x+2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \right]$$

$$= \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < x+1}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \bigwedge_{i=x+2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)$$

Case 3 : $p_x \in \tau$ and p_{x+1} added in ϕ

$$[\![\mathsf{down}(n, \langle \tau, \phi \cup \{p_{x+1}\} \rangle)]\!] = \neg \mathsf{true} \vee [\![\mathsf{down}(n, \langle \tau, \phi \rangle)]\!]|_{p_{x+1}=\mathsf{false}}$$

$$= \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < x}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \left(s_{x+1} \vee \bigvee_{j=1}^{x} \neg s_j \right) \wedge \bigwedge_{i=x+2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)$$

$$= \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ \boldsymbol{i < x+1}}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \left(s_{x+1} \vee \bigvee_{j=1}^{x} \neg s_j \right) \wedge \bigwedge_{i=x+2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)$$

Again, we can replace $i < x$ by $i < x + 1$ in the line above because $p_x \notin \phi$.

Case 4 : $p_x \in \phi$ and p_{x+1} added in ϕ

$$[\![\mathsf{down}(n, \langle \tau, \phi \cup \{p_{x+1}\}\rangle)]\!] = \neg \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) \vee [\![\mathsf{down}(n, \langle \tau, \phi\rangle)]\!]|_{p_{x+1}=\mathsf{false}}$$

$$= \neg \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) \vee \left[\left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < x}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \left(s_x \vee \bigvee_{j=1}^{x-1} \neg s_j \right) \wedge \left(s_{x+1} \vee \bigvee_{j=1}^{x} \neg s_j \right) \wedge \bigwedge_{i=x+2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \right]$$

$$= \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < x+1}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \left(s_{x+1} \vee \bigvee_{j=1}^{x} \neg s_j \right) \wedge \bigwedge_{i=x+2}^{k-1} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right)$$

We have now shown that our claim (6.11,6.12) is correct. Let $n = \mathsf{SNF}([\![\alpha^k \wedge p_k]\!])$, let $\langle \tau, \phi \rangle$ be a partial truth assignment of $[\![\alpha^k \wedge p_k]\!]$ such that $\tau \cup \phi = \{p_1, \ldots, p_{k-1}\}$, and let $n' = \mathsf{down}(n, \langle \tau, \phi \rangle)$. Thanks to claim (6.11,6.12) we know that:

$$[\![n']\!] = \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < k-1}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \qquad \text{if } p_{k-1} \in \tau$$

$$[\![n']\!] = \left(p_k \vee \bigvee_{\substack{p_i \in \phi \\ i < k-1}} \left(\neg s_i \wedge \bigwedge_{j=1}^{i-1} s_j \right) \right) \wedge \left(s_{k-1} \vee \bigvee_{j=1}^{k-2} \neg s_j \right) \qquad \text{if } p_{k-1} \in \phi$$

Therefore, depending on the partial truth assignment $\langle \tau, \phi \rangle$, there are 2^{k-2} possibilities for $\mathsf{down}(n, \langle \tau, \phi \rangle)$ when $p_{k-1} \in \tau$, and 2^{k-2} possibilities for

$\mathsf{down}(n, \langle \tau, \phi \rangle)$ when $p_{k-1} \in \phi$. Since these two sets are disjoint, there are 2^{k-1} possibilities for $\mathsf{down}(\mathsf{SNF}([\![\alpha^k \wedge p_k]\!]), \langle \tau, \phi \rangle)$ when $\tau \cup \phi = \{p_1, \ldots, p_{k-1}\}$. This implies that $|\mathsf{SNF}([\![\alpha^k \wedge p_k]\!])| \geq 2^{k-1}$.

Proof of (C)

Let $n = \mathsf{SNF}([\![\alpha^k \vee p_k]\!])$, let $\langle \tau, \phi \rangle$ be a partial truth assignment of $[\![\alpha^k \vee p_\ell]\!]$ such that $\tau \cup \phi = \{p_1, \ldots, p_\ell\}$ and $\ell \geq 1$, and let $n' = \mathsf{down}(n, \langle \tau, \phi \rangle)$. We claim that:

$$[\![n']\!] = p_k \vee \left[\bigwedge_{p_i \in \phi} \left(s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \wedge \bigwedge_{i=\ell+1}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \right] \tag{6.13}$$

Like for (A) and (B) we proceed by induction. For $\ell = 1$ we have that:

$$\begin{aligned}
[\![\mathsf{down}(n, \langle \{p_1\}, \{\} \rangle)]\!] &= (\exists\, \mathbb{P} : [\![\alpha^k \vee p_k]\!]) \rightarrow ([\![\alpha^k \vee p_k]\!]|_{p_1 = \mathsf{true}}) \\
&= p_k \vee \bigwedge_{i=2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \\
[\![\mathsf{down}(n, \langle \{\}, \{p_1\} \rangle)]\!] &= (\exists\, \mathbb{P} : [\![\alpha^k \vee p_k]\!]) \rightarrow ([\![\alpha^k \vee p_k]\!]|_{p_1 = \mathsf{false}}) \\
&= p_k \vee \left[s_1 \wedge \bigwedge_{i=2}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \right]
\end{aligned}$$

This shows that the claim (6.13) is true for the base case. For the induction step, we assume that our claim above is correct for $\ell = x$, and show that it is correct for $\ell = x + 1$. This time, we must only consider two cases, depending on whether p_{x+1} is added in τ or ϕ. Also, it is easy to see that $\exists\, \mathbb{P} : [\![\mathsf{down}(n, \langle \tau, \phi \rangle)]\!] = \mathsf{true}$ when $x < k$ (that is when $\tau \cup \phi = \{p_1, \ldots, p_x\}$ and $x < k$).

Case 1 : p_{x+1} added in τ

$$\begin{aligned}
&[\![\mathsf{down}(n, \langle \tau \cup \{p_{x+1}\}, \phi \rangle)]\!] = \neg \mathsf{true} \vee [\![\mathsf{down}(n, \langle \tau, \phi \rangle)]\!]|_{p_{x+1} = \mathsf{true}} \\
&= p_k \vee \left[\bigwedge_{p_i \in \phi} \left(s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \wedge \bigwedge_{i=x+1}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \right]
\end{aligned}$$

Case 2 : p_{x+1} added in ϕ

$$\begin{aligned}
&[\![\mathsf{down}(n, \langle \tau, \phi \cup \{p_{x+1}\}\rangle)]\!] = \neg\mathsf{true} \vee [\![\mathsf{down}(n, \langle \tau, \phi\rangle)]\!]|_{p_{x+1}=\mathsf{false}} \\
&= p_k \vee \left[\bigwedge_{p_i \in \phi \cup \{p_{x+1}\}} \left(s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \wedge \bigwedge_{i=x+1}^{k} \left(p_i \vee s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \right]
\end{aligned}$$

We have now shown that our final claim (6.13) is correct. Let $n = \mathsf{SNF}([\![\alpha^k \vee p_k]\!])$, let $\langle \tau, \phi \rangle$ be a truth assignment of $[\![\alpha^k \vee p_k]\!]$ such that $\tau \cup \phi = \{p_1, \ldots, p_k\}$, and let $n' = \mathsf{down}(n, \langle \tau, \phi \rangle)$. Thanks to claim (6.13) we know that:

$$[\![n']\!] = \mathsf{true} \qquad \text{if } p_k \in \tau$$

$$[\![n']\!] = \bigwedge_{p_i \in \phi} \left(s_i \vee \bigvee_{j=1}^{i-1} \neg s_j \right) \qquad \text{if } p_k \in \phi$$

Therefore, there are exactly 2^{k-1} possibilities for $\mathsf{down}(n, \langle \tau, \phi \rangle)$ when $\tau \cup \phi = \{p_1, \ldots, p_k\}$, which implies that $|\mathsf{SNF}([\![\alpha^k \vee p_k]\!])| \geq 2^{k-1}$. □

The direct consequence of Lemma 6.8.3 is that, in general, both the meet and join operations are exponential (in the worst case) in the number of nodes of the operands. However, this negative result should be mitigated by the following observations. First, the result of Lemma 6.8.3 applies to LVBDD defined over *any* lattice. More restricted forms of lattices may lead to more efficient algorithms. Second, the LVBF $[\![\alpha^k]\!]$ of Lemma 6.8.3 has an image which *grows exponentially* in k. If LVBDD are used to encode LVBF that have relatively small images, then this negative result does not apply.

6.9 Empirical Evaluation

In this section, we apply lattice-valued binary decision diagrams to the satisfiability problem for finite-word LTL via a reduction to sAFA emptiness.

In our experiments, we consider two encodings for the LVBF of sAFA transitions. The first encoding uses LVBDD in shared normal form, while

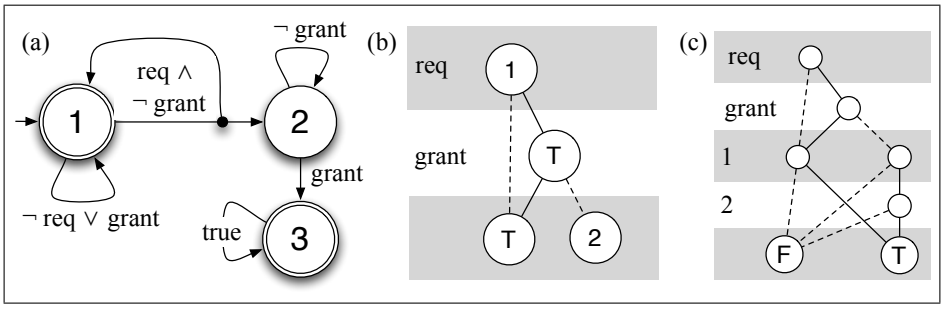


Figure 6.10: An sAFA (a) with the LVBDD (b) and ROBDD (c) encoding the transitions of location 1. The lattice values $\uparrow\{\{1\}\}$ and $\uparrow\{\{2\}\}$ are respectively abbreviated "1", and "2" in the LVBDD.

the second uses ROBDD. The LVBDD encoding uses one decision variable per proposition of the LTL formula. Each node of these LVBDD is labeled with a lattice value, here an upward-closed set of sets of states of the automaton. In all of our experiments, we encode these upward-closed sets with ROBDD, to emphasize the gains of using LVBDD over ROBDD to encode LVBF, independently of the gains of using other representations than ROBDD to encode upward-closed sets. For the ROBDD encoding of LVBF of sAFA transitions, we use one variable per proposition of the LTL formula and location of the automaton (see Chapter 4). Both encodings are illustrated at Fig. 6.10 (b) and (c).

We consider three series of parametric scalable LTL formulas: *mutex* formulas, *lift* formulas and *pattern* formulas. The mutex and lift formulas have been used previously as LTL benchmark formulas in [GMR09], and the pattern formulas were used as benchmark formulas by Vardi *et al.* in [RV07] and previously in [GH06].

For each set of formulas, we supply an *initial ordering* of the propositions. Providing a sensible initial ordering is critical to the fair evaluation of ROBDD-like structures, as these are known to be very sensitive to variable ordering.

The mutex formulas describe the behavior of n concurrent processes involved in a mutual exclusion protocol. The proposition c_i indicates that

process i is in its critical section, r_i that it would like to enter the critical section, and d_i that it has completed its execution. The initial ordering on the propositions is $r_1, c_1, d_1, \ldots, r_n, c_n, d_n$. We check that $\mu(n) \wedge \neg(\mathsf{F}r_1 \rightarrow \mathsf{F}d_1)$ is unsatisfiable.

$$\mu(n) \equiv \bigwedge_{i=1}^{n} \big(\mathsf{G}(c_i \Rightarrow \bigwedge_{j \neq i} \neg c_j) \wedge \mathsf{G}(r_i \Rightarrow \mathsf{F}c_i) \wedge \mathsf{G}(c_i \Rightarrow \mathsf{F}\neg c_i) \wedge \neg d_i \wedge \mathsf{G}((c_i \wedge \mathsf{X}\neg c_i) \Rightarrow d_i)\big)$$

The lift formula describes the behavior of a lift system with n floors. The proposition b_i indicates that the button is lit at floor i, and f_i that the lift is currently at floor i. The initial variable ordering is $b_1, f_1, \ldots, b_n, f_n$. We check that $\lambda(n) \wedge \neg(b_{n-1} \Rightarrow (\neg f_{n-1}\ \mathsf{U} f_n))$ is unsatisfiable.

$$\lambda(n) \equiv f_1 \wedge \bigwedge_{i=1}^{n} \Big(\mathsf{G}(b_i \Rightarrow (b_i\ \mathsf{U} f_i) \wedge \mathsf{G}\big(f_i \Rightarrow (\neg b_i \wedge \bigwedge_{j \neq i}^{n} \neg f_j \wedge \neg \mathsf{X} f_j) \wedge (\bigvee_{i-1 \leq j \leq i+1} \mathsf{XX} f_j)\big)\Big)$$

The pattern formulas of [RV07] are found below, and their initial proposition ordering is set to $p_1, \ldots, p_n$.

$$\begin{array}{rlrl}
E(n) & = \bigwedge_{i=1}^{n} \mathsf{F}p_i & C_1(n) & = \bigvee_{i=1}^{n} \mathsf{GF}p_i \\
U(n) & = (\ldots(p_i\ \mathsf{U} p_2)\ \mathsf{U}\ldots)\ \mathsf{U} p_n & C_2(n) & = \bigwedge_{i=1}^{n} \mathsf{GF}p_i \\
R(n) & = \bigwedge_{i=1}^{n} (\mathsf{GF}p_i \vee \mathsf{FG}p_{i+1}) & Q(n) & = \bigwedge_{i=1}^{n-1} (\mathsf{F}p_i \vee \mathsf{G}p_{i+1}) \\
U_2(n) & = p_1\ \mathsf{U}(p_2\ \mathsf{U}(\ldots p_{n-1}\ \mathsf{U} p_n)\ldots) & S(n) & = \bigwedge_{i=1}^{n} \mathsf{G}p_i
\end{array}$$

In order to evaluate the practical performances of LVBDD, we have implemented two nearly identical C++ prototypes. Both prototypes first translate the LTL formula into a sAFA, and then perform the antichain-based forward fixed point using the algorithms described in Chapter 3 and Chapter 4. The two prototypes differ only in the encoding of the LVBF of the sAFA transitions: one uses LVBDD while the other uses the BuDDy [LN] implementation of ROBDD with the SIFT reordering method enabled. We also compare the performance of our two prototypes with the NuSMV tool. The use of NuSMV for finite-word LTL satisfiability is straightforward: we translate the formula into a sAFA, which is then encoded into an SMV module with one input variable (`IVAR`) per proposition and one state variable (`VAR`) per location of the automaton. To check for satisfiability, we ask NuSMV to verify the property "`CTLSPEC AG !accepting`" where `accepting` is a formula which denotes the set of accepting configurations of the automaton. We invoke NuSMV with the "`-AG`" and "`-dynamic`"

command-line options which respectively enable a single forward reachability computation and dynamic reordering of BDD variables. We now present two sets of experiments which illustrate the practical efficiency of LVBDD in terms of *running time* and *compactness*.

6.9.1 Running Time Comparison

In our first set of experiments, we have compared the respective running times of our prototypes and NuSMV on the benchmarks described above. These results are reported in Table 6.9, where we highlight the best running times in bold.

It is well-known that ordered decision diagrams in general are very sensitive to the ordering of the variables. In practice, ROBDD packages implement *dynamic variable reordering techniques* to automatically avoid bad variable orderings. However, these techniques are known to be sensitive to the *initial variable ordering*, so a sensible initial ordering is a necessary component to the fair evaluation of ordered decision diagrams. In our experiments, we have two sets of variables which respectively encode the LTL propositions and the sAFA states. We provide an initial sensible ordering for both sets of variables; the LTL propositions are initially ordered as described previously, and the AFA locations are ordered by following a topological sort[1]. Finally, for the ROBDD-based tools, we provide an initial ordering such that the LTL propositions variables *precede* the sAFA location variables. In Table 6.9, the "NuSMV + ord" and "NuSMV" columns respectively contain the running times of NuSMV when provided with our initial ordering, or without any initial ordering.

On most of the examples, the LVBDD-based prototype performs better than NuSMV and the ROBDD prototype. For the *mutex* and *lift* benchmarks, LVBDD seem to scale much better than ROBDD. We have investigated the scalability of ROBDD on these instances with profiling tools, which revealed that a huge proportion of the run time is spent on variable reordering. Disabling dynamic reordering for either the ROBDD-based prototype or NuSMV on these instances made matters even worse, with

[1]This ordering is sensible because the translation from LTL produces AFA that are *very weak*.

neither NuSMV nor our ROBDD-based prototype being able to solve them for parameter values beyond 30. These observations shed light on one of the key strengths of LVBDD in the context of LVBF representation. While ROBDD-based encodings must find a *suitable interleaving* of the domain and codomain variables, which can be very costly, LVBDD avoid this issue altogether, even when codomain values are encoded using ROBDD.

Finally, the results for the pattern formulas confirm earlier research, by showing that the antichain approach (i.e., columns ROBDD, LVBDD) and the fully-symbolic approach (NuSMV in our case) exhibit performance behaviors that are incomparable in general; in the Q benchmark, the antichains grow exponentially in length, while the S benchmark makes the ROBDD reordering-time grow exponentially.

6.9.2 Compactness Comparison

In this set of experiments, we compare the compactness of LVBDD and ROBDD when encoding LVBF occurring along the computation of the fixed point that solves the satisfiability for the *lift* formulas. These experiments are reported in Table 6.9. We report on the largest and average structure sizes encountered along the fixed point. We performed the experiments for ROBDD both with and without dynamic reordering enabled, and for two different reordering techniques provided in the BuDDy package: SIFT and WIN2. The sizes reported for LVBDD is equal to the number of decision nodes of the LVBDD *plus* the number of unique ROBDD nodes that are used to encode the lattice values labeling the LVBDD. This metric is thus an *accurate* representation of the total memory footprint of LVBDD and is *fair* for the comparison with ROBDD. These experiments show that, as expected, LVBDD are more compact than ROBDD in the context of LVBF representation, although ROBDD can achieve sizes that are comparable with LVBDD, but at the price of a potentially very large reordering overhead. This increased compactness explains the better running times of the LVBDD prototype reported in Table 6.9.

All experiments were performed with a timeout of 1000 seconds on an Intel Core i7 3.2 Ghz CPU with 12 GB of RAM. A preliminary version of our C++ LVBDD library is freely available at `http://www.ulb.ac.be/di/ssd/nmaquet/`.

6.10 Discussion

In this work, we have developed LVBDD, a new ROBDD-inspired symbolic data structure for the efficient encoding of functions of the form $2^{\mathbb{P}} \mapsto \mathcal{L}$, where $\mathcal{L}$ is a finite and distributive lattice.

We have defined two distinct semantics for LVBDD (the meet-semantics and join-semantics), and shown that each one is exponentially more compact than the other on certain families of LVBF. As stated in the introduction, we developed our theory only for meet-semantics LVBDD, but it is very likely that all results presented in this chapter are applicable to join-semantics LVBDD also (see *dual relative pseudo-complement* in the literature [Bir67]).

We introduced two normal forms for LVBDD. The unshared normal form (UNF) requires that all non-terminal nodes be labeled with the value $\top$. In terms of algorithms and complexity, LVBDD in UNF are nearly identical to ADD and MTBDD, so they are not discussed in great detail. The shared normal form (SNF) uses the lattice-theoretic *relative pseudo-complement* operation and existential quantification to find suitable intermediary lattice values to label non-terminal nodes, in order to increase sharing in the LVBDD DAG. We prove in this work that the SNF is indeed exponentially more compact than the UNF in the best case (it also cannot be less compact).

A large part of this work was to define efficient algorithms for the manipulation of LVBDD in shared normal forms. The soundness proofs of these algorithms required the study of a number of *algebraic properties* of the RPC operator on finite and distributive lattices, in which we have used Birkhoff's representation theorem in order to devise proofs that apply to any finite and distributive lattice. Thanks to these generic proofs, our LVBDD algorithms are shown to be sound for any finite and distributive lattice.

We have studied the worst-case complexity of LVBDD algorithms. Computing the meet and join of LVBDD in UNF has a $\mathcal{O}(n^2)$ worst-case complexity, where n is the number of nodes of the operands, similarly to ROBDD, MTBDD or ADD. On the other hand, we have shown that the computation of meet and join on LVBDD in SNF is worst-case exponential. As discussed previously, this negative result should be mitigated by the fact that it is a very general result, valid for any lattice, and that the size of the lattice

codomain must also grow exponentially for this result to hold.

Finally, we have implemented our LVBDD data structure as a C++ template library and made it available to the research community. We have compared the performance of LVBDD against ROBDD in the context of solving the satisfiability problem for LTL over finite words. Our experiments reveal that LVBDD have the advantage of *decoupling* the domain and codomain constraints of LVBF, while ROBDD must find an appropriate interleaving of domain and codomain variables. We believe that our preliminary experimental results with LVBDD are very encouraging.

Several lines of future works can be considered. First, the join-semantics for LVBDD should be studied in depth, in order to formally verify that all results discussed in this chapter also hold for this semantics. Also, a more fine-grained complexity analysis of LVBDD in SNF could be devised. In particular, we conjecture that the worst-case exponential behavior of the meet operation does not occur on the lattice of upward-closed sets (the join is still exponential, however). It would be also interesting to know the worst-case complexity of LVBDD algorithms when the size of the codomain lattice is bounded by a constant. Finally, in order to confirm the usefulness of LVBDD, we should try to apply them other verification problems and see how they fare in these contexts. In particular, we believe that LVBDD should be useful in the context of *abstract interpretation* [CC77], where they would allow to manipulate more efficiently the abstract domains that mix Boolean variables (to encode some control state for instance), and some numerical abstract domain (to encode more fine-grained information about the numerical variables).

	n	locs	props	iters	ROBDD	LVBDD	NuSMV + ord	NuSMV
Mutex	40	328	121	3	12	**2**	12	11
	80	648	241	3	74	**12**	31	34
	120	968	361	3	284	**37**	87	180
	160	1288	481	3	t.o.	**79**	206	325
	200	1608	601	3	-	**132**	t.o.	t.o.
Lift	20	98	42	41	1	**1**	1	1
	40	178	82	81	10	**3**	6	10
	60	258	122	121	36	**8**	24	44
	80	338	162	161	83	**17**	53	162
	100	418	202	201	188	**31**	185	581
	120	498	242	241	330	**51**	341	t.o.
	140	578	282	281	t.o.	**81**	t.o.	-
E	100	103	101	2	12	**0.1**	1	1
	200	203	201	2	110	**0.3**	4	4
	300	303	301	2	392	**0.6**	19	19
U	100	102	101	2	**1**	1	2	2
	200	202	201	2	**7**	12	23	19
	300	302	301	2	**39**	44	117	116
R	6	27	8	2	0.2	0.4	**0.1**	**0.1**
	8	35	10	2	20	21	**0.1**	**0.1**
	10	43	12	2	t.o.	t.o.	**0.1**	**0.1**
U2	15	17	16	2	0.1	0.5	**0.1**	**0.1**
	20	22	21	2	0.2	23	**0.1**	**0.1**
	25	27	26	2	0.3	t.o.	**0.1**	**0.1**
C_1	100	203	101	2	**0.1**	**0.1**	2	2
	200	403	201	2	**0.2**	**0.2**	8	11
	300	603	301	2	**0.5**	**0.6**	25	20
C_2	100	203	101	2	8	**0.2**	2	2
	200	403	201	2	73	**0.7**	8	11
	300	603	301	2	t.o.	**1.6**	65	21
Q	10	23	12	2	1	1	**0.1**	**0.1**
	15	33	17	2	205	234	**0.1**	**0.1**
	20	43	22	2	t.o.	t.o.	**0.2**	**0.1**
S	200	203	201	2	**0.1**	**0.1**	3	4
	300	303	301	2	**0.1**	**0.3**	6	11
	400	403	401	2	**0.1**	**0.4**	16	25

Table 6.1: Running times in seconds. 'n' is the parameter of the formula, 'locs' the number of locations of the AFA, 'props' the number of propositions of the LTL formula and 'iters' the number of iterations of the fixed point. Time out set at 1,000 seconds.

				LVBDD		ROBDD					
						no reord.		SIFT		WIN2	
n	locs	props	iters	avg.	max.	avg.	max.	avg.	max.	avg.	max.
10	58	22	21	58	284	2,374	8,190	2,374	8,190	2,374	8,190
12	66	26	25	65	300	9,573	32,766	682	32,766	695	32,766
14	74	30	29	75	306	t.o.	t.o.	269	33,812	1,676	131,070
50	218	102	101	201	654	t.o.	t.o.	270	1,925	t.o.	t.o.
100	418	202	201	376	1,304	t.o.	t.o.	469	1,811	t.o.	t.o.

Table 6.2: Comparison of ROBDD and LVBDD sizes on the lift example.

Chapter 7

Conclusion

The purpose of this thesis was to develop new algorithms for the emptiness problem of alternating automata on both finite and infinite words. Our work comprises three distinct approaches to that problem.

First, we have studied how the use of antichains can be combined with reduced ordered binary decision diagrams, in order to obtain new algorithms for the analysis of alternating automata over symbolic alphabets. We have experimentally validated this approach and shown that even prototypical implementations of these techniques are competitive with state-of-the-art tools in terms of efficiency.

Second, we have shown how the analysis of alternating automata can benefit from the combined use of antichains and abstract interpretation, when the number of states of the automaton grow very large. In this case also, we have empirically demonstrated that, for automata with sufficiently many states, the antichain algorithm equipped with abstraction and refinement scales better.

Finally, we have defined a new symbolic data structure, that is especially designed for the encoding of the transition functions of alternating automata over symbolic alphabets. Furthermore, we have defined this data structure in a generic way, in the hope that it could be useful also in other contexts. For this work also, we have provided an implementation and experimentally verified its efficiency on a number of meaningful examples.

Individual future works related to each of the core chapter of this thesis

(Chapters 4, 5, 6) are discussed in the respective *discussions* sections of these chapters, so we do not repeat these arguments here. More generally, we believe that many of the techniques and results found in this thesis could be applied to other settings than alternating automata and model checking. Other applications may for instance be found in solving the *synthesis* or *realizability* problems of LTL (or other logics). Similarly, some of the abstraction and refinement techniques presented in this work could be lifted to other finite-state models. And, finally, LVBDD are likely (in our opinion) to find other fruitful uses, beyond the analysis of alternating automata.

Index

Bibliography

[ACH+10] Parosh Aziz Abdulla, Yu-Fang Chen, Lukás Holík, Richard Mayr, and Tomás Vojnar. When simulation meets antichains. In *TACAS*, pages 158–174, 2010.

[AH97] Henrik Reif Andersen and Henrik Hulgaard. Boolean expression diagrams (extended abstract). In *LICS*, pages 88–98, 1997.

[ATW06] Christoph Schulte Althoff, Wolfgang Thomas, and Nico Wallmeier. Observations on determinization of büchi automata. *Theor. Comput. Sci.*, 363(2):224–233, 2006.

[BCD+08] Dietmar Berwanger, Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, and Sangram Raje. Strategy construction for parity games with imperfect information. In *CONCUR*, pages 325–339, 2008.

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and J L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS'90: Proc. 5th Annual IEEE Symp. on Logic in Computer Science*, pages 1–33. IEEE Computer Society, 1990.

[BFG+97] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[BHH+08] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *CIAA*, volume 5148 of *LNCS*, pages 57–67, 2008.

[BHSV+96] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. Vis: A system for verification and synthesis. In *CAV*, pages 428–432, 1996.

[Bir67] Garrett Birkhoff. Lattice theory. *American Mathematical Society*, 25(3):420, 1967.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BS81] S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Springer, 1981.

[Büc62] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. of the International Congress on logic, Math, and Philosophy of Science (1960)*. Standford University Press, 1962.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and Armando Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *CAV'02: Proc. 14th Int. Conf.*

on Computer Aided Verification, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[CEP01] Marsha Chechik, Steve M. Easterbrook, and Victor Petrovykh. Model-checking over multi-valued logics. In *FME*, pages 72–98, 2001.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGH94] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In *CAV*, volume 818 of *LNCS*, pages 415–427. Springer, 1994.

[CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CGR07] P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In *SAS*, volume 4634 of *LNCS*, pages 333–348. Springer, 2007.

[CKS81] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158. ACM, 1971.

[DC03] Benet Devereux and Marsha Chechik. Edge-shifted decision diagrams for multiple-valued logic. *Multiple-Valued Logic and Soft Computing*, 9(1), 2003.

[DDHR06] M. De Wulf, L. Doyen, T.A. Henzinger, and J-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV*, LNCS 4144, pages 17–30. Springer, 2006.

[DDMR08a] Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-François Raskin. ALASKA : Antichains for logic, automata and symbolic krikpe structures analysis. In *ATVA*, pages 240–245, 2008.

[DDMR08b] Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-François Raskin. Antichains: Alternative algorithms for LTL satisfiability and model-checking. In *TACAS*, LNCS 4963, pages 63–77. Springer, 2008.

[DDR06] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. A lattice theory for solving games of imperfect information. In *HSCC*, pages 153–168, 2006.

[DGV99] M. Daniele, F. Giunchiglia, and M. Vardi. Improved automata generation for linear temporal logic. In *CAV*, volume 1633 of *LNCS*, pages 249–260. Springer, 1999.

[DP04] Alexandre Duret-Lutz and Denis Poitrenaud. Spot: An extensible model checking library using transition-based generalized büchi automata. In *MASCOTS*, pages 76–83, 2004.

[DR07] L. Doyen and J-F. Raskin. Improved algorithms for the automata-based approach to model-checking. In *TACAS*, volume 4424 of *LNCS*, pages 451–465. Springer-Verlag, 2007.

[DR09] Laurent Doyen and Jean-François Raskin. Antichains for the automata-based approach to model-checking. *Logical Methods in Computer Science*, 5(1), 2009.

[DR10] Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In *TACAS*, pages 2–22, 2010.

[DRB04] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Covering sharing trees: a compact data structure for parameterized verification. *STTT*, 5(2-3):268–297, 2004.

[EL86] E. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *LICS*, pages 267–278. IEEE, 1986.

[FJR09] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for LTL realizability. In *CAV*, pages 263–277, 2009.

[FMY97] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[Fri03] C. Fritz. Constructing Büchi automata from LTL using simulation relations for alternating Büchi automata. In *CIAA*, volume 2759 of *LNCS*, pages 35–48. Springer, 2003.

[Gan07] P. Ganty. *The Fixpoint Checking Problem: An Abstraction Refinement Perspective*. PhD thesis, Université Libre de Bruxelles, 2007.

[GH06] Jaco Geldenhuys and Henri Hansen. Larger automata and less work for ltl model checking. In *SPIN*, pages 53–70, 2006.

[GKL+10] Gilles Geeraerts, Gabriel Kalyon, Tristan Le Gall, Nicolas Maquet, and Jean-François Raskin. Lattice-valued binary decision diagrams. In *ATVA*, pages 158–172, 2010.

[GKSV03] Sankar Gurumurthy, Orna Kupferman, Fabio Somenzi, and Moshe Y. Vardi. On complementing nondeterministic büchi automata. In *CHARME*, pages 96–110, 2003.

[GMR09] Pierre Ganty, Nicolas Maquet, and Jean-François Raskin. Fixpoint guided abstraction refinement for alternating automata. In *CIAA*, pages 155–164, 2009.

[GMR10] Pierre Ganty, Nicolas Maquet, and Jean-François Raskin. Fixed point guided abstraction refinement for alternating automata. *Theor. Comput. Sci.*, 411(38-39):3444–3459, 2010.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *CAV*, pages 53–65, 2001.

[GRB08] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. From many places to few: Automatic abstraction refinement for petri nets. *Fundam. Inform.*, 88(3):275–305, 2008.

[Har05] A. Harding. *Symbolic Strategy Synthesis For Games With LTL Winning Conditions*. PhD thesis, University of Birmingham, 2005.

[HHK95] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 453–, 1995.

[HKM05] Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly LTL model checking. In Nicolas Halbwachs and Lenore Zuck, editors, *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, Lecture Notes in Computer Science, page 191, Edinburgh, Scotland, April 2005. Springer-Verlag.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997.

[HPY96] Gerard Holzmann, Doron Peled, and Mihalis Yannakakis. On nested depth first search (extended abstract). In *In The Spin Verification System*, pages 23–32. American Mathematical Society, 1996.

[iM93] Shin ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC*, pages 272–277, 1993.

[KL07] Orna Kupferman and Yoad Lustig. Lattice automata. In *VMCAI*, pages 199–213, 2007.

[Kup06] Orna Kupferman. Sanity checks in formal verification. In *CONCUR*, pages 37–51, 2006.

[Kur94] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, Princeton, NJ, USA, 1994.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *ACM*, 17(8):453–455, 1974.

[LN] Jørn Lind-Nielsen. BuDDy - a binary decision diagram package. `http://sourceforge.net/projects/buddy/`.

[McM99] K. L. McMillan. The SMV system. Technical report, 1999. Technical Report.

[Mer01] Stephan Merz. Model checking: A tutorial overview. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer Berlin / Heidelberg, 2001.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on omega-words. In *CAAP*, pages 195–210, 1984.

[Mic88] M. Michel. Complementation is more difficult with automata on infinite words. CNET, Paris, 1988.

[MS72] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:125–129, 1972.

[MS03] Stephan Merz and Ali Sezgin. Emptiness of Linear Weak Alternating Automata. Technical report, 2003. Technical Report.

[Pap94] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pnu77] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[RCDH07] J.-F. Raskin, K. Chatterjee, L. Doyen, and T. A. Henzinger. Algorithms for omega-regular games with imperfect information. *Logical Methods in Computer Science*, 3(3), 2007.

[Roh97] S. Rohde. *Alternating Automata and the Temporal Logic of Ordinals*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[RRT08] Francesco Ranzato, Olivia Rossi Doria, and Francesco Tapparo. A forward-backward abstraction refinement algorithm. In *VMCAI '08: Proc. 9th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, LNCS. Springer, 2008.

[RV07] K Rozier and M. Y. Vardi. LTL satisfiability checking. In *14th Int'l SPIN Workshop*, volume 4595 of *LNCS*, pages 149–167. Springer, 2007.

[Saf88] Shmuel Safra. On the complexity of omega-automata. In *FOCS*, pages 319–327, 1988.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.

[Som98] F. Somenzi. CUDD: CU Decision Diagram Package. University of Colorado, 1998.

[STV05] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: On hybrid approaches for LTL symbolic model checking. In *CAV*, pages 350–363, 2005.

[Tau03] Heikki Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2003.

[TV05] Deian Tabakov and Moshe Y. Vardi. Experimental evaluation of classical automata constructions. In *LPAR*, pages 396–411, 2005.

[Var95] Moshe Vardi. Alternating automata and program verification. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer Berlin / Heidelberg, 1995. 10.1007/BFb0015261.

[Var08] Moshe Y. Vardi. Automata-theoretic model checking revisited. In *Haifa Verification Conference*, page 2, 2008.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Logic in Computer Science*, pages 332–344, 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.