THESIS

submitted to the

University of Toulouse

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science, specialty: Artificial Intelligence

delivered by the

Toulouse III–Paul Sabatier University (EDMITT)

Knowledge Compilation for Online Decision-Making: Application to the Control of Autonomous Systems

---

Presented and defended by

Alexandre NIVEAU

on March the 27th, 2012

Advisors:

Hélène FARGIER	dir. de recherche CNRS	Paul Sabatier University
Cédric PRALET	ingénieur de recherche	ONERA Toulouse
Gérard VERFAILLIE	directeur de recherche	ONERA Toulouse

Dissertation Examiners:

Enrico GIUNCHIGLIA	professor	University of Genova
Pierre MARQUIS	professor	University of Artois

Jury Members:

Martin COOPER	professor	Paul Sabatier University
Philippe JÉGOU	professor	Paul Cézanne University
Bruno ZANUTTINI	maître de conférences	Univ. of Caen–Basse Normandie

This document was typeset using XeLaTeX and the TeXLive library. Thanks to all contributors.

Version 2.2.1-en, compiled October 15, 2012.

To L.C.,

without whom I would, undoubtedly,
still be in the process of writing Chapter 1.

Abstract

Controlling autonomous systems requires to make decisions depending on current observations and objectives. This involves some tasks that must be executed online—with the embedded computational power only. However, these tasks are generally combinatory; their computation is long and requires a lot of memory space. Entirely executing them online thus compromises the system's reactivity. But entirely executing them offline, by anticipating every possible situation, can lead to a result too large to be embedded. A tradeoff can be provided by knowledge compilation techniques, which shift as much as possible of the computational effort before the system's launching. These techniques consists in a translation of a problem into some language, obtaining a compiled form of the problem, which is both easy to solve and as compact as possible. The translation step can be very long, but it is only executed once, and offline. There are numerous target compilation languages, among which the language of binary decision diagrams (BDDs), which have been successfully used in various domains of artificial intelligence, such as model-checking, configuration, or planning.

The objective of the thesis was to study how knowledge compilation could be applied to the control of autonomous systems. We focused on realistic planning problems, which often involve variables with continuous domains or large enumerated domains (such as time or memory space). We oriented our work towards the search for target compilation languages expressive enough to represent such problems.

In a first part of the thesis, we present various aspects of knowledge compilation, as well as a state of the art of the application of compilation to planning. In a second part, we extend the BDD framework to real and enumerated variables, defining the interval automata (IAs) target language. We draw the compilation map of IAs and of some restrictions of IAs, that is, their succinctness properties and their efficiency with respect to elementary operations. We describe methods for compiling into IAs problems that are represented as continuous constraint networks. In a third part, we define the target language of set-labeled diagrams (SDs), another generalization of BDDs allowing the representation of discretized IAs. We draw the compilation map of SDs and of some restrictions of SDs, and describe a method for compiling into SDs problems expressed as discrete continuous networks. We experimentally show that using IAs and SDs for controlling autonomous systems is promising.

Keywords

knowledge compilation, planning, BDD, automata, model-checking, CSP

Acknowledgements

You would not be reading this thesis but for a number of people, whom I thank wholeheartedly. First of all, I owe a deep gratitude to my advisors Hélène Fargier, Cédric Pralet, and Gérard Verfaillie, for their amazing availability and dedication,[1] for having patiently spurred on my work, leaving me all latitude to go in the directions I wanted without imposing anything—and of course for having proofread this thesis (multiple times).

I am also grateful to all members of the jury: Enrico Giunchiglia and Pierre Marquis, who accepted the burden of being dissertation examiners; Martin Cooper, who presided the committee, and made numerous corrections and suggestions to the text; Philippe Jégou, who helped me relax during the defense by asking questions in French; and Bruno Zanuttini, whose enthusiasm was much appreciated. Also, special thanks to Pierre, if only for having insightfully answered many of my questions about compilation throughout my PhD.

Furthermore, I am thankful to Christophe Garion, Thomas Schiex, and Florent Teichteil-Königsburg for their helpful comments and advice on the occasion of my first-year thesis committee. I owe a lot to Christophe, since he is also the one who introduced me to formal logic, who organized the introductory SUPAERO courses thanks to which I discovered artificial intelligence, and who suggested that I contact Cédric and Gérard for my Master research project.

I would like to offer my regards to people working at ONERA Toulouse, and in particular to thank secretaries and librarians, not only at ONERA but also at IRIT and at the university, for having made my life easier. Thanks to the PRES and ONERA for having awarded me the stipend thanks to which I could carry out my work; also, I stand indebted to people at the CRIL and the IUT de Lens for their warm welcome during the final months of my writing of this dissertation.

Thanks to all students whom I met during my Master and PhD at the ONERA-DCSD. I hold you greatly responsible for having made these years a quite enjoyable experience. I cannot afford being exhaustive—I would necessarily forget someone anyway; but I wanted to at least cite Caroline, Julie, Mario, Pascal, Prof. Jacquet,

[1] How many PhD students have thesis meetings virtually every week?

Quentin, Thibs; the latecomers Damien, Gaëtan, Pierre, Sergio, Simon; and the old timers Julien, Patrice, and Stéphane. If you think you should have been listed here, you just won a beer; drop me an email. Among these, special thanks to Simon for his invaluable help with the cover, to Damien the little delivery boy, and to the mailmen Thibs and Simon (again).[2] Rest assured that I will not hesitate to exploit you again in the future.

Last but not least, I am utterly grateful to my family for a fair numbers of reasons, among which having encouraged my curiosity, and simply having been there when I needed it (and also when I thought I did not). Above all, thanks to my wife for her patience, wisdom, and unqualified support.

[2]While I am at it, let me stress that this work owes nothing at all to the following people: Ben, François, Paul, and Rod.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

A system (vehicle, instrument) is *autonomous* if it is not controlled by a human operator. Its actions are driven by an inner program, which is not modified while the system is working; the program is *embedded* into the system, and allows it to "make decisions" all by itself. Embedded systems are generally subject to a lot of limitations (such as price, weight, or power) that depend on the specific task they have to handle. As a consequence, embedded systems do not have many resources at their disposal, in terms of *memory space* and *computational power* available onboard. However, they generally require high *reactivity*. These constraints are even more significant when dealing with embedded systems controlling aeronautical and spatial autonomous systems, such as drones or satellites: their resources can be severely limited, and lack of reactivity can be a crucial issue.

Decision-making is one of the targets of an artificial intelligence field, namely *planning*. To produce decisions fitting the situation, the program has to solve a *planning problem*. Tools have been designed in order to solve such problems; using these tools, the system can compute suitable decisions to make. However, in the general case, planning problems are hard to solve—they have a high computational complexity. The basic consequence of this fact is that making decisions by solving planning problems takes time, especially on autonomous systems, that have limited computational power. Reactivity is thus not ensured if the problem is solved *online*, i.e., each time it is needed, using only the system's power and memory.

To settle this issue, one possibility is to compute decisions *offline*, that is, before the system is set up. Anticipating all possible situations, and solving the problem for each of them, we could provide the system with a decision table, containing a set of "decision rules" of the form "in *this* situation, make *that* decision". This would guarantee a maximal reactivity for the system, since using such a table online does not require a high computational power.

However, this enhancement in reactivity may imply an important increase in memory space. Indeed, decision-making depends on lots of parameters, including for instance measurements, current (supposed) position, state of components, current data, current objectives, and even some former values of these parameters. This leads to a huge number of possible situations, and thus to a huge number of

decision rules in the decision table, whereas the memory space available online is drastically limited.

We see that a compromise is necessary between reactivity and spatial compactness. *Knowledge compilation* is a way of achieving such a compromise: the objective of this discipline is to study how one can *translate* a problem offline, to facilitate its online resolution. It examines to what extent *target languages*, into which problems can be *compiled*, allow online requests to be tractable, while keeping the representation as compact as possible. Using knowledge compilation amounts to shifting as much computational effort as possible before the system's setting up, thus obtaining good reactivity, while deteriorating space efficiency as little as possible.

Knowledge compilation techniques have proven useful for various applications, including planning. The goal of this thesis was to study how it is possible to advantageously apply these techniques to *real* planning problems related to aeronautical and spatial autonomous systems. We considered a number of real problems, and remarked in particular that the target languages used in state-of-the-art planners are not specifically designed to handle some aspects of our problems, namely continuous domains. We oriented our work towards the application of these state-of-the-art techniques to other (possibly new), more expressive target languages.

This thesis is divided into three parts. The first one develops the *context* of the work, detailing knowledge compilation [Chapter 1] and planning [Chapter 2]. It explains [Chapter 3] why we felt that our subject raised the need for new target compilation languages. The second part deals with *interval automata*, one of the new languages we defined. It presents some theoretical aspects of this language [Chapter 4], details how we can build elements of this language in practice [Chapter 5], and provides experimental results [Chapter 6]. The third part is about another language we defined, *set-labeled diagrams*. Using a similar outline to that of Part II, it begins with definitions and general properties about the language [Chapter 7], presents our compilation algorithm [Chapter 8], and ends with experimental results [Chapter 9].

Part I

Context

CHAPTER

1

Knowledge Compilation

This chapter details the main topic of this work, namely knowledge compilation. In a nutshell, knowledge compilation consists in transforming a problem offline, in order to facilitate its online resolution. This transformation is actually considered as a *translation* from the original *language* in which the problem is described, into a *target language* that has good properties, to wit, allowing the problem to be solved "easily" while remaining as compact as possible.

After having presented and illustrated the basic idea and concepts of knowledge compilation [§ 1.1], we formally define notions pertaining to *languages* [§ 1.2]. A number of state-of-the-art languages are then defined [§ 1.3]; they belong to the particular class of graph-based Boolean languages, on which we focused during the thesis. The next section details some properties of this class, and provides theoretical results from the literature [§ 1.4]. We then give some insight about non-Boolean languages [§ 1.5]. The last two sections of the chapter are dedicated to practical aspects of knowledge compilation [§ 1.6] and successful applications [§ 1.7].

1.1 Presentation

1.1.1 Concepts and History

Knowledge compilation can be seen as a form of "factorization", since the offline phase is dedicated to executing computations that are both *hard* and *common to several online operations*. This idea is not new, as illustrated by the example of tables of logarithms [Mar08]. These tables were used to facilitate hand-made calculations, at a time when calculators were not as small and cheap as today. They contain couples $\langle x, \log_{10}(x) \rangle$, for a lot of different real values of x; these couples could then be used to make complex calculations, such as root extraction. For example, calculating $\sqrt[9]{876}$ by hand is long and hard work. Now, since

$\sqrt[9]{876} = (8.76 \times 10^2)^{1/9}$, it holds that $\log_{10}(\sqrt[9]{876}) = \log_{10}((8.76 \times 10^2)^{1/9}) = (\log_{10}(8.76) + 2)/9$. The user willing to get an approximate value of $\sqrt[9]{876}$ only has to look up the value of $\log_{10}(8.76) \approx 0.942504106$ in the table, compute the fraction $(0.942504106 + 2)/9 = 0.326944901$ (which is easy to do by hand), and look up the antecedent of 0.326944901 by $\log_{10}$ in the table. The user then obtains that $\sqrt[9]{876} \approx 2.122975103$, having used only simple operations.

This example fits the definition of knowledge compilation, even if the context is a bit different (knowledge compilation is not used to facilitate *handmade* calculations). Indeed, the offline phase consists in calculating logarithms, which are hard to obtain *and* potentially useful for several online operations. Each entry can then be used in many different cases (the value of $\log_{10}(8.76)$ can be used to compute the roots of 876, 87.6, $8.76\ldots$), and for each case, the work left to the user is easy.

Let us take an example closer to our subject. In the introduction, the first solution we proposed to tackle the question of autonomous systems' reactivity was to solve the problem beforehand for all possible situations, providing the program with a table of decision rules. Albeit naive, this solution actually relies on knowledge compilation. We can identify the three main points of the definition: the offline phase proceeds with computations that are hard (solving the planning problem for multiple initial situations) and repeated over time (a given situation can be met multiple times; the suitable decision is computed once and for all), and the online phase left to the system is simple (look up in the table the next decision to make).

What is usually (and historically) designated as *compilation* is the translation of a program, from a high-level programming language, easily understandable and modifiable by human beings, to a low-level machine language, much more efficient from a computational point of view. This is also a case of pre-processing, in which we transform something offline to facilitate its online use. The specific *knowledge compilation* domain differs from the general study of pre-processing in two ways.

First, following Marquis [Mar08], the name "knowledge compilation" tautologically limits it to "compilation of knowledge": it deals with the translation of knowledge bases (that is, a set of pieces of information, generally represented as logic formulæ) and with the exploitation of this knowledge, i.e., automated reasoning. The definition remains quite general—lots of things can be understood as "pieces of information"—but typically excludes the classical program compilation from the scope of knowledge compilation.

The second difference between standard compilation and knowledge compilation is a consequence of the previous one [CD97]. Being linked to knowledge representation and reasoning, knowledge compilation inevitably stumbles upon problems considered hard with respect to *computational complexity theory* [see Pap94 for an overview], for example, NP-complete problems, Σ_2^p-complete problems, or even PSPACE-complete problems—all widely conjectured to not be solvable in polynomial time. The goal of researchers in knowledge compilation is not just to make a problem *simpler*, but to make it *drastically* simpler, by changing its complexity class to a more tractable one, for example from NP-complete to P, from PSPACE-complete to NP, etc.

In Chapter 2, we show how solving a planning problem amounts to applying some operations on some functions. Our application, controlling autonomous systems, requires a minimal online complexity; this is why we focused on the possible ways to achieve *polynomial* complexity for the online operations. This removes from our study a number of representations, on which polytime reasoning is not possible [e.g., the high-level logics listed in GK$^+$95]. We basically limit ourselves to graph-based structures.

Research has been made to decide whether knowledge compilation is useful to some given reasoning problem. Roughly speaking, a problem P is considered *compilable into a complexity class* C if and only if there exists a compilation function transforming P into another problem P', such that P' is of size polynomial in the size of P, and P' is in class C. In other words, the compilation step may be really hard—there is no restriction on the time complexity of the compilation function—but the compiled form can take only polynomially more space than the original problem. Compilability has been formalized by Cadoli et al. [CD$^+$96, for an extended review see CD$^+$00].

Since we want our operations to be of polynomial complexity, it would seem that we are only interested in problems *compilable into* P, the complexity class of polytime problems. However, it has been proven by Liberatore [Lib98] that planning (in its *classical* form [§ 2.1.3.3]) is *not* compilable into P. Trying to compile planning problems can nevertheless be useful for several reasons. First, this non-compilability result means that there can be no polysize compilation function transforming *any* planning problem into a tractable one; the compilation function must be common to all problems. It does not imply that a specific planning problem, when taken apart from the others, cannot be transformed into a tractable one. Second, the compilability framework only takes *worst-case complexity* into account—it does not say anything about average or practical space complexity. Last, studying the compilation of subparts of planning problems could be worthwhile anyway.

Before moving on to a practical example of a target compilation language, let us also specify that this work only concerns knowledge compilation in the context of classical propositional inference; we did not study its application to the non-classical inference relations designed for reasoning under inconsistency. We will thus not consider stratified or weighted belief bases, though there exists some literature regarding their compilation [BYD07, DM04].

1.1.2 Example of a Target Language: OBDDs

Compiling a problem is modifying its form (this modification being potentially hard to do), so that the problem in its compiled form is tractable, yet as compact as possible. This can be seen as a *translation* from an *input language* into a *target language*. We give a formal definition of a "language" in the next section [§ 1.2]; before that, let us illustrate what it can look like and how it is generally handled, using the influential language of *ordered binary decision diagrams*, better known as *OBDDs*.

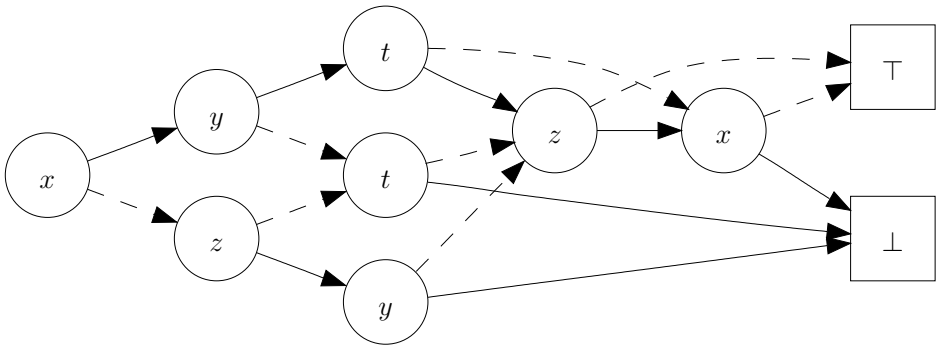

Figure 1.1: An example of a BDD. Solid (resp. dashed) edges are $\top$-labeled ones (resp. $\bot$-labeled ones).

Binary Decision Diagrams

Introduced by Lee [Lee59] and Akers [Ake78], *binary decision diagrams (BDDs)* are rooted directed acyclic graphs that represent Boolean functions of Boolean variables. They have exactly two leaves, respectively labeled $\top$ ("true") and $\bot$ ("false"); their non-leaf nodes are labeled by a Boolean variable and have exactly two outgoing edges, also respectively labeled $\top$ and $\bot$. Figure 1.1 gives an example of a BDD.

A BDD represents a function, in the sense that it associates a unique Boolean value with each *assignment* of the variables it mentions. Let us illustrate this on the simple example of Figure 1.1. Four variables are mentioned; each one of them can take two values. If we choose one value for each variable, for example $x = \top$, $y = \bot$, $z = \top$, and $t = \bot$, we get an assignment of these four variables, among the $2^4 = 16$ possible ones. How does the BDD associate a truth value with this assignment? To get the result, one simply has to start from the root, and follow a path to one of the leaves. The path is completely determined by the chosen assignment: to each node corresponds a variable, the next edge to cross being the one being labeled by the value assigned to this variable. The path ends up either at the $\top$-leaf or at the $\bot$-leaf: the label is the value that the function associates with the chosen assignment.

The name "binary decision diagram" faithfully transcribes this behavior: starting from the root, each node corresponds to a possible decision—"which value do I choose for this variable?"—directing users to the subgraph fitting their choice—"if you decide this, go here, else go there". Note that each possible assignment corresponds to exactly one path; BDDs thus represent *functions* (and not *relations*).

The "Decision Diagram" Family

By imposing structural restrictions on BDDs, we obtain interesting *sublanguages*. For example, a *free BDD (FBDD)* [GM94] is a BDD that satisfies the read-once property: each path contains at most one occurrence of each variable. But the most

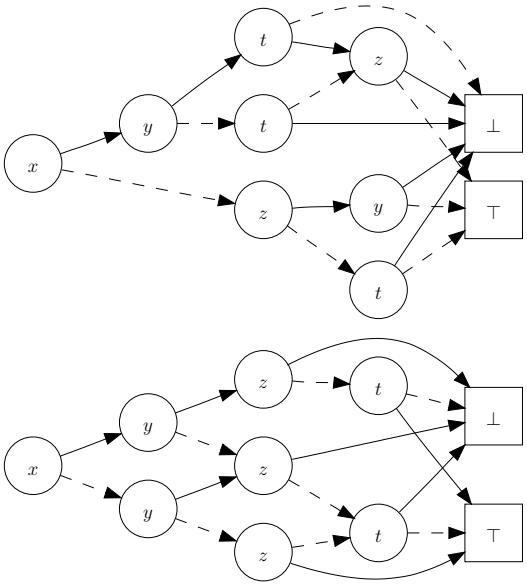

Figure 1.2: An FBDD (top) and an OBDD (bottom), both representing the same function as the BDD of Figure 1.1.

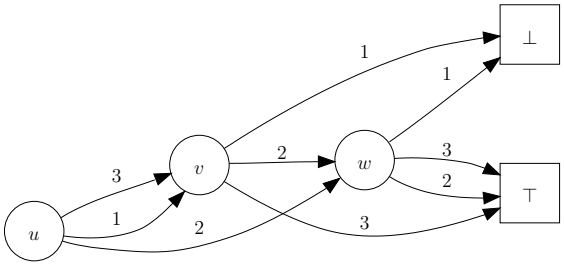

Figure 1.3: An example of an MDD, over variables u, v, and w, of domain $\{1, 2, 3\}$.

influential kind of BDD is the *ordered BDD (OBDD)* [Bry86], in which the order of variables is imposed to be the same in every path. Figure 1.2 shows an FBDD and an OBDD representing the same function as the BDD in Figure 1.1.

The concept of "decision diagram" is not inherently related to Boolean values; the idea has been extended to non-Boolean variables, yielding *multi-valued decision diagrams (MDDs)* [§ 1.3.6], and also to non-Boolean functions, yielding *algebraic decision diagrams (ADDs)* [§ 1.5.1]. Figure 1.3 gives an example of an MDD.

OBDDs Are Compact and Efficient

We showed what OBDDs are, and how they work; but why are they interesting as a knowledge compilation language? There are two simple informal reasons to this: they are efficient, and they are small. Of course, these two aspects are not dissociable. OBDDs are not the smallest possible representation of a function:

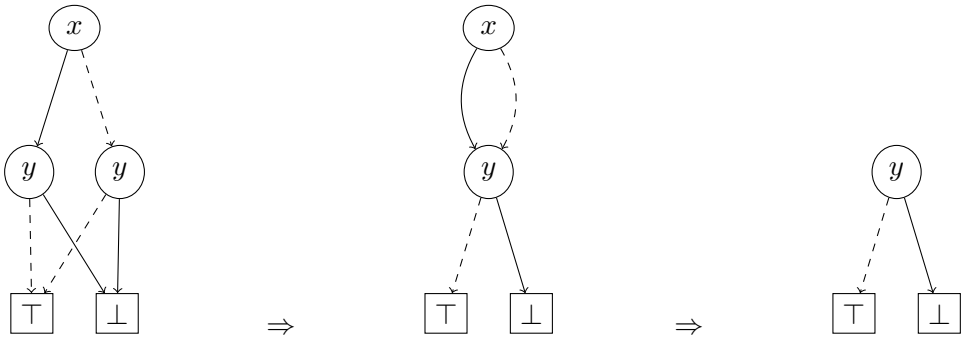

Figure 1.4: Illustration of the reduction procedure on binary decision diagrams. The BDD on the left represents formula $(x \wedge \neg y) \vee (\neg x \wedge \neg y)$. The reduction procedure merges the two y-nodes, which are *isomorphic* (they represent the same function); the result is the BDD in the middle. Then the x-node is removed, because it is *redundant*: the value of x does not matter. The reduced BDD is on the right.

propositional formulæ are generally smaller. Neither are OBDDs the most efficient representations for all applications: computing the conjunction of several functions is easier using truth tables, for example. What is advantageous with OBDDs is that they are *both* efficient and small—they are a good compromise between efficiency and size.

What makes OBDDs so small? The answer lies in their graph structure, that allows *factorization* of "identical" subgraphs, i.e., subgraphs representing the same function—they are said to be *isomorphic*. This factorization can be done thanks to a polytime reduction procedure, described by Bryant [Bry86], and illustrated in Figure 1.4. Not only does this reduction procedure allow exponential gains in space for some cases, but it is also the key to the efficiency of OBDDs. The polynomiality of the reduction procedure is important; it indeed makes it possible to always work on reduced OBDDs. In the following, we always implicitly consider OBDDs to be reduced.

The efficiency of OBDDs is considered with respect to their performances on common, useful operations. Checking whether there exists an assignment satisfying a propositional formula is hard, yet it can be done in constant time if the formula is represented as a reduced OBDD (the only OBDDs that have no satisfying assignment are those that can be reduced to the $\bot$-leaf). More importantly, checking whether two OBDDs using the same order of variables represent the exact same function can be done in time linear in the sum of the two OBDDs' sizes. Moreover, building an OBDD representing the conjunction (or disjunction, or application of any Boolean operator) of two OBDDs using the same variable ordering is only linear in the product of the sizes of the two OBDDs. These remarkable properties of OBDDs were discovered by Bryant [Bry86].

They are not the only language having such interesting properties: all the operations described have for example a linear worst-case complexity on truth ta-

bles. However, OBDDs have these properties while being potentially exponentially smaller than truth tables. Indeed, the memory size necessary to store a truth table is proportional to the number of satisfying assignments of the represented formula, whereas it is not the case for OBDDs.

OBDDs: The Best Language?

Being such a good compromise between spatial and operational efficiency, OBDDs have been widely used for multiple applications over the years. Does this mean they are the best structures to compile Boolean functions? The answer is generally *no*, because it depends on the target application. When one needs to often check whether two functions are equivalent, and often compute conjunctions, OBDDs are probably the best choice; but if the equivalence-checking test is superfluous, and all one needs is a model-checking test, the less constrained BDDs are much better, since they can take exponentially less memory space.

*
**

Choosing a language for one's application is thus an important conception step, that must not be neglected, for a bad choice can lead to great loss in time or space performances—this is particularly crucial in the case of embedded systems. This is rendered difficult by the fact that there exists a lot of target languages, especially in the family of structures representing Boolean functions of Boolean variables. A user thus needs *tools* to compare languages and make a good choice. Introduced by Darwiche and Marquis [DM02], the *knowledge compilation map* provides such tools; we present them in the following section.

1.2 A Framework for Language Cartography

The goal of the *knowledge compilation map* is to compare languages according to their spatial complexity and their efficiency (in terms of worst-case complexity) for different possible operations, such as querying some information on the compiled form, or applying transformations. The map makes it easy to compare fragments from a theoretical complexity point of view, retaining only the best ones for a given application.

The purpose of this section is to present the knowledge compilation map framework. We formally define the notion of *language*, and present the concepts that are used to compare languages—but prior to this, let us introduce our notation conventions. Note that an index of symbols is available at the very end of this thesis [p. 237].

1.2.1 Notation

We denote as $\mathbb{R}$ the set of real numbers, as $\mathbb{Z}$ the set of integers, as $\mathbb{N}$ the set of natural numbers (including 0), as $\mathbb{N}^*$ the set of positive integers (excluding 0), and as $\mathbb{B}$ the set of Boolean constants. We use $\top$ and $\bot$ to refer to the Boolean constants "true" and "false" respectively, but often implicitly identify $\mathbb{B}$ with $\{0, 1\}$, so that the following inclusions hold:

$$\mathbb{B} \subseteq \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}.$$

We use $\mathbb{S}A$ to denote the set of singletons of a set A. Thus, for example, $\mathbb{S}\mathbb{B} = \big\{\{\top\}, \{\bot\}\big\}$, and $\mathbb{S}\mathbb{N} = \bigcup_{n\in\mathbb{N}} \big\{\{n\}\big\}$.

For the sake of generalization, the variables we consider can have any kind of domain—discrete or continuous, finite or infinite…as long as it is not empty. We call $\mathcal{V}$ the set of all possible variables; $\mathcal{V}$ is of course non-empty, and we assume that it is ordered by some total strict order $<_{\mathcal{V}}$. When considering a set of subscripted variables, such as $\{x_1, x_2, \ldots, x_k\}$, it will always be implicitly supposed that $x_1 <_{\mathcal{V}} x_2 <_{\mathcal{V}} \ldots <_{\mathcal{V}} x_k$.

For $x \in \mathcal{V}$, we denote by $\mathrm{Dom}(x)$ its *domain*, which, from a general point of view, can be any non-empty set. Nevertheless, we often need to consider variables defined on specific domains: given a set S, we define $\mathcal{V}_S = \{\, x \in \mathcal{V} \mid \mathrm{Dom}(x) = S \,\}$. Important classes of variables include $\mathcal{B}$, the set of Boolean variables (simply defined as $\mathcal{V}_{\mathbb{B}}$), and $\mathcal{E}$, the set of enumerated variables, that we arbitrarily define as the set of all variables with a finite integer interval domain (formally, $\mathcal{E} = \bigcup_{(a,n)\in\mathbb{Z}\times\mathbb{N}} \mathcal{V}_{\{a,a+1,\ldots,a+n\}}$).

For $X = \{x_1, \ldots, x_k\} \subseteq \mathcal{V}$, $\mathrm{Dom}(X)$ denotes the set of *assignments of variables from* X (or X*-assignments*), that is, $\mathrm{Dom}(X) = \mathrm{Dom}(x_1) \times \mathrm{Dom}(x_2) \times \cdots \times \mathrm{Dom}(x_k)$. We denote as $\vec{x}$ an X-assignment, i.e., $\vec{x} \in \mathrm{Dom}(X)$. Set X is called the *support* of $\vec{x}$. The empty set can have only one assignment, denoted $\vec{\varnothing}$.

Let $X, Y \subseteq \mathcal{V}$, and let $\vec{x}$ be an X-assignment. The *restriction* of $\vec{x}$ to the variables of Y, denoted $\vec{x}_{|Y}$, is the $X \cap Y$-assignment in which each variable takes the same value as in $\vec{x}$. Note that this definition allows Y to be disjoint from X (in which case $\vec{x}_{|Y}$ is always equal to $\vec{\varnothing}$). For a given variable $x_i \in X$, $\vec{x}_{|\{x_i\}}$ thus gives the value taken by x_i in $\vec{x}$; we simply denote it by $\vec{x}_{|x_i}$.

Let $X, Y \subseteq \mathcal{V}$, and let $\vec{x}$ and $\vec{y}$ be some X- and Y-assignments. If X and Y are disjoint, $\vec{x} \,.\, \vec{y}$ denotes the *concatenation* of the two assignments, i.e. the $X \cup Y$-assignment that coincides with $\vec{x}$ on variables from X and with $\vec{y}$ on variables from Y.

Given S and E some sets, we use the notation E^S to designate the set of all functions of the form $f\colon S \to E$; S is called the *input set* of f, and E its *valuation set*. The restriction of f to a set S', denoted $f_{|S'}$, is the function from $S \cap S'$ to E which coincides with f on $S \cap S'$. We generally consider functions verifying $S = \mathrm{Dom}(V)$, with $V \subseteq \mathcal{V}$ some set of variables. We call them functions over the variables from V to the set E. For such a function $f\colon \mathrm{Dom}(V) \to E$, V is called the *scope* of f, and is denoted $\mathrm{Scope}(f)$. Note that we authorize functions to take

as input assignments the support of which is larger than the scope of the function: it will always be implicitly considered that $f(\vec{v})$ means $f(\vec{v}_{|\,\mathrm{Scope}(f)})$.

1.2.2 Representation Language

Let us now introduce the basic elements of the map. At the highest level of abstraction, these elements are called *representation languages*, and have been formalized in the propositional case by Fargier and Marquis [FM09]. We extend their definition here so that it captures all languages presented thereafter.[1]

What Is Expressed: Interpretation Domain

Generally speaking, knowledge compilation languages represent *functions*. These functions are of various kinds: some languages are used to handle Boolean functions, some others to handle real-valued functions; some hold on Boolean variables, some others on enumerated variables. We say that they have different *interpretation domains*—an interpretation domain being the set of functions that are admissible in the language.

Definition 1.2.1 (Interpretation domain)**.** Let $V \subseteq \mathcal{V}$ be a *finite* set of variables, and E any set. The *interpretation domain* associated with V and E is the set

$$\mathfrak{D}_{V,E} = \bigcup_{V' \subseteq V} E^{\mathrm{Dom}(V')}$$

of the functions holding on some variables from V and returning elements in E.

For example, $\mathfrak{D}_{\mathcal{B},\mathbb{B}}$ is the set of Boolean functions of Boolean variables; it is the interpretation domain of OBDDs [§ 1.1.2].

How It Is Expressed: Data Structures

Knowledge compilation aims at expressing functions as instances of specific *data structures*. A data structure [see e.g. CL$^+$01, Part III] is a particular organization of computer data, together with some algorithms allowing data to be handled efficiently, by taking advantage of this organization. Well-known data structures include stacks, queues, hash tables, trees, graphs…

A mathematical object, such as a function, is an abstract concept. To represent it using data structures, it is necessary to *model* it, by identifying the basic operations by means of which it is handled. For instance, *sets* are "things" one can *intersect*, *unite*, *enumerate*, etc. Once a model is described, it is possible to define a data structure implementing this model conveniently. This data structure is not unique; different data structures can be used to express the same mathematical object. This will raise differences in terms of efficiency, but does not necessarily affect the correctness of the representation itself.

[1] Being the most general brick of the map, what we define here as a "representation language" is different from the informal version of Darwiche and Marquis [DM02].

We consider data structures in a rather abstract way, without focusing on the implementation details. One important attribute of data structures that we often refer to is the *memory size* its instances take. We join to each data structure an abstract *characteristic size function*, associating with each instance φ of this data structure a positive integer representative of its size. The characteristic size of φ, that we denote by $\|\varphi\|$ (to distinguish between the characteristic size and the cardinal, denoted as $|S|$), may not be directly equal to its actual memory size $\text{Mem}(\varphi)$. The key point is that characteristic size must have the same growth rate as memory size: $\|\varphi\| \in \Theta(\text{Mem}(\varphi))$, that is, there exists two positive constants k_1 and k_2 such that for sufficiently large φ, $\text{Mem}(\varphi) \cdot k_1 \leqslant \|\varphi\| \leqslant \text{Mem}(\varphi) \cdot k_2$.

For example, let us consider a data structure used to represent enumerated sets of real numbers, and another one used to represent enumerated sets of integers. It is likely that instances of the former need more memory space than instances of the latter, since real numbers are more spatially costly than integers. However, if the data structures are similar except for the type of the numbers, they have the same *characteristic* size, since they are equivalent modulo a multiplicative constant.

A given instance of a given data structure can represent various mathematical objects. For example, the linked list containing $\boxed{\text{a}} \rightarrow \boxed{\text{b}} \rightarrow \boxed{\text{c}}$ can represent the sequence $\langle a, b, c\rangle$, the set $\{a, b, c\}$, the function from $\{1, 2, 3\}$ to $\{a, b, c\}$ associating a with 1, b with 2, and c with 3, etc. This interpretation entirely depends on the context in which it is used: this is where the notion of *language* is needed.

Linking Interpretation Domains to Data Structures: Languages

We can now introduce representation languages, following Fargier and Marquis [FM09].

Definition 1.2.2. A *representation language* is a triple $\mathtt{L} = \langle \mathfrak{D}_{V,E}, \mathcal{R}, [\![\cdot]\!]\rangle$, where:

- $\mathfrak{D}_{V,E}$ is an interpretation domain, associated with some finite set of variables $V \subseteq \mathcal{V}$ (called the *scope* of $\mathtt{L}$) and some set E (called the *valuation set* of $\mathtt{L}$);
- $\mathcal{R}$ is a set of instances of some data structure, also called $\mathtt{L}$-*representations*;
- $[\![\cdot]\!] \colon \mathcal{R} \to \mathfrak{D}_{V,E}$ is the *interpretation function* (or *semantics*) of $\mathtt{L}$, associating with *each* $\mathtt{L}$-representation a function holding on some variables of V and returning elements of E.

A representation language is basically a set of structures called *representations*, each one being associated with a function, called its *interpretation*. This definition is quite general[2] and allows us to introduce several notions (sublanguage, fragment, succinctness…) without making assumptions on variables or data structures.

For the sake of simplicity, when considering a representation language $\mathtt{L}$, we implicitly define it as $\mathtt{L} = \langle \mathfrak{D}_{V_\mathtt{L},E_\mathtt{L}}, \mathcal{R}_\mathtt{L}, [\![\cdot]\!]_\mathtt{L}\rangle$. Given an $\mathtt{L}$-representation φ, we use

[2] It notably is more general than the original definition [FM09], which is in particular limited to the $\mathfrak{D}_{\mathcal{B},\mathbb{B}}$ interpretation domain.

$\text{Scope}_{\mathtt{L}}(\varphi)$ to denote the set of variables on which the interpretation of φ holds: $\text{Scope}_{\mathtt{L}}(\varphi) = \text{Scope}(\llbracket\varphi\rrbracket_{\mathtt{L}})$. When there is no ambiguity, we drop the $\mathtt{L}$ subscript, simply writing $\llbracket\varphi\rrbracket$ and $\text{Scope}(\varphi)$.

To illustrate the notion of representation language, let us take once again the example of BDDs. Section 1.1.2 presented the "binary decision diagram" data structure, and explained how it could be interpreted as a Boolean function over Boolean variables. Those are all the elements of a representation language: denoting $\mathcal{R}_{\text{BDD}}$ the set of all BDDs, and $\llbracket\cdot\rrbracket_{\text{BDD}}$ the interpretation function that has been informally described, the BDD representation language can be defined as $\mathtt{L}_{\text{BDD}} = \langle\mathfrak{D}_{\mathcal{B},\mathbb{B}}, \mathcal{R}_{\text{BDD}}, \llbracket\cdot\rrbracket_{\text{BDD}}\rangle$.

Representation languages form a *hierarchy*, in that each language is obtained by imposing a restrictive property to a more general language—as illustrated in § 1.1.2, FBDDs are read-once BDDs, OBDDs are ordered FBDDs, and so on. This leads us to define the notion of sublanguage.

Definition 1.2.3 (Sublanguage)**.** Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two representation languages. $\mathtt{L}_2$ is a *sublanguage* of $\mathtt{L}_1$ (denoted $\mathtt{L}_2 \subseteq \mathtt{L}_1$) iff each of the following properties hold:

- $\mathfrak{D}_{V_{\mathtt{L}_2},E_{\mathtt{L}_2}} \subseteq \mathfrak{D}_{V_{\mathtt{L}_1},E_{\mathtt{L}_1}}$;
- $\mathcal{R}_{\mathtt{L}_2} \subseteq \mathcal{R}_{\mathtt{L}_1}$;
- $\llbracket\cdot\rrbracket_{\mathtt{L}_2} = (\llbracket\cdot\rrbracket_{\mathtt{L}_1})_{|\mathcal{R}_{\mathtt{L}_2}}$.

It is thus possible to obtain a sublanguage by restricting variables, values, or representations of a more general language; but the interpretation of representations from the child language must remain the same as in the parent language. This definition allows the language of OBDDs to be considered as a sublanguage of the language of MDDs [see § 1.1.2], $\mathtt{L}_{\text{OBDD}} \subseteq \mathtt{L}_{\text{MDD}}$, since these two languages have the same structural properties and valuation domain (Boolean), but are defined on different variables—integer variables being more general than Boolean ones.

Note that one cannot generally restrict the interpretation domain of a language without removing some representations, since the interpretation function $\llbracket\cdot\rrbracket$ must both remain the same and be defined on all of the sublanguage's representations. This can be seen with the next definition.

Definition 1.2.4 (Restriction on variables)**.** Let $\mathtt{L}$ be a representation language, and $X \subseteq \mathcal{V}$. The *restriction of* $\mathtt{L}$ *to variables from* X, denoted $\mathtt{L}_X$, is the most general (with respect to inclusion of representation sets) sublanguage $\mathtt{L}' \subseteq \mathtt{L}$ such that $E_{\mathtt{L}'} = E_{\mathtt{L}}$ and $V_{\mathtt{L}'} = X \cap V_{\mathtt{L}}$.

The main use of this definition is to restrict languages to Boolean variables; we use the simple notation $\mathtt{L}_{\mathcal{B}}$ to represent the restriction of $\mathtt{L}$ to Boolean variables. Thus $\mathtt{L}_{\text{OBDD}} = (\mathtt{L}_{\text{MDD}})_{\mathcal{B}}$. As noted above, restricting the interpretation domain of $\mathtt{L}_{\text{MDD}}$ is sufficient for a number of representations to be removed: any representation *mentioning* a non-Boolean variable cannot be an $(\mathtt{L}_{\text{MDD}})_{\mathcal{B}}$-representation, since it would be interpreted as a function depending on a non-Boolean variable. This is

formally summarized by the following proposition.

Proposition 1.2.5. Let $\mathtt{L}$ be a representation language, and $X \subseteq \mathcal{V}$. The representation set of $\mathtt{L}_X$ is $\{ \varphi \in \mathcal{R}_\mathtt{L} \mid \text{Scope}_\mathtt{L}(\varphi) \subseteq X \}$.

Proof. Note that, from the definitions, the interpretation domain and interpretation function of $\mathtt{L}_X$ are respectively $\mathfrak{D}_{X \cap V_\mathtt{L}, E_\mathtt{L}}$ and $(\llbracket \cdot \rrbracket_\mathtt{L})_{|\mathcal{R}_{\mathtt{L}_X}}$. This notably implies that for any $\mathtt{L}_X$-representation φ, $\text{Scope}_\mathtt{L}(\varphi) \subseteq X \cap V_\mathtt{L}$. Now, let us consider an $\mathtt{L}$-representation ψ the scope of which is included in X. By definition of the scope, it is also included in $V_\mathtt{L}$, and thus $\llbracket \psi \rrbracket_\mathtt{L} \in \mathfrak{D}_{X \cap V_\mathtt{L}, E_\mathtt{L}}$. If $\psi \notin \mathcal{R}_{\mathtt{L}_X}$, then we can define another sublanguage $\mathtt{L}'$ of $\mathtt{L}$ that is equal to $\mathtt{L}_X$ except that $\mathcal{R}_{\mathtt{L}'} = \mathcal{R}_{\mathtt{L}_X} \cup \{\psi\}$; this is impossible, since $\mathtt{L}_X$ is the most general sublanguage of $\mathtt{L}$ on this interpretation domain. Hence $\psi \in \mathcal{R}_{\mathtt{L}_X}$, and the result follows. □

Using this, we can already give a simple yet useful result on sublanguages: restricting variables maintains language hierarchy. This is due to the following lemma.

Lemma 1.2.6. Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two representation languages, and $X \subseteq \mathcal{V}$ be a set of variables. For all representations $\varphi_1 \in \mathcal{R}_{\mathtt{L}_1}$ and $\varphi_2 \in \mathcal{R}_{\mathtt{L}_2}$,

$$\llbracket \varphi_1 \rrbracket_{\mathtt{L}_1} = \llbracket \varphi_2 \rrbracket_{\mathtt{L}_2} \qquad \Longrightarrow \qquad \varphi_1 \in \mathcal{R}_{\mathtt{L}_{1X}} \iff \varphi_2 \in \mathcal{R}_{\mathtt{L}_{2X}}.$$

Proof. Supposing $\llbracket \varphi_1 \rrbracket_{\mathtt{L}_1} = \llbracket \varphi_2 \rrbracket_{\mathtt{L}_2}$, we get that $\text{Scope}_{\mathtt{L}_1}(\varphi_1) = \text{Scope}_{\mathtt{L}_2}(\varphi_2)$ directly. Now, using Proposition 1.2.5, we know that the representation set of $\mathtt{L}_{1X}$ is $\{ \varphi \in \mathcal{R}_{\mathtt{L}_1} \mid \text{Scope}_{\mathtt{L}_1}(\varphi) \subseteq X \}$; we deduce from this that if $\varphi_1 \in \mathcal{R}_{\mathtt{L}_{1X}}$, then its scope is included in X. Therefore the scope of φ_2 is also included in X. Since the representation set of $\mathtt{L}_{2X}$ is $\{ \varphi \in \mathcal{R}_{\mathtt{L}_2} \mid \text{Scope}_{\mathtt{L}_2}(\varphi) \subseteq X \}$ (still using Proposition 1.2.5), this proves that φ_2 is an $\mathtt{L}_{2X}$-representation. Switching the roles of $\mathtt{L}_1$ and $\mathtt{L}_2$, we get the equivalence and hence the result. □

Proposition 1.2.7. Let $\mathtt{L}$ and $\mathtt{L}'$ be two representation languages, and $X \subseteq \mathcal{V}$ be a set of variables; it holds that $\mathtt{L} \subseteq \mathtt{L}' \implies \mathtt{L}_X \subseteq \mathtt{L}'_X$.

Proof. Since $\mathtt{L} \subseteq \mathtt{L}'$, we know that $\mathfrak{D}_{V_\mathtt{L}, E_\mathtt{L}} \subseteq \mathfrak{D}_{\mathcal{V}_{\mathtt{L}'}, E_{\mathtt{L}'}}$. Therefore $V_\mathtt{L} \subseteq V_{\mathtt{L}'}$, and $V_\mathtt{L} \cap X \subseteq V_{\mathtt{L}'} \cap X$. Thus, using the definitions, $\mathfrak{D}_{V_{\mathtt{L}_X}, E_{\mathtt{L}_X}} \subseteq \mathfrak{D}_{V_{\mathtt{L}'_X}, E_{\mathtt{L}'_X}}$.

Let φ be an $\mathtt{L}_X$-representation. It is obviously an $\mathtt{L}$-representation and also an $\mathtt{L}'$-representation; since it has the same interpretation in both of these languages, we can use Lemma 1.2.6. We obtain that $\varphi \in \mathcal{R}_{\mathtt{L}'_X}$. Hence, $\mathcal{R}_{\mathtt{L}_X} \subseteq \mathcal{R}_{\mathtt{L}'_X}$.

We now prove the equality of interpretation functions. Using definitions again,

$$\llbracket \cdot \rrbracket_{\mathtt{L}_X} = (\llbracket \cdot \rrbracket_\mathtt{L})_{|\mathcal{R}_{\mathtt{L}_X}} = \Big((\llbracket \cdot \rrbracket_{\mathtt{L}'})_{|\mathcal{R}_\mathtt{L}} \Big)_{|\mathcal{R}_{\mathtt{L}_X}} = (\llbracket \cdot \rrbracket_{\mathtt{L}'})_{|\mathcal{R}_\mathtt{L} \cap \mathcal{R}_{\mathtt{L}_X}},$$

so $\llbracket \cdot \rrbracket_{\mathtt{L}_X} = (\llbracket \cdot \rrbracket_{\mathtt{L}'})_{|\mathcal{R}_{\mathtt{L}_X}}$, since $\mathcal{R}_{\mathtt{L}_X} \subseteq \mathcal{R}_\mathtt{L}$. Also,

$$(\llbracket \cdot \rrbracket_{\mathtt{L}'_X})_{|\mathcal{R}_{\mathtt{L}_X}} = \Big((\llbracket \cdot \rrbracket_{\mathtt{L}'})_{|\mathcal{R}_{\mathtt{L}'_X}} \Big)_{|\mathcal{R}_{\mathtt{L}_X}} = (\llbracket \cdot \rrbracket_{\mathtt{L}'})_{|\mathcal{R}_{\mathtt{L}'_X} \cap \mathcal{R}_{\mathtt{L}_X}}$$

and since we proved $\mathcal{R}_{\mathtt{L}_X} \subseteq \mathcal{R}_{\mathtt{L}'_X}$, it holds that $(\llbracket \cdot \rrbracket_{\mathtt{L}'_X})_{|\mathcal{R}_{\mathtt{L}_X}} = (\llbracket \cdot \rrbracket_{\mathtt{L}'})_{|\mathcal{R}_{\mathtt{L}_X}}$.

Hence, we obtain that $[\![\cdot]\!]_{\mathtt{L}_X} = ([\![\cdot]\!]_{\mathtt{L}'_X})_{|\mathcal{R}_{\mathtt{L}_X}}$; each item of the definition of a sublanguage thus holds, and consequently $\mathtt{L}_X \subseteq \mathtt{L}'_X$. □

All in all, our definition of a sublanguage allows languages on different interpretation domains to be identified as belonging to the same "family". Yet, in the language hierarchy, the transition from a superlanguage to a sublanguage is most often a *structural* restriction. For example, OBDDs are a specific kind of BDDs, in which variables are encountered in the same order on all paths; but these two languages are defined on the same variables and values. To reflect this, we refine the notion of sublanguage.

Definition 1.2.8 (Fragment)**.** Let $\mathtt{L}$ be a representation language. A *fragment* of $\mathtt{L}$ is a sublanguage $\mathtt{L}' \subseteq \mathtt{L}$ verifying $V_{\mathtt{L}'} = V_{\mathtt{L}}$ and $E_{\mathtt{L}'} = E_{\mathtt{L}}$.

A fragment is thus a particular kind of sublanguage, having the same interpretation domain as its parent. $\mathtt{L}_{\mathrm{OBDD}}$ is a fragment of $\mathtt{L}_{\mathrm{BDD}}$, but it is not a fragment of $\mathtt{L}_{\mathrm{MDD}}$. In the literature, there is to our knowledge no distinction between sublanguage and fragment; we introduce it here to clarify later definitions. As far as we know, this notion of fragment corresponds to what is usually used. Going on with concepts related to the compilation map hierarchy, let us introduce a notation that will lighten later definitions.

Definition 1.2.9 (Operations on fragments)**.** Let $\mathtt{L}$ be a representation language, and $\mathcal{P}$ a property on $\mathtt{L}$-representations (that is, a function associating a Boolean value with every element of $\mathcal{R}_{\mathtt{L}}$).

The *restriction of* $\mathtt{L}$ *to* $\mathcal{P}$ (also called *fragment of* $\mathtt{L}$ *satisfying* $\mathcal{P}$) is the fragment $\mathtt{L}' \subseteq \mathtt{L}$ that verifies

$$\forall \varphi \in \mathcal{R}_{\mathtt{L}}, \qquad \mathcal{P}(\varphi) \iff \varphi \in \mathcal{R}_{\mathtt{L}'}.$$

Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two fragments of $\mathtt{L}$. The *intersection* (resp. *union*) of $\mathtt{L}_1$ and $\mathtt{L}_2$ is the fragment of $\mathtt{L}$ the representation set of which is exactly the intersection (resp. union) of the representation sets of $\mathtt{L}_1$ and $\mathtt{L}_2$.

1.2.3 Language Comparison

Evaluation of a language's efficiency for a given application is based on four general criteria:

1. expressivity (which functions is it able to represent?);
2. succinctness (how much space does it take to represent functions?);
3. efficiency of queries (how much time does it take to obtain information about a function?);
4. efficiency of transformations (how much time does it take to apply operations on the functions?).

The first two criteria aim at comparing space efficiency of languages, while the last two allow one to compare time efficiency. In the following, we formally define these notions on representation languages in general. In practice, however, they are mainly used to compare *fragments* or *sublanguages* of a given language.

Space Efficiency

The notion of *expressivity* was introduced in the form of a preorder on languages by Gogic et al. [GK$^+$95].

Definition 1.2.10 (Relative expressivity)**.** Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two representation languages. $\mathtt{L}_1$ is *at least as expressive* as $\mathtt{L}_2$, which is denoted $\mathtt{L}_1 \leqslant_e \mathtt{L}_2$, if and only if for each $\mathtt{L}_2$-representation φ_2, there exists an $\mathtt{L}_1$-representation φ_1 such that $[\![\varphi_1]\!]_{\mathtt{L}_1} = [\![\varphi_2]\!]_{\mathtt{L}_2}$.

Relation $\leqslant_e$ is clearly reflexive and transitive; it is thus a (partial) preorder, of which we denote by $\sim_e$ the symmetric part and by $<_e$ the asymmetric part.[3]

Obviously, if two representation languages have disjoint scope or valuation set, they are incomparable with respect to $\leqslant_e$. This explains why this criterion is considered apart from the others: users are supposed to know which expressivity they need for their application. They will thus just discard languages that do no allow them to represent the functions they need. But even if two languages are comparable with respect to $\leqslant_e$, it is unlikely that users will concern themselves with the *relative* expressivity of languages. For example, knowing that the language of Horn formulæ is strictly less expressive than the language of CNFs does not help them; what is good to know is that the former is *not complete*, whereas the latter is.

In order to define the completeness of a language, let us first introduce its (absolute) expressivity.

Definition 1.2.11 (Expressivity)**.** The *expressivity* of a representation language $\mathtt{L}$ is the set $\mathrm{Expr}(\mathtt{L}) = \{ f \in \mathfrak{D}_{V_{\mathtt{L}},E_{\mathtt{L}}} \mid \exists \varphi \in \mathcal{R}_{\mathtt{L}}, [\![\varphi]\!]_{\mathtt{L}} = f \}$ (it can be seen as the image of $\mathcal{R}_{\mathtt{L}}$ by $[\![\cdot]\!]_{\mathtt{L}}$).

We simply define the expressivity of a language as the set of functions it allows to be expressed. Note that we have in particular $\mathtt{L}_2 \geqslant_e \mathtt{L}_1 \iff \mathrm{Expr}(\mathtt{L}_2) \subseteq \mathrm{Expr}(\mathtt{L}_1)$. Completeness can now be introduced.

Definition 1.2.12 (Complete language)**.** A representation language $\mathtt{L}$ is *complete* if and only if $\mathrm{Expr}(\mathtt{L}) = \mathfrak{D}_{V_{\mathtt{L}},E_{\mathtt{L}}}$.

To be complete, a language must be able to represent any function *of its interpretation domain*. For example, the aforementioned language of BDDs, $\mathtt{L}_{\mathrm{BDD}}$, is

[3] Let $\preceq$ be a partial preorder on a set S. We define the symmetric part $\sim$ and the asymmetric part $\prec$ of $\preceq$ as:

$$\forall \langle a,b \rangle \in S^2, \quad a \sim b \iff a \preceq b \wedge a \succeq b,$$
$$\forall \langle a,b \rangle \in S^2, \quad a \prec b \iff a \preceq b \wedge a \not\succeq b.$$

complete, but the language of "BDDs having at most 3 nodes" defined on the same interpretation domain is incomplete (it is impossible to represent propositional formula $x \vee y \vee z \vee t$ with a 3-node BDD). Completeness is defined with respect to the interpretation domain; two languages that differ only by their interpretation domains have the same expressivity, but not necessarily the same completeness. Thus, when defined on interpretation domain $\mathfrak{D}_{\mathcal{V}_{\mathbb{R},\mathbb{B}}}$, the language of BDDs is incomplete. All in all, a *fragment* of an incomplete language *cannot* be complete, but a *sublanguage* can. Examples of more realistic incomplete languages include the language of propositional Horn clauses on interpretation domain $\mathfrak{D}_{\mathcal{B},\mathbb{B}}$ [§ 1.3.3.3].

Once users have identified languages being expressive enough with regard to their application, they must be able to compare these languages according to their spatial efficiency. This is the role of *succinctness*, also introduced by Gogic et al. [GK+95] (and further detailed by Darwiche and Marquis [DM02]).

Definition 1.2.13 (Succinctness)**.** Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two representation languages. $\mathtt{L}_1$ is *at least as succinct* as $\mathtt{L}_2$, which is denoted by $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$, if and only if there exists a polynomial $P(\cdot)$ such that for each $\mathtt{L}_2$-representation φ_2, there exists a $\mathtt{L}_1$-representation φ_1 such that $\|\varphi_1\| \leqslant P(\|\varphi_2\|)$ and $[\![\varphi_1]\!]_{\mathtt{L}_1} = [\![\varphi_2]\!]_{\mathtt{L}_2}$.

In the same way as $\leqslant_e$, $\leqslant_s$ is a partial preorder, of which we denote by $\sim_s$ the symmetric part and by $<_s$ the asymmetric part. Preorder $\leqslant_s$ is a refinement of $\leqslant_e$, since for all representation languages $\mathtt{L}_1$ and $\mathtt{L}_2$, it holds that $\mathtt{L}_1 \leqslant_s \mathtt{L}_2 \implies \mathtt{L}_1 \leqslant_e \mathtt{L}_2$.

It is important to notice that succinctness only requires the *existence* of a polysize equivalent, be it computable in polytime or not. An interesting sufficient condition for $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$ to hold is that there exist a *polyspace* algorithm matching any $\mathtt{L}_2$-representation to an $\mathtt{L}_1$-representation of equal interpretation. If we impose said algorithm to be polytime, which is more restrictive ($\mathsf{P} \subsetneq \mathsf{PSPACE}$ unless $\mathsf{P} = \mathsf{NP}$ [see Pap94]), we get the following relation, introduced by Fargier and Marquis [FM09].

Definition 1.2.14 (Polynomial translatability)**.** Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two representation languages. $\mathtt{L}_2$ is *polynomially translatable* into $\mathtt{L}_1$, which is denoted as $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$, if and only if there exists a polytime algorithm mapping each $\mathtt{L}_2$-representation φ_2 to an $\mathtt{L}_1$-representation φ_1 such that $[\![\varphi_1]\!]_{\mathtt{L}_1} = [\![\varphi_2]\!]_{\mathtt{L}_2}$.

Once again, $\leqslant_p$ is a partial preorder, of which we denote by $\sim_p$ the symmetric part and by $<_p$ the asymmetric part; it is a refinement of succinctness, since for all representation languages $\mathtt{L}_1$ and $\mathtt{L}_2$, it holds that $\mathtt{L}_1 \leqslant_p \mathtt{L}_2 \implies \mathtt{L}_1 \leqslant_s \mathtt{L}_2$. Obviously enough, if $\mathtt{L}_2$ is a sublanguage of $\mathtt{L}_1$, then $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$ (each $\mathtt{L}_2$-representation *is* an $\mathtt{L}_1$-representation, so the algorithm is trivial).

The following proposition sums up the relationships between the three preorders.

Proposition 1.2.15. Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two representation languages. The following implications hold: $\mathtt{L}_2 \subseteq \mathtt{L}_1 \implies \mathtt{L}_2 \geqslant_p \mathtt{L}_1 \implies \mathtt{L}_2 \geqslant_s \mathtt{L}_1 \implies \mathtt{L}_2 \geqslant_e \mathtt{L}_1$.

These comparison relations have in common that they still hold after restricting both languages to a same set of variables.

Proposition 1.2.16. Let $\mathtt{L}$ and $\mathtt{L}'$ be two representation languages, and $\leqslant$ one of the three comparison preorders we defined, $\leqslant_p$, $\leqslant_s$, or $\leqslant_e$. For any set of variables $X \subseteq \mathcal{V}$, it holds that

$$\mathtt{L} \leqslant \mathtt{L}' \quad \Longrightarrow \quad \mathtt{L}_X \leqslant \mathtt{L}'_X.$$

Proof. This is a corollary of Lemma 1.2.6. For $\mathtt{L}_X \leqslant \mathtt{L}'_X$ to hold, there must exist an algorithm verifying some given property $\mathcal{P}$ (it must be polytime for $\leqslant_p$, have a polynomial output for $\leqslant_s$, and simply exist for $\leqslant_e$), that maps every $\mathtt{L}'_X$-representation to an $\mathtt{L}_X$-representation of equal interpretation. Since $\mathtt{L} \leqslant \mathtt{L}'$, we know that there exists an algorithm A verifying $\mathcal{P}$ and mapping every element in $\mathtt{L}'$ to an $\mathtt{L}$-representation of equal interpretation. Let φ' be an $\mathtt{L}'_X$-representation, and φ the $\mathtt{L}$-representation obtained thanks to A. Since φ and φ' have the same interpretation, we can use Lemma 1.2.6 to get that φ' is an $\mathtt{L}_X$-representation. This proves the proposition. □

We now have tools to compare languages in terms of expressivity and spatial efficiency, but this does not suffice for users to be able to pick a "good" language for their application. Indeed, there are numerous operations that they may want to apply on representations, and their efficiency greatly varies from one language to another.

Operational Efficiency

The fundamental idea of the knowledge compilation map is to compare languages with respect to their ability to *support* (or *satisfy*) elementary operations, that is, their ability to allow these operations to be done in polytime. Users, after having identified the operations they need to be done online, can thus choose a language known to support these operations; they are then ensured that the complexity of their online application is polynomial in the size of the structures handled online.

The idea of classifying languages according to the elementary operations they support has been introduced by [DM02] for propositional languages. The authors distinguish two categories of operations: *queries* and *transformations*. Queries are operations that return information about the function that the compiled form represents; examples of queries in the case of a propositional language are "is the formula consistent?" or "what are the models of this formula?". Transformations are operations that return an element of the language considered; examples of transformations in the case of a propositional language are "what is the negation of this formula?" or "what is the conjunction of these two formulæ?".

We see that in both cases, queries and transformations *return something*; the distinction between the two may appear superficial at first glance. Yet, it is quite important: queries are independent of the language considered, while transformations depend on it. Indeed, again in the case of propositional languages, the output of the query "is the formula consistent?" is the same, be the formula represented

by a CNF, a DNF, an OBDD, or any other Boolean fragment. On the contrary, the output of the transformation "what is the negation of this formula?" is supposed to be in the same language the formula is represented in; that is, if our formula is a BDD, we want its negation to be a BDD, which is easy, but if our formula is a DNF, we want its negation to be a DNF, which is hard.

Let us give here a formal definition of the satisfaction of a query and of a transformation. Each operation is associated with a "request function", that gives an answer (an element of a set *Answers*, often a Boolean) to a question (that depends on parameters, represented by elements of a set *Params*) about some representations of a language. For example, for the "model checking" query, *Params* is the set of all possible assignments, and *Answers* $= \mathbb{B}$.

Definition 1.2.17 (Query and transformation)**.** Let $\mathfrak{D}$ be some interpretation domain, $n \in \mathbb{N}^*$, *Params* and *Answers* some sets, and $f\colon \mathfrak{D}^n \times \mathit{Params} \to \mathit{Answers}$. We say that f is a *request function*. Let $\mathtt{L}$ be a representation language of interpretation domain included in $\mathfrak{D}$.

$\mathtt{L}$ is said to *satisfy* or *support* the *query* **Q** associated with f if and only if there exists a polynomial P and an algorithm mapping every n-tuple of $\mathtt{L}$-representations $\langle \varphi_1, \ldots, \varphi_n \rangle$ and every element $\pi \in \mathit{Params}$ to an element $\alpha \in \mathit{Answers}$ such that $\alpha = f(\llbracket \varphi_1 \rrbracket_{\mathtt{L}}, \ldots, \llbracket \varphi_n \rrbracket_{\mathtt{L}}, \pi)$ in time bounded by $P(\|\varphi_1\|, \ldots, \|\varphi_n\|, \|\pi\|, \|\alpha\|)$.

If *Answers* $= \mathfrak{D}$, $\mathtt{L}$ is said to *satisfy* or *support* the *transformation* **T** associated with f if and only if there exists a polynomial P and an algorithm mapping every n-tuple of $\mathtt{L}$-representations $\langle \varphi_1, \ldots, \varphi_n \rangle$ and every element $\pi \in \mathit{Params}$ to an $\mathtt{L}$-representation ψ such that $\llbracket \psi \rrbracket_{\mathtt{L}} = f(\llbracket \varphi_1 \rrbracket_{\mathtt{L}}, \ldots, \llbracket \varphi_n \rrbracket_{\mathtt{L}}, \pi)$ in time bounded by $P(\|\varphi_1\|, \ldots, \|\varphi_n\|, \|\pi\|)$.

For the sake of illustration, we give a formal definition of the "model checking" query, using this formalism. Let $\mathtt{L}$ be a representation language on interpretation domain $\mathfrak{D}_{V,\mathbb{B}}$; let *Params* be the set of V-assignments; the "model checking" query on $\mathtt{L}$ is hence the query associated with the request function

$$f\colon \begin{array}{ccc} \mathfrak{D}_{V,\mathbb{B}} \times \mathit{Params} & \to & \mathbb{B} \\ \langle I, \vec{v} \rangle & \mapsto & I(\vec{v}) \end{array}.$$

It is important to notice that for a language to support a query, the algorithm needs not have an output of size polynomial in the input, since the bounding polynomial depends on the sizes of both the input and the output. This way, queries the output of which is exponential in the input (for example, enumerating the models of a propositional formula) can actually be satisfied. But regarding transformations, the size of the output compiled form is necessarily polynomial in the size of the input (since any polytime algorithm is also polyspace: $\mathsf{P} \subseteq \mathsf{PSPACE}$ [see Pap94]).

Despite having only a general definition of a query and a transformation, we can already prove the following simple properties.

Proposition 1.2.18. Let $\mathfrak{D}$ be some interpretation domain, and let $\mathtt{L}_1$ and $\mathtt{L}_2$ be two representation languages of interpretation domains included in $\mathfrak{D}$.

- If $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$, then queries supported by $\mathtt{L}_1$ are also supported by $\mathtt{L}_2$.
- If $\mathtt{L}_1 \sim_p \mathtt{L}_2$, then $\mathtt{L}_1$ and $\mathtt{L}_2$ support the same queries and transformations.

The first item of this proposition illustrates the distinction between queries and transformations: $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$ is sufficient for $\mathtt{L}_2$ to support the *queries* supported by $\mathtt{L}_1$, because in order to obtain the answer to a query **Q** on a $\mathtt{L}_2$-representation φ_2, we only have to compute a $\mathtt{L}_1$-representation φ_1 of $[\![\varphi_2]\!]$ (which is done in polytime), and ask the query on φ_1 (which is also polytime since $\mathtt{L}_1$ satisfies **Q**). But we cannot use the same procedure with *transformations*: supposing that $\mathtt{L}_1$ supports a transformation **T** and that $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$, we can obtain in polytime an $\mathtt{L}_1$-representation of the answer of the request for a $\mathtt{L}_2$-representation, but the transformation imposes that the result be an $\mathtt{L}_2$-representation, and we do not know whether we can obtain such a representation in polytime. If we also suppose that $\mathtt{L}_1 \geqslant_p \mathtt{L}_2$, then it is the case: we can translate the result from $\mathtt{L}_1$ back to $\mathtt{L}_2$ in polytime, and therefore $\mathtt{L}_2$ supports **T**—this explains the second item of the proposition.

Of course, the elementary requests differ regarding the kind of function is represented by the target language considered. For example, *consistency* is a property of Boolean functions, *arithmetical sum* is an operation on arithmetic functions. At first glance, it does not seem to make sense to ask whether an ADD [§ 1.1.2] is consistent, or to make the arithmetical sum of two OBDDs. It is certainly possible to generalize, defining queries and transformations that hold on any language; for example, the `VNNF` framework [FM07] contains general queries and transformations that actually capture the more specific "consistency" query and "arithmetical sum" transformation. However, since the present work details the knowledge compilation map only for Boolean languages, we refrain from establishing a complete list of general-range queries and transformations.

The general notions pertaining to the knowledge compilation map are now defined, and apply to languages representing a wide range of functions. In the following section, we define a number of state-of-the-art *Boolean languages*.

1.3 Boolean Languages

We call Boolean languages, the representation languages $\mathtt{L}$ such that $E_{\mathtt{L}} = \mathbb{B}$ (but without restriction on the variables): the interpretation domain of Boolean languages is actually the space of Boolean functions. The study of Boolean languages is the most developed area of knowledge compilation; a possible explanation is that it is a simple yet powerful, historically important framework. Note that Boolean functions can also be seen as *sets of assignments*; this notably allows them to encompass constraint networks [§ 1.6.1], understood as the set of their solutions.

In this section, we define a number of Boolean languages that can be found in the literature, and for which some knowledge compilation results are known. We are not concerned with the higher-level, very succinct logics, like those listed by

Gogic et al. [GK+95]; we limit ourselves to a particular (although quite general) data structure, namely rooted, directed acyclic graphs.

1.3.1 General Rooted DAG Languages

Many representations of Boolean functions have the form of rooted, directed acyclic graphs (DAGs), and what is more, use the same general mechanisms. We define here the general language of rooted DAGs, which we call `GRDAG`, encompassing all other Boolean languages we are going to define.

Let us first formally define the notion of directed graph, and its associated concepts [see e.g. Gib85].

Definition 1.3.1 (Directed graph)**.** A *directed graph* Γ is a couple $\langle \mathscr{N}_\Gamma, \mathscr{E}_\Gamma \rangle$, where $\mathscr{N}_\Gamma$ is a finite set whose elements are called *nodes* (or *vertices*), and $\mathscr{E}_\Gamma \subseteq \mathscr{N}_\Gamma^2$ is a set of node couples, called *edges* (or *arcs*).

Let $E = \langle N_1, N_2 \rangle \in \mathscr{E}_\Gamma$; N_1 is said to be the *source* of edge E, and N_2 its *destination*; N_2 is called a *child* of N_1, and N_1 a *parent* of N_2. The *outgoing edges* of a node N are the edges the source of which is N; the *incoming edges* of a node N are the edges the destination of which is N.

Let $N \in \mathscr{N}_\Gamma$; N is called a *root of* Γ if and only if it has no parent. N is called a *leaf of* Γ if and only if it has no child.

We denote as $\mathrm{Src}(E)$ (resp. $\mathrm{Dest}(E)$) the source (resp. destination) of an edge E, as $\mathrm{Out}(N)$ (resp. $\mathrm{In}(N)$) the set of outgoing (resp. incoming) edges of a node N, and as $\mathrm{Ch}(N)$ (resp. $\mathrm{Pa}(N)$) the set of children (resp. parents) of a node N. Note that definition does not prevent a graph from being *empty*, that is, having no node or edge at all. Note also that it prevents graphs from being infinite.

In knowledge compilation, graphs are almost always *labeled*, that is, nodes and/or edges are associated with some mathematical object, e.g., a variable or a value. With our definition of a directed graph, there can be at most one edge between two given nodes; because of the labeling, we sometimes need to use directed *multigraphs* rather than graphs. Directed multigraphs are defined in the same way as directed graphs, with the difference that $\mathscr{E}_\Gamma$ is a *finite multiset* of node couples—allowing us to define several edges, with possibly different labels, between two given nodes.

Now, to introduce DAGs, we need the following classical notions of *path* and *cycle*.

Definition 1.3.2 (Path and cycle)**.** Let Γ be a directed graph (or multigraph), and N and N' two nodes in Γ. A *path from N to N'* is a (possibly empty, not necessarily finite) sequence of edges $p = \langle E_1, E_2, \ldots, E_k \rangle \in \mathscr{E}_\Gamma^k$ such that

- $\mathrm{Src}(E_1) = N$ and $\mathrm{Dest}(E_k) = N'$;
- $\forall i \in \{1, \ldots, k-1\}, \mathrm{Dest}(E_i) = \mathrm{Src}(E_{i+1})$.

If there exists a path from N to N', N is called an *ancestor* of N', and N' a *descendant* of N.

A *cycle* in Γ is a *non-empty* path from a node $N \in \mathscr{N}_\Gamma$ to itself.

We can now define DAGs and rooted DAGs.

Definition 1.3.3 (DAG)**.** A *directed acyclic graph (DAG)* is a directed graph without any cycle.

It is said to be *rooted* if and only if it is empty or has exactly one root.

Note the importance of the "non-empty path" condition on cycles: without it, all directed graphs would trivially be DAGs. Remark also that non-empty rooted DAGs have at least one leaf—which can also be the root.

We can now proceed with our definition of a general graph-based Boolean language. This generalization is inspired from the work of Fargier and Marquis [FM07]. [4]

Definition 1.3.4 (GRDAG)**.** A *general rooted DAG* (GRDAG) is a non-empty rooted, directed acyclic graph meeting the following requirements:

- each leaf is labeled with a Boolean constant $\top$ or $\bot$ (the leaf is a *constant*), or by a couple $\langle x, A\rangle$, with x a variable and A a set (the leaf is a *literal*);
- each non-leaf (internal) node is either labeled with a binary operator on $\mathbb{B}$ and has two (ordered) children, or is labeled with the unary operator $\neg$ or a quantifier $\exists x$ or $\forall x$ (with $x \in \mathcal{V}$), and has exactly one child.

Figure 1.5 shows an example of a GRDAG. Note that a node can be labeled with any binary operator, not only "$\vee$" or "$\wedge$", but also "$\oplus$" (exclusive or), "$\rightarrow$" (implication), negation of conjunction or disjunction, etc. In particular, label operators do not have to be commutative or associative. The order of the children has to be recorded, as can be seen on the example (implication node). We denote by $\mathbb{B}^{\mathbb{B}^2}$ the set of all binary operators on $\mathbb{B}$, and by Ops the set $\mathbb{B}^{\mathbb{B}^2} \cup \{\neg, \exists, \forall\}$.

In order to define the size function of GRDAGs, we begin by defining the size of a node N: $\|N\| = 1$ in all cases, unless it is a literal $\langle x, A\rangle$, in which case $\|N\| = \|A\|$ (which depends on the way A is represented). Now, for any GRDAG φ, denoting $\mathscr{N}_\varphi$ the set of its nodes, $\|\varphi\| = \sum_{N\in\mathscr{N}_\varphi}\|N\| + \sum_{x\in\text{Scope}(\varphi)}\|\text{Dom}(x)\|$. We have to incorporate the size of the variables' domains, since we cannot consider that the structure just *mentions* variables, without having any information about the variables themselves. This would indeed compromise the polynomiality of some important requests, such as negation. This is however generally harmless—we seldom consider "complicated" domains, but rather domains representable by an interval, specified by its two bounds.

[4] The `VNNF` language they define is much more general, since it encompasses various frameworks, some of which not being in the scope of knowledge compilation. However, our `GRDAG` language is not strictly speaking included in `VNNF`, since we slightly generalize some of its elements [see also § 1.5.4].

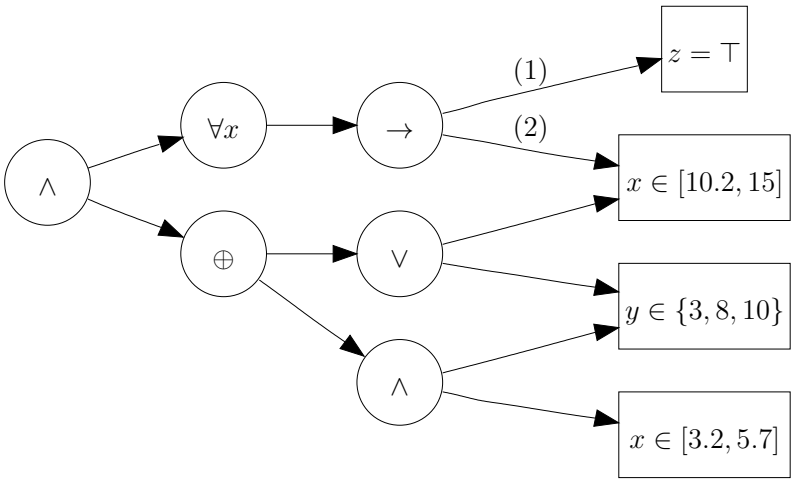

Figure 1.5: An example of a GRDAG. It holds on the following variables: $x \in \mathcal{V}_{\mathbb{R}}$, $y \in \mathcal{V}_{\mathbb{N}}$, and $z \in \mathcal{B}$.

Definition 1.3.5 (Semantics of a GRDAG)**.** The interpretation and scope of a GRDAG φ are inductively defined as follows.

- If the root of φ is a leaf labeled with $\bot$ (resp. $\top$), then $\text{Scope}(\varphi) = \varnothing$, and $[\![\varphi]\!]$ is the constant function always returning $\bot$ (resp. $\top$).
- If the root of φ is a leaf labeled with $\langle x, A\rangle$, then $\text{Scope}(\varphi) = \{x\}$, and $[\![\varphi]\!]$ is the function $\text{Dom}(x) \to \mathbb{B}$ such that for any $\{x\}$-assignment $\vec{x}$, $[\![\varphi]\!](\vec{x}) = \top$ if and only if $\vec{x} \in A$ holds.
- If the root of φ is an internal node labeled with a binary operator $\otimes$, then (denoting N_l and N_r its children, and φ_l (resp. φ_r) the GRDAG rooted at N_l (resp. N_r)) $\text{Scope}(\varphi) = \text{Scope}(\varphi_l) \cup \text{Scope}(\varphi_r)$, and $[\![\varphi]\!]$ is the function $\text{Dom}(\text{Scope}(\varphi)) \to \mathbb{B}$ such that for any $\text{Scope}(\varphi)$-assignment $\vec{v}$, $[\![\varphi]\!](\vec{v}) = [\![\varphi_l]\!](\vec{v}) \otimes [\![\varphi_r]\!](\vec{v})$.
- If the root of φ is an internal node labeled with unary operator $\neg$, then (denoting N_c its child, and φ_c the GRDAG rooted at N_c) $\text{Scope}(\varphi) = \text{Scope}(\varphi_c)$, and $[\![\varphi]\!]$ is the function $\text{Dom}(\text{Scope}(\varphi)) \to \mathbb{B}$ such that for any $\text{Scope}(\varphi)$-assignment $\vec{v}$, $[\![\varphi]\!](\vec{v}) = \neg[\![\varphi_c]\!](\vec{v})$.
- If the root of φ is an internal node labeled with quantification $\forall x$ (resp. $\exists x$), then (denoting N_c its child, and φ_c the GRDAG rooted at N_c) $\text{Scope}(\varphi) = \text{Scope}(\varphi_c) \setminus \{x\}$, and $[\![\varphi]\!]$ is the function $\text{Dom}(\text{Scope}(\varphi)) \to \mathbb{B}$ such that for any $\text{Scope}(\varphi)$-assignment $\vec{v}$, $[\![\varphi]\!](\vec{v}) = \top$ if and only if for all $\vec{x} \in \text{Dom}(x)$, $[\![\varphi_c]\!](\vec{v} \ . \ \vec{x}) = \top$ (resp. there exists an $\vec{x} \in \text{Dom}(x)$ such that $[\![\varphi_c]\!](\vec{v} \ . \ \vec{x}) = \top$).

It is important to notice that GRDAGs are very close to classical logic formulæ; the difference is that instead of holding on Boolean variables, they hold on "membership conditions". This echoes the satisfiability modulo theories (SMT) problem, which consists in checking the consistency of a logic formula, with some Boolean variables being replaced by predicates of given first-order theories [see e.g. Seb07]. A second difference between GRDAGS and classical logic formulæ is that GRDAGS are not represented as sequences of symbols, but as graphs, which avoids repetition of common subformulæ, in the same way as the merging of isomorphic subgraphs in BDDs [§ 1.1.2.3]. Despite these differences, we will sometimes call GRDAGs formulæ or sentences.

Using these various elements, we can now define the GRDAG representation language.

Definition 1.3.6. GRDAG is the representation language $\langle \mathfrak{D}_{\mathcal{V},\mathbb{B}}, \mathcal{R}, [\![\cdot]\!] \rangle$, with $\mathcal{R}$ the set of all GRDAGs and $[\![\cdot]\!]$ the interpretation function given in Definition 1.3.5.

Note the font difference between "GRDAG " (which denotes the representation language) and "GRDAG" (which denotes a GRDAG-representation). In the following, we always use a specific font to distinguish a language from a structure, as it is done in the original knowledge compilation map [DM02].

We further define three useful parameters. The *height* of a GRDAG φ, denoted $\mathrm{h}(\varphi)$, is the number of edges in the longest path from the root to any leaf of φ. The *label set* of φ, denoted $\mathrm{Labels}(\varphi)$, is the set of all sets with which some literal is labeled in φ. The *operator set* of φ, denoted $\mathrm{Ops}(\varphi)$, is the set defined as follows:

- $\mathrm{Ops}(\varphi) \subseteq \mathrm{Ops}$;
- $\mathrm{Ops}(\varphi)$ contains $\otimes \in \mathbb{B}^{\mathbb{B}^2} \cup \{\neg\}$ if and only if at least one node in φ is labeled with $\otimes$;
- $\mathrm{Ops}(\varphi)$ contains $\exists$ (resp. $\forall$) if and only if at least one node in φ is labeled with $\exists x$ (resp. $\forall x$), with $x \in \mathcal{V}$.

Some simplifications are used, in order to improve clarity or lighten notations. First of all, no distinction is generally made between a node N and the graph φ rooted at N; thus $\mathrm{Scope}(N)$ simply corresponds to $\mathrm{Scope}(\varphi)$ (this simplification occurs for all DAGs, not only GRDAGs). We also usually do not distinguish, in the text, between a graph and its interpretation: we for example call "literal" a graph consisting only of a literal leaf. Referring to the fact that GRDAGs can be seen as Boolean formulæ, we will sometimes use the formulation "an $\big[x \in A\big] \wedge \big[y \in B\big]$ node" to talk about a $\wedge$-node having two literals as children, respectively labeled with $\langle x, A \rangle$ and $\langle y, B \rangle$. Another very common simplification is to allow nodes labeled with operators that are associative and commutative to have arbitrarily many children; this is considered harmless since it only changes the characteristic size logarithmically.

A first result on GRDAG is that it is an incomplete language. This is due to the limitation of literals to membership constraints.

Proposition 1.3.7. GRDAG is incomplete.

This can be seen for example on a function $f\colon \mathbb{R}^2 \to \mathbb{B}$ defined by $f\colon \langle x, y\rangle \mapsto \big[x = y\big]$. f can be represented by the formula $f = \bigvee_{r\in\mathbb{R}}(\big[x \in \{r\}\big] \wedge \big[y \in \{r\}\big])$, but this formula cannot be expressed as a GRDAG, since we consider all graphs to be finite. This incompleteness does not imply that all sublanguages of GRDAG are incomplete: as mentioned earlier, completeness depends on the interpretation domain. For instance, $\texttt{GRDAG}_{\mathcal{B}}$ is complete.

The remainder of this section is dedicated to the presentation of state-of-the-art compilation languages. However, the way we define them makes them more general than they are in the literature, since we allow non-Boolean, non-enumerated variables. We will specify in each case on which category of variables the language was originally defined. A consequence of this generalization is that GRDAGs allow literals representing $\big[x \in A\big]$, with A not being a singleton, and not being included in $\mathrm{Dom}(x)$. This will be useful in further chapters of this thesis [Part II], but in the literature all languages only allow singleton, domain-included literals. We thus need to introduce a restriction of GRDAGs.

Definition 1.3.8 (Restricted literal expressivity)**.** Let $\mathcal{A}$ be a set of sets and φ a GRDAG. We say that *the literal expressivity of φ is restricted to $\mathcal{A}$* if and only if $\mathrm{Labels}(\varphi) \subseteq \mathcal{A}$. We denote as $\texttt{GRDAG}^{\mathcal{A}}$ the sublanguage of GRDAG with literal expressivity restricted to $\mathcal{A}$.

The main restriction of literal expressivity that we will need to characterize languages from the literature, is the restriction to literals of the form $\langle x, \top\rangle$ and $\langle x, \bot\rangle$, excluding those of the form $\langle x, \varnothing\rangle$ and $\langle x, \{\bot, \top\}\rangle$. We will thus often consider fragments of $\texttt{GRDAG}_{\mathcal{B}}^{\mathbb{SB}}$, the set $\mathbb{SB}$ being the set of singletons of $\mathbb{B}$. Sometimes fragments of $\texttt{GRDAG}_{\mathcal{E}}^{\mathbb{SZ}}$ will be encountered, $\mathcal{E}$ being the set of enumerated variables (of finite integer interval domain) and $\mathbb{SZ}$ the set of singletons of $\mathbb{Z}$.

1.3.2 Fragments Based on Operator Restrictions

We obtain various sublanguages of GRDAG by restricting the operators with which nodes can be labeled.

Restricting to Basic Operators

We begin with a fragment of great importance in knowledge compilation, since it is the original "root" of the propositional compilation map [DM02]. Taking its name from the negation normal form sentences of propositional logic [see e.g. Bar77], it has been introduced as a compilation language by Darwiche [Dar01a].

Definition 1.3.9 (NNF)**.** A GRDAG φ is in *negation normal form (NNF)* if and only if $\mathrm{Ops}(\varphi) \subseteq \{\wedge, \vee\}$.

NNF is the restriction of GRDAG to NNF formulæ.

Using this definition, the name "negation normal form formula" seems weird, since we removed negation from the operators authorized as labels for internal nodes. In

its original propositional definition (which corresponds to $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$ here), this name refers to the fact that all negations are on literals. As we consider NNFs on non-Boolean variables, with leaves of the form $\big[x \in A\big]$, there is no reference to negation anymore (the propositional literal $\neg x$ is represented by a literal $\langle x, \{\bot\}\rangle$ in a GRDAG); we nevertheless keep the name for the sake of clarity.

Adding Negation

The fragment PDAG has been defined by Wachter and Haenni [WH06] as an extension of NNF, adding the possibility to use $\neg$-nodes.

Definition 1.3.10 (PDAG)**.** A GRDAG φ is called a *propositional DAG (PDAG)*, if and only if $\mathrm{Ops}(\varphi) \subseteq \{\wedge, \vee, \neg\}$.

PDAG is the restriction of GRDAG to PDAGs.

In their original sense, PDAGs are actually $\mathtt{PDAG}_{\mathcal{B}}^{\$\mathbb{B}}$-representations, and moreover only contain positive literals; they have been modified to include negative literals by Fargier and Marquis [FM08a], the modification being harmless and making PDAGs definition-wise closer to NNFs. It is indeed trivial that $\mathtt{NNF} \subseteq \mathtt{PDAG}$.

Adding Quantifiers

The following language embraces most usual languages in the knowledge compilation framework; it was introduced [FM08a] to take advantage of closure principles [§ 1.3.7].

Definition 1.3.11 (*Q*PDAG)**.** A GRDAG φ is called a *quantified propositional DAG (QPDAG)* if and only if $\mathrm{Ops}(\varphi) \subseteq \{\wedge, \vee, \neg, \exists, \forall\}$.

*Q*PDAG is the restriction of GRDAG to QPDAGs.

The original *Q*PDAG language [FM08a] corresponds to $\mathit{Q}\mathtt{PDAG}_{\mathcal{B}}^{\$\mathbb{B}}$ here.

1.3.3 "Historical" Fragments

In this section, we recover as fragments of NNF some well-known languages that were used before knowledge compilation had been formalized. They have been linked to the map by Darwiche and Marquis [DM02] and Fargier and Marquis [FM08b].

Flattening NNF Languages

Let us begin by generalizing important elements of propositional logic, viz., clauses and terms, to our GRDAG framework.

Definition 1.3.12 (Clauses and terms)**.** A node in a GRDAG is a *clause* (resp. a *term*) if and only if it is a constant, a literal, or an internal node labeled with $\vee$ (resp. by $\wedge$) whose children are literals. We accordingly call clause (resp. a term) a GRDAG whose root is a clause (resp. term).

We define term as the restriction of GRDAG to terms, and clause as the restriction of GRDAG to clauses.

The following definition [DM02] introduces the first *operationally efficient* languages of this map.

Definition 1.3.13 (Flatness and simple-junction)**.** An NNF sentence φ is *flat* if and only if $\mathrm{h}(\varphi) \leqslant 2$.

An NNF sentence satisfies the *simple-disjunction* (resp. *simple-conjunction*) property if and only if each $\vee$-node (resp. each $\wedge$-node) is a clause (resp. a term) and its children share no variables.

`f-NNF` is the fragment of `NNF` satisfying flatness; `CNF` and `DNF` are the fragments of `f-NNF` respectively satisfying simple-disjunction and simple-conjunction.

`CNF` and `DNF` are named after the form of their representations: indeed, $\mathtt{CNF}_{\mathcal{B}}^{\mathbb{SB}}$ and $\mathtt{DNF}_{\mathcal{B}}^{\mathbb{SB}}$-representations are the well-known conjunctive normal form (CNF) and disjunctive normal form (DNF) formulæ.

Prime Implication

We proceed with other well-known subsets of CNFs and DNFs, for which we keep their original meaning of propositional languages.

Definition 1.3.14 (Prime implicates and `PI`)**.** A CNF formula φ is a *prime implicate* if and only if:

- for every clause δ_1 such that $\varphi \models \delta_1$, there exists a clause δ_2 in φ verifying $\delta_2 \models \delta_1$;
- φ contains no couple $\langle \delta_1, \delta_2 \rangle$ of clauses such that $\delta_1 \models \delta_2$.

`PI` is the restriction of $\mathtt{CNF}_{\mathcal{B}}^{\mathbb{SB}}$ to prime implicates.

Definition 1.3.15 (Prime implicants and `IP`)**.** A DNF formula φ is a *prime implicant* if and only if:

- for every term γ_1 such that $\gamma_1 \models \varphi$, there exists a term γ_2 in φ verifying $\gamma_1 \models \gamma_2$;
- φ contains no couple $\langle \gamma_1, \gamma_2 \rangle$ of terms such that $\gamma_1 \models \gamma_2$.

`IP` is the restriction of $\mathtt{DNF}_{\mathcal{B}}^{\mathbb{SB}}$ to prime implicants.

Krom and Horn

To help introduce the following fragments, which once again keep their original Boolean meaning, let us define a *positive literal* as a literal of the form $\left[x = \top\right]$, and a *negative literal* as a literal of the form $\left[x = \bot\right]$. The *complement* of a literal $\left[x = \top\right]$ (resp. $\left[x = \bot\right]$) is $\left[x = \bot\right]$ (resp. $\left[x = \top\right]$). We now define the classic Krom, Horn, and renamable Horn formulæ, and follow Fargier and Marquis [FM08b] to include them in the compilation map.

Definition 1.3.16 (Krom, Horn)**.** A *Krom clause* is a clause having at most two literals. A *Horn clause* is a propositional clause having at most one positive literal.

A CNF sentence whose all clauses are Krom clauses (resp. Horn clauses) is called a Krom-CNF formula (resp. a Horn-CNF formula).

The languages `KROM-C` and `HORN-C` are the restrictions of $\mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ to Krom-CNFs and Horn-CNFs, respectively. The `K/H-C` language is the union of `KROM-C` and `HORN-C`.

Definition 1.3.17 (Horn-renamability)**.** A propositional CNF φ is *Horn-renamable* if and only if there exists a subset $V \subseteq \text{Scope}(\varphi)$ (called a *Horn renaming for* φ) such that if we substitute in φ every literal l such that $\text{Scope}(l) \in V$ by its complement, we obtain a Horn-CNF formula.

`renH-C` is the fragment of $\mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ satisfying Horn-renamability.

1.3.4 Fragments Based on Node Properties

Following the classical presentation of the propositional knowledge compilation map, this section groups together two fundamental properties on $\wedge$- and $\vee$-nodes.

Darwiche [Dar98] introduced decomposable NNF formulas for characterizing consistency-based diagnoses. This turned out to be a quite influential fragment, occupying an advantageous place in the knowledge map (more succinct than classical languages, yet supporting fundamental operations in polytime). Decomposability was first defined as a property of $\wedge$-nodes, but Fargier and Marquis [FM06] applied the property to $\vee$-nodes, and finally extended the definition to any node (renaming it "simple decomposability") [FM07].

Definition 1.3.18 (Decomposability)**.** A node N of a GRDAG is *decomposable* if and only if its children do not share any variable, that is, for each couple $\langle N_1, N_2 \rangle$ of distinct children of N, $\text{Scope}(N_1) \cap \text{Scope}(N_2) = \varnothing$.

A GRDAG is *decomposable* if and only if all of its $\wedge$-nodes are decomposable.

`DNNF` is the fragment of `NNF` satisfying decomposability.

Note that in a decomposable *graph*, only $\wedge$-nodes must be decomposable. Contrary to decomposability, determinism [Dar01b] is only defined on $\vee$-nodes.

Definition 1.3.19 (Determinism)**.** A node N of a GRDAG is *deterministic* if and only if it is labeled with $\vee$ and its children are pairwise logically contradictory, that is, for each couple $\langle N_1, N_2 \rangle$ of distinct children of N, $[\![N_1]\!] \wedge [\![N_2]\!] \models \bot$.

A GRDAG is *deterministic* if and only if all of its $\vee$-nodes are deterministic.

`d-NNF` is the fragment of `NNF` satisfying determinism; `d-DNNF` is the intersection of `d-NNF` and `DNNF`.

1.3.5 The Decision Graph Family

The previous section defined restrictive properties holding on single nodes; we now present properties that involve several nodes at a time. We recover here in particular

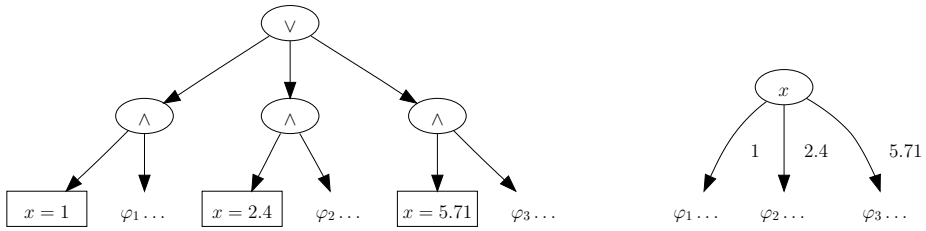

Figure 1.6: A GRDAG decision node (left) and its simplified representation (right). On the GRDAG version, the root $\vee$-node is a decision node, and the three $\wedge$-nodes are assignment nodes. In the simplified representation, the decision node corresponds to the variable-labeled node, and each assignment node corresponds to a labeled edge.

the famous BDDs [§ 1.1.2] as NNF sentences. Following Fargier and Marquis [FM06], we present the decision graph family using "cascading" definitions: we start by defining the notion of *assignment nodes*, then *decision nodes* are defined using the latter notion, and we can then present various languages depending on the properties of the nodes—for example, the language of graphs in which all decision nodes are exclusive, that of graphs in which all $\wedge$-nodes are assignment nodes, etc.

First, let us present assignment nodes [FM06, FM07].

Definition 1.3.20 (Assignment node)**.** A node N of a GRDAG is an *assignment node* if and only if it is labeled with $\wedge$ and has exactly two children, exactly one being a literal.

An assignment node N thus has the form $\big[x \in A\big] \wedge \alpha$; we say that N is an assignment node on x, and denote variable x as $\mathrm{Var}_{\mathrm{ass}}(N)$ and subgraph α as $\mathrm{Tail}_{\mathrm{ass}}(N)$. The name "assignment node" comes from the original Boolean definition; here, such nodes do not really assign values to their variables, but only restrict their possible values. We nonetheless keep the original name, similarly to what is done in the VNNF framework [FM07].

We can now give the definition of a decision node, still following Fargier and Marquis [FM07], who built up on the work of Darwiche and Marquis [DM02].

Definition 1.3.21 (Decision node)**.** A node N of a GRDAG is a *decision node* if and only if it is labeled with $\vee$ and all of its children are assignment nodes on the same variable x.

A decision node N thus has the form $(\big[x \in A_1\big] \wedge \alpha_1) \vee \dots \vee (\big[x \in A_k\big] \wedge \alpha_k)$; we say that N is a decision node on x, and use $\mathrm{Var}_{\mathrm{dec}}(N)$ to denote variable x. Figure 1.6 shows the correspondence between GRDAG decision nodes as we just defined, and decision nodes as they are classically represented, in the context of decision diagrams. The two representations are equivalent; we will refer to them as "GRDAG version" and "decision diagram version" respectively.

We define *exclusive* decision nodes by applying a restriction to the children of decision nodes.

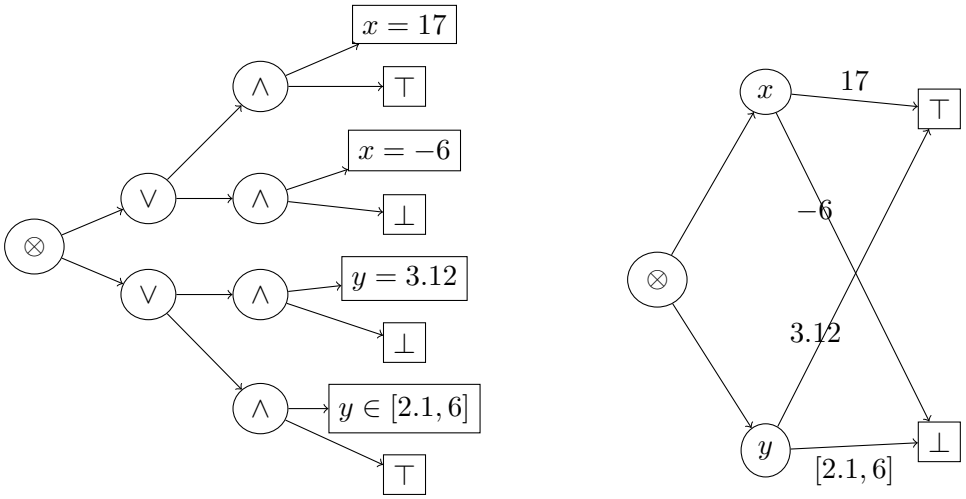

Figure 1.7: On the left, a GRDAG satisfying weak decision; the $\otimes$ label is a placeholder for either $\vee$ or $\wedge$—in both cases, weak decision is satisfied. On the right, the "decision diagram version" of this GRDAG.

Definition 1.3.22 (Exclusiveness)**.** Two assignment nodes on a same variable x in a GRDAG are *exclusive* if and only if their respective literals are disjoint, that is, denoting $\langle x, A_1 \rangle$ and $\langle x, A_2 \rangle$ the two literals, $A_1 \cap A_2 = \varnothing$.

A decision node is *exclusive* if and only if its children are pairwise exclusive assignment nodes.

A GRDAG φ satisfies the *exclusive decision* property if and only if all of its decision nodes are exclusive.

If a graph satisfies the exclusive decision property, it is guaranteed that the branches of each decision node be mutually disjoint. In the decision diagram version (using variable-labeled nodes and labeled edges), this implies that there can be at most one path compatible with a given assignment: no choice is possible at a decision node. In particular, GRDAGs satisfying the exclusive decision property are deterministic—but the reverse is not true.

The following properties, holding on the graphs' structure, finally allow us to define languages.

Definition 1.3.23 (Decision properties)**.** Let φ be a GRDAG.

- φ satisfies the *weak decision* property if and only if:
  - each literal in φ has at least one parent, and all its parents are assignment nodes, and
  - each assignment node in φ has at least one parent, and all its parents are decision nodes.

- φ satisfies the $\vee$-*simple decision* property if and only if it satisfies weak decision, and all of its $\vee$-nodes are decision nodes.
- φ satisfies the $\wedge$-*simple decision* property if and only if it satisfies weak decision, and all of its $\wedge$-nodes are assignment nodes.
- φ satisfies the *(strong) decision* property if and only if it satisfies both the $\vee$- and $\wedge$-simple decision.

This definition comes from the work of Fargier and Marquis [FM06], with a few modifications. In particular, we refined the "simple decision" property into three.

Weak decision is simply the condition for a GRDAG to belong to the "decision family". Indeed, it ensures that no literal or assignment node is "free", since such elements have no representation in the "decision diagram version". For example, the GRDAG $\left[x = 1\right] \vee \left[y = 3\right]$ does not satisfy weak decision, because literals are not children of assignment nodes. On the contrary, the GRDAG in Figure 1.7 satisfies weak decision.

In addition to this requirement, $\vee$-simple decision (resp. $\wedge$-simple decision) ensures that there is no "free" $\vee$-node (resp. $\wedge$-node). For example, going back to the GRDAG in Figure 1.7, if the $\otimes$ operator is a $\wedge$, then the GRDAG satisfies $\vee$-simple decision: all $\vee$-nodes are decision nodes, but there can be $\wedge$-nodes linking these decision nodes. The decision diagram version of these "free" $\wedge$-nodes are simply pure conjunctive nodes. Similarly, if the $\otimes$ operator is a $\vee$, then the GRDAG satisfies $\wedge$-simple decision: there can be pure disjunctive nodes in the decision diagram version, but no pure conjunctive node.

The strongest requirement is to forbid both pure disjunctive nodes and pure conjunctive nodes, that is, to satisfy $\wedge$-simple and $\vee$-simple decision altogether: the decision diagram version of the graph then contains only variable-labeled nodes. We call this property "strong decision" to emphasize its meaning; in the original work [FM06], this property was simply referred to as the "decision property".

We can now define various decision graph languages, still following this paper.

Definition 1.3.24. A *decision graph (DG)* is an NNF sentence satisfying $\vee$-simple and exclusive decision. `DG` is the restriction of `NNF` to decision graphs.

`DDG` is the fragment of `DG` satisfying decomposability.

A *basic decision diagram (BDD)* is an NNF sentence satisfying strong and exclusive decision. `BDD` is the restriction of `NNF` to BDDs.

A *free(-ordered) BDD (FBDD)* is a BDD satisfying decomposability. `FBDD` is the restriction of `NNF` to FBDDs.

Note here that, as with `NNF`, we used the names of Boolean fragments for languages that are not necessarily Boolean. Hence, using the previous definition, a BDD can here hold on non-Boolean variables: the classical `BDD` language actually corresponds to $\texttt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$. For that reason, we changed the meaning of the initials BDD from binary decision diagram to basic decision diagram, since our BDDs have no more reason to be called "binary" than any other structure in this map.

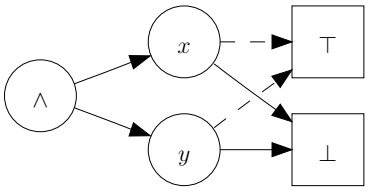

Figure 1.8: A very simple example of a DG on Boolean variables x and y; the pure conjunctive node allows us to consider the two subgraphs independently.

Remark that the FBDD language is the language of BDDs in which variables cannot be repeated along a path (in the decision diagram version of the graphs). This requirement boils down to imposing decomposability [DM02]: we just want that for each assignment node N, $\text{Var}_{\text{ass}}(N) \notin \text{Scope}(\text{Tail}_{\text{ass}}(N))$, which is equivalent to imposing that the scopes of the children of N be disjoint.

Decision graphs (DG-representations) basically add to decision diagrams the possibility to use pure conjunctive nodes between decision nodes, allowing parallel assignments, as shown in Figure 1.8.

1.3.6 Ordering for Decision Graphs

General Ordering

This section deals with the ordering of decision graphs, which is not as straightforward as it was when introduced by Bryant [Bry86] on binary decision diagrams. Although the following definition of an ordered graph is very close to the one introduced by Bryant, it must be noted that the order needs not be total—which allows one to take advantage of the "parallel" structure of decision graphs.

Definition 1.3.25 (Ordering)**.** Let $<$ be a strict order on some $V \subseteq \mathcal{V}$; a GRDAG φ is *ordered by* $<$ if and only if $\text{Scope}(\varphi) \subseteq V$ and for each couple $\langle N_1, N_2 \rangle$ of distinct decision nodes in φ such that N_1 is an ancestor of N_2, it holds that $\text{Var}_{\text{dec}}(N_1) < \text{Var}_{\text{dec}}(N_2)$.

O-DDG$_<$ is the restriction of DDG to graphs ordered by $<$. O-DDG is the union of all O-DDG$_<$, for every possible strict order $<$.

Let $<$ be a *total* strict order on some $V \subseteq \mathcal{V}$; OBDD$_<$ is the restriction of BDD to graphs ordered by $<$. OBDD is the union of all OBDD$_<$, for every possible total strict order $<$.

$\text{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ contains ordered binary decision diagrams as they have been introduced by Bryant [Bry86]. Recalling that $\mathcal{E}$ is the set of enumerated variables [§ 1.2.1], $\text{OBDD}_{\mathcal{E}}^{\$\mathbb{Z}}$ contains multivalued decision diagrams (MDDs) [SK$^+$90] and finite-state automata [Vem92]—we use the notation $\text{MDD} = \text{OBDD}_{\mathcal{E}}^{\$\mathbb{Z}}$. In a similar way, we can recover a more exotic language, BED, defined by Andersen and Hulgaard [AH97] as an extension of OBDDs. Its particularity is that it allows any binary operator

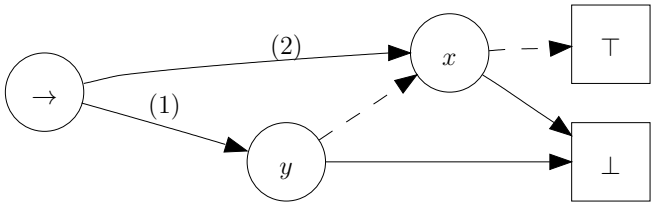

Figure 1.9: An example of a Boolean expression diagram, with decision nodes represented in simplified form. Variables x and y are Boolean.

between decision nodes; it is thus not a fragment of *Q*PDAG. We can nevertheless recover it with previous definitions.

Definition 1.3.26 (Boolean expression diagrams)**.** A GRDAG φ is a *Boolean expression diagram (BED)* if and only if $\mathrm{Ops}(\varphi) \subseteq \mathbb{B}^{\mathbb{B}^2}$, φ satisfies weak and exclusive decision [Def. 1.3.23], and is ordered by some strict order.

BED is the restriction of GRDAG to Boolean expression diagrams.

A simple example of a BED can be found in Figure 1.9.

Another language is that of *interval diagrams* [ST98], also defined as an extension of OBDDs. It is a sublanguage of OBDD, the edges being labeled by intervals of integers. We do not define them formally here, because it would require specific definitions (there are indeed restrictions on the outgoing edges of a node). This language is nonetheless encompassed by the language of *set-labeled diagrams* that we introduce later in this work [Chapter 7].

Strong Ordering

O-DDG is quite general, but this definition of ordering does not allow canonicity, since the order does not constrain the use of pure conjunctive nodes. Fargier and Marquis [FM06] thus defined a refinement of this language, allowing only tree orders.

Definition 1.3.27 (Tree order)**.** Let $<$ be a strict order on some set X. $<$ is a *tree order* if and only if the graph $\langle X, <_{\min}\rangle$ is a tree, with $<_{\min}$ being the smallest binary relation (w.r.t. inclusion) the transitive closure of which is $<$.

Tree orders are thus midway between partial strict orders and total strict orders, since they refine strict orders, and are refined by total strict orders. For example, the strict order $<_{\mathrm{a}}$ defined on $\{x, y, z, t\}$ by $x <_{\mathrm{a}} y <_{\mathrm{a}} z$ and $y <_{\mathrm{a}} t$ is a tree order, while $<_{\mathrm{b}}$ defined by $x <_{\mathrm{b}} y <_{\mathrm{b}} z$ and $t <_{\mathrm{b}} y$ is not. Let us now impose tree ordering on a decision graph's variables.

Definition 1.3.28 (Strong ordering)**.** Let $<$ be a tree order on some $V \subseteq \mathcal{V}$; a GRDAG φ is *strongly ordered by* $<$ if and only if

- φ is ordered by $<$ (which implies that $\mathrm{Scope}(\varphi) \subseteq V$);
- for each decision node N on some variable $y \in V$ of φ, if there exists $x \in V$

such that $x < y$, then N has an ancestor which is a decision node on x.

SO - DDG$_<$ is the restriction of DDG to graphs strongly ordered by $<$. SO - DDG is the union of all SO - DDG$_<$, for all possible tree order $<$.

SO - DDG$_{\mathcal{E}}^{\mathbb{S}\mathbb{Z}}$ contains both AND/OR multivalued decision diagrams (AOMDDs) [MD06] and tree-driven automata [FV04].

1.3.7 Closure Principles

Closure principles have been formally defined by Fargier and Marquis [FM08a]. They enable us to define new languages "above" an existing one, by allowing the use of some operators not contained in the original language. We here slightly extend their concepts to GRDAGs. We first need to consider proper formulæ.

Definition 1.3.29 (Proper formula)**.** Let φ be a GRDAG, l a literal in φ with Scope$(l) = x$, and Q a node labeled with a quantification on x.

We say that l is *bound by* Q if and only if there exists a path from the child of Q to l containing no quantification on x.

We say that l is *free* if and only if there exists a path from the root to l containing no quantification on x.

φ is said to be *proper* if and only if each of its literals is either free or bound by exactly one quantification on x. GRDAG$_p$ is the restriction of GRDAG to proper formulæ; Q_pPDAG is the restriction of QPDAG to proper formulæ.

Using only proper GRDAG sentences, we are sure that there is no literal that depends on two different quantifiers, or is together free and bound. This simplifies some operations, that would require such literals to be duplicated (for example, during a conditioning operation, free literals have to be changed, but not bound literals, raising problems if some literals fall under both qualifications).

Definition 1.3.30 (Closure of a language)**.** Let L be a sublanguage of GRDAG, and Δ a subset of Ops. The Δ-*closure of* L, denoted L$[\Delta]$, is the sublanguage of GRDAG$_{V_{\mathtt{L}}}$ the representation set of which is inductively defined as follows:

- if $\varphi \in$ L, then $\varphi \in$ L$[\Delta]$;
- if $\otimes \in \Delta \cap \mathbb{B}^{\mathbb{B}^2}$, φ_l and φ_r are L-representations, then $\varphi_l \otimes \varphi_r \in$ L$[\Delta]$;
- if $\neg \in \Delta$ and $\varphi \in$ L, then $\neg\varphi \in$ L$[\Delta]$;
- if q $\in \Delta \cap \{\exists, \forall\}$, $\varphi \in$ L, and $x \in V_{\mathtt{L}}$, then q$x.\varphi \in$ L$[\Delta]$.

Using closures is especially useful on incomplete languages, since it can allow one to build a complete language while keeping the main operations tractable, as we will see in Section 1.4.3. Following Fargier and Marquis [FM08b], we consider the following closures: KROM - C$[\vee]$, HORN - C$[\vee]$, K/H - C$[\vee]$, and renH - C$[\vee]$.

1.4 Compilation Map of Boolean Languages

This section sums up the knowledge compilation results known from the literature about the various Boolean languages (i.e., languages with valuation set $E = \mathbb{B}$) introduced in the previous section. We start by presenting the queries and transformations used to classify Boolean languages. They come from the knowledge compilation map literature, with some modifications so that they fit our context involving non-Boolean variables.

1.4.1 Boolean Queries and Transformations

Semantical Notions

This section formally extends some notions of logic (model, consistency, validity, etc.) to Boolean functions in general.

Definition 1.4.1 (Model and countermodel)**.** Let f be a Boolean function over $V \subseteq \mathcal{V}$; a V-assignment $\vec{v}$ is a *model* (resp. *countermodel*) of f, denoted $\vec{v} \models f$ (resp. $\vec{v} \not\models f$), if and only if $f(\vec{v}) = \top$ (resp. $f(\vec{v}) = \bot$).

The *model set* of f is denoted by $\text{Mod}(f) \subseteq \text{Dom}(V)$; its *countermodel set* is denoted by $\text{Mod}^{\text{c}}(f)$. Quite clearly, $\{\text{Mod}(f), \text{Mod}^{\text{c}}(f)\}$ is a partition of $\text{Dom}(V)$.

We extend the definition of a model to every assignment $\vec{x}$ of variables from any set $X \subseteq \mathcal{V}$: denoting $Y = V \setminus X$, $\vec{x}$ is a model of f if and only if for every Y-assignment $\vec{y}$, $f(\vec{x}_{|V} \cdot \vec{y}) = \top$. We also write $\vec{x} \models f$, and extend countermodels accordingly. Note that the model and countermodel sets still contain only V-assignments.

Definition 1.4.2 (Consistency and validity)**.** Let f be a Boolean function over $V \subseteq \mathcal{V}$. f is *consistent* if and only if $\text{Mod}(f) \neq \varnothing$. f is *valid* if and only if $\text{Mod}^{\text{c}}(f) = \varnothing$.

Definition 1.4.3 (Context)**.** Let f be a Boolean function over $V \subseteq \mathcal{V}$, $x \in \mathcal{V}$ be a variable, and $\omega \in \text{Dom}(x)$. ω is said to be a *consistent value* of x in f if and only if there exists a $V \cup \{x\}$-assignment $\vec{v}$ such that $\vec{v}_{|x} = \omega$ and $\vec{v} \models f$.

The set of all consistent values of x in f is called the *context* of x in f and is denoted as $\text{Ctxt}_f(x)$.

Consistent values are sometimes called *generalized arc consistent (GAC)* or *globally consistent* in constraint programming [see e.g. LS06].

Proposition 1.4.4. Let f be a Boolean function over $V \subseteq \mathcal{V}$. The following statements are equivalent:

(1) f is consistent;

(2) $\exists x \in \mathcal{V}, \text{Ctxt}_f(x) \neq \varnothing$;

(3) $\forall x \in \mathcal{V}, \text{Ctxt}_f(x) \neq \varnothing$.

Proof. We prove that $(2) \implies (1) \implies (3) \implies (2)$.

- $(2) \implies (1)$**:** Let $x \in \mathcal{V}$; suppose that $\text{Ctxt}_f(x) \neq \varnothing$. Let $\omega \in \text{Ctxt}_f(x)$; by definition of the context, there exists a $V \cup \{x\}$-assignment $\vec{v}$ such that $\vec{v}_{|x} = \omega$ and $\vec{v} \models f$. Obviously, $\vec{v}_{|V} \models f$, and thus $\text{Mod}(f) \neq \varnothing$: f is consistent.

- $(1) \implies (3)$**:** Suppose that f is consistent. Let $x \in \mathcal{V}$; we show that $\text{Ctxt}_f(x) \neq \varnothing$. Let $\vec{v}$ be a model of f.

  – If $x \in V$, by definition, $\vec{v}_{|x}$ is a consistent value of x in f, and thus $\text{Ctxt}_f(x) \neq \varnothing$.

  – If $x \notin V$, then let $\omega \in \text{Dom}(x)$ and let $\vec{x}$ be the $\{x\}$-assignment such that $\vec{x}_{|x} = \omega$. It holds that $f(\vec{v} \,.\, \vec{x}) = \top$, since $f(\vec{v}) = \top$ and $x \notin V$ (and by convention [§ 1.2.1], $f(\vec{v} \,.\, \vec{x}) = f(\vec{v} \,.\, \vec{x}_{|V}) = f(\vec{v})$). All assignments $\vec{y}$ from $V \setminus X = \varnothing$ verify $f\left((\vec{v} \,.\, \vec{x})_{|V} \,.\, \vec{y}\right) = \top$, hence $\vec{v} \,.\, \vec{x} \models f$ holds; ω is a consistent value of x in f, and thus $\text{Ctxt}_f(x) \neq \varnothing$.

  Consequently, for any $x \in \mathcal{V}$, it holds that $\text{Ctxt}_f(x) \neq \varnothing$.

- $(3) \implies (2)$**:** This is trivial (since $\mathcal{V} \neq \varnothing$).

This proves by transitivity the equivalence of the three statements. □

Definition 1.4.5 (Entailment and equivalence)**.** Let f, g be Boolean functions. f *entails* g, denoted $f \models g$, if and only if every model of f is a model of g. f is *equivalent* to g, denoted $f \equiv g$, if and only if both $f \models g$ and $g \models f$ hold.

All the preceding notions can also be simply defined on *representations*, that is, on elements of some language, using the interpretation function.

Definition 1.4.6. Let $\mathtt{L}$ be a Boolean representation language, and φ and ψ some $\mathtt{L}$-representations.

- A $\text{Scope}(\varphi)$-assignment $\vec{v}$ is a *model* (resp. a *countermodel*) of φ, denoted $\vec{v} \models \varphi$ (resp. $\vec{v} \not\models \varphi$), if and only if it is a model (resp. countermodel) of $[\![\varphi]\!]$.
- We write $\text{Mod}(\varphi) = \text{Mod}([\![\varphi]\!])$ and $\text{Mod}^{\text{c}}(\varphi) = \text{Mod}^{\text{c}}([\![\varphi]\!])$.
- φ is *consistent* if and only if $[\![\varphi]\!]$ is consistent.
- φ is *valid* if and only if $[\![\varphi]\!]$ is valid.
- $\varphi \models \psi$ if and only if $[\![\varphi]\!] \models [\![\psi]\!]$.
- $\varphi \equiv \psi$ if and only if $[\![\varphi]\!] \equiv [\![\psi]\!]$.

- A *consistent value* of variable x in φ is defined as a consistent value of x in $[\![\varphi]\!]$. We write $\text{Ctxt}_{\varphi}(x) = \text{Ctxt}_{[\![\varphi]\!]}(x)$.

We now introduce operations on Boolean functions The following definitions pertain to operations on Boolean functions, to wit, application of binary operators, negation, quantification, and restriction.

Definition 1.4.7 (Binary operations)**.** Let $\otimes \in \mathbb{B}^{\mathbb{B}^2}$ be any binary operator on $\mathbb{B}$, and f, g be Boolean functions over $V_f \subseteq \mathcal{V}$ and $V_g \subseteq \mathcal{V}$, respectively. The Boolean function $f \otimes g$ is defined on variables from $V_f \cup V_g$ by $\forall \vec{x} \in \text{Dom}(V_f \cup V_g), (f \otimes g)(\vec{x}) = f(\vec{x}_{|V_f}) \otimes g(\vec{x}_{|V_g})$.

Definition 1.4.8 (Negation)**.** Let f be a Boolean function over $V \subseteq \mathcal{V}$; the Boolean function $\neg f$ is defined on variables from V by $\forall \vec{x} \in \text{Dom}(V), (\neg f)(\vec{x}) = \neg f(\vec{x})$.

Definition 1.4.9 (Quantification)**.** Let f be a Boolean function over $V \subseteq \mathcal{V}$, and $X \subseteq \mathcal{V}$ be a set of variables. The *existential quantification of f by X* (also called *forgetting of X in f* and *existential projection of f on $V \setminus X$*), denoted $\exists X.f$, is the Boolean function defined on $Y = V \setminus X$ by

$$\forall \vec{y} \in \text{Dom}(Y), \quad (\exists X.f)(\vec{y}) = \top \iff \exists \vec{x} \in \text{Dom}(X), f(\vec{y} \,.\, \vec{x}_{|V}) = \top.$$

The *universal quantification of f by X* (also called *ensuring of X in f* and *universal projection of f on $V \setminus X$*), denoted $\forall X.f$, is the Boolean function defined on $Y = V \setminus X$ by

$$\forall \vec{y} \in \text{Dom}(Y), \quad (\forall X.f)(\vec{y}) = \top \iff \forall \vec{x} \in \text{Dom}(X), f(\vec{y} \,.\, \vec{x}_{|V}) = \top.$$

Proposition 1.4.10. Let f be a Boolean function over $V \subseteq \mathcal{V}$. f is consistent if and only if $\exists V.f$ is consistent. f is valid if and only if $\forall V.f$ is consistent.

Proof. $\text{Scope}(\exists V.f) = \varnothing$; $\exists V.f$ is consistent if and only if $(\exists V.f)(\vec{\varnothing}) = \top$. By definition of the existential quantification, this condition is equivalent to $\exists \vec{v} \in \text{Dom}(V), f(\vec{v}) = \top$, which is the definition of f's consistency.

Similarly, $\text{Scope}(\forall V.f) = \varnothing$. $\forall V.f$ is consistent if and only if $(\forall V.f)(\vec{\varnothing}) = \top$. This holds if and only if $\forall \vec{v} \in \text{Dom}(V), f(\vec{v}) = \top$, i.e., if and only if f is valid. □

Definition 1.4.11 (Restriction)**.** Let f, g be Boolean functions over $V_f \subseteq \mathcal{V}$ and $V_g \subseteq \mathcal{V}$, respectively. The *restriction* of f to g, denoted $f_{|g}$, is the Boolean function defined on $Y = V_f \setminus V_g$ by $f_{|g} = \exists V_g.(f \wedge g)$.

Let $V \subseteq \mathcal{V}$ and let $\vec{v}$ be a V-assignment. The *restriction* of f to $\vec{v}$, denoted $f_{|\vec{v}}$, is the Boolean function defined on $Y = V_f \setminus V$ by $f_{|\vec{v}}(\vec{y}) = f(\vec{y} \,.\, \vec{v})$.

Note that the restriction to an assignment can be seen as a special case of restriction to a function. The restriction operation is useful to compose two functions or to fix the values of some variables. We also write $f_{|x=\omega}$ to denote the restriction of f to the assignment of x to ω.

Proposition 1.4.12 (Shannon decomposition)**.** Let f be a Boolean function and $x \in \text{Scope}(f)$. The following properties hold:

- $\exists x.f \equiv \bigvee_{\vec{x} \in \text{Dom}(\{x\})} f_{|\vec{x}}$;
- $\forall x.f \equiv \bigwedge_{\vec{x} \in \text{Dom}(\{x\})} f_{|\vec{x}}$.

Proof. Let $Z = \text{Scope}(f) \setminus \{x\}$, and $\vec{z}$ a Z-assignment. Let us prove that $\exists x.f \equiv \bigvee_{\vec{x} \in \text{Dom}(\{x\})} f_{|\vec{x}}$:

(⇒) Suppose that $(\exists x.f)(\vec{z}) = \top$. This means there exists an $\{x\}$-assignment $\vec{x}$ such that $f(\vec{z} \ . \ \vec{x}) = \top$, i.e. $f_{|\vec{x}}(\vec{z}) = \top$. It is hence obvious that $(\bigvee_{\vec{x} \in \text{Dom}(\{x\})} f_{|\vec{x}})(\vec{z}) = \top$.

(⇐) Suppose that $(\exists x.f)(\vec{z}) = \bot$. Then any $\{x\}$-assignment $\vec{x}$ verifies $f(\vec{z} \ . \ \vec{x}) = \bot$, and therefore $f_{|\vec{x}}(\vec{z}) = \bot$. Consequently, it is clear enough that $(\bigvee_{\vec{x} \in \text{Dom}(\{x\})} f_{|\vec{x}})(\vec{z}) = \bot$.

The proof of $\forall x.f \equiv \bigwedge_{\vec{x} \in \text{Dom}(\{x\})} f_{|\vec{x}}$ is dual from the previous one:

(⇒) Supposing $(\forall x.f)(\vec{z}) = \top$, any $\{x\}$-assignment $\vec{x}$ verifies $f(\vec{z} \ . \ \vec{x}) = \top$, so $f_{|\vec{x}}(\vec{z}) = \top$. Hence $(\bigwedge_{\vec{x} \in \text{Dom}(\{x\})} f_{|\vec{x}})(\vec{z}) = \top$.

(⇐) Supposing $(\forall x.f)(\vec{z}) = \bot$. Then there is a $\{x\}$-assignment $\vec{x}$ such that $f(\vec{z} \ . \ \vec{x}) = \bot$, i.e., $f_{|\vec{x}}(\vec{z}) = \bot$, hence $(\bigwedge_{\vec{x} \in \text{Dom}(\{x\})} f_{|\vec{x}})(\vec{z}) = \bot$. □

Queries on Boolean Languages

Let us now define the standard queries used to compare the efficiency of Boolean languages. Introduced by Darwiche and Marquis [DM02], they have been generalized to representation languages by Fargier and Marquis [FM09], from whom we adapted the following definitions.

We identified three types of query; let us begin with the first category, which contains queries common to all Boolean languages.

Definition 1.4.13 (General queries)**.** Let L be a Boolean representation language.

- L satisfies **CO** (consistency) (resp. **VA**, validity) if and only if there exists a polytime algorithm mapping every L-representation φ to 1 if φ is consistent (resp. valid), and to 0 otherwise.
- L satisfies **MC** (model checking) if and only if there exists a polytime algorithm mapping every L-representation φ and every assignment $\vec{v}$ of the variables in $\text{Scope}(\varphi)$ to 1 if $\vec{v} \models \varphi$, and to 0 otherwise.
- L satisfies **EQ** (equivalence) (resp. **SE**, sentential entailment) if and only if there exists a polytime algorithm mapping every couple of L-representations $\langle \varphi, \psi \rangle$ to 1 if $\varphi \equiv \psi$ (resp. $\varphi \models \psi$), and to 0 otherwise.

These queries are straightforward extensions of the classical definitions [DM02] to languages on non-Boolean variables. Thanks to their simple definitions, these queries apply to any Boolean language. This is due to the fact that, contrary to the next queries, they have simple inputs and output, raising no questions about how these are represented.

The queries in the second category are problematic in this regard: their output is still simple (Boolean), but their parameters are "complex" objects, namely clauses and terms—that are only defined as GRDAGs. For this reason, we chose to define these queries on GRDAG sublanguages only.

Definition 1.4.14 (Queries with complex parameters)**.** Let L be a sublanguage of GRDAG.

- L satisfies **CE** (clausal entailment) if and only if there exists a polytime algorithm mapping every L-representation φ and every clause γ in L to 1 if $\varphi \models \gamma$, and to 0 otherwise.
- L satisfies **IM** (implicant checking) if and only if there exists a polytime algorithm mapping every L-representation φ and every term γ in L to 1 if $\gamma \models \varphi$, and to 0 otherwise.

In this definition, φ and γ all are GRDAGs, and in particular, L-representations (of the same L). If it were not the case, there would be expressivity problems, that could complicate the reasons why a language supports these queries or not—whereas the purpose of the map is to give insight about the *intrinsic* power of each language. For instance, for $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$ to satisfy **CE**, the polytime algorithm must be able to decide whether some $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$-representation entails $\big[x = \top\big] \vee \big[y = \bot\big]$, but *not* whether it entails non-GRDAG "clauses" such as $\big[x = y\big] \vee \big[y = z\big]$, which has a completely different expressivity.

The queries in the third category have no complex parameters, but their output (related to the notion of model set) is non-Boolean. We could not simply extend these queries from the literature; indeed, the number of models being possibly infinite, we could not consider counting or enumerating them. Regarding the "model enumeration" query, we had to decide on an arbitrary representation of the model set. We chose to require the model set to be in the form of a DNF, since this is a natural way of representing continuous sets of assignments—they are called *union of boxes* in this context. For the "model counting" query, this is harder; we thought of using the characteristic size of the *minimal* equivalent DNF-representation, but proving minimality is not simple. In the end, we chose to keep the original meaning of "counting", and output the number of models if it is finite, and ∞ otherwise.

Definition 1.4.15 (Queries with complex output)**.** Let L be a sublanguage of GRDAG, over the set of variables $V_{\mathtt{L}}$.

- L satisfies **ME** (model enumeration) if and only if there exists a polynomial $P(\cdot, \cdot)$ and an algorithm mapping every L-representation φ to a $\mathtt{DNF}_{V_{\mathtt{L}}}$-representation ψ of $[\![\varphi]\!]$ in time $P(\|\varphi\|, \|\psi\|)$.

- L satisfies **ME**$^{\mathbf{c}}$ (counter-model enumeration) if and only if there exists a polynomial $P(\cdot,\cdot)$ and an algorithm mapping every L-representation φ to a $\mathtt{DNF}_{V_\mathtt{L}}$-representation ψ of $\neg[\![\varphi]\!]$ in time $P(\|\varphi\|,\|\psi\|)$.
- L satisfies **CT** (model counting) if and only if there exists a polytime algorithm mapping every L-representation φ to $|\mathrm{Mod}(\varphi)|$ if it is finite, and to ∞ otherwise.

Once again, these queries are only defined on sublanguages of GRDAG, because of expressivity issues. Indeed, we want a $\mathtt{DNF}_{V_\mathtt{L}}$-representation of φ to exist, yet this is not guaranteed if φ is not a GRDAG. Consider for instance the Boolean function f defined on real variables x and y by $f(x,y) = \top \iff x = y$; there exists no GRDAG-representation of f, and hence no DNF-representation. To be able to use DNF to represent the model set, we have to restrict the input language to a language with expressivity similar to DNF, which is the case of GRDAG.

Before going on with transformations, let us note that the queries defined here are classical ones; we introduce new queries later in this work [§ 3.3.3].

Transformations on Boolean Languages

We now present the standard transformations on Boolean languages, also introduced by Darwiche and Marquis [DM02] and generalized to representation languages by Fargier and Marquis [FM09].

Definition 1.4.16 (Transformations)**.** Let L be a Boolean representation language, over the set of variables $V_\mathtt{L}$.

- L satisfies **CD** (conditioning) if and only if there exists a polytime algorithm mapping every L-representation φ and every assignment $\vec{v}$ of variables from $V_\mathtt{L}$ to an L-representation of the restriction $[\![\varphi]\!]_{|\vec{v}}$ of $[\![\varphi]\!]$ to $\vec{v}$.
- L satisfies **FO** (forgetting) (resp. **SFO**, single forgetting) if and only if there exists a polytime algorithm mapping every L-representation φ and every set (resp. singleton) $X \subseteq V_\mathtt{L}$ to an L-representation of $\exists X.[\![\varphi]\!]$.
- L satisfies **EN** (ensuring) (resp. **SEN**, single ensuring) if and only if there exists a polytime algorithm mapping every L-representation φ and every set (resp. singleton) $X \subseteq V_\mathtt{L}$ to an L-representation of $\forall X.[\![\varphi]\!]$.
- L satisfies $\wedge$**C** ($\wedge$-closure) (resp. $\vee$**C**, $\vee$-closure) if and only if there exists a polytime algorithm mapping every finite set Φ of L-representations to an L-representation of $\bigwedge_{\varphi\in\Phi}[\![\varphi]\!]$ (resp. $\bigvee_{\varphi\in\Phi}[\![\varphi]\!]$).
- L satisfies $\wedge$**BC** ($\wedge$-closure under binary $\wedge$) (resp. $\vee$**BC**, $\vee$-closure under binary $\vee$) if and only if there exists a polytime algorithm mapping every couple of L-representations $\langle\varphi,\psi\rangle$ to an L-representation of $[\![\varphi]\!] \wedge [\![\psi]\!]$ (resp. $[\![\varphi]\!] \vee [\![\psi]\!]$).
- L satisfies $\neg$**C** ($\neg$-closure) if and only if there exists a polytime algorithm mapping every L-representation φ to an L-representation of $\neg[\![\varphi]\!]$.

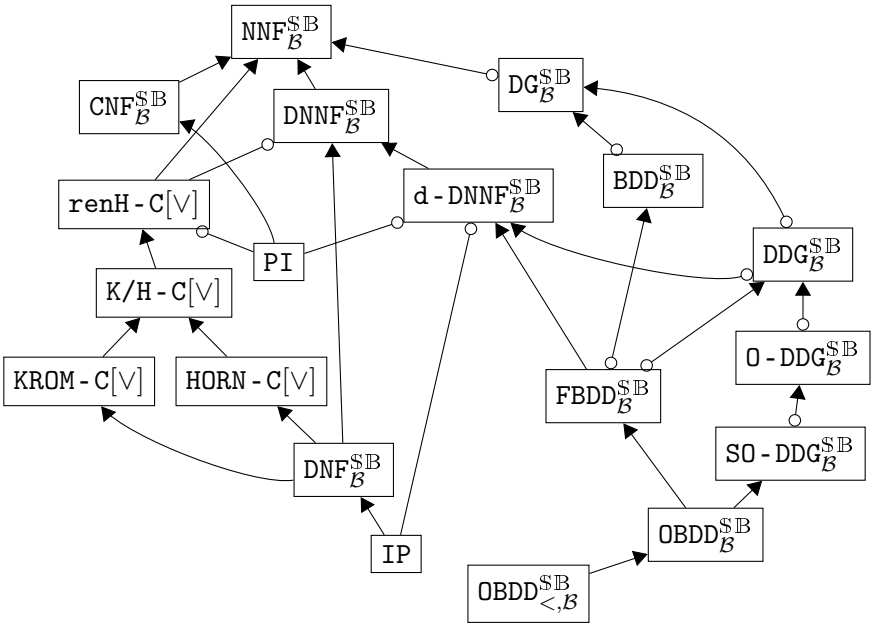

Figure 1.10: Succinctness of some $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$ fragments. On an edge linking $\mathtt{L}_1$ and $\mathtt{L}_2$, an arrow pointing towards $\mathtt{L}_1$ means that $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$. If there is no symbol on $\mathtt{L}_1$'s side (neither an arrow nor a circle), it means that $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$. If there is a circle on $\mathtt{L}_1$'s side, it means that it is unknown whether $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$ or $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$ holds. Relations deducible by transitivity are not represented, which means that two fragments not being ancestors to each other are incomparable with respect to succinctness.

The only transformation that was problematic to generalize to non-Boolean variables is the first one. In its original definition [DM02], conditioning is a "restriction to a term"; but since literals were of the form "$x = \top$" or "$x = \bot$", it boils down to a restriction to an assignment. When generalizing conditioning to Boolean languages with non-Boolean variables, we had the choice between extending the "term restriction" definition or keeping the "assignment restriction" usage. We chose the second alternative (which has the advantage of having no representation-dependant parameter—see discussion of Definition 1.4.14), and extended the original definition into another transformation, explicitly called *term restriction* [see § 3.3.3].

1.4.2 Succinctness Results

Figure 1.10 presents succinctness results about some of the Boolean languages we presented, in the form of a *succinctness graph*.

Theorem 1.4.17. The succinctness relations in Figure 1.10 hold.

Proof. The results come from various papers [DM02, FM08b, FM06]. □

L	CO	VA	CE	IM	EQ	SE	CT	ME
$\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$	○	○	○	○	○	○	○	○
$\mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	○	○	○	○	√
$\mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$	○	√	○	√	○	○	○	○
PI	√	√	√	√	√	√	○	√
IP	√	√	√	√	√	√	○	√
KROM-C	√	√	√	√	√	√	○	√
KROM-C[∨]	√	○	√	○	○	○	○	√
HORN-C	√	√	√	√	√	√	○	√
HORN-C[∨]	√	○	√	○	○	○	○	√
K/H-C	√	√	√	√	√	√	○	√
K/H-C[∨]	√	○	√	○	○	○	○	√
renH-C	√	√	√	√	√	√	○	√
renH-C[∨]	√	○	√	○	○	○	○	√
$\mathtt{DNNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	○	○	○	○	√
$\mathtt{d\text{-}NNF}_{\mathcal{B}}^{\$\mathbb{B}}$	○	○	○	○	○	○	○	○
$\mathtt{d\text{-}DNNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	○	√	√
$\mathtt{DG}_{\mathcal{B}}^{\$\mathbb{B}}$	○	○	○	○	○	○	○	○
$\mathtt{DDG}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	○	○	√	√
$\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$	○	○	○	○	○	○	○	○
$\mathtt{FBDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	○	√	√
$\mathtt{O\text{-}DDG}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	○	√	√
$\mathtt{O\text{-}DDG}_{<,\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	?	√	√
$\mathtt{SO\text{-}DDG}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	○	√	√
$\mathtt{SO\text{-}DDG}_{<,\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	√	?	√	√
$\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	√	○	√	√
$\mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	√	√	√	√

Table 1.1: Queries satisfied by fragments of $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$. Meaning of symbols is as follows: √ means "satisfies", ○ means "does not satisfy, unless $\mathsf{P} = \mathsf{NP}$", and ? indicates an unknown result.

A succinctness graph allows one to quickly examine the relative succinctness of a group of languages. Having identified the languages supporting their desired operations, users just have to select those being as close to the root as possible. Of course, when some "hard" operations are necessary, such as equivalence checking, there is no choice but to select leaf languages.

1.4.3 Satisfaction of Queries and Transformations

Tables 1.1 and 1.2 show which queries and transformations are known to be satisfied or not by the languages we presented.

Theorem 1.4.18. The results in Tables 1.1 and 1.2 hold.

Proof. The results come from various papers [DM02, FM08b, FM06]. □

L	CD	FO	SFO	∧C	∧BC	∨C	∨BC	¬C
$\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	√	√	√	√	√
$\mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	•	√	√	√	•
$\mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	√	√	•	√	•
PI	√	√	√	•	•	•	√	•
IP	√	•	•	•	√	•	•	•
KROM-C	√	√	√	√	√	!	!	!
KROM-C[∨]	√	√	√	○	√	√	√	•
HORN-C	√	•	√	√	√	!	!	!
HORN-C[∨]	√	?	√	○	√	√	√	•
K/H-C	√	•	√	○	○	!	!	!
K/H-C[∨]	√	?	√	○	○	√	√	•
renH-C	√	•	√	!	!	!	!	!
renH-C[∨]	√	?	√	○	○	√	√	○
$\mathtt{DNNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	○	○	√	√	○
d-$\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	√	√	√	√	√
d-$\mathtt{DNNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	○	○	○	○	○	?
$\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	√	√	√	√	√
$\mathtt{FBDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	•	○	•	○	•	○	√
$\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	•	√	•	○	•	○	√
$\mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}}$	√	•	√	•	√	•	√	√

Table 1.2: Transformations satisfied by fragments of $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$. Meaning of symbols is as follows: √ means "satisfies", • means "does not satisfy", ○ means "does not satisfy, unless P = NP", ! indicates that the transformation is not always feasible within the fragment, and ? indicates an unknown result.

This classical presentation of the knowledge compilation map is meant to be easy to read; checking whether a language satisfies some given query or transformation is immediate, as is finding the set of languages that satisfy some given query or transformation. To achieve this, tables do not provide precise information about the complexity class of each problem, but rather only indicate whether the query or transformation is satisfied.

*
**

We have presented a state-of-the-art knowledge compilation map of graph-based Boolean languages. Let us now introduce a few non-Boolean languages.

1.5 Non-Boolean Languages

In this section, we take a glimpse at non-Boolean representation languages, designed to compactly express functions with a non-Boolean output: $E_{\mathrm{L}} \neq \mathbb{B}$. Note

that the "non-Boolean" character of a language only concerns the target set of its interpretation domain, not necessarily its variables. We do not develop this section as much as we developed the Boolean languages portion, for several reasons: much fewer non-Boolean languages have been defined, there are fewer applications using them, and there are fewer works about their knowledge compilation properties.

1.5.1 ADDs

As it was hinted in § 1.1.2.1, the basic principle of OBDDs has been adapted to non-Boolean functions. The first such extension[5] is due to Clarke et al. [CF$^+$93] who called it multi-terminal binary decision diagrams (MTBDDs). The difference between OBDDs and MTBDDs is basically an increase in the number of leaves, so that every possible output of the function corresponds to a unique leaf. Obviously, there has to be a finite number of leaves: the interpretation domain of MTBDDs is restricted to $\mathfrak{D}_{\mathcal{B},S}$, with S a finite subset of $\mathbb{R}$.

The idea of MTBDDs was extended by Bahar et al. [BF$^+$97], to express functions taking their values on (finite) algebraic structures. Let us present their more general definition.

Definition 1.5.1 (ADD)**.** Let $\mathcal{S} = \langle S, O, D\rangle$ be an algebraic structure, composed of a finite carrier S, a set of operations O and a set of distinguished elements $D \subseteq S$. An *algebraic decision diagram (ADD)* on S is a rooted DAG meeting the following requirements:

- each leaf is labeled with an element of S;
- each internal node N is labeled with a variable from $\mathcal{B}$ (denoted $\mathrm{Var}(N)$), and has exactly two outgoing edges, respectively labeled with $\top$ and $\bot$;
- it is ordered: there exists a total strict order $<$ of the mentioned variables such that for each couple of distinct internal nodes $\langle N, N'\rangle$, if N' is a descendant of N, then $\mathrm{Var}(N) < \mathrm{Var}(N')$.

The semantics of an ADD φ is defined inductively as follows.

- If the root of φ is a leaf, with label s, then $\mathrm{Scope}(\varphi) = \varnothing$, and $[\![\varphi]\!]$ is the constant function returning s.
- If the root of φ is an internal node, denoting x its label variable, $\varphi_\top$ the child pointed by its $\top$-outgoing edge, and $\varphi_\bot$ the child pointed by its $\bot$-outgoing edge, then $\mathrm{Scope}(\varphi) = \mathrm{Scope}(\varphi_\bot) \cup \mathrm{Scope}(\varphi_\top) \cup \{x\}$, and for any $\mathrm{Scope}(\varphi)$-assignment $\vec{v}$,

$$[\![\varphi]\!](\vec{v}) = \begin{cases} [\![\varphi_\top]\!](\vec{v}) & \text{if } \vec{v}_{|x} = \top, \\ [\![\varphi_\bot]\!](\vec{v}) & \text{if } \vec{v}_{|x} = \bot. \end{cases}$$

[5] Ignoring the fact that the original definition of MDDs [SK$^+$90] also formally allowed multiple leaves, since this feature appears to not have been used by the community.

The ADD language is the union of all representation languages $\langle \mathfrak{D}_{\mathcal{B},S}, \mathcal{R}_S, [\![\cdot]\!] \rangle$, with $\mathcal{R}_S$ the set of all ADDs on S, for any algebraic structure S.

The authors use ADDs to represent matrices over specific algebraic structures, namely quasirings and semirings. They present an algorithm returning in polytime an ADD representing the multiplication of two matrices expressed as ADDs, and apply it to the search of shortest paths. In the same way as OBDDs, ADDs are canonical structures, which allows a very efficient equivalence checking.

1.5.2 AADDs

Sanner and McAllester [SM05] noticed that ADDs are not efficient in representing functions involving additions or multiplications. Indeed, contrary to disjunctions and conjunctions, these operations can highly increase the number of possible outputs, and hence the number of leaves. Using ADDs to perform such operations can lead to structures of size exponential in the size of the initial ones. Yet, these operations are often used when reasoning with probabilities.

The authors introduce affine algebraic decision diagrams (AADDs) to tackle this problem. The idea is to associate an additive and multiplicative value with each edge to "factorize" leaves.

Definition 1.5.2 (AADD)**.** An *affine algebraic decision diagram (AADD)* φ is an extended ADD: each edge E in φ is associated with a couple $\langle c_E, b_E \rangle$ of real values from $[0, 1]$. It moreover has a unique leaf, labeled with 0.

The semantics of an AADD φ is defined inductively as follows.

- If the root of φ is the leaf, then $\text{Scope}(\varphi) = \varnothing$, and $[\![\varphi]\!]$ is the constant function returning 0.
- If the root of φ is an internal node, denoting x its label variable, $E_\top$ (resp. $E_\bot$) its $\top$-outgoing edge (resp. $\bot$-outgoing edge), and $\varphi_\top$ (resp. $\varphi_\bot$) the subgraph rooted at $\text{Dest}(E_\top)$ (resp. at $\text{Dest}(E_\bot)$), then $\text{Scope}(\varphi) = \text{Scope}(\varphi_\bot) \cup \text{Scope}(\varphi_\top) \cup \{x\}$, and for any $\text{Scope}(\varphi)$-assignment $\vec{v}$,

$$[\![\varphi]\!](\vec{v}) = \begin{cases} c_{E_\top} + b_{E_\top} \times [\![\varphi_\top]\!](\vec{v}) & \text{if } \vec{v}_{|x} = \top, \\ c_{E_\bot} + b_{E_\bot} \times [\![\varphi_\bot]\!](\vec{v}) & \text{if } \vec{v}_{|x} = \bot. \end{cases}$$

The AADD language is the representation language $\langle \mathfrak{D}_{\mathcal{B},[0,1]}, \mathcal{R}, [\![\cdot]\!] \rangle$, with $\mathcal{R}$ the set of all AADDs.

Note that even if AADDs are based on ADDs, AADD is not a sublanguage of ADD, their interpretation function being completely different. Using the definition we provided, AADDs are not canonical, but it is possible to obtain canonicity by imposing some restricting properties to the additive and multiplicative coefficients, creating *normalized AADDs* [see SM05 for details].

As shown by the authors, AADDs are spatially much more efficient than ADDs. In addition, the worst-case complexity of the "apply" operation (which is the basis

of the sum, product, etc. of two decision diagrams) on AADDs is only within a multiplicative constant of that of ADDs. This results in very good operational performance, since AADDs can be exponentially smaller than ADDs.

The language of *edge-valued binary decision diagrams (EVBDDs)*, introduced by Lai and Sastry [LS92], can be seen as a restriction of AADD, in which no multiplicative coefficient is allowed, and $\bot$-labeled edges are always associated with a zero additive coefficient. Multiplicative coefficients were added by Tafertshofer and Pedram [TP97], yielding *factored EVBDDs*. Let us also mention the language of *semiring-labeled decision diagrams (SLDD)* [Wil05], designed for the compilation of weighted constraint networks; it differs from AADDs in that variables are enumerated rather than Boolean, and there is only one coefficient per edge, which is taken from a semiring. A given path in an SLDD corresponds to an assignment of the variables and to a value, which is the semiring product of all the coefficients encountered along the path.

1.5.3 Arithmetic Circuits

Arithmetic circuits [Dar03] are similar to Boolean NNFs, but apply this mechanism to numerical variables.

Definition 1.5.3. An *arithmetic circuit (AC)* is a rooted DAG, such that:

- its leaves are labeled with a numeric constant c or with a variable $x \in \mathcal{V}_{[0,1]}$;
- its internal nodes are labeled with a numerical operator, either $+$ or $\times$.

The semantics of an arithmetic circuit φ is the function from $\mathfrak{D}_{\mathcal{V}_{[0,1]},\mathbb{R}}$ inductively defined as follows.

- If the root of φ is a leaf of label constant c, then $\text{Scope}(\varphi) = \varnothing$, and $[\![\varphi]\!]$ is the constant function returning c.
- If the root of φ is a leaf of label variable x, then $\text{Scope}(\varphi) = \{x\}$, and $[\![\varphi]\!]$ is the function $x \mapsto x$.
- If the root N of φ is an internal node, denoting $\otimes$ its label operator, then $\text{Scope}(\varphi) = \bigcup_{E \in \text{Out}(N)} \text{Scope}(\text{Dest}(E))$, and for any $\text{Scope}(\varphi)$-assignment $\vec{v}$, $[\![\varphi]\!](\vec{v}) = \bigotimes_{E \in \text{Out}(N)} [\![\text{Dest}(E)]\!](\vec{v})$.

The AC language is the representation language $\langle \mathfrak{D}_{\mathcal{V}_{[0,1]},\mathbb{R}}, \mathcal{R}, [\![\cdot]\!] \rangle$, with $\mathcal{R}$ the set of all arithmetic circuits.

ACs have been designed to represent *Bayesian networks*, a kind of probability distribution modeled by a directed acyclic graph [Pea88]. Indeed, a Bayesian network can be expressed as a polynomial of degree 1 over variables from $\mathcal{V}_{[0,1]}$, which is straightforward to convert into an arithmetic circuit. Having been created for this specific application, arithmetic circuits always represent functions over $[0, 1]$ variables in the literature. We decided to restrict the interpretation domain of AC

accordingly, even if in the original work of Darwiche [Dar03], nothing prevents the variables in an arithmetic circuit from having any real-valued domain.

Several probabilistic queries on Bayesian networks boil down to computing some partial derivative of its polynomial representation. Now, differentiating a polynomial expressed as an arithmetic circuit is easy (linear in the size of the structure), which explains the interest of this compilation.

1.5.4 A Unified Framework: VNNF

The VNNF fragment (VNNF stands for valued negation normal form) has been introduced by Fargier and Marquis [FM07] as a general framework for the representation of non-Boolean functions. It subsumes all languages we presented in this section, and also NNF, constraint networks [see Section 1.6.1], and various frameworks extending constraint networks with valuations. One of the interests of this generalization is the possibility to identify some common behaviour and structure among these different formalisms, and to classify them with respect to their operational efficiency—it is the first step towards a knowledge compilation map of non-Boolean languages. VNNF has inspired our definition of GRDAG, in that we wanted to define a common formalism to encompass all graph-based Boolean languages.

We do not give the definition of VNNFs; we only emphasize the main differences between VNNFs and GRDAGs. The first difference is of course that the target set of the interpretation domain of VNNFs is not Boolean: $E_{\texttt{VNNF}} \neq \mathbb{B}$. The authors define the target set as a *valuation structure*, i.e. an ordered set with a least element and a greatest element. The operators with which internal nodes are labeled are of course operators on this valuation structure; however, not any operator can be used (there are restrictive conditions on operators, they must for example be binary and commutative).

The second main difference holds on leaves: in VNNFs, they are labeled with functions from the interpretation domain. These functions are called "local functions". Of course, if a local function is hard to compute, the semantics of the overall VNNF is also hard to handle. In GRDAGs, leaves are also labeled with some sort of local functions—but only very specific local functions can be used, namely constants or functions of the form $x \mapsto \big[x \in A\big]$.

We do not detail the VNNF framework any further; let us just emphasize that it enables one to define quite general queries and transformations (such as partial consistency, optimal satisfaction, quantification with respect to any operator), and to extend some structural notions such as decomposability or determinism.

1.6 Compiling

After this rather theoretical presentation of knowledge compilation, including numerous examples of representation languages and their compilation properties, the

question of *practical use* arises. From a theoretical point of view, the online manipulation of compiled forms is simple: it consists in combining elementary queries and transformations. But how are compiled forms obtained? What does the compilation step consist in? This section aims at answering these questions, first by presenting compilation techniques, and then by listing some state-of-the-art compilers implementing these techniques.

1.6.1 Compilation Methods

Let us take the example of a person—say, Alice—who needs an autonomous system to run a program, requiring the online solving of a reasoning problem. She decides to use knowledge compilation, after having noticed that the problem can be viewed as a large fixed knowledge base on which elementary operations are performed. Using the knowledge compilation map, she identifies the target Boolean language L that fits her application the best. She now needs to *compile* her problem into an L-representation.

Being meant to be efficiently manipulable by computers, target compilation languages are generally not easy to handle "by hand". Darwiche and Marquis [DM02] have distinguished two categories of languages, those that are designed for humans to encode knowledge directly, and those that are designed to be tractable.[6] It is very likely that Alice's reasoning problem is represented by means of a "natural" or "human-manipulable' language. The *compilation* step, in practice, is hence the translation from this natural language into the identified target compilation language.

Of course, this compilation step depends on the two concerned languages; but this does not prevent us from identifying general compilation methods. To do this, we consider a unique encoding language, general enough to encompass most practical uses, namely the *constraint network*.

Definition 1.6.1. A *constraint network (CN)* is a couple $\Pi = \langle V, \mathscr{C} \rangle$, where $V \subseteq \mathcal{V}$ is a set of variables (denoted as $\text{Scope}(\Pi)$) and $\mathscr{C}$ is a set of *constraints*. Each constraint $C \in \mathscr{C}$ has an associated scope, denoted as $\text{Scope}(C)$, and consists of a set of $\text{Scope}(C)$-assignments: these are the assignments *allowed* by C.

A *solution* of Π is a V-assignment $\vec{v}$ compatible with every constraint:

$$\forall C \in \mathscr{C}, \vec{v}_{|\,\text{Scope}(C)} \in C.$$

The set of all solutions of Π is called its *solution set*, and is denoted $\text{Sol}(\Pi)$.

This definition is quite generic; in particular, there are various ways of expressing constraints. The most direct way is to explicitly enumerate the allowed tuples—this is of course only possible when variables in the scope have a finite domain.

[6] They use the words *representation languages* for the former and *target compilation languages* for the latter—we do not use this terminology in this thesis, since our definition of representation language encompasses the two categories.

The most natural way is to directly use high-level functions and relations over the variables, leading to constraints such as $x = y$, $z \to x \wedge y$ (on Boolean variables),[7] $3x + 2y > z$ (on numeric variables), or even global constraints such as alldiff$(x_1, \ldots, x_k)$, which means "variables $x_1, \ldots, x_k$ have pairwise distinct values". The problem of answering the question "does Π have at least one solution?" is a fundamental problem in artificial intelligence, called the *constraint satisfaction problem (CSP)*.[8]

Constraint networks are a very natural way to express a knowledge base, and thus a Boolean function (identifying the solutions of a CN with the models of a Boolean function). We now study methods to translate a constraint network into a Boolean language.

Combination of Small Structures

The classical way of compiling a constraint network into some Boolean language is often called *bottom-up fashion*. It consists in combining "elementary" representations, applying operator-based transformations, such as $\wedge$**BC** (conjunction) or $\neg$**C** (negation). For example, the constraint $(x \wedge y) \to z$ can be compiled by applying the *conjunction* operator to the L-representations of $\big[x = \top\big]$ and $\big[y = \top\big]$, then applying the *implication* operator to the result and the L-representation of $\big[z = \top\big]$. Once each constraint has been compiled, a representation of the whole CN can be obtained by conjoining compiled forms of all constraints.

Of course, this method does not work with just any target language; the operator application must be tractable enough for the compiled form not to inflate exponentially during construction. The bottom-up fashion is thus typically used to compile decision diagrams [§ 1.3.5], as shown for example by Bryant [Bry86] on OBDDs, Srinivasan et al. [SK$^+$90] on MDDs, Gergov and Meinel [GM94] on FBDDs, or Vempaty [Vem92] on finite-state automata.

With well-chosen target languages, this method has the advantage of being fast. It nonetheless has a serious drawback: during construction, the temporary compiled forms can be much larger than the final one. In case the available memory is not sufficient to contain all intermediate structures, compilation fails—even if the final compiled form would have been small enough.

Solver Tracing

Another approach is to exploit the relationship between the search trace of a problem solving and the representation of this problem using a compilation language. This idea has lead to the *"DPLL with a trace"* algorithm, originated in a work by Huang and Darwiche [HD04], and further refined by Wille, Fey, and Drechsler [WFD07] and Huang and Darwiche [HD05a]. The DPLL algorithm [DLL62] is a

[7] Constraint networks can thus be used to represent knowledge bases consisting of logic formulæ.

[8] The "constraint network" terminology is sometimes used to refer to binary constraints only. There is no such restriction in our definition—we use "constraint network" to refer to the structure associated with a CSP, following for example Lecoutre and Szymanek [LS06].

seminal method to check the consistency of a CNF. Roughly speaking, it works recursively, by selecting a variable and making a case analysis on its value, based on the fact that the formula is satisfiable if and only if at least one of the cases results in a consistent formula.

Huang and Darwiche [HD05a] remarked that the search tree of the DPLL algorithm, extended so that it enumerates all models of the formula rather than stopping when the first is found, corresponds exactly to a non-reduced FBDD. They adapted their algorithm so that it can also compile OBDDs and d-DNNFs. This technique is not limited to Boolean languages; any solver using branching can be used to compile decision diagrams. It has notably been adapted to the compilation of MDDs [HH$^+$08], and was proposed by Wilson [Wil05] for the compilation of SLDDs. Contrary to the bottom-up fashion, this *top-down* construction does not generate intermediate structures that are larger than the final one. It is however generally slower, since it needs to explore the whole search tree and enumerate solutions. This drawback can be limited by the use of *caching*, that can avoid equivalent subproblems to be explored multiple times.

Approximate Compilation

A common practical problem with knowledge compilation is the size of the compiled form; even when using the most appropriate language with respect to the needed operations, the available memory of embedded systems may not be sufficient. O'Sullivan and Provan [OP06] proposed a method to reduce the size of a compiled form, by considering only the most interesting solutions.

The idea is to associate a valuation with each solution, representing its likeliness (such as a probability: the closer to 1 it is, the more interesting the solution is), and to keep in the compiled form only solutions the value of which is greater than a given threshold, fixed beforehand. Considering the value of a set of solutions to be the sum of the solutions' values, we can calculate the value v_c of the complete compiled form φ_c (containing all the possible solutions), and the value v_p of a partial compilation φ_p (containing all solutions exceeding the threshold).

The $v_\mathrm{p}/v_\mathrm{c}$ ratio is called the *valuation coverage ratio*, representing the proportion of "good" solutions covered by the partial compiled form. The $\|\varphi_\mathrm{p}\|/\|\varphi_\mathrm{c}\|$ ratio is the *memory reduction ratio*, representing the proportion of space the partial compilation allows to be saved. Thanks to these two ratios, it is possible to determine the quality of an approximate compilation, and choose the threshold accordingly.

Iterative Compilation

The "approximate compilation" idea has been further improved by Venturini and Provan [VP08a], who proposed algorithms (for languages `IP` and `DNNF`) that compute *partial solutions*, that is, incomplete assignments of the variables. These solutions are then iteratively refined, by completing the assignments, while making sure that the valuation of each partial solution always remains greater than a given threshold. The final compilation structure thus contains all and only the preferred models of the initial function.

1.6.2 Existing Compilers: Libraries and Packages

This section quickly surveys existing compilers (some being unavailable to the public) for a number of languages.

BDDs, ADDs, and AADDs

There exists a decent number of libraries providing functions to manipulate Boolean OBDDs, for various programming languages. Let us cite ABCD [Bie00], BuDDy [Lin02], JDD [Vah03], JavaBDD [Wha07], and Crocopat [Bey08]. One of them, CUDD [Som05], is quite complete (all classic operations mentioned in the knowledge compilation map [§ 1.4.1] are implemented), and widely used by the community. It also allows the manipulation of ADDs.

Sanner and McAllester [SM05] have implemented an `AADD` compiler, most likely based on the combination of small AADDs to which algebraic operations are applied, since the "operation applying" algorithm is extensively described in the paper. It was however never publicly released.

DNNFs

A `d-DNNF`$^{\mathbb{S}\mathbb{B}}_{\mathcal{B}}$ compiler, C2D, has been implemented by Darwiche [Dar04]; it is available online. Let us mention also that the solver trace compilation method [§ 1.6.1.2] can be used to compile Boolean d-DNNFs [HD05a].

There is no known compiler able to build "pure", non-deterministic Boolean DNNFs; it would be quite interesting to have one, since `DNNF`$^{\mathbb{S}\mathbb{B}}_{\mathcal{B}}$ is strictly more succinct than `d-DNNF`$^{\mathbb{S}\mathbb{B}}_{\mathcal{B}}$, and supports a number of important operations.

MDDs

Vempaty [Vem92] has implemented a CN compiler able to build finite-state automata (which are similar to MDDs). It works by bottom-up fashion, using a CSP solver to enumerate the solutions of each constraint, combine them into elementary automata, then conjoin all these elementary automata using a conjunction operation. A compiler relying on this approach has been implemented and made available by Amilhastre [Ami99].

Regarding decision graphs, Mateescu and Dechter [MD08] showed how one can obtain AOMDDs by performing AND/OR searches on input constraint networks. This is very similar to the "DPLL with a trace" approach of Huang and Darwiche [HD05a]. Another approach has been taken by Fargier and Vilarem [FV04], who build tree-driven automata (that are equivalent to AOMDDs) in bottom-up fashion, from tree-structured CSPs or hyper-trees of constraints.

Arithmetic Circuits

Darwiche [Dar03] presented various methods to compiles ACs, either using the bayesian network's *jointree*, or exploiting local structures, which turns out to be more efficient when the jointree is large. His ACE compiler [DC07] is available online.

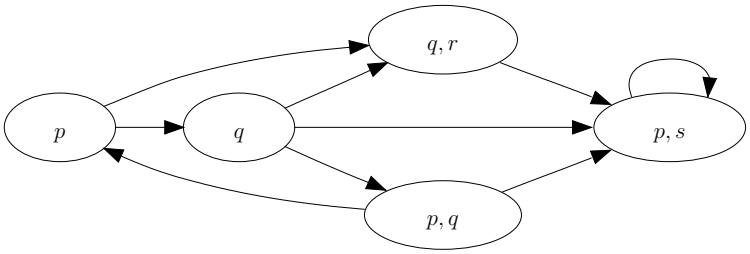

Figure 1.11: A Kripke structure represented as a graph. The worlds are the nodes of the graph, the initial world is the leftmost one, the transitions are the edges, and the labeling atoms are written inside the nodes.

1.7 Applications of Knowledge Compilation

Knowledge compilation has proven useful in a variety of domains, in particular *model checking*, *diagnosis*, *planning*, and *configuration*. Applications of compilation to planning are detailed in the next chapter [§ 2.3]. In this section, we present the three other fields, with a particular emphasis on model checking, which is a quite general domain, also used in planning [§§ 2.2, 2.3.3].

1.7.1 Model Checking

The *model checking* problem [see CGP99 for a survey, Mer01 for a tutorial, GV08 for a general perspective of the field] consists in determining whether a given property holds in some model. For example, suppose one needs a model of a complex factory machine. It is not possible to *formally prove* that the model is entirely correct, i.e., that it totally reflects the behavior of the machine in every possible case. Still, it can be interesting to verify that the model works in some given cases; thus, it is for example important that the model respects the following property: "each time a failure is detected, an alarm is triggered". Using model checking techniques, it is possible to *prove* that this property holds on the model. Defining the *expected behaviour* of some model with a set of properties, model checking thus allows this model to be proven correct with respect to this expected behaviour.

The Model Checking Problem

In the model checking framework, system models are represented as *Kripke structures* [see e.g. CGP99].

Definition 1.7.1. A *Kripke structure* is a quadruple $K = \langle W, W_0, T, L \rangle$, where:

- W is a finite set of *worlds* (also called *states*);
- $W_0 \subseteq W$ is the set of *initial worlds*;

- $T \subseteq W \times W$ is a binary relation on W, called the *transition relation*, indicating the possible transitions between worlds. T must be *left-total*: $\forall w \in W, \exists w' \in W, \langle w, w' \rangle \in T$.
- $L \colon W \to 2^{\mathcal{P}}$ is the *labeling function*, where $\mathcal{P}$ is a set of propositional atoms. L indicates which propositions hold in each world.

A Kripke structure is representable as a graph, as shown on Figure 1.11. It is meant to reflect the possible evolutions of the modeled system, passing from one world to another; each path in the graph corresponds to a given sequence of events that can occur on the system. Note that since T is left-total, the graph cannot be a tree: a path p is thus an *infinite* sequence of worlds $\langle w_0, w_1, w_2, \ldots \rangle$ such that for all $i \in \mathbb{N}$, $\langle w_i, w_{i+1} \rangle \in T$.

Using the labeling function, we can express properties on each world of a Kripke structure. To express more general properties, that hold on the whole model, we need some *modal logic*. We will use here a *temporal* logic, namely *Computation Tree Logic (CTL)* [Eme90]. Basically, CTL formulæ are propositional sentences with *temporal operators*: for example, the CTL formula $\mathbf{EX}p$ means "p holds in *at least one* of the *immediate* successors of the current world"; and $\mathbf{AX}p$ means "p holds in *all* the *immediate* successors of the current world". There are other temporal operators in CTL, that we do not detail here.

The semantics of CTL formulæ is defined on some world w of some Kripke structure K. For example, $K, w \models \mathbf{EX}p$ holds if and only if *there exists* a path $\langle w, w', w'', \ldots \rangle$ such that $K, w' \models p$. Similarly, $K, w \models \mathbf{AX}p$ holds if and only if *all* paths $\langle w, w', w'', \ldots \rangle$ verify $K, w' \models p$. When $K, w \models \varphi$ holds (with φ a CTL formula), we say that φ is true in w. For φ to be true in K as a whole, denoted $K \models \varphi$, it must hold that $K, w \models \varphi$ for all $w \in W_0$.

All in all, a model checking problem is defined as follows.

Definition 1.7.2. A *model checking problem* is a couple $\langle K, \varphi \rangle$, where K is a Kripke structure and φ a CTL formula on the same propositional atoms. The model checking problem consists in deciding whether $K \models \varphi$ holds.

From Classical Algorithms to Symbolic Model Checking

To decide whether a CTL formula holds on some Kripke structure, classical model checking algorithms directly use the definition of the semantics of CTL. That is, the truth value of atomic formulæ is computed by simply checking whether it is true in all worlds of W_0. The truth value of $\mathbf{EX}p$ is computed by checking whether p holds in some world w' such that $(w, w') \in T$, for each $w \in W_0$. Similar methods are used for all temporal operators, always implying the computation of sets of successor or predecessor worlds. All in all, the basic operations needed by model checking algorithms are the following [Mer01]:

1. basic operations on sets—union, intersection, complement;

2. computation of the sets of successor and predecessor worlds of a current set of worlds;

3. verification of inclusion and equality between two sets.

These classical algorithms unfortunately could not deal with realistic problems, which generally need huge numbers of worlds. *Symbolic* model checking [BC$^+$92, McM93] has been invented to overcome this obstacle. The idea is to directly handle *sets* of worlds, rather than individual worlds, by representing these sets as propositional formulæ. For example, given two propositional atoms p and q, the formula $p \wedge q$ represents the set of all worlds in which $p \wedge q$ holds. Assuming that the labeling function is injective[9] (two different worlds are not labeled by the same set of atoms), any set of worlds corresponds to at least one propositional formula. First works on symbolic model checking succeeded to handle models with more than 10^{20} worlds by representing these formulæ as OBDDs; let us explain why.

Symbolic Model Checking Using Knowledge Compilation

Using the notation of our knowledge compilation framework, we can see the atoms of $\mathcal{P}$ as Boolean variables, and thus express worlds as $\mathcal{P}$-assignments: a world w such that $L(w) = \{p, q, r\}$ corresponds to the unique $\mathcal{P}$-assignment in which p, q, and r are assigned to $\top$, and all the other variables are assigned to $\bot$. Finally, we can express a set of worlds S as a Boolean function of the form $f_S\colon \mathrm{Dom}(\mathcal{P}) \to \mathbb{B}$, that associates $\top$ with all $\mathcal{P}$-assignments corresponding to a world in S, and $\bot$ with all $\mathcal{P}$-assignments corresponding to a world *not* in S.

The trick used in symbolic model checking to represent transition relations is the *duplication* of variables. Each atomic variable $p \in \mathcal{P}$ is associated with another variable p', interpreted as the proposition p on *successor worlds*. Hence, while Boolean functions on variables from $\mathcal{P}$ represent sets of *current* worlds, Boolean functions on variables from $\mathcal{P}'$ represent sets of *next* worlds. A transition from a world $\vec{p}$ to a world $\vec{q}$ can thus be represented as the $\mathcal{P} \cup \mathcal{P}'$-assignment $\vec{p}\ .\ \vec{q}'$. Finally, the transition relation is simply a Boolean function F_T over variables from $\mathcal{P} \cup \mathcal{P}'$, of which the possible transitions are models, and the impossible transitions are countermodels.

All the elements of a Kripke structure that are used in model checking algorithms (i.e., sets of worlds and transition relation) are now represented as Boolean functions of Boolean variables. Depending on how they are represented, operations used by model checking algorithms may be computationally hard. It seems therefore interesting to make use of knowledge compilation on these elements.

In order to choose a language, we must identify the operations on Boolean functions that correspond to the operations on sets used by model checking algorithms.

1. Basic operations on sets are simply basic operations on Boolean functions: intersection is conjunction, union is disjonction, complement is negation. We thus need the following transformations: $\wedge$**BC**, $\vee$**BC**, $\neg$**C**.

[9] This can be done without loss of generality, by adding a sufficient number of "fresh" propositional atoms to distinguish similar worlds.

2. The set of successors of a set of worlds S is obtained with the following operation: $\exists\mathcal{P}.(f_S \wedge F_T)$. The set of predecessors of S' is $\exists\mathcal{P}'.(F_T \wedge f_{S'})$. We thus need the **SFO** transformation (the number of variables to forget is bounded).

3. Two sets S_1 and S_2 are equal if and only if $f_{S_1} \equiv f_{S_2}$. $S_1 \subseteq S_2$ if and only if $f_{S_2} \models f_{S_1}$. We thus need queries **SE** and **EQ**.

We need all these operations to be made online, that is, during the execution of the algorithm. According to Tables 1.1 and 1.2, reasons for the success of the $\mathtt{OBDD}^{\$\mathbb{B}}_{\mathcal{B}}$ language in symbolic model checking are quite clear; it is the only language supporting all necessary operations.

Finally, compiling the set of initial worlds W_0 and the transition relation T as OBDDs is easy, using for example the bottom-up fashion. With a good variable ordering, compiled forms can be exponentially smaller than explicit enumerations. Classical model checking algorithms adapted to OBDDs can thus be much more efficient. This is a first example of how knowledge compilation can allow a practical scaling obstacle to be overcome. Symbolic model checkers include the original SMV [McM93], its reimplementation NuSMV [CC+99], and its extension to SAT-based model-checking, NuSMV2 [CC+02].

1.7.2 Diagnosis

Diagnosis is an artificial intelligence field the purpose of which is to help determining whether a system behaves correctly, and if it is not the case, indicating the cause and origin of the malfunction. We focus here on *model-based diagnosis* [see e.g. HCK92], in which diagnoses are computed automatically online, leaning on a model of the system, as well as current *observations*.

The Diagnosis Problem

We consider a simplified framework, in which the system to be diagnosed is modeled as a propositional formula.

Definition 1.7.3. A *system description* is a triple $S = \langle \Delta, \mathcal{A}, \mathcal{O} \rangle$, where Δ is a propositional formula over variables $\mathcal{X}$, $\mathcal{A} \subseteq \mathcal{X}$ is the set of *assumable* variables (each one encoding the working condition of some component), and $\mathcal{O} \subseteq \mathcal{X} \setminus \mathcal{A}$ the set of *observable* variables (an assignment of which corresponds to a possible observation, or measure).

Formula Δ encodes the behavior of the system, in normal conditions, but also in presence of component failures. Each model of Δ corresponds to a possible situation, associating an observation with a "health state" of the system's components. For the sake of simplicity, this definition of a system description does not allow dynamic behavior to be modeled; we limit ourselves to static systems. The diagnosis problem that we study is, given an observation, to compute all the compatible health states—they are known as *consistency-based diagnoses*. We do not study

how these diagnoses are exploited; an example could be to identify component failure that is common to all diagnoses. We simply focus on the computation of the set of *consistency-based diagnoses*.

Definition 1.7.4. A *model-based diagnosis problem* is a couple $\langle S, \vec{o} \rangle$, where $S = \langle \Delta, \mathcal{A}, \mathcal{O} \rangle$ is a system description and $\vec{o}$ is an $\mathcal{O}$-assignment. The problem consists in computing all $\mathcal{A}$-assignments $\vec{a}$ such that $\vec{a} \ . \ \vec{o}$ is logically consistent with Δ (that is, $\Delta_{|\vec{a}.\vec{o}} \not\models \bot$).

The consistency-based diagnoses are the $\mathcal{A}$-assignments. Note that we ignore variables from $\mathcal{X} \setminus (\mathcal{O} \cup \mathcal{A})$, called the *nonobservable variables*: we want $\vec{a} \ . \ \vec{o}$ to be consistent with Δ, but we do not want it to be a model of Δ.

Our diagnosis problem can be hard, depending on how Δ is represented. Yet, we want to be able to obtain diagnoses online. It hence seems interesting to use knowledge compilation.

Knowledge Compilation for Model-Based Diagnosis

Knowledge compilation is well-adapted to model-based diagnosis. Different languages have been used throughout the years, from PI [De 90] to $\mathtt{OBDD}^{\$\mathbb{B}}_{\mathcal{B}}$ [SM96 for dynamic systems, TT03 for static systems] and DNNF [Dar01a, HD05b]. Let us apply knowledge compilation on our simplified diagnosis problem.

Of course, what we want to compile here is the model of the system: Δ can be a huge formula, and we know it never changes (it does not depend on observations). We hence translate Δ into a Boolean language, since it is a Boolean function on Boolean variables. The operations we need are the following:

1. *restriction* of Δ to the current observation $\vec{o}$: this is the **CD** transformation;
2. *existential projection* of the result on the assumable variables: this is the **SFO** transformation (on the non-observable variables);
3. finally, we must be able to retrieve diagnoses, *enumerating models* of the resulting formula: this is the **ME** query.

The knowledge compilation map (Tables 1.1 and 1.2) shows that $\mathtt{OBDD}^{\$\mathbb{B}}_{<,\mathcal{B}}$, $\mathtt{DNNF}^{\$\mathbb{B}}_{\mathcal{B}}$, PI, and $\mathtt{DNF}^{\$\mathbb{B}}_{\mathcal{B}}$ are suitable languages for this application, and that among these languages, the best one is probably $\mathtt{DNNF}^{\$\mathbb{B}}_{\mathcal{B}}$ (it is unsure whether $\mathtt{DNNF}^{\$\mathbb{B}}_{\mathcal{B}} \leqslant_s \mathtt{PI}$). The diagnosis problem we defined was very simplistic; previously cited works include applications of knowledge compilation to more complex problems, such as the computation of diagnoses involving as few faults as possible (minimum-cardinality diagnoses).

1.7.3 Product Configuration

Product configuration [Stu97, SW98a] is an important area of research about assisted decision, with many successful practical applications in industry. It takes

place between the seller and the buyer of a customizable product. The buyer can *configure* the product by deciding on his preferred values for a set of *attributes* (such as color or options). An assignment of all attributes is called an *alternative*. Generally, it is not possible to choose any alternative, notably because some options are not compatible. The customer nevertheless wants to be able to fix some attribute, then to see which values are available for the remaining attributes, to fix another attribute, and so on. The *product configurator* is the system that allows this, reasoning on a model of the customizable product.

The Configuration Problem

We define a basic configuration problem, adapted from Hadzic, Jensen, and Andersen [HJA07].

Definition 1.7.5. A *configuration problem* is a couple $\Pi = \langle C, \vec{a} \rangle$, where C is a constraint network $\langle V, \mathscr{C} \rangle$ [Def. 1.6.1], called the *catalog*, of which solutions are possible alternatives, and $\vec{a}$ is an assignment of a set of variables $A \subseteq V$, representing some *assumptions* made by the user. The problem consists in computing all consistent values (i.e., the context [Def. 1.4.3]) for each variable in $V \setminus A$, in the restriction of the catalog to alternatives compatible with $\vec{a}$.

It corresponds exactly to the requirements we stated in the previous paragraph. In practice, product configuration is more complex. For example, customer Bob needs to be able to relax any assumption he made, so that he can *restore* an option rendered unavailable by his previous choices. The configurator must be able to guide Bob through this, explaining why the choice is not possible, and indicating the best relaxation according to his defined preferences. Nonetheless, even our simplistic configuration problem can be computationally hard, depending on the way the catalog is represented.

Knowledge Compilation for Configuration

Our problem clearly shows the interest of using knowledge compilation to implement the configurator. There is a fixed knowledge base, possibly huge, viz., the catalog; and the only parameter that varies online is the set of assumptions. The catalog can be seen as a Boolean function on enumerated variables; as usual, we identify the queries and transformations needed online, to decide which Boolean languages are suitable for this application.

1. The *restriction* of C to the current set of assumptions $\vec{a}$ corresponds to the **CD** transformation.

2. The *enumeration of compatible values* can be done by checking, for each value of each variable, whether the function is still consistent when restricted to this variable assignment. This needs the **CD** transformation again, and the **CO** query. Another possibility would be to use **FO** and **ME**, or a specific *context extraction* query that we will define in Section 3.3.3.

These requirements are met by a number of Boolean languages on Boolean variables, notably PI and $\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$; they have been used to compile the configuration problem, using several Boolean variables to encode each enumerated domain, respectively by Sinz [Sin02] and Hadzic, Jensen, and Andersen [HJA07]. Suitable languages on enumerated variables notably include $\mathtt{OBDD}_{\mathcal{E}}^{\$\mathbb{Z}}$, used for configuration in the form of finite-state automata [AFM02]. Let us also cite Pargamin [Par03], who used *cluster trees* (which is not included in our knowledge compilation map). All these works use more complex frameworks than the one we defined—for example by assigning *weights* to constraints, to model user preferences or prices.

*
**

In this chapter, we presented knowledge compilation, a technique consisting in translating a problem offline to speed up its online resolution. We put the emphasis on the knowledge compilation map, detailing its concepts in a quite general framework. We presented a number of graph-based Boolean languages from the literature, classified as sublanguages of a wide-range language we introduced, GRDAG. Inspired from the VNNF framework, it notably allows all decision diagram languages to be grouped together, be their variables Boolean or not. Then, we gathered together the knowledge compilation results of all those languages. After presenting the best-known non-Boolean languages, we finally detailed practical aspects of knowledge compilation—compilation methods and existing compilers; the last section was dedicated to practical applications of knowledge compilation. We can now move on to the second chapter, which deals with the other AI field related to our topic, namely automated planning.

CHAPTER

2

Planning

An autonomous system needs to *make decisions* that fit its current situation and mission. Making a decision implies reasoning about the possible alternatives, their outcomes, and their combination, in order to select a good "first step" towards the success of the mission. By definition of an autonomous system, this reasoning cannot be made by some human operator. A possibility for an autonomous system to make decisions is to rely on *automated planning*.

In this chapter, we present a broad outline of automated planning, starting with a general, formal definition of the field [§ 2.1], then moving on to a description of various frameworks and techniques in planning, which we call *planning paradigms* [§ 2.2], and finally providing insight about the application of knowledge compilation to planning [§ 2.3].

2.1 General Definition

2.1.1 Intuition

From a general point of view, planning aims at answering the following question: "what should be done for a given objective to be achieved?". In *automated* planning, this question must be answered by an algorithm, called a *planner*. This first intuitive definition is deliberately quite vague, since there exists various forms of planning. Let us detail why it is ambiguous.

First of all, the answer to the question "what should be done?" obviously depends on what *can* be done, that is, on *possible actions*. This brings up the question of the *representation* of the world. In planning, the world is generally modeled as a *state-transition system*, with actions and events modifying its current state.

Second, the notion of "objective to be achieved" can designate various requirements. For example, it can be the finite, fixed goal of a mission (for example,

reaching a given place), or continuous, perpetual ones (for example, continuously following a given path). In the latter case, the goal is never *achieved*, but must be permanently ensured; planning problems with such objectives are called *control problems*. There can also be several goals, of various importance to the mission, in which case the planner must *optimize* its output with respect to given preferences.

The last ambiguity concerns the desired form of the answer. Do we want one sequence of actions? Do we need the choice of actions to depend on some condition? How robust should it be to uncertainties and aleas?

We discuss all of these ambiguities in the remainder of this section; the question of the representation of the world is addressed first [§ 2.1.2], then the other elements of a planning problem are detailed [§ 2.1.3]; last, we present different kinds of solutions [§ 2.1.4]. Let us emphasize that our overview is not meant to be exhaustive; the interested reader can consult the reference book by Ghallab, Nau, and Traverso [GNT04], or recent tutorials such as Rintanen's [Rin11] or Geffner's [Gef11].

2.1.2 Description of the World

When defining a planning problem, one obviously does not take the whole world into account, but rather wants to describe a number of elements of interest, with which the *agent* (i.e., the autonomous system concerned by the problem) can interact. The reasoning then only applies to these elements—the world is restricted to them. This section presents the way these elements are described, allowing them to be handled by automated reasoning.

Basic Model

The general model of the world is often a state-transition system [GNT04, § 1.4].

Definition 2.1.1. A *state-transition system* is a quadruple $\Sigma = \langle S, A, E, \gamma \rangle$, where

- S is a recursively enumerable set of *states*;
- A is a recursively enumerable set of *actions*;
- E is a recursively enumerable set of *events*;
- $\gamma \colon S \times A \times E \to 2^S$ is a *state-transition function*.

This model allows one to represent the dynamics of the world. The current state can be modified by an event, on which the agent has no control, or by an action, triggered by the agent. Events and actions are called *state transitions* (hence the name of the system). A state-transition system can be viewed as a multigraph the nodes of which are the states of the system, and the edges of which are the actions and events. The diagram of Figure 2.1 is a graphical representation of the state-transition system modeling the world of a quite simple planning problem.

The state-transition system model is generally not used in this basic form. There are numerous planning paradigms, each one making restrictive assumptions on the

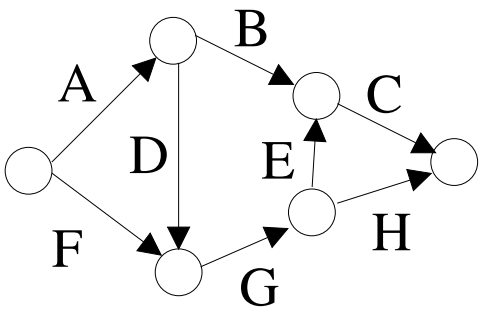

Figure 2.1: Elements of a planning problem. Circles represent states, and edges (A, B, C…) represent actions, that make the world go from one state to another.

model or enhancing it by adding information on some elements. These modifications are aimed either at making reasoning simpler, or at fitting the specific features of some application area.

Common Model Restrictions

Here are the most common restrictions of the basic definition of a state-transition system:

- *finite* sets of states, actions, and events;
- *static* system—there is no uncontrollable dynamics, E is a singleton containing the "empty" event;
- *deterministic* system, in which the effect of actions is known for sure—applying a given action in a given state always leads to the same state, that is, $\forall s \in S, \forall \langle a, e\rangle \in A \times E, |\gamma(s, a, e)| = 1$.

These assumptions are sometimes made all at once, as it is the case in *classical planning* [§ 2.1.3.3], but this is not necessarily true. Each planning paradigm has its own set of restrictions.

Common Model Extensions

Even when none of these restrictions is applied, the basic model of Definition 2.1.1 still has a quite limited expressivity. Planning paradigms often refine it, by adding information to one element or another.

- States can have various levels of *observability*, meaning that the current state may not be fully known by the system (this can even be the case of the initial state);
- state transitions can be *probabilistic*, inducing a hierarchy in the possible outcomes of an action [§ 2.2.4];

- actions can be made durative, enhancing the model so that time can be taken into account [§ 2.2.5.4].

All these modifications—restrictions and extensions—only affect the *theoretical* model of the world. In practice, a planner handles a *representation* of this abstract model. The choice of the concrete representation also differs from one planning paradigm to another.

Model Representation

There can be a huge number of states in a world. This number is not even necessarily finite in the general case. Yet, the planner must take a description of the world as input; it thus requires some *implicit representation* of the world, in which states are not enumerated *in extenso*.

A possible representation, based on propositional logic, expresses states as *assignments* of some *state variables* and transitions as *formulæ* over these variables. For example, let us consider two Boolean state variables *light-on* and *door-open*. Their assignments describe four possible states:

- *light-on* $= \bot$, *door-open* $= \bot$: the light is off and the door is closed;
- *light-on* $= \bot$, *door-open* $= \top$: the light is off and the door is open;
- *light-on* $= \top$, *door-open* $= \bot$: the light is on and the door is closed;
- *light-on* $= \top$, *door-open* $= \top$: the light is on and the door is open.

Actions are generally defined by two types of formulæ on state variables, called *preconditions* and *effects*. For a given action to be *executable* in a given state, the preconditions of this action must hold in this state. Once it has been applied, the world ends up in a state in which the effects hold.

Evolution of the world is often considered as a sequence of steps; each step corresponds to a given state, and thus to a given assignment of the state variables. In this case, state variables are called *fluents*, and preconditions and effects are usually represented as sets of fluents: those that must be true (preconditions), those that become true (positive effects), and those that become false (negative effects). An example of state-transition system expressed using this typical fluent formulation is given on Figure 2.2.

2.1.3 Defining a Planning Problem

We now have an idea of how the world is described in a planning problem. This is however not sufficient to fully define a problem; a given world model can be the basis for many different problems, depending on the initial state and the goal.

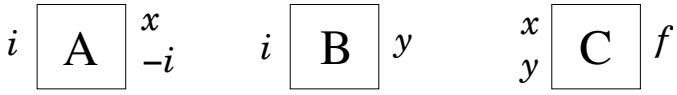

Figure 2.2: A state-transition system defined with fluents. Each square represents an action, with its preconditions on the left (fluents that must be true for the action to be executable), and its effects on the right (fluents that become true or false once the action has been executed).

Initial States

In a given state-transition system, it is of course possible to choose various states to be the initial one. For example, if the world model describes the possible paths of an automated vehicle, we can consider the problem of reaching a goal position starting from different locations. In classical planning problems defined with fluents, the initial state is specified by a set of fluents, namely the fluents that are true in this state. In the example of Figure 2.2, we could set the initial state to $\{i\}$; it means that initially, i is true, and all the other fluents are false.

It is not mandatory that there be a single initial state. Some specific planning frameworks, such as conformant planning [SW98b] or planning as model-checking [GT99], use multiple initial states; this can be a way to introduce some uncertainty in the problem, without using nondeterministic actions.

The Notion of Goal

Intuitively, the purpose of planning is to find a way for some desired state to be reached; but goals can actually be much more complex. The basic goals are called *reachability goals*; they are simply defined by a state, or a set of states. With the fluent representation, this set of states may be expressed as a set of fluents—the ones that are required to be true at the end.

With reachability goals, a plan can only fall into two cases: either it is a solution or it is not. But let us consider for example an explorer robot that must gather as many samples as possible. This goal is obviously not "reachable", but rather induces a classification among possible outcomes; a solution is thus a plan that leads to the *preferred outcome*.

There are other examples that do not even involve valuation; suppose the explorer robot must gather samples, such that it always stay within the range of the radio control, so that it can come back to the station whenever it is asked to. Here the goal is a *condition* that must be fulfilled during the whole execution. Such goals are called *extended goals* [DLPT02, PT01].

Planning Problems

Planning problems are defined using the various elements we presented before: a generic planning problem is defined as a state-transition system, an initial set of states, and a goal. Problems are classified with respect to the type and properties of their elements. A *classical* planning problem is thus defined as follows [FN71].

Definition 2.1.2. A *classical planning problem* is a triple $P = \langle \Sigma, s_0, G \rangle$, where $\Sigma = \langle S, A, \gamma \rangle$ is a finite, static, and deterministic state-transition system, $s_0 \in S$ is the initial state, and $G \subseteq S$ is the set of goal states.

Relaxing one of the state-transition system's restrictions, extending it, or using extended goals, raises different kinds of planning problems. Except for classical planning, categories of planning problems have generally no name of their own; they are named after the paradigm that uses them, like "MDP planning problem" [§ 2.2.4], or simply described extensively, like "problem of nondeterministic planning with reachability goals under partial observability".

With the purpose of unifying planning problem representations, and accounting, in a clear and normalized way, for respective abilities of different planning algorithms, various *description languages* have been created. The first to be widely used was the STRIPS language; it was designed to express entries for the Stanford Research Institute problem solver [FN71], and as such, only aimed at representing problems that the solver couldn't handle.

STRIPS has been extended by ADL (action description language) [Ped89], which can express disjunctions of fluents in preconditions and effects. The current standard is PDDL (planning domain definition language) version 3.1 [see Kov11 for a complete specification]. It is more expressive than STRIPS and ADL, notably allowing representation of durative actions, numeric fluents, action costs, preferences over states, and derived predicates. An extension of PDDL, named PDDL+, allows mixed (discrete and continuous) domains to be handled [FL06].

2.1.4 Solutions to a Planning Problem

As we saw, there are multiple kinds of planning problems. Let us now define what a *solution* to a given planning problem is.

Form of Solutions

There are various *forms* of possible solutions. The basic solution to a planning problem is simply a sequence of actions, called a *plan*.

Definition 2.1.3 (Plan)**.** Let $\Sigma = \langle S, A, \gamma \rangle$ be a state-transition system. A *plan* π for Σ is a finite sequence of actions from A: $\pi = \langle a_1, \ldots, a_n \rangle \in A^n$, with $n \in \mathbb{N}$ the *horizon* of π.

Not all plans are solutions. Some are not even feasible—this is formally addressed in the next subsection. Intuitively, a plan is a solution to a problem if it "allows the goal to be reached" when the actions are executed in the given order, starting from the initial state. For example, taking the classical planning problem defined in Figure 2.2, if the initial state is defined by $\{i\}$, a solution plan to the problem of reaching a state where f is true is $\langle B, A, C \rangle$.

There exists various kinds of plans: they can be either *sequential* or *parallel* (if actions are allowed to be executed simultaneously). If the actions to be executed depend on observations, plans are called *conditional*. Different forms of solutions than plans also exist, namely *policies*.

Definition 2.1.4 (Decision policy)**.** Let $\Sigma = \langle S, A, \gamma \rangle$ be a state-transition system. A *decision policy* δ for Σ is a function $\delta \colon S \to A$. Policy δ may be *partial*, that is, not defined on the entire set of states.

Contrary to solution plans, that provide an ordered sequence of actions to execute, a policy is a *function* going from the set of states to the set of actions. It actually yields the next decision to make, given the current state. Once this decision has been executed, the system has to ask the policy once more to determine the next action, etc.

Policies are more robust to uncertainty than solution plans: during execution, if one of the decisions made has not had the expected effect, a plan may become useless (if remaining decisions do not fit the unexpected state). Using a policy, even if the current state is unexpected as an effect of last decision, a good action for that state may still be available.

The form of the expected solution to a problem thus greatly depends on the model: policies are suitable to nondeterministic problems, but require a modification of the model (adding special state variables accounting for the "history" of executed actions) to allow fixed sequencing of actions. Conditional plans are suitable to partially observable problems, but they do not enable a given action to be repeated indefinitely until some given state is reached.

Feasibility of Solutions

In order to define what a *solution plan* and a *solution policy* are, we have to express the fact that plans and policies are not necessarily *executable*, or *feasible*. Intuitively, feasibility means that the plan or policy is compatible with the transitions of the model. The formal definition of a feasible plan [adapted from FG00] involves the notion of *history*.

Definition 2.1.5 (History)**.** Let $\Sigma = \langle S, A, \gamma \rangle$ be a state-transition system, and $\pi = \langle a_1, \ldots, a_n \rangle$ a plan for Σ. A *history for* π is a sequence of states $\langle h_0, \ldots, h_k \rangle \in S^k$, with $0 \leqslant k \leqslant n$, verifying

$$\forall i \in \{1, \ldots, k\}, h_i \in \gamma(h_{i-1}, a_i).$$

State h_0 is called the *starting state* of this history, h_k its *ending state*, and k its *size*.

A history is thus a sequence of states that is compatible with plan π: applying this plan in the starting state *may* lead to the ending state.

Definition 2.1.6 (Feasible plan)**.** Let $\Sigma = \langle S, A, \gamma \rangle$ be a state-transition system, and $s_0 \in S$. A plan $\pi = \langle a_1, \ldots, a_n \rangle$ for Σ is said to be *feasible from* s_0, or *(always) executable from* s_0, if and only if any history for π of size $k < n$ and starting from s_0 is such that $|\gamma(h_k, a_{k+1})| > 0$.

A plan is feasible from a given initial state if the first action is applicable in the initial state, and the second action is applicable in any of the possible resulting states, etc. For a plan to be feasible, there must be no history leading to a state in which the current action is not executable.

To define histories on solution policies, we need to introduce the following concept of *execution structure*.

Definition 2.1.7 (Execution structure)**.** Let $\Sigma = \langle S, A, \gamma \rangle$ be a state-transition system, and δ a policy. The *execution structure* induced by δ on Σ is the set $\Sigma_\delta = \{ \langle s, s' \rangle \in S \times S \mid s' \in \gamma(s, \delta(s)) \}$.

An execution structure is thus a simplified state-transition system, in which no action is left; it can be viewed as a directed graph the edges of which are the transitions that are compatible with the policy. We can now define histories for decision policies.

Definition 2.1.8 (History)**.** Let $\Sigma = \langle S, A, \gamma \rangle$ be a state-transition system, $s_0 \in S$, and δ a policy. A *history of δ from s_0* is a complete path in the execution structure Σ_δ that starts in state s_0.

Note that contrary to histories for plans, histories for policies may be infinite. Note also that they are *complete paths*, that is, they can only end in states for which the policy does not provide any action to apply. We use the notion of history in the next subsection; for now, let us define *executable policies* [GT99].

Definition 2.1.9 (Executable policy)**.** Let $\Sigma = \langle S, A, \gamma \rangle$ be a state-transition system. A policy δ for Σ is said to be *executable* if and only if for every couple $\langle s, a \rangle$ such that $a = \delta(s)$, there exists a state $s' \in S$ verifying $s' = \gamma(s, a)$.

A policy is executable if each action it returns is executable in the current state. Note that this definition is simpler than that of a feasible plan.

Strength of Solutions

Being executable is a necessary condition for a plan or policy to be a solution to a given planning problem. It is not sufficient, of course; the second requirement is that it allows the goal to be reached.

Definition 2.1.10 (Classical planning solution)**.** Let $P = \langle \Sigma, s_0, G \rangle$ be a classical planning problem. A plan π for Σ is a *solution* to P if and only if it is feasible, and there exists a history for π from s_0 that ends in one of the goal states $g \in G$.

For a non-classical problem, the definition of a solution is actually not unique; indeed, this problem can be more or less prone to *uncertainty*, for example through nondeterminism or partial observability. Requirements regarding the soundness of the solution may not always be the same. For some applications, the goal must be reached in all cases in spite of nondeterminism: a *strong* solution is hence sought. For some other applications this soundness is not strictly necessary; it is possible to be optimistic and seek a *weak* solution.

The strength of a solution plan is related to the notion of *history*, that we defined in the previous subsection. Various types of solution can be defined for each category of planning problems; let us give some examples, which we refer to in the following. The simplest kind of solution is the *weak* one, that only requires the *existence* of a history [GT99].

Definition 2.1.11 (Weak solution)**.** Let $P = \langle \Sigma, S_0, G \rangle$ be a planning problem with several initial states and reachability goals. A plan π (resp. a policy δ) for Σ is a *weak solution* to P if and only if it is executable from s_0, and for all $s_0 \in S_0$, there exists a history for π (resp. δ) from s_0 that ends in one of the goal states $g \in G$.

For a policy to be a weak solution, there simply must exist at least one path from each initial state to one of the goal states in the execution structure.[1] For it to be a strong solution, there must *not* exist any path not leading to a goal state [CRT98b].

Definition 2.1.12 (Strong solution)**.** Let $P = \langle \Sigma, S_0, G \rangle$ be a planning problem with several initial states and reachability goals. A policy δ for Σ is a *strong solution* to P if and only if it is executable, and for all $s_0 \in S_0$, all histories for δ from s_0 are finite and end in one of the goal states $g \in G$.

When referring to plans, strong solutions are called conformant solutions.

Definition 2.1.13 (Conformant solution)**.** Let $P = \langle \Sigma, S_0, G \rangle$ be a planning problem with several initial states and reachability goals. A plan π for Σ, of horizon n, is a *conformant solution* to P if and only if for all $s_0 \in S_0$, it is feasible from s_0, and all histories for π of size n and starting from s_0, end in one of the goal states $g \in G$.

Conformant solutions are generally sought for fully non-observable problems, in which the plan must be valid even though the system has no idea of the state it is in [see e.g. SW98b]. If the problem is (partially) observable, this kind of solution is rather called *contingent* [DHW94]. Let us finally define a midterm between weak and strong solutions [CRT98a].

Definition 2.1.14 (Strong cyclic solution)**.** Let $P = \langle \Sigma, S_0, G \rangle$ be a planning problem with several initial states and reachability goals. A policy δ for Σ is a *strong cyclic solution* to P if and only if it is executable, and for all $s_0 \in S_0$, all finite histories for δ from s_0 end in one of the goal states $g \in G$.

Note the difference with strong solutions: there can be infinite histories, but they are not taken into account. The requirement is only that finite histories cannot end in a non-goal state. The idea is to consider infinite loops to be highly unlikely—a strong cyclic solution hence being able to reach the goal "almost certainly".

2.2 Planning Paradigms

Throughout the years, many techniques have been used to tackle planning problems. Some techniques aim at solving specific categories of problems; some others endeavor to be as generic as possible. In this section, we give some insight about several planning paradigms that are particularly related to our subject.

[1] Without loss of generality, we consider goal states to be dead-end states, in which no action is possible.

2.2.1 Forward Planning in the Space of States

The simplest planning paradigm consists in reasoning directly on the state-transition system, that is, using a *state-space search* [see e.g. GNT04, Chapter 4]. The idea is to try to reach a goal, starting from the initial state; in other terms, to find a *path*, in the graph representing the state-transition system, from the initial state to a goal state. If there is no such path, the problem has no solution. If there is such a path, the plan consisting of the corresponding sequence of actions is a solution to the problem.

Algorithm 2.1 Forward state-space search.

```
1: function: ForwardSS(P, s)
2: input: a classical planning problem P = ⟨Σ, s_0, S_G⟩
3: input: a current state s
4: output: a plan π from s to the goal if there exists one, and nil otherwise
5: if s ∈ S_G then
6:     return the empty plan
7: let A_s be the set of actions a that verify |γ(s, a)| ≠ 0
8: while A_s ≠ ∅ do
9:     pick some a in A_s (and remove it)
10:    let s' := γ(s, a)
11:    let π := ForwardSS(P, s')
12:    if π ≠ nil then
13:        return a . π
14: return nil
```

Algorithm 2.1 presents this procedure using a recursive fashion. This approach can also be used backwards—starting from the goals and trying to reach the initial state. The drawback of this rather straightforward technique is that the search space is generally huge: in the worst case, *all possible sequences of actions* are explored. As a consequence, this paradigm is never used as such, but with *heuristics* for the choice of the next action [see Section 2.2.5.2], or with restrictions on possible actions, as was done in the STRIPS algorithm [FN71].

2.2.2 Planning as Satisfiability

The idea behind the *planning as satisfiability* [KS92b] paradigm is to solve planning problems using SAT solvers (programs able to decide whether a given propositional formula is satisfiable). Indeed, the constant improvement of these solvers is the aim of a very lively branch of artifical intelligence; efficient algorithms have been developed, and it is possible to take advantage of their performance in a planning context.

To do so, a planning problem $P = \langle \Sigma, s_0, g \rangle$ is encoded as a propositional formula φ, such that any model of φ corresponds to a solution plan to P. The

usual method is to decompose the horizon into *steps*, each step corresponding to the application of some action in some state of the world. There must thus be specific state and action variables for each step, the assignment of which indicates the current state and the action applied at that step. This obviously means that we cannot deal with an unbounded horizon: this would imply an infinite number of states and actions—and hence an infinite propositional formula.

The planning problem is therefore restricted to a more constrained problem: finding a plan *of horizon* n (with n some fixed integer). The number of steps is then finite; each action variable a is duplicated into n versions $a_0, a_1, \ldots, a_{n-1}$, and each state variable s into $n+1$ versions $s_0, s_1, \ldots, s_n$. The encoding of the problem is now straightforward. We illustrate it on a classical planning problem P; in this case, the encoding φ_P^n is basically the conjunction of formulæ corresponding to the following requirements:

- the assignment of state variables at step 0 must represent the initial state;
- the assignment of state variables at step n must represent the goal state;
- for all i between 0 and $n-1$, denoting a_i the action represented by the assignment of action variables at step i, state variables at step i must respect the preconditions of a_i, and state variables at step $i+1$ must respect its effects.

This is not all; this encoding does not prevent state variables that are not modified by the current action from changing, whereas it is supposedly forbidden by the *frame axiom*—and it does not even prevent several actions from being applied at the same step. An actual encoding must of course include formulæ forbidding this kind of behaviour.

The final formula φ_P^n is equivalent to the original bounded planning problem, in the sense that every model of the formula is a solution to the problem. We can indeed extract a plan from a model $\vec{x}$: it is simply the sequence of actions $\langle a_0, \ldots, a_{n-1} \rangle$—let us denote this plan by $\pi_{\vec{x}}$. By definition, this plan is a solution to the planning problem, and the reverse is also true, as summed up in the following proposition [KS92b].

Proposition 2.2.1. Let P be a classical planning problem, and φ_P^n its propositional encoding (following the rules described above). Let $\vec{x}$ be a Scope(φ_P^n)-assignment; it holds that $\vec{x} \in \text{Mod}(\varphi_P^n)$ if and only if $\pi_{\vec{x}}$ is a n-step solution plan to P.

To find a solution plan using this property, one thus just has to run a SAT solver on φ_P^n, with increasing values of n (that is, starting from $n = 1$, run the solver, and and while no solution is found, increment n and start again). Many refinements can be made to that paradigm, based for example on the kind of SAT solver used, the encoding of actions, or the encoding of the frame axiom [KMS96, GMS98]. This approach has notably been extended to conformant planning [FG00, CGT03].

2.2.3 Planning as Model-Checking

Planning using model-checking [CG$^+$97, GT99] is a paradigm aiming at solving nondeterministic planning problems. It is based on the fact that an execution structure (i.e., a state-transition system "restricted" by some policy [Def. 2.1.7]), corresponds exactly to a Kripke structure [Def. 1.7.1], given that states are encoded using propositional variables. For the policy to be a weak solution to the planning problem, it is sufficient that there exist a path from the initial state to the goal state. For the policy to be a strong solution, all paths starting from the initial state must lead to the goal state. Basically, it is possible to use a CTL formula to express the fact that the policy is a solution; hence, verifying whether the policy is a solution boils down to applying *model checking* on the execution structure.

This enables one to use model-checking techniques to solve a planning problem; but this is not the only asset of this approach. It indeed allows one to apply the expressivity of CTL to planning, and thus deciding on the strength of the desired plan [CG$^+$97 for weak planning, CRT98b for strong planning, CRT98a for strong cyclic planning], but also to look for plans fulfilling a continuous condition, such as "keep the radio contact" [PT01]. Let us illustrate this approach by focusing on strong planning.

Definition 2.2.2 (Execution structure as a Kripke structure)**.** Let $P = \langle \Sigma, S_0, S_g \rangle$ be a planning problem, and δ a policy on $\Sigma = \langle S, A, \gamma \rangle$. The Kripke structure corresponding to the execution structure Σ_δ is $K_{P,\delta} = \langle W, W_0, T, L \rangle$, with

- $W = S$;
- $W_0 = S_0$;
- $T = \{ \langle w, w' \rangle \in W^2 \mid w' \in \gamma(w, \delta(w)) \}$;
- L the function associating with each world w (which is also a state) the state variable assignment $\vec{s}$ representing w.

As previously stated, the fact that δ be a strong solution to a planning problem P is equivalent to the fact that a certain property hold on $K_{P,\delta}$. This property is the CTL formula **AF**g, where g is the formula on state variables that holds on states in S_g, and **AF** is the CTL operator meaning "all paths eventually lead to a world where the formula is true" [GT99].

Proposition 2.2.3. Let $P = \langle \Sigma, S_0, S_g \rangle$ be a planning problem, and δ a policy on $\Sigma = \langle S, A, \gamma \rangle$. δ is a strong solution to P if and only if $K_{P,\delta} \models$ **AF**g.

Using this property, it is possible to take advantage of a model-checking algorithm to build policies respecting the requirements.

2.2.4 Planning Using Markov Decision Processes

Using the standard state-transition system of Definition 2.1.1, there is no way to state that one possible outcome of a nondeterministic action is more likely to happen

than another. Representing the state-transition system as a Markov decision process [Bel57] is a way of quantifying nondeterminism using probabilities.

Definition 2.2.4. A *Markov decision process (MDP)* is defined as a quadruple $\Sigma = \langle S, A, P, R \rangle$, where

- S is a finite set of states;
- A is a finite set of actions;
- $P\colon S \times A \times S \to [0,1]$ is a probability distribution over the state transitions. $P(s,a,s')$ is usually denoted as $P_a(s,s')$, and represents the probability of reaching state s' when executing action a in state s. It must verify

$$\forall s \in S, \forall a \in A, \quad \sum_{s' \in S} P(s,a,s') \ \in \ \{0,1\}$$

  (the sum equals 0 if a is not executable in s, and 1 otherwise);
- $R\colon S \times A \times S \to \mathbb{R}$ is the reward function. $R(s,a,s')$, often denoted $R_a(s,s')$, is the immediate reward earned by the agent when it reaches state s' after having executed action a in state s.

Because of nondeterminism, on a given MDP, a policy δ that associates a *single* action with *each* state can have several outcomes. Thanks to the probabilities on transitions, it is possible to quantify the likelihood of outcomes. An outcome of a policy applied on an MDP[2] is called a *history*, as in the case of non-quantified transitions [Definition 2.1.8]. It is simply an infinite sequence of states, e.g. $h = \langle s_1, s_4, s_6, s_6, s_1, \ldots \rangle$. Denoting $h = \langle h_i \rangle_{i \in \mathbb{N}}$, the probability of h induced by δ is given by $P(h \mid \delta) = \prod_{i \in \mathbb{N}} P_{\delta(h_i)}(h_i, h_{i+1})$.

The reward function can be used as the "goal" in an MDP; it is used to classify histories with respect to their desirability. Rewards indeed allow *utility functions* to be defined on histories: $V(h \mid \delta) = \sum_{i \in \mathbb{N}} \gamma^i R_{\delta(h_i)}(h_i, h_{i+1})$, with γ being a *discount factor* (i.e., a parameter in $[0,1[$ ensuring that the utility of infinite histories is finite—more precisely, it makes first rewards count more than later ones). Thus, in practice, the reward function can for example indicate states that must be avoided (by associating them with a negative reward), or states that must be visited (by associating them with a high reward).

Putting together these two properties of a history, we can compute the *expected utility* of a policy, by summing utilities of all possible histories balanced with their probabilities:

$$E(\delta) = \sum_{h \in S^{\mathbb{N}}} P(h \mid \delta) \cdot V(h \mid \delta),$$

with $S^{\mathbb{N}}$ being the set of all possible histories.

[2] When associated with a policy, an MDP becomes a simpler object, called a *Markov chain*. It is basically an execution structure [Definition 2.1.7] with probabilistic transitions.

This finally allows us to define what an MDP planning problem is.

Definition 2.2.5. An *MDP planning problem* is simply defined by an MDP $\Sigma = \langle S, A, P, R \rangle$. A *solution* to an MDP planning problem is a policy δ^* verifying $E(\delta^*) = \max_{\delta \in A^S} E(\delta)$.

In other words, solving a planning problem expressed as an MDP consists in finding an *optimal* policy, that maximizes the expected utility.

2.2.5 More Paradigms

There are numerous other planning paradigms. We rapidly present a few more in this section; the interested reader can refer to the comprehensive manual by Ghallab, Nau, and Traverso [GNT04].

Planning in the Space of Partial Plans

Plan-space planning [Sac75] is a classical planning paradigm which consists in searching in the space of partial plans, instead of in the space of states. There is no *current state*: the reasoning is at a global level, on the whole horizon. Search proceeds by identifying actions that should appear in the solution plan, trying to add *ordering constraints*. Backtracking is guided by information about the actions, explaining how they depend on one another, using *causal links* and *variable bindings*.

Planning as Heuristic Search

The idea underlying *planning as heuristic search* [BG01, HG00, HN01] is to use heuristics to guide a state-space search. Typical heuristics involve the computation of some *distance to the goal*; it is a hard problem in general, so the distance is often not calculated on the real problem, but on a *relaxation* of the problem—for example, ignoring negative effects of actions.

Hierarchical Task Network Planning

Hierarchical task network (HTN) planning [EHN96] is a paradigm to solve problems that are different from the ones we have seen so far. Indeed, goals are not defined as states, conditions, or reward functions, but as *tasks* to be done. An HTN problem consists of a classical planning problem, together with a set of *methods*, that are "recipes" explaining how to decompose some tasks into simpler subtasks, themselves being decomposed into subtasks, etc. The simplest, indivisible tasks are called *primitive tasks*, and correspond to actions. The interest of this paradigm is that the definition of methods allows one to discard irrelevant sequences of actions—this is a means of incorporating human expertise into the planning model.

Temporal Planning

The purpose of *temporal planning* is to take into account the duration of actions, and in particular to define more complex conditions and effects of actions—such as conditions that must hold during the whole execution, or effects that hold only at a specific point in time. Temporal planning is a wide research area, involving multiple frameworks; let us cite the work by Cushing et al. [CK$^+$07] that presents an interesting hierarchy of sublanguages of PDDL depending on their temporal abilities.

Resource Scheduling

Scheduling is a branch of artificial intelligence studying problems of resource and time allocation for a number of tasks. Typically, for a task to be done, a fixed set of *activities* must be applied; each activity uses some resources and takes a certain time to be done. The objective is then to find a certain *schedule* for activities, that is, a plan indicating *when* activities must be executed and *which resources* must be allocated to them. The schedule must respect some constraints, such as not sharing a given resource between two activities, not exceeding the allotted time, etc. Scheduling is often separated from planning, but the two areas are converging, since many practical problems are neither pure scheduling nor pure planning ones [see e.g. SFJ00].

2.3 Knowledge Compilation for Planning

After this general overview of automated planning, let us explain how some of the paradigms we presented take advantage of knowledge compilation. We denote as $\mathcal{S}$ and $\mathcal{A}$ the sets of state and action variables, respectively.

2.3.1 Planning as Satisfiability

Classical Planning

Barrett [Bar03] addressed a deterministic, fully observable planning problem for real-time embedded systems with varying goals. He proposed to use an online planning as satisfiability approach. The solution plan is not computed offline once and for all; a new plan is made for each current state, which makes the autonomous system more robust to aleas.

Of course, solving a SAT problem is not tractable in the general case; the author made use of knowledge compilation, compiling the problem into $\mathtt{DNNF}_{\mathcal{B}}^{\$\mathbb{B}}$ [§ 1.3.4]. The work left to the online phase is the conditioning of the compiled problem by current observations, the forgetting of intermediary states, and the extraction of a model to obtain a solution plan. Algorithm 2.2 presents this procedure. It has been adapted from the original work to show more clearly which queries and transformations are actually used.

Algorithm 2.2 The online "planning as satisfiability" procedure proposed by Barrett [Bar03] for real-time embedded systems.

1: **input:** a Boolean DNNF φ representing a planning problem $\left\langle \langle S, A, \gamma \rangle, S, S_G \right\rangle$ (with no specified initial state) on horizon n
2: **input:** an $\mathcal{S}_0$-assignment $\overrightarrow{s_0}$ of the state variables at step 0
3: **output:** a solution plan
4: compute a DNNF φ' representing $[\![\varphi]\!]_{|\overrightarrow{s_0}}$ *// condition φ by $\overrightarrow{s_0}$*
5: compute a DNNF φ'' representing $\exists \mathcal{S}.[\![\varphi']\!]$ *// forget the intermediary states*
6: **return** a model of φ''

A problem with this approach is that the plan is always of length n, since the planning problem is encoded with a fixed horizon n. A workaround is to select in priority models in which actions are performed early.

Conformant Planning

Palacios et al. [PB$^+$05] studied the possibility of using the planning as satisfiability paradigm to solve conformant planning problems. More precisely, they aimed at seeking conformant parallel plans for a planning problem with deterministic actions and uncertainty on the initial state. In the classical planning as satisfiability framework, each model of the formula encoding the problem is a solution; this is not the case here, since the plan must lead to the goal *whatever the initial state may be*. Usual SAT solving procedures do not allow this requirement to be fulfilled. Indeed, they work as follows: after a value has been chosen for a variable, inconsistent assignments of other variables are discarded. However, it is not guaranteed that *all* remaining assignments correspond to strong solutions.

In order to find conformant solution plans, the authors proposed to use a pruning step based on *validity* with respect to the initial states. A partial assignment of the action variables indeed corresponds to a partial plan; if this partial plan does not cover all possible initial states, it is useless to try to assign remaining variables—the partial plan is said to be invalid, and can be pruned. The problem is that testing whether a partial plan is valid with respect to the initial states is hard in general; the authors hence identified a target language for the encoding that allows this operation in polytime.

Checking this specific validity boils down to existentially project the current assignment on the initial state variables, and count models: indeed, if the number of models is less than the number of initial states, it means that some initial state is not covered. The operations needed on the compiled form are thus **FO** and **CT**; the authors proposed to use the $\mathtt{d\text{-}DNNF}_{\mathcal{B}}^{\mathbb{SIB}}$ language to encode the planning problem in their solver, using a special decomposition tree during compilation, that ensures that the forgetting operation maintains determinism on $\mathtt{DNNF}_{\mathcal{B}}^{\mathbb{SIB}}$. The procedure is described in Algorithm 2.3.

Algorithm 2.3 The conformant planner of Palacios et al. [PB$^+$05].

input: a Boolean d-DNNF φ, representing some given planning problem $\langle\langle S, A, \gamma\rangle, S_0, S_G\rangle$ on horizon n
output: a conformant plan
while some action variable is not determined **do**
    select an action variable a_i *// see paper for selection details*
    compute a d-DNNF φ' representing $[\![\varphi]\!]_{|a_i=\top}$ *// assign the action variable to* $\top$
    if the number of models of $\exists\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}.[\![\varphi']\!]$ is equal to $|S_0|$ **then**
        assign a_i to $\top$ in the current plan
        $\varphi \coloneqq \varphi'$
    else
        let φ' be a d-DNNF representing $[\![\varphi]\!]_{|a_i=\bot}$ *// assign the action variable to* $\bot$
        if the number of models of $\exists\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}.[\![\varphi']\!]$ is equal to $|S_0|$ **then**
            assign a_i to $\bot$ in the current plan
            $\varphi \coloneqq \varphi'$
        else
            backtrack, unassigning some action variable

2.3.2 Planning as Heuristic Search

Bonet and Geffner [BG06] applied knowledge compilation to planning in an original way. They studied deterministic planning with penalties and rewards associated with fluents (the classical approach having been to associate them only with actions); this allows planning problems to be modeled without real goals, but rather with preferences among the possible outcomes, like it is the case in MDP planning.

They presented a rather simple heuristics, that roughly corresponds to the cost of the optimal plan for the relaxed problem (that is, ignoring negative effects of actions). The inconvenient of this heuristics is that hard computations are necessary in each visited state. The proposed solution is to compile the relaxed problem into a Boolean d-DNNF, using different encodings à la planning as satisfiability; the preferences are simply associated with each literal in the formula. From there, the computation of the heuristic value in each state can be done in time linear in the compiled structure (this uses a result from Darwiche and Marquis [DM04], applying knowledge compilation to weighted bases—we do not address this part of the knowledge compilation field in this thesis). They obtained good practical results, but had to use workarounds to compile some problems, because regular compilation took too much time or memory.

2.3.3 Planning as Model-Checking

The planning as (symbolic) model-checking paradigm has, since the beginning [CG$^+$97], taken advantage of knowledge compilation, through the use of OBDDs.

Weak Planning

The first "planning as symbolic model-checking" algorithm, designed by Cimatti et al. [CG$^+$97], aimed at solving weak planning problems. The procedure is described in Algorithm 2.4. It starts from the initial states, and recursively computes the set of reachable states. It stops when one of the goals is reached—or when it reaches a fixed point, in which no state is added anymore. In practice, the authors used the SMV model-checker [McM93], which itself relies on the $\texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ language; however, Algorithm 2.4 does not specify any language. It can practically be used with any Boolean language, as long as the corresponding operations are supported, viz., $\wedge$**BC**, **FO**, **CO**, and **EQ**.

Algorithm 2.4 Weak planning as model-checking algorithm with forward search.

input: a planning problem $\langle\langle S, A, \gamma\rangle, S_0, S_g\rangle$
output: 1 if there exists a weak solution, and 0 otherwise
initialize φ so that $\text{Mod}(\varphi) = S_0$
initialize G so that $\text{Mod}(G) = S_g$
initialize T so that $\text{Mod}(T) = \{\, \langle \vec{s}, \vec{a}, \vec{s}' \rangle \in S \times A \times S' \mid \vec{s}' \in \gamma(\vec{s}, \vec{a}) \,\}$
repeat
    if $[\![\varphi]\!] \wedge [\![G]\!]$ is consistent **then** *// at least one of the reached states is a goal state*
        return 1
    $\varphi_{\text{prec}} \coloneqq \varphi$
    let φ' represent $\exists\mathcal{S}.\exists\mathcal{A}.([\![\varphi]\!] \wedge [\![T]\!])$ *// the set of reachable next states*
    $\varphi \coloneqq \varphi'_{|\mathcal{S}\leftrightarrow\mathcal{S}'}$ *// next states are replaced by current states*
until $\varphi \equiv \varphi_{\text{prec}}$ *// we repeat while some state is added*
return 0

Algorithm 2.4 only checks whether a solution exists; to exhibit a solution, the authors added another procedure (which we do not detail here) going backwards in the successive sets of states.

Strong Planning

Cimatti, Roveri, and Traverso [CRT98b] used the planning as model-checking approach to build strong plans, that is to say, plans ensured to always be valid despite non-determinism. The procedure is different from the previous one because it uses a *backward search*: instead of computing a set of reachable states, it builds a set of *states to reach*, starting from the goals. First, it computes the set of state-action pairs ensuring that the goal be reached in one step, add the corresponding states to the set of "states to reach", then computes the set of state-action pairs ensuring that one of these states be reached in one step, etc. The procedure is presented in Algorithm 2.5; the authors implemented it using Boolean OBDDs, but once again, any Boolean language supporting the necessary operations can fit.

Algorithm 2.5 keeps in δ the current policy, and in φ the set of "covered states", that is, the states from which the current policy is *ensured* to lead to one of the goals.

Algorithm 2.5 Strong planning as model checking algorithm with backward search.

1: **input:** a planning problem $\langle\langle S, A, \gamma\rangle, S_0, S_g\rangle$
2: **output:** a strong solution policy
3: initialize I so that $\text{Mod}(I) = S_0$
4: initialize φ so that $\text{Mod}(\varphi) = S_g$
5: initialize T so that $\text{Mod}(T) = \{\, \langle \vec{s}, \vec{a}, \vec{s}' \rangle \in S \times A \times S' \mid \gamma(\vec{s}, \vec{a}) = \vec{s}' \,\}$
6: initialize δ to $\bot$
7: **repeat**
8: **if** $\varphi \models I$ **then** *// all initial states are covered*
9: **return** δ
10: $\varphi' \coloneqq \varphi_{|\mathcal{S}' \leftrightarrow \mathcal{S}}$ *// current states are replaced by next states*
11: $\delta_{\text{step}} \coloneqq \neg\varphi \wedge \forall \mathcal{S}'.(T \rightarrow \varphi')$ *// compute the state-action pairs ensuring to lead to a covered state*
12: $\delta \coloneqq \delta \vee \delta_{\text{step}}$
13: $\varphi_{\text{step}} \coloneqq \exists \mathcal{A}.\delta_{\text{step}}$ *// compute the set of newly covered states*
14: $\varphi \coloneqq \varphi \vee \varphi_{\text{step}}$
15: **until** φ_{step} is inconsistent *// we repeat while some state is added*

Indeed, the states added to φ are those for which there exists an action (line 13) such that all resulting states are already covered (line 11).

2.3.4 Planning with Markov Decision Processes

Hoey et al. [HS$^+$99] explored the compilation of MDPs. Their approach was similar to what is done in planning as model-checking: representing an MDP using Boolean variables to express the current state $\vec{s}$, the chosen action $\vec{a}$, and the state $\vec{s}'$ obtained after having applied the action in the current state. But in the case of MDPs, contrary to Kripke structures, transitions between states have a *value*; it is hence not possible to use a Boolean language. The authors employed the ADD language [§ 1.5.1], on interpretation domain $\mathfrak{D}_{\mathcal{S}\cup\mathcal{A}\cup\mathcal{S}',[0,1]}$.

Then, the value of the ADD for each assignment is the probability associated with the corresponding states and transition in the MDP. For example, denoting φ the ADD, for some given assignments $\vec{s}$, $\vec{a}$, and $\vec{s}'$, $[\![\varphi]\!](\vec{s} \,.\, \vec{a} \,.\, \vec{s}')$ is the probability to end up in $\vec{s}'$ when applying $\vec{a}$ in $\vec{s}$—that is, the value of $P_{\vec{a}}(\vec{s}, \vec{s}')$. Using another ADD to similarly represent the reward function, the authors present an algorithm, named SPUDD (stochastic planning using decision diagrams), to solve an MDP planning problem, based on the classical *value iteration* procedure [Bel57]. The interest of SPUDD is that the operations are made on ADDs, and thus on sets of states, rather than on single states, in the same way as in the planning as model-checking paradigm. This allowed the authors to solve problems that classical techniques could not tackle. This kind of approach has since then been extended to the search for *approximate* policies [SHB00] and to the solving of *first-order MDPs* [JKK09].

*
**

This chapter provided an overview of automated planning, and examined various state-of-the-art applications of knowledge compilation to this general problem. In the next chapter, we refocus on our subject, which more specifically involves embedded systems.

CHAPTER

3

Orientation of the Thesis

After this general overview of knowledge compilation, planning, and how they relate, we center on our specific subject: studying the application of knowledge compilation to realistic problems of autonomous system control. We identify several specific problems, for which knowledge compilation could be useful, and apply state-of-the-art techniques to one of them. Realizing their shortcomings, we decide to study new target languages, more suitable to the compilation of our problems than existing ones.

We first informally describe the problems we consider [§ 3.1]; then we present our first attempt at compiling these problems [§ 3.2]; finally, we detail the general orientation of our work [§ 3.3].

3.1 Benchmarks

We consider four benchmarks that we deem representative of realistic decision-making problems for embedded systems. We give a rough presentation of each generic problem; complete specification can be found either in the paper it originates from, or in Appendix A.

3.1.1 Drone Competition Benchmark

The *Drone* benchmark is about the management of objectives for a micro air vehicle (MAV) in a competition.[1] This kind of competition generally involves MAVs having to achieve a number of goals. In the *Drone* problem, which is adapted from Verfaillie and Pralet [VP08b] (see Appendix A.1 for the specification of our version), the field is divided into *zones*, each zone containing a target. There are three

[1] For example, the competition organized within the International Micro Air Vehicle Conference; see http://www.imav2011.org/.

different types of target, each type accounting for some specific abilities of the MAV.

- Targets to be *identified*: the MAV must be able to identify which color a target is, for example. This involves image processing, so the MAV must have an embedded video camera, and must make a specific maneuver (an "eight") above the target to get enough material.
- Targets to be *localized*: the MAV has no information about the location of the target in its zone, and must find it. To perform the localization task, the MAV must make a "scanning" maneuver and inspect the whole zone using a video camera.
- Targets to be *touched*: the MAV drops a ball that must reach the target. Coordinates of the target are known, so we consider that no video camera is necessary for this objective.

There is a special "home" zone, from which the MAV takes off and to which it must return at the end. The overall goal of the *Drone* problem is for the MAV to take off, accomplish all objectives, and land in the home zone, all of this within the allotted time.

The problem is formulated as a classical planning problem using fluents, with a constraint network representing the preconditions of actions, a constraint network representing their effects, and some constraints representing the initial and goal states. These constraint networks involve discrete variables, but also a continuous one, representing the "remaining time" state variable.

We would like to compile these constraint networks, so that we can apply the planning as satisfiability [§ 2.3.1.1] or planning as model-checking [§ 2.3.3] approaches.

3.1.2 Satellite Memory Management Benchmark

The *ObsToMem* problem [PV$^+$10] manages connections between the observation instrument and the mass memory of a satellite. The observation instrument consists of sensors, which are organized into a set of *detector lines*. The information gathered by each line must be compressed; this is done thanks to *memory compressors (COMs)*. Each detector line can be connected to only a given number of COMs; there is generally a different number of detector lines and of COMs. If at some point, there is no COM available to compress the output of a detector line, data is lost. To avoid this, the satellite is equipped with more COMs than detector lines—each COM being associated with a given set of lines.

Compressed information is then stored into the mass memory of the satellite. This memory is divided into *memory banks*; each COM can write only into a given set of memory banks. Overall, for the information at a given point to be correctly written into memory, each detector line must be connected to an available COM,

which must be connected to an available memory bank. The objective of this problem is to build a *controller* ensuring (as much as possible) that data can be written into mass memory, even in case of COM and memory bank failures. To this end, the controller must be able to change the configuration of the connections between all elements.

ObsToMem is thus a *control problem*: its solution is a decision policy ensuring that the requirements are permanently fulfilled. It is given in the form of several constraint networks, representing the transition relation, the safety property (that is, the permanent goal), and the initial state. These constraint networks do not involve continuous variables, but a fair number of enumerated variables, with domain sizes depending on the number of detector lines, COMs, and memory banks.

We want to compile these constraint networks, to apply for example some planning as model-checking algorithm [§ 2.3.3].

3.1.3 Transponder Connections Management Benchmark

The *Telecom* problem also aims at building a controller managing connections between some elements of a satellite. The object of the problem is the *transponder* of a communications satellite, that is, the series of elements forming the communications channel between the receiving and the transmitting antennæ of the satellite. In the *Telecom* benchmark, we only consider three elements: *input* and *output* channels, and signal amplifiers. The objective is to build a controller ensuring that input data is correctly amplified and connected to an output channel, even in case of failures of amplifiers and channels. The problem is modelized in a different way than *ObsToMem*, in that the possible configurations are given in the form of a set of *paths* linking input channels, amplifiers, and output channels. See Appendix A.2 for a complete specification.

The problem is given as a constraint network on state and decision variables, representing respectively which devices are working or not, and which path is assigned to each input channel. The solutions of the constraint networks are configurations connecting input channels to working output channels via working amplifiers.

We want to compile this constraint network, either to use it directly online (each time a failure is declared, a new configuration is sought), or to build a policy.

3.1.4 Attitude Rendezvous Benchmark

The *Satellite* benchmark is a part of a decision-making problem involving an Earth-observing agile satellite, equipped with a cloud detection instrument [BVC07]. The satellite must make very quick decisions, depending on whether the zones to be observed are cloudy or not. One of its possible decisions is “point to the Sun and recharge batteries”. Pointing to the sun takes time, so this is not something the satellite should do whenever it has no observation to make—the next observation must be sufficiently remote in time (among other requirements).

Computing the time necessary to point towards the Sun is not trivial; it is based on equations involving a number of continuous parameters, such as the angles representing the current attitude of the satellite. The *Satellite* problem consists in deciding whether it is possible to achieve the Sun pointing maneuver within the given time. The result is then used as a parameter by the main decision-making program embedded in the satellite.

Our objective is to compile the constraint network representing this subproblem, in such a way that the decision-making program uses the compiled form and gets a solution as fast as possible. This can be seen as a *partial compilation*—we only compile a part of the problem, and leave the rest to another kind of algorithm.

3.2 A First Attempt

3.2.1 Our Approach to the *Drone* Problem

We started our study by trying to apply a strong planning as model-checking algorithm [§ 2.3.3.2] to the *Drone* problem. Theoretically speaking, the strong and weak planning problems are the same in this case, since actions are deterministic; however, building a policy instead of a plan is more robust, because it can cope with the imperfections of the deterministic model in practice. As long as the current state is covered by the policy, the autonomous system is able to reach the goal. Using the strong planning algorithm, which starts from the goal and iteratively adds covered states, we can provide a decision even for states that are not supposed to be encountered.

We implemented this algorithm, modifying it so that instead of stopping whenever it covers all initial states, it stops when no new covered states are added—that is, when a fixed point is reached. At a fixed point, we are sure that all states from which the goal is reachable are covered. The resulting decision policy thus covers as much states as possible; remaining states are considered as "dead-ends".

Our procedure compiles each constraint network (viz., preconditions and effects of actions, and goal) into $\texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$. It then builds the policy as an OBDD, as described above [see also GT99]. Figure 3.1 shows the resulting policy for the instance number 1. In order to compile constraint networks involving the real-valued "remaining time" variable, the procedure arbitrarily discretizes it into small time units, thus creating an enumerated variable of domain large enough for results to remain significative despite the loss in precision. Then, this enumerated variable, together with the other integer-domain variables in the problem, are replaced by a number of Boolean variables, each one corresponding to a given bit in the binary representation of the domain's values. This transformation is called "*log encoding*" [see e.g. SK$^+$90, Wal00].

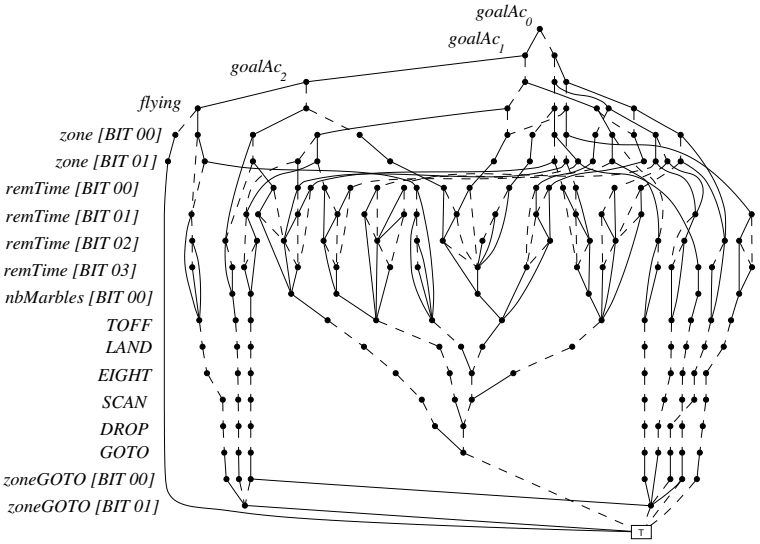

Figure 3.1: Example of a policy for the *Drone* problem, in the form of an OBDD, obtained using the planning as model-checking procedure. This is the instance number 1, with 4 zones. The $\bot$-node and all its incoming edges are not represented.

instance	policy	input	largest
0	29	4274	4225
1	383	63 869	85 411
2	1219	164 176	323 222
3	4547	248 631	661 505
4	16 494	361 043	1 241 057
5	144 087	781 995	3 595 802

Table 3.1: Number of nodes of various OBDDs—solution policy, input constraint networks, and largest OBDD encountered during computation—for six instances of the *Drone* problem.

3.2.2 Results for the *Drone* problem

Table 3.1 shows some results obtained using this approach on a simplified version of the *Drone* problem (all actions arbitrarily take the same time). We considered different instances of this problem, with the number of objectives as a varying parameter: instance n has n zones per type of objective, which means it has $3n + 1$ zones in total (including the "home" zone).

Regarding the resulting policy, this approach is fruitful: a hundred-thousand-node policy can be handled by the embedded system. However, the largest OBDD used by the planning algorithm contained more than 3.5 million nodes. Because of this, we could not compile the $n = 6$ instance, due to a lack of memory.

3.3 Towards More Suitable Target Languages

In this section, we describe the direction we follow in this work, with respect to the characteristics of our subject.

3.3.1 General Orientation

Instead of compiling our problems into OBDD, we decide to investigate the possibility of using a suitable target language. In particular, we would like it to be able to represent constraint networks on Boolean variables together with enumerated and continuous variables.

There exists several Boolean languages on *enumerated* variables, among which MDD and other related decision diagrams. However, there exists no knowledge compilation map of these languages. More importantly, we did not find studies about the compilation of problems involving *continuous* variables.

We thus decide to define a language over continuous variables, inspired from the decision diagram family. Yet, we do not directly transpose ordered binary decision diagrams or multivalued decision diagrams in a continuous context; indeed, these languages satisfy more queries and transformations than we need. We begin by defining a quite general language, then apply restrictions to it so that it satisfies the set of queries and transformations we are interested in, while remaining as succinct as possible.

3.3.2 Identifying Important Operations

In order to search for languages both as general as possible and suitable for our problems, we draw up a list of the queries and transformations our languages must satisfy for our applications to be tractable. Depending on the problem, we have two different objectives:

- either compiling a solution to the problem, in the form of a policy;
- or compiling a part of the problem (typically, the transition relation) for the online resolution to be tractable.

Handling Decision Policies

In the *Drone* or *ObsToMem* benchmarks, we need to compile a decision policy, that is, a function associating with each state a set of suitable actions—or equivalently, a function mapping each state-action couple $\langle s, a \rangle$ to a Boolean value indicating whether making action a in state s is a good choice. Such a function can be compiled into a Boolean representation language.

In order to exploit the policy online, two basic operations are required. First, each time a new state is observed—corresponding to a given assignment of the state variables—the set of suitable actions must be processed. This corresponds to *conditioning* the compiled form by the given assignment. The obtained structure represents the set of "good" actions; depending on the case, it might be useful either to list them all, which corresponds to *model enumeration*, or just to pick one, which is *model extraction*. Contrary to the former two, the latter operation is not defined in the existing knowledge compilation map. We introduce it in the following section [§ 3.3.3].

Handling Transition Relations

In all our benchmarks, except for *Satellite*, we must handle transition relations, that is, relations linking a given state s and a given action a to the state s' resulting from the application of a in s. These relations are representable as Boolean functions over state variables and action variables. Once they are compiled, they can be handled in different ways.

The simplest one is to compute, given a current state s and an action a, the set of all possible successor states. To do this, it is necessary to condition the compiled form by the assignment representing s and a, then to enumerate the models of the resulting structure. Alternatively, one may need to compute the set of all possible successor states, independently from the action to be made; in this case, it is still necessary to condition by s, but the following step is to *forget* the action variables, before enumerating the models.

An interesting use of a compiled transition relation is the construction of a decision policy, using a planning as model-checking approach (for example, the procedure we described in Section 3.2). The main operations involved are bounded conjunction and disjunction, negation, forgetting, and ensuring. The ensuring transformation is not necessary when all actions are deterministic.

3.3.3 New Queries and Transformations

The operation of *model extraction* is needed to handle compiled decision policies online. Since it is not considered in the knowledge compilation map literature, we introduce it here, together with other interesting new queries and transformations.

Definition 3.3.1 (Extraction queries)**.** Let L be a sublanguage of GRDAG.

- L satisfies **MX** (model extraction) if and only if there exists a polytime algorithm that maps every L-representation φ to a model of φ if there exists one, and stops without returning anything otherwise.
- L satisfies **CX** (context extraction) if and only if there exists a polytime algorithm that maps every L-representation φ and every variable y to $\text{Ctxt}_\varphi(y)$.

Extracting the context of a variable, that is, the set of its consistent values, is especially interesting when domains are non-Boolean. It can be used for example to examine some property of the successor states in a transition relation; it has also applications in configuration (which values are still available for this parameter?) and diagnosis (which are the possible failure modes of this component?).

Observe that satisfaction of model or context extraction is a stronger result than satisfaction of **CO**.

Proposition 3.3.2. Let L be a sublanguage of GRDAG. If L satisfies **MX** or **CX**, then it satisfies **CO**.

Proof. If L satisfies **MX**, there exists a polynomial P and an algorithm that takes any L-representation φ as input, stops after at most $P(\|\varphi\|)$ operations, returning a model of φ if there exists one or nothing otherwise. We can use this algorithm to decide whether some L-representation is consistent: if it returns a model, φ is consistent, if it returns nothing φ is not consistent. We know that the algorithm will answer after at most $P(\|\varphi\|)$ operations, P being the same for all φ: we hence have a polytime algorithm for deciding whether any φ is consistent or not. L satisfies **CO**.

If L satisfies **CX**, we can obtain in polytime the context of a variable y in an L-representation φ. Once again, this information is sufficient for deciding whether φ is consistent: thanks to Proposition 1.4.4, if we obtain a non-empty set, we can conclude that φ is consistent, and if we obtain an empty set, we can conclude that φ is not consistent. Using the same mechanism as for **MX**, we have a polytime algorithm for deciding whether any φ is consistent or not, hence L satisfies **CO**. □

Now, as we explained in Section 1.4.1.3, there are several ways to extend the "conditioning" query to non-Boolean domains. We chose to keep the idea that conditioning a function is assigning fixed values to some variables. But it is sometimes necessary to restrict variables to a *subset* of their domain, and not to a value. This is the role of the following transformation.

Definition 3.3.3 (Term restriction)**.** Let L be a sublanguage of GRDAG. L satisfies **TR** (term restriction) if and only if there exists a polytime algorithm mapping every L-representation φ and every consistent term γ in L, to an L-representation of the restriction $[\![\varphi]\!]_{|[\![\gamma]\!]}$ of $[\![\varphi]\!]$ to $[\![\gamma]\!]$.

The term restriction transformation is thus an extension of conditioning. When applied to Boolean variables, these two transformations are totally equivalent, as we explained in Section 1.4.1.3. Term restriction can be used for example on a compiled form representing a decision policy: if the observation of the current

state is not accurate, instead of conditioning the state variables, they can be "term restricted". The resulting structure then represents the set of actions that may be suitable to the unknown current state. Of course, **TR** is strongly related to **CD**, and also to **FO**, by definition.

Term restriction is also related to conjunction, but not the conjunction of any two Boolean functions. It is actually a conjunction with a term, followed by the forgetting of the term's variables. It is generally easy to conjoin a structure with a term, even if the language in question does not satisfy $\wedge$**C** or even $\wedge$**BC**. Since this fact is often used in proofs, we introduce the specific "conjunction with a term" transformation, and for similar reasons, its dual "disjunction with a clause" transformation.

Definition 3.3.4. Let L be a sublanguage of GRDAG.

- L satisfies $\wedge$**tC** (closure under conjunction with a term) if and only if there exists a polytime algorithm mapping every L-representation φ and every term γ in L, to an L-representation of $[\![\varphi]\!] \wedge [\![\gamma]\!]$.
- L satisfies $\vee$**clC** (closure under disjunction with a clause) if and only if there exists a polytime algorithm mapping every L-representation φ and every clause γ in L, to an L-representation of $[\![\varphi]\!] \vee [\![\gamma]\!]$.

*
**

To sum up, we decide to examine the possibility of compiling our problems into more suitable target languages, able to handle both continuous and discrete variables. We remain in the GRDAG framework, but try to be as general as possible, with the purpose of keeping succinctness maximal. The queries and transformations [see Definitions 1.4.15 and 1.4.16] we want the new languages to satisfy are the following:

- **CD** and **MX**—mandatory, for basic handling of decision policies and transition tables;
- **ME** and **FO**—desirable, for more complex handling of these elements;
- $\wedge$**BC**, $\vee$**BC**, $\neg$**C**—necessary to apply our policy building algorithm based on "strong planning as model-checking";
- **EN**—desirable, for the aforementioned algorithm to be able to handle nondeterministic effects of actions.

Part II

Interval Automata

Introduction

When we tried to apply knowledge compilation to one of our target problems, we stumbled upon a problem with *real variables*. They are usually not considered in knowledge compilation, even though they can be useful for representing real-world problems, notably in a planning context. Parameters having a continuous range, such as time or energy, are generally arbitrarily *discretized*, so that they can be represented as enumerated variables. For example, in the *Drone* problem, we represented the "remaining time" parameter as an integer variable, decreasing precision to a fixed number of seconds. Such methods lead to variables with large domains, which has an impact on the size of compiled structures.

We decided to investigate the efficiency of compilation languages capturing continuous variables without requiring an arbitrary discretization, as well as discrete variables with large domains. Interval diagrams [ST98] seemed a good basis, but they were never practically applied to continuous variables nor used for planning. Moreover, since they were meant to be used for symbolic model checking, they were designed to support equivalence, whereas we do not need it in our applications; we thus tried to relax some of the structural constraints of interval diagrams, and defined a more general language, that we called *interval automata*.

This part is dedicated to the study of this language. We begin by defining interval automata, giving their main properties and their knowledge compilation map [Chapter 4]; then we present how they can be compiled [Chapter 5]; and finally, we experiment their use for our target applications [Chapter 6].

CHAPTER

4

Interval Automata Framework

In this chapter, we define *interval automata* as a target compilation language for problems involving continuous variables. We identify a structural restriction of interval automata allowing them to support common queries and transformations, and in particular the main ones we need for planning.

The chapter is divided into three sections: first, we define the general language of interval automata [§ 4.1], then introduce the sublanguage of focusing interval automata [§ 4.2], and finally provide the knowledge compilation map of the interval automata family [§ 4.3]. Most proofs are gathered at the end of the chapter [§ 4.4].

4.1 Language

4.1.1 Definition

The language of interval automata can be defined as a sublanguage of GRDAG; we need to define a few notions beforehand.

Definition 4.1.1 (Interval)**.** Let S be a set totally ordered by $\leqslant$. Subset $A \subseteq S$ is an *interval of* S if and only if it is a convex part of S, i.e., all $\langle x, y\rangle \in A^2$ and all $z \in S$ verify $x \leqslant z \leqslant y \implies z \in A$.

An interval is said to be *left-bounded* (resp. *right-bounded*) if and only if it has a lower bound (resp. an upper bound), i.e., $\exists m \in S, \forall x \in A, m \leqslant x$ (resp. $m \geqslant x$)).

The greatest lower bound (resp. least upper bound) of a left-bounded (resp. right-bounded) interval is simply called its *left bound* (resp. *right bound*).

An interval is said to be *left-open* (resp. *right-open*) if and only if it is left-bounded (resp. right-bounded) and does not contain its left bound (resp. its right bound), that is, $\forall x \in A, \exists m \in A, m < x$ (resp. $m > x$).

A *closed interval* is an interval that is neither left-open nor right-open.

Note that a closed interval needs not be finite: $\mathbb{R}_+$ is a closed interval of $\mathbb{R}$. We will focus on intervals of $\mathbb{R}$, and denote as $[a, b]$ the closed interval with left bound a and right bound b, $[a, +\infty]$ the closed interval with left bound a and no right bound, and $[-\infty, b]$ the closed interval with right bound b and no left bound. $\mathbb{I}S$ denotes the set of closed intervals of S. We will need, on rare occasions, to handle non-closed intervals; we write them with inverted brackets, such as $[a, b[$ for the interval corresponding to $[a, b] \setminus \{b\}$.

Definition 4.1.2 (Tileability)**.** Let S be a totally ordered set. A subset $A \subseteq S$ is *S-tileable* if and only if there exists $n \in \mathbb{N}$ and a sequence $\langle A_1, \ldots, A_n \rangle \in (\mathbb{I}S)^n$ such that $A = \bigcup_{i=1}^{n} A_i$.

An S-tileable set is thus simply a finite union of closed intervals from S. We denote as $\mathbb{T}S$ the set of S-tileable sets. $\mathbb{T}\mathbb{R}$ includes of course all closed intervals and all finite subsets of $\mathbb{R}$; but note that it does not include $\mathbb{N}$. We use this notion to define the variables used in this part; it will also be useful later [§ 7.1.1].

In this part, we need to handle real variables, as well as enumerated ones. We choose to use $\mathbb{R}$-tileable domains to cover both possibilities. Let us denote by $\mathcal{T} = \mathcal{V}_{\mathbb{T}\mathbb{R}}$ the set of $\mathbb{R}$-tileable variables; it has the interesting property of including variables with a finite enumerated domain, such as $\{1, 3, 56, 4.87\}$, along with variables with a continuous domain, such as $[1, 7] \cup [23.4, 28]$.

$\mathbb{R}$-tileable domains are also interesting because they are directly representable using a simple data structure, such as a list of bounds (which is not the case of an *infinite* union of intervals, for example). We need to define the *characteristic size* of a tileable set, that we consider representative of the memory size taken by the data structure, independently from its actual implementation [§ 1.2.2.2].

Definition 4.1.3 (Size of a tileable set)**.** Let $S \in \mathbb{T}\mathbb{R}$ be an $\mathbb{R}$-tileable set; we define the characteristic size of S as the smallest number of intervals needed to cover S:

$$\|S\| = \min \{\, n \in \mathbb{N} \mid \exists \langle A_1, \ldots, A_n \rangle \in (\mathbb{I}\mathbb{R})^n, S = \textstyle\bigcup_{i=1}^{n} A_i \,\}\,.$$

We can now give the formal definition of our language.

Definition 4.1.4. We define the `IA` language as the restriction of $\mathtt{NNF}_{\mathcal{T}}^{\mathbb{I}\mathbb{R}}$ to representations satisfying $\wedge$-simple decision [Def. 1.3.23].

In other words, `IA`-representations are GRDAGs on $\mathbb{R}$-tileable variables, with literals of the form "$x \in [a, b]$", and with structural restrictions belonging to the "decision diagram" family [§ 1.3.5]. The name `IA` stands for *interval automata*; but following the usage for BDDs and finite-state automata, we generally do not handle interval automata directly in their NNF form. In the following section, we present the equivalent "decision diagram" representation of interval automata, that will simplify further manipulations.

4.1.2 Interval Automata

The following definition describes an interval automaton in its usual form.

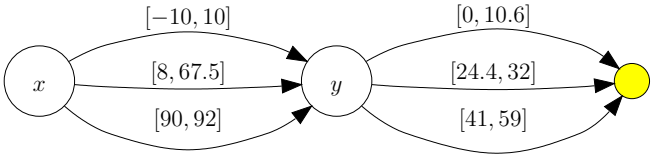

Figure 4.1: An example of an interval automaton. Its model set [Def. 1.4.6] is $[-10, 10] \times [0, 10.6] \cup [-10, 10] \times [24.4, 32] \cup [-10, 10] \times [41, 59] \cup [8, 67.5] \times [0, 10.6] \cup [8, 67.5] \times [24.4, 32] \cup [8, 67.5] \times [41, 59] \cup [90, 92] \times [0, 10.6] \cup [90, 92] \times [24.4, 32] \cup [90, 92] \times [41, 59]$.

Definition 4.1.5. An *interval automaton* (IA) is a directed acyclic multigraph with at most one root and at most one leaf (the *sink*); each non-leaf node is labeled with a variable of $\mathcal{T}$ or with the disjunctive symbol $\veebar$, and each edge is labeled by a closed interval of $\mathbb{R}$.

This definition of interval automata makes them different from the elements of the `IA` language we defined in the previous section: `IA` is defined as a subset of `NNF`, whereas the structure of Definition 4.1.5 is clearly not an NNF. However, this is harmless, since there is a one-to-one relation between the two structures, exactly as for BDDs [§ 1.3.5].

In the following, we always implicitly consider interval automata in this form. Figure 4.1 gives an example of an interval automaton and of its model set. This model set has a particular form: it is a union of cartesian products of closed intervals, which we call "union of boxes". It is not surprising, given the structure of interval automata: they behave exactly like BDDs, and as such, their model set is composed of all assignments compatible with at least one path. Since each path represents a *term*, for example $\big[x \in [-10, 10]\big] \wedge \big[y \in [0, 10.6]\big]$ for the uppermost path, the set of its compatible assignments can only be a box. Hence, an IA always having a finite number of paths, its model set can only be a finite union of boxes.

Using this fact, we can already state that `IA` is not a complete language [Definition 1.2.12].

Proposition 4.1.6. The `IA` language is not complete.

Proof. Let us recall that a language $\mathtt{L} = \langle \mathfrak{D}, \mathcal{R}, [\![\cdot]\!] \rangle$ is complete if and only if every function from its interpretation domain $\mathfrak{D}$ has a representation in $\mathcal{R}$ with respect to $[\![\cdot]\!]$.

There are several angles from which `IA` can be proven incomplete; one can use the fact that it cannot represent open intervals, for example. We use here the argument of infinity: let $x \in \mathcal{T}$ be a variable of domain $\mathbb{R}$. The Boolean function $\big[x \in \mathbb{N}\big]$ is not representable as an interval automaton: if it were, $\mathbb{N}$ would be a finite union of closed intervals from $\mathbb{R}$. □

Note that the Boolean function always returning $\top$ can be represented by the sink-only automaton, similar to the case of BDDs. However, since IAs have only

one leaf, the Boolean function always returning $\bot$ can be represented by an *empty* automaton. This is authorized by the definition, and is consistent with the interpretation function of decision diagrams [§ 1.1.2.1]: since "the function returns true if and only if there exists a compatible path", it never returns true, as there exists no path at all.

We treat the disjunctive symbol $\veebar$ as a peculiar variable, arbitrarily imposing $\mathrm{Dom}(\veebar) = \{0\}$. Let φ be an interval automaton, N a node and E an edge in φ. We define the following elements:

- $\mathrm{Root}(\varphi)$ is the root of φ and $\mathrm{Sink}(\varphi)$ its sink;
- $\mathrm{Var}(N)$ is the variable with which N is labeled (by convention, it holds that $\mathrm{Var}(\mathrm{Sink}(\varphi)) = \veebar$);
- $\mathrm{Lbl}(E)$ is the interval with which E is labeled;
- $\mathrm{Var}(E)$ is the variable associated with E, viz., $\mathrm{Var}(E) = \mathrm{Var}(\mathrm{Src}(E))$.

As suggested by Figure 4.1, the size of the automaton can be exponentially lower than the size of its extended model set (described as a union of boxes). This notably comes from the fact that IAs can be *reduced* by suppressing redundancies, in the manner of BDDs and NNFs. Before detailing this reduction operation, let us present the relationship between IAs and this kind of structures.

4.1.3 Relationship with the BDD family

Interval automata can be understood as a generalization of binary decision diagrams. The interpretation of BDDs is indeed similar to that of IAs: for a given assignment of the variables, the function's value is $\top$ if and only if there exists a path from the root to the $\top$-labeled leaf such that the given assignment is consistent with each edge along the path. When restricted to the same set of variables, BDDs are even particular IAs.

Proposition 4.1.7. $\mathtt{BDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \mathtt{IA}$.

Proof. $\mathtt{BDD}_{\mathcal{T}}^{\mathbb{IR}}$ is a sublanguage of $\mathtt{NNF}_{\mathcal{T}}^{\mathbb{IR}}$, by Proposition 1.2.7. It satisfies strong and exclusive decision. Since, by definition, strong decision implies $\wedge$-simple decision, $\mathtt{BDD}_{\mathcal{T}}^{\mathbb{IR}}$ is a fragment of $\mathtt{IA}$. □

In particular, since we consider $\mathbb{B}$ as a subset of $\mathbb{N}$, Boolean variables have a tileable domain (that is, $\mathcal{B} \subseteq \mathcal{T}$) and of course $\mathbb{SB} \subseteq \mathbb{IR}$; we thus obtain the following important result.

Corollary 4.1.8. $\mathtt{BDD}_{\mathcal{B}}^{\mathbb{SB}} \subseteq \mathtt{IA}$.

Interval automata are thus more general than binary decision diagrams in three ways:

- they allow more general variables and labels;

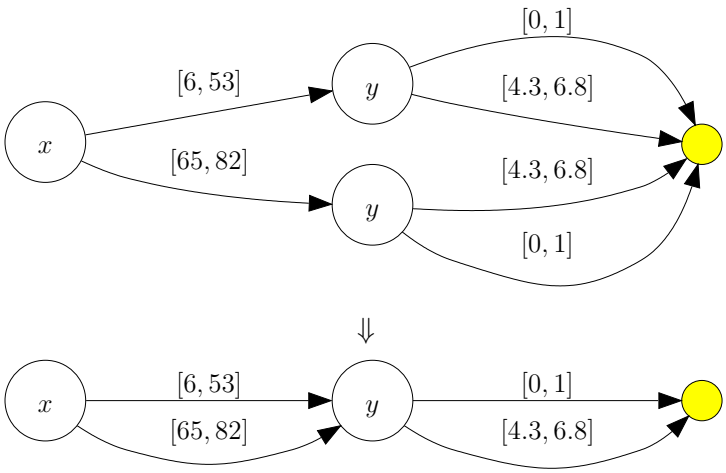

Figure 4.2: Merging of isomorphic nodes.

- they allow non-exclusive decision nodes, and can thus be non-deterministic;
- they allow pure disjunctive nodes.

4.1.4 Reduction

Like a BDD, an interval automaton can be reduced in size without changing its semantics by merging some nodes or edges. The reduction operations that we introduce thereafter are based on the notions of *isomorphic*, *stammering*, and *undecisive* nodes, and of *contiguous* and *dead* edges. Some of these notions are straightforward generalizations of definitions introduced in the context of Boolean BDDs [Bry86], while others are specific to interval automata.

Definition 4.1.9 (Isomorphic nodes)**.** Two non-leaf nodes N_1 and N_2 of an IA φ are *isomorphic* if and only if:

(i) $\text{Var}(N_1) = \text{Var}(N_2)$;

(ii) there exists a bijection σ from $\text{Out}(N_1)$ to $\text{Out}(N_2)$, such that for each edge $E \in \text{Out}(N_1)$, $\text{Lbl}(E) = \text{Lbl}(\sigma(E))$ and $\text{Dest}(E) = \text{Dest}(\sigma(E))$.

Isomorphic nodes are redundant, as they represent the same function; only one of the two is necessary (see Figure 4.2). This corresponds to the usual procedure on BDDs (and generally on DAG languages).

Definition 4.1.10 (Undecisive node)**.** A node N of an IA φ is *undecisive* if and only if $|\text{Out}(N)| = 1$, and $E \in \text{Out}(N)$ is such that $\text{Dom}(\text{Var}(E)) \subseteq \text{Lbl}(E)$.

An undecisive node does not restrict the solutions corresponding to the paths it is in; it is "automatically" crossed (see Figure 4.3).

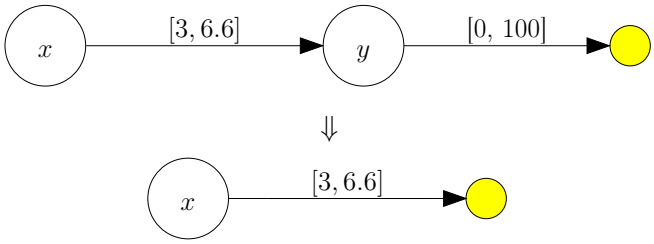

Figure 4.3: Elimination of an undecisive node (here $\text{Dom}(y) = [0, 10]$).

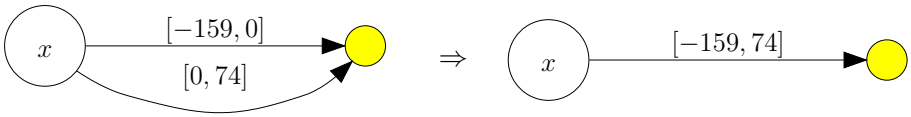

Figure 4.4: Merging of contiguous edges.

Definition 4.1.11 (Contiguous edges)**.** Two edges E_1 and E_2 of an IA φ are *contiguous* if and only if:

(i) $\text{Src}(E_1) = \text{Src}(E_2)$;

(ii) $\text{Dest}(E_1) = \text{Dest}(E_2)$;

(iii) there exists an interval $A \subseteq \mathbb{R}$ such that $A \cap \text{Dom}(\text{Var}(E_1)) = (\text{Lbl}(E_1) \cup \text{Lbl}(E_2)) \cap \text{Dom}(\text{Var}(E_1))$.

Two contiguous edges come from the same node, point to the same node, and are not disjoint (modulo the domain of their variable): they can be replaced by a single edge (see Figure 4.4). For example, in the case of an integer-valued variable, a couple of edges respectively labeled $[0, 3]$ and $[4, 8]$ is equivalent to a single edge labeled $[0, 8]$.

Elimination of undecisive nodes and redundant edges altogether corresponds to the elimination of "redundant nodes" in the context of Boolean BDDs (nodes having only one child).

Definition 4.1.12 (Stammering node)**.** A non-root node N of an IA φ is *stammering* if and only if all parent nodes of N are labeled by $\text{Var}(N)$, and either $|\text{Out}(N)| = 1$ or $|\text{In}(N)| = 1$.

Stammering nodes are not necessary, because the information they bring can harmlessly be transferred to their parents (see Figure 4.5).

Definition 4.1.13 (Dead edge)**.** An edge E of an IA φ is *dead* if and only if $\text{Lbl}(E) \cap \text{Dom}(\text{Var}(E)) = \varnothing$.

A dead edge can never be crossed, as no value in its label is consistent with the variable domain (see Figure 4.6).

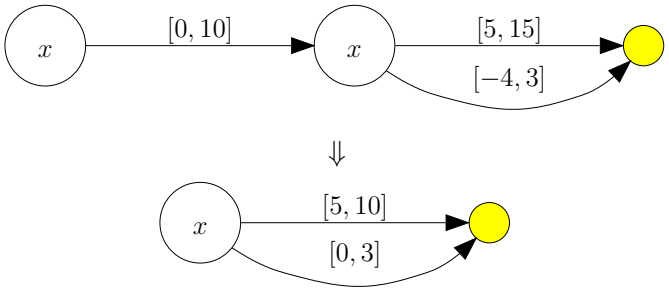

Figure 4.5: Merging of a stammering node.

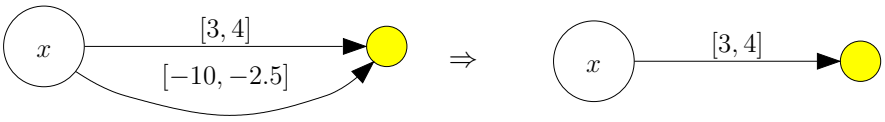

Figure 4.6: Elimination of a dead edge (here $\mathrm{Dom}(x) = \mathbb{R}_+$).

Definition 4.1.14 (Reduced interval automaton)**.** An interval automaton φ is said to be *reduced* if and only if:

(i) no node of φ is isomorphic to another, stammering, or undecisive;

(ii) no edge of φ is contiguous to another or dead.

In the following, we generally consider only reduced IAs; reduction is indeed "harmless", since it can be done in time polynomial in the size of the structure.

Proposition 4.1.15 (Reduction of an IA)**.** There exists a polytime algorithm that transforms any IA φ into an equivalent reduced IA φ' such that $\|\varphi'\| \leqslant \|\varphi\|$.

Detailed proof p. 112.

Algorithm 4.1 is an example of a polynomial reduction procedure—although it is certainly not the best one. Since each operation can modify previously reduced nodes, traversals must be repeated until a fixed point is reached. All elementary operations are polynomial, and no node or edge is added throughout each traversal, therefore these elementary operations are not executed more than $\|\varphi\|$ times (see the complete proof for details).

Like reduced non-ordered BDDs on Boolean variables, reduced IAs are *not* canonical: as we show in the knowledge compilation map, `IA` does not support the equivalence query. However, reduction has interesting properties; it notably removes superfluous disjunctive nodes, as expressed by the following proposition.

Proposition 4.1.16. The only reduced IAs φ such that $\mathrm{Scope}(\varphi) = \varnothing$ are the empty and the sink-only automata.

Detailed proof p. 113.

Algorithm 4.1 Reduction algorithm for IAs. At any time during process, if an edge has no source or destination, it is removed; so are non-leaf nodes with no outgoing edge and non-root nodes with no incoming edge.

```
1: repeat
2:     for each node N in φ, from the root to the sink do
3:         if N is stammering then
4:             for each ⟨E_in, E_out⟩ ∈ In(N) × Out(N) do
5:                 add an edge from Src(E_in) to Dest(E_out) labeled with the
                   interval Lbl(E_in) ∩ Lbl(E_out)
6:             remove N
7:         else
8:             for each E ∈ Out(N) do
9:                 if E is dead then
10:                    remove E
11:                else
12:                    mark E
13:                    for each E′ ∈ Out(N) such that E′ is not marked do
14:                        if E and E′ are contiguous then
15:                            let U ≔ Lbl(E) ∪ Lbl(E′)
16:                            label E with [min(U), max(U)]
17:                            remove E′
18:            if N is undecisive then
19:                for each E_in ∈ In(N) do
20:                    redirect E_in to the child of N
21:                remove N
22:            else
23:                for each node N′ in φ, from the root to the sink do
24:                    if N and N′ are isomorphic then
25:                        for each E_in ∈ In(N) do
26:                            redirect E_in to N′
27:                        remove N
28: until φ has not changed during process
```

4.2 Efficient Sublanguage

We now study the efficiency of IA on the queries needed in our planning context [§ 3.3.2]. This raises the need for sublanguages of IA having better performances; we then propose a sublanguage that is more efficient than IA.

4.2.1 Important Requests on Interval Automata

The first operation we study is *conditioning*, that consists in assigning values to some variables. It is actually possible to do this using a simple, syntactic procedure, detailed in Algorithm 4.2. In broad outline, for each edge in the graph, the procedure checks whether it contains the value to assign; it keeps the edges that do and removes the edges that do not.

Proposition 4.2.1. IA satisfies **CD**.

Detailed proof p. 113.

Algorithm 4.2 Conditioning of an IA φ by an assignment $\overrightarrow{y}$ of a set of variables Y.

```
1: for each node N in φ, from the sink to the root do
2:     if Var(N) ∈ Y then
3:         for each E ∈ Out(N) do
4:             if y⃗_{|Var(N)} ∈ Lbl(E) then
5:                 label E by {0}
6:             else
7:                 label E by ∅
8:         label N by ⊻
```

Thus, conditioning is quite direct on interval automata. The next operation we need is the *model extraction* query. Unfortunately, this query is not supported by IA, basically because it is not supported by $\mathtt{BDD}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$, and $\mathtt{BDD}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$ is a sublanguage of IA [Corollary 4.1.8].

Proposition 4.2.2. IA does not satisfy **MX** unless $\mathsf{P} = \mathsf{NP}$.

Detailed proof p. 114.

This relates to the fact that deciding whether an interval automata is consistent is not tractable. One of the reasons for this, is that sets restricting a variable along a path can be conflicting: it is possible to have an edge "$x \in [1, 4]$" followed by an edge "$x \in [8, 10]$", which makes the path inconsistent. Hence, in the worst case, all paths must be explored before a consistent one is found. Note that reduction does not help; Figure 4.7 shows an example of a reduced IA containing no consistent path at all.

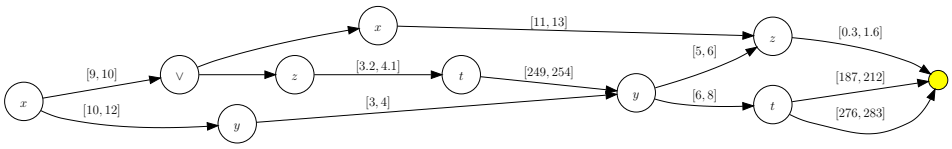

Figure 4.7: Example of a reduced IA in which no path is consistent. Before the IA is proven inconsistent, every path must be checked.

Since the **MX** query is fundamental in our applications, we cannot work with raw interval automata; we must impose structural restrictions allowing the model extraction query, and hence the consistency query too (by Prop. 3.3.2) to be polynomial.

4.2.2 Focusingness

One of the aspects that makes consistency checking hard on interval automata, is that paths can be inconsistent, because labels can be disjoint on a given path. We want to define a sublanguage of `IA` that forbids this behavior. The solution we propose is to force intervals pertaining to a given variable to *shrink* from the root to the sink.

Definition 4.2.3 (Focusingness)**.** An edge E in an interval automaton φ is *focusing* if and only if all edges E' on all paths from the root of φ to $\mathrm{Src}(E)$ such that $\mathrm{Var}(E) = \mathrm{Var}(E')$ verify $\mathrm{Lbl}(E) \subseteq \mathrm{Lbl}(E')$.

An interval automaton φ is *focusing with respect to* x, with $x \in \mathrm{Scope}(\varphi)$, if and only if each edge E in φ for which $\mathrm{Var}(E) = x$ is focusing.

A *focusing* interval automaton (FIA) is an IA φ that is focusing with respect to all variables in $\mathrm{Scope}(\varphi)$. `FIA` is the restriction of `IA` to focusing interval automata.

An example of an FIA can be found on Figure 4.8. Note that focusingness is only defined on variables from $\mathrm{Scope}(\varphi)$: $\veebar$-nodes are not concerned by this restriction. This makes the definition a little more complex, but simplifies later proofs.

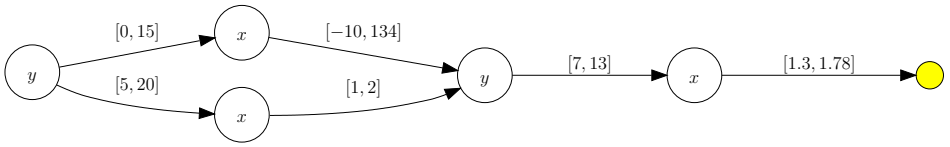

Figure 4.8: An example of a focusing interval automaton. Variable domains are as follows: $\mathrm{Dom}(x) = [0, 100]$, $\mathrm{Dom}(y) = [0, 100]$ and $\mathrm{Dom}(z) = \{0, 3, 7, 10\}$.

Imposing focusingness prevents labels from "conflicting"—in the sense that there cannot be some path containing "$x \in [1, 8]$" and then "$x \in [5, 15]$". However, focusingness alone does not make all paths consistent: nothing prevents labels from being empty, or disjoint with the domain. Fortunately, these cases are removed by

reduction. Polynomiality of FIA reduction is thus an important result, which is the object of the following proposition.

Proposition 4.2.4 (Reduction of an FIA)**.** There exists a polytime algorithm that transforms any FIA φ into an equivalent reduced FIA φ' such that $\|\varphi'\| \leqslant \|\varphi\|$.

Detailed proof p. 114.

The proof relies on the fact that the procedure described in Algorithm 4.1, initially designed to reduce general interval automata, actually maintains focusingness. After reduction, all paths in an FIA are consistent—exactly as is the case for read-once decision diagrams. For that matter, we get an interesting result about the relationship between FIAs and FBDDs.

Proposition 4.2.5. It holds that $\texttt{FBDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \texttt{FIA}$, and in particular that $\texttt{FBDD}_{\mathcal{B}}^{\mathbb{SB}} \subseteq \texttt{FIA}$.

Proof. We know that $\texttt{BDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \texttt{IA}$ [Prop. 4.1.7], therefore of course $\texttt{FBDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \texttt{IA}$. Now, in a read-once decision diagram, variables are never repeated along a path, which obviously means that all edges are focusing: each one has *no* ancestor pertaining to its variable. This proves that read-once decision diagrams are focusing, and hence that $\texttt{FBDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \texttt{FIA}$. Since $\mathbb{SB} \subseteq \mathbb{IR}$ and $\mathcal{B} \subseteq \mathcal{T}$, we get that $\texttt{FBDD}_{\mathcal{B}}^{\mathbb{SB}} \subseteq \texttt{FIA}$. □

Let us now show that the operations we are interested in are tractable on FIAs, starting with conditioning. Once again, the algorithm designed for general IAs maintains focusingness. This is used to prove the following proposition.

Proposition 4.2.6. FIA satisfies **CD**.

Detailed proof p. 115.

Before examining model extraction, let us remark that reduction makes consistency checking tractable on focusing interval automata. It can actually be proven that a reduced FIA is inconsistent *if and only if* it is empty: roughly speaking, the only thing that can compromise the consistency of an FIA is the fact that labels can go out of a variable's domain, which is impossible on a reduced FIA.

Proposition 4.2.7. FIA satisfies **CO**.

Detailed proof p. 115.

It can be seen in a different way: every path in a reduced FIA is consistent, hence if there exists a path, then the FIA is consistent. We use this property on paths to extract a model from an FIA.

Proposition 4.2.8. FIA satisfies **MX**.

Detailed proof p. 115.

The procedure is described in Algorithm 4.3. The idea is to reduce the FIA and choose an arbitrary path in the automaton (the choice does not matter, since all

paths are consistent), then choose an arbitrary assignment that is compatible with this path.

Algorithm 4.3 Extraction of a model from an FIA φ.

reduce φ
if φ is empty **then**
    exit
let $\vec{y} \in \text{Dom}(\text{Scope}(\varphi))$
let $S \coloneqq \{\underline{\vee}\}$
let node $N \coloneqq \text{Sink}(\varphi)$
while $N \neq \text{Root}(\varphi)$ **do**
    select an edge $E \in \text{In}(N)$
    let $x \coloneqq \text{Var}(E)$
    if $x \notin S$ **then**
        select a value $\omega \in \text{Lbl}(E) \cap \text{Dom}(x)$
        let $\vec{x}$ be the $\{x\}$-assignment of value ω
        $\vec{y} \coloneqq \vec{x} \cdot \vec{y}_{\mid \text{Scope}(\varphi) \setminus \{x\}}$
        $S \coloneqq S \cup \{x\}$
    $N \coloneqq \text{Src}(E)$
return $\vec{y}$

Hence, unlike raw IAs, FIAs support the main requests needed for planning applications. Moreover, as we show in Chapter 5, interval automata obtained exploiting the *trace* of an interval-based constraint solver are naturally focusing. `FIA` is thus a good candidate for our applications; this is why all experiments on interval automata are done using FIAs [Chapter 6].

Before examining the construction of FIAs, and experimenting their practical use, let us build the knowledge compilation map of `IA` and `FIA`, including all the queries and transformations we mentioned in Chapters 1 and 3.

4.3 The Knowledge Compilation Map of `IA`

This section contains the knowledge compilation map of the `IA` family. For the sake of readability, we gathered most of the proofs at the end of the chapter [§ 4.4]. Before stating the complete results, we emphasize in the next section a few key points on which most proofs depend.

4.3.1 Preliminaries

Context on FIAs

To extract the context of a variable x in an FIA, we use the same property that we used to check consistency and extract a model. Namely, on a path from the root to

the sink of a reduced FIA, values included in the label of the last x-edge are either consistent or out of the domain. Algorithm 4.4 uses this property to compute the context of a variable in an FIA.

Proposition 4.3.1. `FIA` satisfies **CX**.

Detailed proof p. 116.

Algorithm 4.4 Extraction of the context of a variable x in an FIA φ

```
reduce φ
if φ is empty then
    return ∅
let C ≔ ∅
mark the sink of φ
for each node N in φ, ordered from the sink to the root do
    if N is marked then
        for each E ∈ In(N) do
            if Var(E) = x then
                C ≔ C ∪ Lbl(E) ∩ Dom(x)
            else
                mark Src(E)
if the root of φ is marked then
    return Dom(x)
return C
```

The idea is to find the x-frontier of the sink, that is to say the set of x-labeled nodes N such that there exists a path from a child of N to the sink that does not mention x. In other words, nodes on the x-frontier are the x-nodes that are the closest to the sink. The x-frontier is computed by pulling up a mark, that means no x-labeled node has been encountered yet. If a mark reaches the root, it means there is at least one path from the root to the sink containing no x-node. In this case, the context of x in φ is $\mathrm{Dom}(x)$ (because φ is reduced, so the path is trivially satisfiable, in the same way that a non-empty reduced FIA is satisfiable). If no mark reaches the root, the context of x is the intersection of the domain of x and of the union C of the labels of the edges by which the x-frontier accesses the sink.

Clausal Entailment on FIAs

The goal of the clausal entailment query **CE** is to decide whether a Boolean function entails a clause. In the `IA` framework, it consists in checking whether $\varphi \models \big[x_1 \in I_1\big] \vee \cdots \vee \big[x_k \in I_k\big]$, with φ an IA, $x_1, \ldots x_k$ some variables, and $I_1, \ldots I_k$ some *closed intervals*. It is equivalent to checking whether $[\![\varphi]\!] \wedge \big[x_1 \notin I_1\big] \wedge \cdots \wedge \big[x_k \notin I_k\big]$ is inconsistent. As it involves open intervals, this formula is generally not representable as an interval automaton; however, we can build in polytime the restriction of φ to the "term" $\big[x_1 \notin I_1\big] \wedge \cdots \wedge \big[x_k \notin I_k\big]$, which is consistent if and only if the non-representable formula is consistent.

Proposition 4.3.2. `FIA` satisfies **CE**.

Detailed proof p. 118.

Combining Automata

Let us remark that computing the disjunction of interval automata is trivial—thanks to disjunctive nodes.

Proposition 4.3.3. `IA` and `FIA` satisfy $\vee$**C**, $\vee$**BC**, and $\vee$**clC**.

Proof. It is sufficient to "join" all IAs together using a $\veebar$-node. That is, denoting as $\Phi = \{\varphi_1, \dots, \varphi_k\}$ the set of IAs to be disjoined, we copy all IAs in Φ into ψ, fuse their sinks, and create a new root node for ψ, labeled $\veebar$, with $|\Phi|$ outgoing edges labeled by $\{0\}$, each pointing to the root of a different $\varphi \in \Phi$. This procedure is obviously linear in $\sum_{\varphi\in\Phi}\|\varphi\|$. It moreover maintains focusingness (no label is modified). □

Computing the conjunction of interval automata is also quite simple, as long as the resulting automaton is not required to be focusing. The idea is to connect the automata one to another.

Proposition 4.3.4. `IA` satisfies $\wedge$**C**, $\wedge$**BC**, and $\wedge$**tC**.

Proof. It is sufficient to "join" all IAs together into a *chain*, that is, denoting as $\Phi = \{\varphi_1, \dots, \varphi_k\}$ the set of IAs to be conjoined, to replace the sink of φ_i by the root of φ_{i+1}, for all $i \in \{1, \dots, k-1\}$. The root of the resulting automaton ψ is then set to be $\mathrm{Root}(\varphi_1)$, and its sink is $\mathrm{Sink}(\varphi_k)$. This procedure is obviously linear in $\sum_{\varphi\in\Phi}\|\varphi\|$. □

This simple procedure does not maintain focusingness; it is actually impossible to obtain the conjunction of two FIAs in polytime, unless $\mathsf{P} = \mathsf{NP}$.

Proposition 4.3.5. `FIA` does not satisfy $\wedge$**BC** or $\wedge$**C** unless $\mathsf{P} = \mathsf{NP}$.

Detailed proof p. 118.

We can nevertheless translate any term into an FIA in polytime.

Lemma 4.3.6. It holds that $\texttt{FIA} \leqslant_p \texttt{term}^{\mathbb{IR}}_{\mathcal{T}}$.

Proof. If the term contains only one literal for each variable in its scope, we can directly use the procedure described for making the conjunction of raw IAs: each literal can be turned into a one-edge IA, and since all the IAs obtained do not share variables, they can be put in a chain satisfying the read-once property, and thus also focusingness. If the term contains several literals on a same variable, it suffices to transform all these literals into a unique one, labeled with the intersection of all intervals, hence falling back into the first case. □

Meshes and Single Quantification

A number of proofs of queries or transformations use the fact that labels of all x-edges in an interval automaton induce a *partition of* $\mathrm{Dom}(x)$, as shown in Fig-

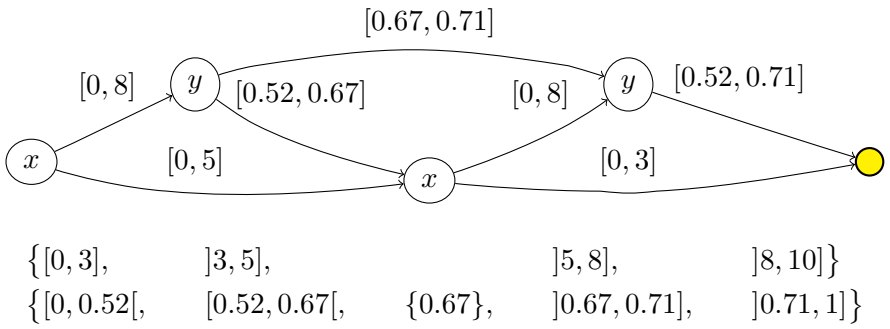

Figure 4.9: An IA over variables x and y, of domain $[0, 10]$ and $[0, 1]$, respectively. The partitions (meshes) it induces over these two domains is shown below; note that each element of a partition is either fully included in or completely disjoint from any label.

ure 4.9. The elements of this partition have the property of being *finer* than any x-label in the automaton. We call such a partition a *mesh*.

Definition 4.3.7 (Mesh)**.** Let φ be an IA and $x \in \text{Scope}(\varphi)$. A *mesh* of x in φ is a partition $\mathcal{M} = \{M_1, \ldots, M_n\}$ of $\text{Dom}(x)$ such that for any edge E in φ verifying $\text{Var}(E) = x$, for all $i \in \{1, \ldots, n\}$,

$$M_i \cap \text{Lbl}(E) \neq \varnothing \quad \Longrightarrow \quad M_i \subseteq \text{Lbl}(E).$$

Meshes have very interesting properties, that come from the following fact: if $[a, b]$ is one of the intervals of a mesh of x in φ, then conditioning φ by assigning x to any value in $[a, b]$ always yields the same result. We can say that all values in $[a, b]$ are "interchangeable". In particular, quantification operations on interval automata boil down to conditioning operations; both the proof of **SFO** on IA and FIA, and the proof of **SEN** on IA, use this mechanism.

Proposition 4.3.8. Let L be a sublanguage of IA; if L satisfies **CD** and $\vee$**C**, then it satisfies **SFO**. If L satisfies **CD** and $\wedge$**C**, then it satisfies **SEN**.

Detailed proof p. 120.

4.3.2 Succinctness

An important result concerns the relationship between the IA family and the CNF and DNF languages.

Proposition 4.3.9. It holds that

$$\begin{matrix}\texttt{IA}\\ \texttt{FIA}\end{matrix} \quad \leqslant_p \quad \texttt{DNF}_{\mathcal{T}}^{\mathbb{IR}}$$

and

$$\texttt{IA} \quad \leqslant_p \quad \texttt{CNF}_{\mathcal{T}}^{\mathbb{IR}}.$$

Proof. IA satisfies $\wedge$C and $\vee$C, so CNFs and DNFs can be directly transformed into IAs. FIA does not satisfy $\wedge$C, but since terms can be transformed into FIAs in polytime thanks to Lemma 4.3.6, and FIA satisfies $\vee$C, any DNF can be transformed into an FIA in polytime. □

In order to prove negative results about succinctness, we rely on an important lemma, due to Kautz and Selman [KS92a].

Lemma 4.3.10. It is impossible to find a polysize compilation function comp such that for any propositional CNF Σ and any propositional clause γ, checking whether $\Sigma \models \gamma$ using comp(Σ) can be done in polytime, unless the polynomial hierarchy PH collapses at the second level.

We use in particular a corollary of this lemma, using a more "knowledge compilation map"-oriented terminology.

Lemma 4.3.11. Let L be a Boolean representation language satisfying **CE**. It holds that $\texttt{L} \not\leqslant_s \texttt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ unless PH collapses at the second level.

This allows us to prove that FIA is strictly less succinct than IA, modulo the collapse of PH.

Proposition 4.3.12. It holds that $\texttt{FIA} \not\leqslant_s \texttt{IA}$ unless PH collapses.

Proof. We know that FIA supports **CE** [Prop. 4.3.2]. Thanks to Lemma 4.3.11, we get that FIA cannot be at least as succinct as $\texttt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ unless PH collapses.

Now, $\texttt{IA} \leqslant_s \texttt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ holds, as a direct consequence of Proposition 4.3.9. Therefore, if $\texttt{FIA} \leqslant_s \texttt{IA}$ were true, this would mean that $\texttt{FIA} \leqslant_s \texttt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ by transitivity—yet it is impossible unless PH collapses. Hence $\texttt{FIA} \not\leqslant_s \texttt{IA}$, unless PH collapses. □

Theorem 4.3.13. The results in Figure 4.10 hold.

Proof. Direct from Propositions 4.1.7, 4.2.5, 4.3.9, and 4.3.12, and from various inclusions between languages. □

4.3.3 Queries and Transformations

Theorem 4.3.14. The results in Tables 4.1 and 4.2 hold.

Detailed proof p. 124.

Many of the proofs directly follow some result from the existing compilation map [Theorem 1.4.18]. For example, the fact that IA does not support any query except for **MC** comes from the fact that $\texttt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$ does not either: since $\texttt{BDD}_{\mathcal{B}}^{\$\mathbb{B}} \subseteq \texttt{IA}$, IA cannot support queries unsupported by $\texttt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$, as stated in Proposition 1.2.18.

This makes IA weak with respect to the vast majority of queries. Imposing focusingness makes more queries tractable, including **CO** and **MX**, that are especially

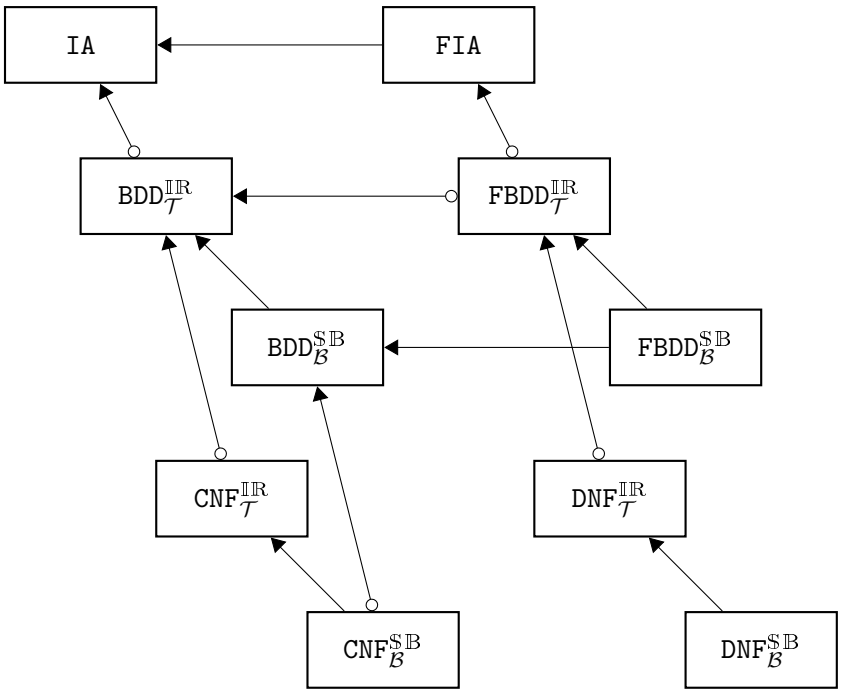

Figure 4.10: Relative succinctness of IA, FIA, and various languages over $\mathbb{R}$-tileable and Boolean variables. On an edge linking $\mathtt{L}_1$ and $\mathtt{L}_2$, if there is an arrow pointing towards $\mathtt{L}_1$, it means that $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$. If there is no symbol on $\mathtt{L}_1$'s side (neither an arrow nor a circle), it means that $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$. If there is a circle on $\mathtt{L}_1$'s side, it means that it is unknown whether $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$ or $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$ holds. Relations deducible by transitivity are not represented, which means that two fragments not being ancestors to each other are incomparable with respect to succinctness.

L	CO	VA	MC	CE	IM	EQ	SE	MX	CX	CT	ME
IA	○	○	√	○	○	○	○	○	○	○	○
FIA	√	○	√	√	○	○	○	√	√	○	√

Table 4.1: Results about queries; √ means "satisfies", and ○ means "does not satisfy, unless P = NP".

L	CD	TR	FO	SFO	EN	SEN	$\vee$C	$\vee$BC	$\vee$clC	$\wedge$C	$\wedge$BC	$\wedge$tC
IA	√	○	○	√	○	√	√	√	√	√	√	√
FIA	√	√	√	√	○	○	√	√	√	○	○	√

Table 4.2: Results about transformations; √ means "satisfies", and ○ means "does not satisfy, unless P = NP".

important in our context. Interestingly enough, not so many transformations are rendered untractable by this restrictive property. Some even become polynomial, such as forgetting. This makes FIA close to DNNF in terms of knowledge compilation "profile", despite FIA *not* being a decomposable language.

Note that the negation transformation ($\neg$C) does not appear in the table. This is a matter of expressivity: indeed, the IA language is not closed under negation, because open intervals cannot be represented.

As a side note, a corollary of this theorem is that neither reduced IAs nor reduced FIAs are canonical—contrary to OBDDs—since if they were, **EQ** would be polytime.

*
**

Now that theoretical aspects of IAs are known, next chapter (starting page 127, after the proofs) deals with a more practical side, namely the actual *compilation* into IA, and more precisely, into FIA.

4.4 Chapter Proofs

4.4.1 Reduction

Proof of Proposition 4.1.15 [p. 101]. We prove that Algorithm 4.1 [p. 102] is polytime and transforms any IA φ into an equivalent reduced IA φ' such that $\|\varphi'\| \leqslant \|\varphi\|$. Let us apply Algorithm 4.1 on an IA φ.

For a given node N:

- the operation of line 3 suppresses N if it is stammering;
- the operation of line 9 suppresses all dead outgoing edges of N;
- the operation of line 14 suppresses all contiguous outgoing edges of N;
- the operation of line 18 suppresses N if it is undecisive;
- the operation of line 24 suppresses all nodes that are isomorphic to N.

and the algorithm stops when no operation has to be applied, so obviously the resulting IA is reduced. Moreover, it is easy to verify that each operation leaves the semantics of φ unchanged. Finally, every operation removes strictly more edges or nodes than it creates; indeed, the only operation that creates anything is one removing stammering nodes, and recall that either $\text{In}(N_i)$ or $\text{Out}(N_i)$ contains only one element.

This proves that

(i) the algorithm eventually stops (once the IA is empty, it cannot change anymore);

(ii) the size of the resulting IA is lower than $\|\varphi\|$ (the only case where the size does not change is when the input IA is already reduced).

The computation for each node is clearly polynomial; the only difficulty is checking whether two disjoint edges are contiguous. This can simply be done by verifying whether $(I_1 \cup I_2) \cap \text{Dom}(x) = I_{\text{min}} \cap \text{Dom}(x)$, where I_{min} is the narrowest interval covering $(I_1 \cup I_2) \cap \text{Dom}(x)$. It can be shown that if the property is not true for I_{min}, it cannot be true for any I. Now, the traversal loop (line 2) handles each node once; as a result, what is inside the outer loop (from line 1 to line 27) is processed in polytime.

Since reducibility properties are not mutually independent, the traversal must be repeated while it has modified something in φ (line 28). This does not change polynomiality: a traversal always lowers the size of φ (except of course for the last one), therefore the traversal loop is not repeated more than $\|\varphi\|$ times.

Note that there obviously exists more efficient methods to reduce an IA, but the only point here is to show that this operation is polytime. □

Proof of Proposition 4.1.16 [p. 101]. Let φ be an IA such that $\text{Scope}(\varphi) = \varnothing$. We show that φ is either the empty or the sink-only automaton.

$\text{Scope}(\varphi) = \varnothing$ means that φ does not mention any variable; if it has non-leaf nodes, they can thus only be $\veebar$-nodes. We show that φ does not contain $\veebar$-nodes. Suppose it contains at least one $\veebar$-node: there must exist a node N all the outgoing edges of which point to the sink (because IAs are acyclic DAGs, which means that there always exists a topological sorting of their nodes [see e.g. CL$^+$01, § 22.4]). Since φ is reduced, each outgoing edge $E \in \text{Out}(N)$ verifies $\text{Lbl}(E) \cap \text{Dom}(\text{Var}(E)) \neq \varnothing$; since $\text{Var}(E) = \veebar$ and $\text{Dom}(\veebar) = \{0\}$, it implies that $\bigcap_{E \in \text{Out}(N)} \text{Lbl}(E) = \{0\}$. Hence, there is only one edge E going out of N (φ is reduced, so there are no contiguous edges). This edge verifies $\text{Dom}(\text{Var}(E)) \subseteq \text{Lbl}(E)$, hence N is undecisive, which is impossible since φ is reduced. Therefore, φ does not contain any $\veebar$-node—and since it cannot contain any other non-leaf node, it is either the empty or the sink-only automaton. □

4.4.2 Sublanguages

Proof of Proposition 4.2.1 [p. 103]. We prove that there exists a linear algorithm mapping every IA φ and every assignment $\vec{y}$ to an IA equivalent to the restriction $[\![\varphi]\!]_{|\vec{y}}$ of $[\![\varphi]\!]$ to $\vec{y}$.

Let φ be an IA, $Y \subseteq \mathbb{TR}$ a set of $\mathbb{R}$-tileable variables, and $\vec{y}$ a Y-assignment. We denote as $\varphi_{|\vec{y}}$ the IA obtained by applying Algorithm 4.2 [p. 103] on φ and $\vec{y}$. Automaton $\varphi_{|\vec{y}}$ is obtained in time linear in $\|\varphi\|$; let us now show that $[\![\varphi_{|\vec{y}}]\!] \equiv [\![\varphi]\!]_{|\vec{y}}$.

By definition of the conditioning, $\vec{z} \in \text{Dom}(\text{Scope}(\varphi) \setminus Y)$ is a model of $[\![\varphi]\!]_{|\vec{y}}$ if and only if $\vec{y} \,.\, \vec{z}$ is a model of φ.

($\Leftarrow$) Suppose that $\overrightarrow{y} \cdot \overrightarrow{z}$ is a model of φ. Then there is in φ a path p compatible with $\overrightarrow{z} \cdot \overrightarrow{y}$. By construction, a copy p' of p exists in $\varphi_{|\overrightarrow{y}}$, and $\overrightarrow{z}$ is compatible with p': $\overrightarrow{z}$ is a model of $\varphi_{|\overrightarrow{y}}$.

($\Rightarrow$) Suppose that $\overrightarrow{y} \cdot \overrightarrow{z}$ is a not model of φ. Any path p in φ contains an edge E such that $(\overrightarrow{y} \cdot \overrightarrow{z})_{|\operatorname{Var}(E)} \notin \operatorname{Lbl}(E)$. Recall that paths in $\varphi_{|\overrightarrow{y}}$ are the same as those in φ, and let p' be the one corresponding to p. If $\operatorname{Var}(E) \in Y$, the fact that $(\overrightarrow{y} \cdot \overrightarrow{z})_{|\operatorname{Var}(E)} \notin \operatorname{Lbl}(E)$ has led the algorithm to label the corresponding edge in $\varphi_{|\overrightarrow{y}}$ by $\varnothing$. Consequently, $\overrightarrow{z}$ cannot be compatible with this path. If $\operatorname{Var}(E) \notin Y$, $(\overrightarrow{y} \cdot \overrightarrow{z})_{|\operatorname{Var}(E)} \notin \operatorname{Lbl}(E)$ means that $\overrightarrow{z}_{|\operatorname{Var}(E)} \notin \operatorname{Lbl}(E)$: because such edges remain unchanged in $\varphi_{|\overrightarrow{y}}$, $\overrightarrow{z}$ cannot be compatible with p'. Therefore $\overrightarrow{z}$ is incompatible with all paths in $\varphi_{|\overrightarrow{y}}$, and $\overrightarrow{z}$ is thus not a model of $\varphi_{|\overrightarrow{y}}$.

Hence $[\![\varphi_{|\overrightarrow{y}}]\!] \equiv [\![\varphi]\!]_{|\overrightarrow{y}}$. □

Lemma 4.4.1. `IA` does not satisfy **CO**, **VA**, **CE**, **IM**, **EQ**, **SE**, **MX**, **CX**, **CT**, or **ME**, unless $\mathsf{P} = \mathsf{NP}$.

Proof. Corollary 4.1.8 states that $\texttt{BDD}_{\mathcal{B}}^{\$\mathbb{B}} \subseteq \texttt{IA}$. This implies that $\texttt{BDD}_{\mathcal{B}}^{\$\mathbb{B}} \geqslant_p \texttt{IA}$ [Prop. 1.2.15], and thus all queries supported by `IA` are also supported by $\texttt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$ [Prop. 1.2.18]. Since Theorem 1.4.18 states that $\texttt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$ does not satisfy **CO**, **VA**, **CE**, **IM**, **EQ**, **SE**, **CT**, or **ME**, unless $\mathsf{P} = \mathsf{NP}$, `IA` cannot satisfy any of these queries.

In addition, since both **MX** and **CX** respectively imply **CO** [Prop. 3.3.2], it proves that `IA` does not satisfy **MX** or **CX** unless $\mathsf{P} = \mathsf{NP}$. □

Proof of Proposition 4.2.2 [p. 103]. The result is stated in Lemma 4.4.1. □

Proof of Proposition 4.2.4 [p. 105]. We first show that Algorithm 4.1 [p. 102] maintains the property of focusing with respect to a given variable. Let φ be an IA that is focusing w.r.t. $y \in \operatorname{Scope}(\varphi)$. Let us suppose that the algorithm is treating node N, with $\operatorname{Var}(N) = y$.

- "Stammering" operation: let $E \in \operatorname{Out}(N)$ and E' be an edge on a path from $\operatorname{Dest}(E)$ to the sink such that $\operatorname{Var}(E') = y$. E' is focusing, so $\operatorname{Lbl}(E') \subseteq \operatorname{Lbl}(E)$. Thus $\operatorname{Lbl}(E') \subseteq \operatorname{Lbl}(E) \cap \operatorname{Lbl}(E_{\text{in}})$ for any $E_{\text{in}} \in \operatorname{In}(N)$. Hence E' is still focusing after the "stammering" operation.
- "Dead" operation: suppressing edges does not have any influence on the focusingness of other edges in the graph.
- "Contiguous" operation: the two contiguous edges point to the same node, so every descendant edge E associated with y is such that $\operatorname{Lbl}(E)$ is included in the label of both contiguous edges; hence it is included in their union.
- "Undecisive" operation: every descendant edge of the child of N is also a descendant edge of N, so this operation does not compromise their focusingness.

- "Isomorphism" operation: since every outgoing edge of every node isomorphic to N is focusing, redirecting all the parent edges to N is harmless.

Hence Algorithm 4.1 maintains the property of focusing with respect to a given variable.

Now, suppose it is applied to an FIA φ; since by definition, φ is focusing with respect to all variables in Scope(φ), the resulting IA also is. The algorithm thus maintains the focusing property, hence the result. □

Proof of Proposition 4.2.6 [p. 105]. We show that Algorithm 4.2 [p. 103] maintains focusingness. Let φ be a FIA, $Y \subseteq \mathbb{TR}$ a set of $\mathbb{R}$-tileable variables, and $\vec{y}$ a Y-assignment. Let $\varphi_{|\vec{y}}$ be the IA defined in the proof of Proposition 4.2.1. We show that $\varphi_{|\vec{y}}$ is focusing.

Indeed, in $\varphi_{|\vec{y}}$ the only edges that have been modified are those that correspond to edges in φ associated with a variable in Y. In $\varphi_{|\vec{y}}$, these modified edges are all associated with $\veebar$, and thus are not concerned by the focusingness restriction (see Definition 4.2.3).

As for the other edges in $\varphi_{|\vec{y}}$, since they *all* remain unchanged, they are still focusing. Consequently, $\varphi_{|\vec{y}}$ is focusing w.r.t. all variables from Scope(φ) $\setminus Y =$ Scope($\varphi_{|\vec{y}}$); it is thus a FIA. Hence, Algorithm 4.2 maintains focusingness, which proves that `FIA` supports **CD**. □

Proof of Proposition 4.2.7 [p. 105]. Let φ be a *reduced* FIA; we prove that φ is empty if and only if it is inconsistent. The implication is trivial; now let us suppose that φ is not empty. If it is the sink-only automaton, it is obviously consistent. In the other case, it has at least one non-$\veebar$ edge, since a reduced IA cannot contain only $\veebar$-nodes [Prop. 4.1.16].

Let us consider a non-$\veebar$ edge E. Because φ is reduced, there is no dead edge: Lbl(E) contains at least one value $\omega \in$ Dom(Var(E)). As E is focusing, ω is coherent with all the preceding edges in φ. Since it is the case for all non-$\veebar$ edges, φ cannot be inconsistent.

Thanks to the fact that reduction is polynomial on FIAs [Proposition 4.2.4], we overall get that `FIA` satisfies **CO**. □

Proof of Proposition 4.2.8 [p. 105]. We prove that Algorithm 4.3 [p. 106] is polynomial, and that when given an FIA φ, it returns a model of φ if φ is consistent, and stops without returning anything otherwise.

Let us first show that it is polynomial in $\|\varphi\|$. It is clear enough, since reduction is polynomial, and the edge and value selection steps can be done in constant time. Since the procedure explores a single path, it cannot loop more than $\|\varphi\|$ times, hence the polynomiality of the whole procedure.

Now, the reduction step eliminates the case when φ is inconsistent. Let us prove that the assignment the procedure returns is a model of φ. Starting from the sink, we choose a path to the root. On the chosen path, each time a new variable is encountered, a value is selected, that is compatible with the edge label and with the

variable domain. Such a value exists, since φ is reduced. Because we do this only when we encounter each variable for the first time, we can be sure that the selected value is consistent: it is included in all other labels pertaining to this variable from this node upwards, since the automaton is focusing. Since variables in Scope(φ) that have not been encountered on the chosen path had already been set to a random value in their domain (line 4), the assignment $\vec{y}$ returned by this algorithm is a model of φ. □

4.4.3 Preliminaries to the Map

Context on FIAs

Proof of Proposition 4.3.1 [p. 107]. Algorithm 4.4 [p. 107] is polynomial: each node and edge is encountered at most once, and the per-edge treatment is polynomial. Now, we prove that when given an FIA φ and a variable x, it returns the context of x in φ.

First, the procedure reduces the FIA φ. If it is empty, it is inconsistent: an empty context is returned. If it is not empty, it is consistent [see proof of Prop. 4.2.7, p. 115], and every path is a consistent path.

If a node N is marked, by contruction, it means that there exists a path from N to the sink not mentioning x. If N is the root, it obviously means that $\text{Ctxt}_\varphi(x) = \text{Dom}(x)$, since all paths are consistent.

Let us suppose that the root is not marked. We consider a marked node N and an edge E pointing to N. If $\text{Var}(E) = x$, there exists a path from the root to the sink of φ in which E is the last x-edge (since N is marked). Thanks to the fact that φ is focusing, E is included in the label of all its ancestor edges: all values in $\text{Lbl}(E)$ are consistent—except for those that are not in $\text{Dom}(x)$.

This proves that $C \subseteq \text{Ctxt}_\varphi(x)$. Let us now prove that $C \supseteq \text{Ctxt}_\varphi(x)$: consider a value $\omega \in \text{Ctxt}_\varphi(x)$. Since the root is not marked, every path in φ contains at least one x-edge. There must exist an edge E such that $\omega \in \text{Lbl}(E)$, otherwise ω could obviously not be a consistent value. Suppose there exists no path in which the label of the last x-edge contains ω. Then all paths from $\text{Dest}(E)$ to the sink contain an x-edge not mentioning ω. Thus, there cannot be a model of φ in which x is assigned to ω. This is a contradiction, since ω is supposed to be a consistent value. As a consequence, there exists a path in which the label of the last x-edge contains ω. By construction of C, this means that $\omega \in C$. Since this is true for any $\omega \in \text{Ctxt}_\varphi(x)$, we get that $C \supseteq \text{Ctxt}_\varphi(x)$. □

Clausal Entailment on FIAs

The following lemma helps proving Proposition 4.3.2.

Lemma 4.4.2. Let φ be an FIA, and γ be a term over literals of the form "$x \in I$", with I an *open or closed* interval.

Algorithm 4.5, given as input φ and γ, has the following properties:

- it is linear in $\|\varphi\| \cdot \|\gamma\|$;

Algorithm 4.5 Restriction of an FIA φ to a term γ.

```
1:  for each node N in φ, from the sink to the root do
2:      let x := Var(N)
3:      if x ∈ Scope(γ) then
4:          for each E ∈ Out(N) do
5:              let S := Lbl(E)
6:              for each literal ⟨x, A⟩ in γ do
7:                  S := S ∩ A
8:              if S = ∅ then
9:                  label E by ∅
10:             else
11:                 label E by {0}
12:         label N by ⊻
```

- it preserves focusingness;
- the IA it returns is equivalent to the restriction of φ to γ.

Proof. Each node of φ is handled once; for each node, each literal of γ is considered at most once. S always remains an interval (be it open or closed), so the intersection of line 7 can be done in constant time. Overall, the procedure is therefore linear in $\|\varphi\| \cdot \|\gamma\|$.

The second point is trivial, since the procedure only replaces variable nodes by $\veebar$-nodes: edges that were focusing in φ remain so after the application of this procedure.

To prove the last point, we denote the returned automaton as ψ, $V_\gamma = \text{Scope}(\gamma)$, $V_\varphi = \text{Scope}(\varphi)$, and $V_\psi = \text{Scope}(\psi)$. The goal is to show that $[\![\psi]\!] \equiv [\![\varphi]\!]_{|\gamma}$, i.e., $V_\psi = V_\varphi \setminus V_\gamma$ and for all $\vec{v} \in \text{Dom}(V_\psi)$,

$$[\![\psi]\!](\vec{v}) = \top \iff \exists \vec{g} \in \text{Dom}(V_\gamma), ([\![\varphi]\!] \wedge [\![\gamma]\!])(\vec{v} \cdot \vec{g}) = \top.$$

It is obvious that $V_\psi = V_\varphi \setminus V_\gamma$, since all nodes labeled with a variable in V_γ are removed from φ. Now, let us consider a V_ψ-assignment $\vec{v}$.

First, we suppose there exists a $\vec{g} \in \text{Dom}(V_\gamma)$ such that $([\![\varphi]\!] \wedge [\![\gamma]\!])(\vec{v} \cdot \vec{g}) = \top$. That means $[\![\varphi]\!](\vec{v} \cdot \vec{g}) = \top$ and $[\![\gamma]\!](\vec{v} \cdot \vec{g}) = \top$. Hence, there exists a path p in φ compatible with $\vec{v} \cdot \vec{g}$ and all literals $\langle x, A\rangle$ in γ are such that $\vec{g}_{|x} \in A$.

Let us consider the path p' in ψ corresponding to path p in φ. Since nodes labeled with variables not in V_γ are left unchanged in ψ, $\vec{v}$ is compatible with p'. The other nodes are replaced by $\veebar$-nodes in ψ. Let us consider an x-edge E on path p, with $x \in V_\gamma$. Since all literals $\langle x, A\rangle$ in γ are such that $\vec{g}_{|x} \in A$, the set S computed in the procedure cannot be empty. Hence the edge corresponding to E in ψ is labeled $\{0\}$, and is thus trivially compatible with $\vec{v}$. As it is the case for all x-edges on p with $x \in V_\gamma$, we get that p' is compatible with $\vec{v}$, and therefore that $[\![\psi]\!](\vec{v}) = \top$. This proves the first part of the last point.

Now, let us suppose that $[\![\psi]\!](\vec{v}) = \top$. There exists a path in ψ compatible with $\vec{v}$. We call this path p', and denote the corresponding path in φ as p. Obviously, $\vec{v}$ is compatible with p (nodes labeled with variables not in V_γ are not modified in φ).

If p contains no node labeled with a variable in V_γ, then any model $\vec{g}$ of γ trivially satisfies the requirement $([\![\varphi]\!] \wedge [\![\gamma]\!])(\vec{v} \,.\, \vec{g}) = \top$. Otherwise, let us consider an edge E on path p, labeled by some variable $x \in V_\gamma$. Since p' is compatible with $\vec{v}$, it means that E has been replaced by a nonempty $\trianglelefteq$-edge in ψ, and thus that the set S computed in the procedure was not empty. This implies that there exists a value ω such that $\omega \in \mathrm{Lbl}(E)$ and ω is a consistent value for x in γ. We can find such a value for any variable in V_γ that is mentioned in p; that means we can easily find a V_γ-assignment verifying $([\![\varphi]\!] \wedge [\![\gamma]\!])(\vec{v} \,.\, \vec{g}) = \top$. This proves the second part of the last point. □

Proof of Proposition 4.3.2 [p. 108]. Let φ be an FIA. Let $\{x_1, \ldots, x_k\} \subseteq \mathcal{T}$. Let $\{I_1, \ldots, I_k\} \subseteq \mathbb{IR}$. We consider the clause $\gamma = \bigvee_{1 \leqslant i \leqslant k} \big[x_i \in I_i\big]$. It holds that $\varphi \models \gamma$ if and only if $[\![\varphi]\!] \wedge \neg[\![\gamma]\!]$ is inconsistent. This is in turn equivalent to checking whether $[\![\varphi]\!]_{|\neg[\![\gamma]\!]}$ is inconsistent.

Since $\neg[\![\gamma]\!]$ is equivalent to $\bigwedge_{1 \leqslant i \leqslant k} \big[x_i \notin I_i\big]$, restricting φ to the negation of γ boils down to restricting it to a conjunction of (possibly open) intervals. Thanks to Lemma 4.4.2, the application of which is not limited to terms on *closed* intervals, we know we can obtain in polytime an FIA equivalent to $[\![\varphi]\!]_{|\neg[\![\gamma]\!]}$. Since FIA supports **CO** [Prop. 4.2.7], we can also check in polytime whether this FIA is consistent or not—and thus, whether or not φ entails clause γ. □

Combining Automata

Lemma 4.4.3. Let L be a Boolean representation language; if $\mathtt{L} \leqslant_p \mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ and L satisfies **CO**, then L does not satisfy $\wedge$**BC** unless $\mathsf{P} = \mathsf{NP}$.

Proof. The proof is adapted from Darwiche and Marquis [DM02]: assuming $\mathtt{L} \leqslant_p \mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ and L satisfies **CO**, if L satisfied $\wedge$**BC**, we would have a polytime algorithm to decide whether the conjunction of two OBDDs (the variable orderings being possibly different in each OBDD) is consistent; yet, this problem is NP-complete, as shown by Meinel and Theobald [MT98, Lemma 8.14]. Therefore L does not support $\wedge$**BC** unless $\mathsf{P} = \mathsf{NP}$. □

Proof of Proposition 4.3.5 [p. 108]. Proposition 4.2.5 states that $\mathtt{FBDD}_{\mathcal{B}}^{\$\mathbb{B}} \subseteq \mathtt{FIA}$. Therefore $\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}} \subseteq \mathtt{FIA}$, and $\mathtt{FIA} \leqslant_p \mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ [Prop. 1.2.15]. Since FIA satisfies **CO** [Prop. 4.2.7], Lemma 4.4.3 implies that FIA does not satisfy $\wedge$**BC**, and thus $\wedge$**C**, unless $\mathsf{P} = \mathsf{NP}$. □

Meshes and Single Quantification

The key property about meshes is that they are easy to obtain.

Lemma 4.4.4 (Obtaining a mesh)**.** It is possible to build in quasi-linear time a mesh $\mathcal{M} = \{M_1, \ldots, M_n\}$ of a variable in any IA.

Proof. Let φ be an IA and $x \in \text{Scope}(\varphi)$. We only have to recover the finite bounds of all intervals associated with x, i.e., lower and upper bounds of all disjoint intervals constituting $\text{Dom}(x)$ and of all x-edge labels; this process is linear (simple traversal of the graph). Let $B = \{b_1, \ldots, b_k\}$ be the obtained set of bounds, sorted in ascending order (quasi-linear process). We consider the following set:

$$\mathcal{P} = \Big\{ [-\infty, b_1[, \{b_1\},]b_1, b_2[, \{b_2\}, \ldots, \\ ]b_{i-1}, b_i[, \{b_i\},]b_i, b_{i+1}[, \ldots, \{b_k\},]b_k, +\infty] \Big\}.$$

Set $\mathcal{P}$ is obviously a partition of $\mathbb{R}$: all sets are pairwise disjoint and their union is $\mathbb{R}$. Now, $\mathcal{M} = \{\, p \in \mathcal{P} \mid p \subseteq \text{Dom}(x) \,\}$ is a mesh of x in φ.

- We show that $\mathcal{M}$ is a partition of $\text{Dom}(x)$. All elements of $\mathcal{M}$ are pairwise disjoint and included in $\text{Dom}(x)$; let us prove that $\text{Dom}(x)$ is entirely covered by $\mathcal{M}$. Let $\omega \in \text{Dom}(x)$, and let p be the unique element of $\mathcal{P}$ that contains ω. If p is a singleton, then obviously $p \in \mathcal{M}$, since $p \subseteq \text{Dom}(x)$. If p is not a singleton, it is by construction an open interval containing no bound of any interval constituting $\text{Dom}(x)$; since it contains one value from $\text{Dom}(x)$, it must be included in one of the intervals constituting $\text{Dom}(x)$, and thus $p \in \mathcal{M}$. Consequently, $\mathcal{M}$ covers the whole $\text{Dom}(x)$ and is thus a partition.
- Let E be an edge in φ such that $\text{Var}(E) = x$, and i an integer such that $M_i \cap \text{Lbl}(E) \neq \varnothing$. By construction of $\mathcal{M}$, M_i is either a singleton, or an interval opened on both sides. In the first case, it is straightforward that $M_i \subseteq \text{Lbl}(E)$. In the second case, it is impossible for M_i to contain any bound of $\text{Lbl}(E)$, by construction; thus $M_i \subseteq \text{Lbl}(E)$.

All in all, $\mathcal{M}$ is a partition of $\text{Dom}(x)$ such that for all $i \in \{1, \ldots, n\}$, if $M_i \cap \text{Lbl}(E) \neq \varnothing$ holds, then $M_i \subseteq \text{Lbl}(E)$. By definition, it is a mesh of x in φ. □

Meshes have very interesting properties, that come from the following fact: values in a mesh element are interchangeable with respect to conditioning.

Lemma 4.4.5 (Conditioning on a mesh element)**.** Let φ be an IA, $x \in \text{Scope}(\varphi)$, and $\mathcal{M}$ be a mesh of x in φ. Let M be an element of the mesh. For any two values m and m' in M, it holds that $[\![\varphi]\!]_{|x=m} \equiv [\![\varphi]\!]_{|x=m'}$.

Proof. Let $Z = \text{Scope}(\varphi) \setminus \{x\}$, and $\vec{z}$ a Z-assignment. We write $\vec{m}$ the assignment of x to m and $\vec{m}'$ the assignment of x to m'.

(⇒) Suppose that $[\![\varphi]\!]_{|\vec{m}}(\vec{z}) = \top$, then $[\![\varphi]\!](\vec{m} \,.\, \vec{z}) = \top$. Consequently, there exists a path p in φ that is compatible with $\vec{m}$ and $\vec{z}$. Since p is compatible with $\vec{m}$, any x-edge E along p verifies $m \in \text{Lbl}(E)$. Therefore $M \cap \text{Lbl}(E) \neq \varnothing$. By definition of a mesh, $M \subseteq \text{Lbl}(E)$. Hence $m' \in \text{Lbl}(E)$. As this holds for any E, p is also compatible with $\vec{m}'$, and consequently, $[\![\varphi]\!](\vec{m}' \,.\, \vec{z}) = \top$. Finally, we get $[\![\varphi]\!]_{|\vec{m}'}(\vec{z}) = \top$.

($\Leftarrow$) Suppose that $[\![\varphi]\!]_{|\vec{m}}(\vec{z}) = \bot$, then $[\![\varphi]\!](\vec{m} \,.\, \vec{z}) = \bot$. Any path in φ that is compatible with $\vec{z}$ is not compatible with $\vec{m}$.

If there existed a path p compatible with $\vec{z}$ and $\vec{m}'$, this would mean that any x-edge E along p would verify $m' \in \text{Lbl}(E)$. Thus, we would get $M \cap \text{Lbl}(E) \neq \varnothing$, and by definition of a mesh, $M \subseteq \text{Lbl}(E)$. Hence, E, and thus p, would be compatible with $\vec{m}$, which is contradictory.

Therefore, no path in φ can be compatible with $\vec{z}$ and $\vec{m}'$: $[\![\varphi]\!]_{|\vec{m}'}(\vec{z}) = \bot$.

This proves that $[\![\varphi]\!]_{|x=m} \equiv [\![\varphi]\!]_{|x=m'}$. □

The following lemma is a direct consequence of this property.

Lemma 4.4.6 (Quantification with a mesh)**.** Let φ be an IA, $x \in \text{Scope}(\varphi)$, and $\mathcal{M} = \{M_1, \ldots, M_n\}$ be a mesh of x in φ.

Let $\langle m_1, \ldots, m_n \rangle$ be a sequence of values such that for any $i \in \{1, \ldots, n\}$, $m_i \in M_i$. We show that

$$\exists x.[\![\varphi]\!] \equiv \bigvee_{i=1}^{n} [\![\varphi]\!]_{|x=m_i} \qquad \text{and} \qquad \forall x.[\![\varphi]\!] \equiv \bigwedge_{i=1}^{n} [\![\varphi]\!]_{|x=m_i}.$$

Proof. Thanks to Shannon's decomposition [Prop. 1.4.12], we obtain $\exists x.[\![\varphi]\!] \equiv \bigvee_{\omega \in \text{Dom}(x)} [\![\varphi]\!]_{|x=\omega}$ and $\forall x.[\![\varphi]\!] \equiv \bigwedge_{\omega \in \text{Dom}(x)} [\![\varphi]\!]_{|x=\omega}$.

Since $\text{Dom}(x) = \bigcup_{i=1}^{n} M_i$ (by definition of a mesh), we can write

$$\exists x.[\![\varphi]\!] \equiv \bigvee_{i=1}^{n} \bigvee_{\omega \in M_i} [\![\varphi]\!]_{|x=\omega} \qquad \text{and} \qquad \forall x.[\![\varphi]\!] \equiv \bigwedge_{i=1}^{n} \bigwedge_{\omega \in M_i} [\![\varphi]\!]_{|x=\omega}.$$

Now, thanks to Lemma 4.4.5, we know that for all $i \in \{1, \ldots, n\}$, any $\omega \in M_i$ verifies $[\![\varphi]\!]_{|x=\omega} \equiv [\![\varphi]\!]_{|x=m_i}$. Hence

$$\bigvee_{\omega \in M_i} [\![\varphi]\!]_{|x=\omega} \quad \equiv \quad \bigwedge_{\omega \in M_i} [\![\varphi]\!]_{|x=\omega} \quad \equiv \quad [\![\varphi]\!]_{|x=m_i}$$

and we get the result. □

We often use such sequences of values $\langle m_1, \ldots, m_n \rangle$ verifying $m_i \in M_i$ for any $i \in \{1, \ldots, n\}$. We call these values "indexes", and use $\text{Indexes}(x)$ to denote a sequence of indexes for some mesh of x.

Proof of Proposition 4.3.8 [p. 109]. Let φ be an L-representation, and let $x \in \text{Scope}(\varphi)$. Lemma 4.4.4 states that it is possible to build in time quasi-linear in $\|\varphi\|$ a mesh $\mathcal{M} = \{M_1, \ldots, M_n\}$ of x in φ. We can build an "index sequence" $\text{Indexes}(x) = \langle m_1, \ldots, m_n \rangle$ from this mesh.

Now, thanks to Lemma 4.4.6, we know that $\exists x.[\![\varphi]\!] \equiv \bigvee_{i=1}^{n} [\![\varphi]\!]_{|x=m_i}$; to obtain an L-representation of $\exists x.[\![\varphi]\!]$, it is sufficient to make n conditionings, and to disjoin the results together. Thus, if L satisfies **CD** and $\vee$**C**, it satisfies **SFO**.

Similarly, Lemma 4.4.6 states that $\forall x.[\![\varphi]\!] \equiv \bigwedge_{i=1}^{n} [\![\varphi]\!]_{|x=m_i}$. We can thus obtain an L-representation of $\forall x.[\![\varphi]\!]$ by making n conditionings and conjoining the results together. Hence, if L satisfies **CD** and $\wedge$**C**, it satisfies **SEN**. □

Lemma 4.4.7. IA and FIA satisfy **SFO**. IA satisfies **SEN**.

Proof. We get these results by a direct application of Proposition 4.3.8, since IA satisfies **CD** [Prop. 4.2.1], $\vee$**C** [Prop. 4.3.3], and $\wedge$**C** [Prop. 4.3.4]; and FIA satisfies **CD** [Prop. 4.2.6] and $\vee$**C** [Prop. 4.3.3]. □

4.4.4 Queries and Transformations

Remaining Queries

Lemma 4.4.8. IA and FIA satisfy **MC**.

Proof. Let φ be an IA and $\vec{v} \in \mathrm{Dom}(\mathrm{Scope}(\varphi))$. Since IA satisfies the **CD** transformation [Prop. 4.2.1], we can obtain in polytime an IA $\varphi_{|\vec{v}}$ equivalent to the restriction of φ by $\vec{v}$. By definition of the restriction, $\mathrm{Scope}(\varphi_{|\vec{v}}) = \varnothing$; thanks to Proposition 4.1.16, after having reduced $\varphi_{|\vec{v}}$, we obtain either the empty or the sink-only automaton. Since **MC** is a query and FIA $\subseteq$ IA, Propositions 1.2.15 and 1.2.18 show that FIA also supports **MC**. □

Lemma 4.4.9. FIA does not satisfy **VA**, **IM**, **EQ**, **SE**, or **CT**, unless $\mathsf{P} = \mathsf{NP}$.

Proof. We apply Proposition 1.2.18: $\mathtt{DNF}_{\mathcal{B}}^{\mathbb{SB}}$ does not satisfy any of these queries [Th. 1.4.18], and it is polynomially translatable into FIA [Prop. 4.3.9]. □

Lemma 4.4.10. FIA satisfies **ME**.

Proof. Let φ be a FIA. We check in polytime whether φ is consistent [Prop. 4.2.7]; if it is not the case, the empty set is returned. Otherwise, we build a decision tree representing a union of boxes, that is, a DNF, equivalent to $\mathrm{Mod}(\varphi)$. For each variable $x_i \in \mathrm{Scope}(\varphi) = \{x_1, \ldots, x_n\}$, let us build (in time quasi-linear in $\|\varphi\|$ [Lemma 4.4.4]) a mesh $\mathcal{M}^i = \{M_1^i, \ldots, M_{n_i}^i\}$ of x_i in φ. We also build an index sequence $\mathrm{Indexes}(x_i) = \{m_1^i, \ldots, m_{n_i}^i\}$ for each mesh, that is, for all $j \in [1, \ldots, n_i]$, $m_j^i \in M_j^i$. We denote the assignment of x_i to m_j^i as $\vec{m}_j^i$.

Then, we create a tree T, initially containing only one node, labeled with the empty assignment. We complete the tree thanks to the following process.

1: **for** i from 1 to $|\mathrm{Scope}(\varphi)|$ **do**
2: let $\mathcal{F}$ be the set of T's leaves
3: **for each** F in $\mathcal{F}$ **do**
4: let $\vec{z}$ be the $\{x_1, \ldots, x_{i-1}\}$-assignment label of F
5: **for** j from 1 to n_i **do**
6: **if** $[\![\varphi]\!]_{|\vec{z}.\vec{m}_j^i}$ is consistent **then**
7: add a child to F, labeled $\vec{z}\ .\ \vec{m}_j^i$

Once the tree is entirely built, all leaf labels are models of φ, and replacing each assignment with its corresponding term (each value is a mesh index, so it can be "replaced" by its corresponding mesh element), yields the entire model set of φ.

More precisely, for each leaf F in T, we consider its label $\vec{z}$. It is a Scope(φ)-assignment, consisting of mesh indexes. For each i from 1 to n, let $m^i_{j_i}$ be the value assigned to x_i in $\vec{z}$. We know that if we modify $\vec{z}$ so that instead of $m^i_{j_i}$, we assign x_i to any value in $M^i_{i_j}$, the result is still a model of φ—values in each mesh element are interchangeable, thanks to Lemma 4.4.5. Any model of the term $\bigwedge_{i=1}^{n} \left[x_i \in M^i_{j_i}\right]$ is thus a model of φ. The disjunction Δ of the terms associated with all leaf labels in T is hence included in Mod(φ), and it is a $\mathtt{DNF}_{\mathcal{T}}$-representation.

Since a mesh is a partition of the variable's domain, we tested every possible value for each variable, so every model of φ is also a model of at least one of the terms. This proves that our DNF Δ has the same model set as φ.

Let us now show that the process is polynomial. At the end of the algorithm, all leaves of T are at the same level (i.e., all paths of T are of equal length); indeed, for each node, at least one of the tests of line 6 must pass (as the current FIA is consistent). It implies that at each incrementation of i, the number of leaves is less than the number of leaves in the final tree, and thus obviously bounded by $\|\Delta\|$. Moreover, all the n_i are polynomial in $\|\varphi\|$. Hence, the test of line 6 is not done more than a number of times polynomial in $\|\Delta\| \cdot \|\varphi\|$ in the whole algorithm. Now, this test can be made in time polynomial in $\|\varphi\|$, as `FIA` supports **CO** [Prop. 4.2.7] and **CD** [Prop. 4.2.6]. Hence, the global algorithm is polytime in $\|\Delta\|$ and $\|\varphi\|$ (as $n \leqslant \|\varphi\|$). □

Quantification and Restriction

Lemma 4.4.11. `IA` and `FIA` do not satisfy **EN** unless $\mathsf{P} = \mathsf{NP}$.

Proof. If these languages satisfied **EN**, they would also satisfy **VA**. Indeed, if they satisfied **EN**, we could obtain in polytime, for any φ, an automaton ψ equivalent to $\forall V.[\![\varphi]\!]$, with $V = \text{Scope}(\varphi)$. By definition of the quantification [Def. 1.4.9], $\text{Scope}(\psi) = \varnothing$; after reducing ψ, we thus obtain either the empty or the sink-only automaton [Prop. 4.1.16]. It is hence easy to check whether ψ is consistent; and thanks to Proposition 1.4.10, we know that φ is valid if and only if ψ is consistent.

We could thus check in polytime the validity of any automaton; yet none of these languages satisfies **VA** unless $\mathsf{P} = \mathsf{NP}$ [Lemmas 4.4.1 and 4.4.9]. □

Lemma 4.4.12. `IA` does not satisfy **FO** or **TR** unless $\mathsf{P} = \mathsf{NP}$.

Proof. If `IA` satisfied **FO**, it would also satisfy **CO**. Indeed, we could in this case obtain in polytime, for any φ, an automaton ψ equivalent to $\exists V.[\![\varphi]\!]$, with $V = \text{Scope}(\varphi)$. By definition of the quantification [Def. 1.4.9], $\text{Scope}(\psi) = \varnothing$; after reducing ψ, we thus obtain either the empty or the sink-only automaton [Prop. 4.1.16]. Checking whether ψ is consistent is then easy; and thanks to Proposition 1.4.10, we know that φ is consistent if and only if ψ is consistent.

We could thus check in polytime the consistency of any IA; yet it is impossible unless $\mathsf{P} = \mathsf{NP}$ [Lemma 4.4.1]. Hence, `IA` does not satisfy **FO** unless $\mathsf{P} = \mathsf{NP}$.

Now, since forgetting a set of variables $\{x_1, \ldots, x_k\}$ is equivalent to restricting to the valid term $\big[x_1 \in \mathbb{R}\big] \wedge \cdots \wedge \big[x_k \in \mathbb{R}\big]$, if `IA` satisfied **TR**, it would also satisfy **FO**. Hence, `IA` does not satisfy **TR** unless $\mathsf{P} = \mathsf{NP}$. □

Lemma 4.4.13. `FIA` does not satisfy **SEN** unless $\mathsf{P} = \mathsf{NP}$.

Proof. We show that if `FIA` satisfied **SEN**, it would satisfy $\wedge$**BC**. Let φ_1 and φ_2 be two FIAs. Let $Z = \text{Scope}(\varphi_1) \cup \text{Scope}(\varphi_2)$. We consider a variable $x \notin Z$ of domain $\{0, 1\}$. We build the automaton ψ by merging the sinks of φ_1 and φ_2, and adding a root to ψ, labeled x, with one outgoing edge labeled $\{0\}$ and pointing to the root of φ_1, and a second outgoing edge labeled $\{1\}$ and pointing to the root of φ_2. Clearly enough, ψ is focusing, since φ_1 and φ_2 are, and x is mentioned in one node only.

By construction, $[\![\varphi_1]\!] \equiv [\![\psi]\!]_{|x=0}$ (that is, a model $\vec{z}$ of ψ is a model of $[\![\varphi_1]\!]$ if and only if $\vec{z}_{|x} = 0$). Similarly, $[\![\varphi_2]\!] \equiv [\![\psi]\!]_{|x=1}$ (that is, a model $\vec{z}$ of ψ is a model of $[\![\varphi_2]\!]$ if and only if $\vec{z}_{|x} = 1$).

Shannon's decomposition [Prop. 1.4.12] gives $\forall x.[\![\psi]\!] \equiv \bigwedge_{\omega \in \text{Dom}(x)} [\![\psi_{|x=\omega}]\!]$. Hence, $\forall x.[\![\psi]\!] \equiv [\![\psi]\!]_{|x=0} \wedge [\![\psi]\!]_{|x=1}$, that is to say, $\forall x.[\![\psi]\!] \equiv [\![\varphi_1]\!] \wedge [\![\varphi_2]\!]$.

It is possible to build ψ in time linear in the sizes of φ_1 and φ_2. Therefore, if `FIA` supported **SEN**, we would have a polytime algorithm mapping two FIAs to an FIA representing their conjunction. Yet it is impossible, since `FIA` does not satisfy $\wedge$**BC** unless $\mathsf{P} = \mathsf{NP}$ [Prop. 4.3.5]. Hence `FIA` does not support **SEN** unless $\mathsf{P} = \mathsf{NP}$. □

Lemma 4.4.14. `FIA` satisfies **TR** and **FO**.

Proof. It is obvious from Lemma 4.4.2 that `FIA` satisfies **TR**, thanks to Algorithm 4.5. Now, since **TR** implies **FO** (forgetting variables $\{x_1, \ldots, x_k\}$ is equivalent to restricting to the term $\big[x_1 \in \mathbb{R}\big] \wedge \cdots \wedge \big[x_k \in \mathbb{R}\big]$), `FIA` satisfies **FO**. □

Conjunction with a Term

Lemma 4.4.15. `IA` satisfies $\wedge$**tC**.

Proof. The proof is straightforward from the fact that `IA` satisfies $\wedge$**C**. □

Lemma 4.4.16. `FIA` satisfies $\wedge$**tC**.

Proof. Algorithm 4.6 is very similar to Algorithm 4.5, which computes the term restriction of an FIA. The difference is that instead of replacing nodes by $\veebar$-nodes, it replaces edge labels by the intersection S (line 7). It also adds nodes at the top of the automaton—similar to what is done when conjoining two IAs.

For the same reasons as Algorithm 4.5 [see proof of Lemma 4.4.14, p. 123], Algorithm 4.6 is linear. It also maintains focusingness, thanks to the fact that $A \subseteq B \implies (A \cap C) \subseteq (B \cap C)$. All edge labels are intersected with the same intervals, so if they were focusing before, they still are after the procedure. The

Algorithm 4.6 Conjunction of an FIA φ and a term γ.

```
1: for each node N in φ, from the sink to the root do
2:     let x ≔ Var(N)
3:     if x ∈ Scope(γ) then
4:         for each E ∈ Out(N) do
5:             let S ≔ Lbl(E)
6:             for each literal ⟨x, A⟩ in γ do
7:                 S ≔ S ∩ A
8:             label E with S
9: for each variable x ∈ Scope(γ) do
10:    let S ≔ ℝ
11:    for each literal ⟨x, A⟩ in γ do
12:        S ≔ S ∩ A
13:    add to φ an x-node N with one outgoing edge to Root(φ), labeled S
14:    let N be the new root of φ
```

added top nodes do not compromise focusingness either: by construction, all labels are included in these top edge labels.

Now, we show that when applied to an FIA φ and a term γ, the FIA ψ that is returned verifies $[\![\psi]\!] \equiv [\![\varphi]\!] \wedge [\![\gamma]\!]$. The scope of ψ, which we denote as V_ψ, is equal to $\text{Scope}(\varphi) \cup \text{Scope}(\gamma)$, since ψ is basically γ on top of a modified version of φ. Let us consider a V_ψ-assignment $\vec{v}$.

Suppose that $[\![\psi]\!](\vec{v}) = \top$. Then there exists a path p from the root to the sink of ψ, that is compatible with $\vec{v}$. Path p consists of a top part (from the root of ψ to the node corresponding to the root of φ) and a bottom part (from the node corresponding to the root of φ to the sink). Since the top part of p (which is the same for all paths from the root to the sink of ψ) is compatible with $\vec{v}$, this proves that $\vec{v} \models \gamma$, by construction. Let us consider the path p' in φ that corresponds to the bottom part of p. Edge labels in p' are less restrictive than in p, so if p is compatible with $\vec{v}$, p' also is. This proves that $\vec{v} \models \varphi$. Hence $([\![\varphi]\!] \wedge [\![\gamma]\!])(\vec{v}) = \top$.

Now, suppose that $([\![\varphi]\!] \wedge [\![\gamma]\!])(\vec{v}) = \top$. $\vec{v}$ is compatible with all literals in γ, and there exists a path p in φ that is compatible with $\vec{v}$. Let us consider the path p' in ψ, that consists of the common top part of all paths in ψ, and of the bottom path corresponding to p. We know that $\vec{v}$ is compatible with all literals in γ, so it is compatible with the top part of p'. It is also compatible with the bottom part: considering a variable $x \in \text{Scope}(\gamma)$ and an x-edge E on path p, $\vec{v}_{|x} \in S$, since it is in $\text{Lbl}(E)$ and also in all x-literals of γ. Hence, p' is compatible with $\vec{v}$, and consequently $[\![\psi]\!](\vec{v}) = \top$. All in all, Algorithm 4.6 is polytime and outputs an FIA representing the conjunction of the given FIA and term: `FIA` supports $\wedge$**tC**. □

Final Proof

Proof of Theorem 4.3.14 [p. 110]. All results of this theorem come from proposition and lemmas. Tables 4.3 and 4.4 associate with each claim its proposition. □

Query	IA	FIA
CO	○ [Lem. 4.4.1]	√ [Prop. 4.2.7]
VA	○ [Lem. 4.4.1]	○ [Lem. 4.4.9]
MC	√ [Lem. 4.4.8]	√ [Lem. 4.4.8]
CE	○ [Lem. 4.4.1]	√ [Prop. 4.3.2]
IM	○ [Lem. 4.4.1]	○ [Lem. 4.4.9]
EQ	○ [Lem. 4.4.1]	○ [Lem. 4.4.9]
SE	○ [Lem. 4.4.1]	○ [Lem. 4.4.9]
MX	○ [Lem. 4.4.1]	√ [Prop. 4.2.8]
CX	○ [Lem. 4.4.1]	√ [Prop. 4.3.1]
CT	○ [Lem. 4.4.1]	○ [Lem. 4.4.9]
ME	○ [Lem. 4.4.1]	√ [Lem. 4.4.10]

Table 4.3: Results about queries, and proposition corresponding to each result. √ means "satisfies", and ○ means "does not satisfy, unless $\mathsf{P} = \mathsf{NP}$".

Transfo.	IA	FIA
CD	√ [Prop. 4.2.1]	√ [Prop. 4.2.6]
TR	○ [Lem. 4.4.12]	√ [Lem. 4.4.14]
FO	○ [Lem. 4.4.12]	√ [Lem. 4.4.14]
SFO	√ [Lem. 4.4.7]	√ [Lem. 4.4.7]
EN	○ [Lem. 4.4.11]	○ [Lem. 4.4.11]
SEN	√ [Lem. 4.4.7]	○ [Lem. 4.4.13]
∨**C**	√ [Prop. 4.3.3]	√ [Prop. 4.3.3]
∨**BC**	√ [Prop. 4.3.3]	√ [Prop. 4.3.3]
∨**clC**	√ [Prop. 4.3.3]	√ [Prop. 4.3.3]
∧**C**	√ [Prop. 4.3.4]	○ [Prop. 4.3.5]
∧**BC**	√ [Prop. 4.3.4]	○ [Prop. 4.3.5]
∧**tC**	√ [Lem. 4.4.15]	√ [Lem. 4.4.16]

Table 4.4: Results about transformations, and proposition corresponding to each result. √ means "satisfies", and ○ means "does not satisfy, unless $\mathsf{P} = \mathsf{NP}$".

CHAPTER

5

Building Interval Automata

From a theoretical point of view, focusing interval automata make for efficient handling of Boolean functions over continuous and enumerated variables. It is hence possible to adapt existing "planning using compilation" approaches [§ 2.3], and in particular those we decided to focus on [§ 3.1], to the use of such variables. An embedded system can thus, for example, rely on a decision policy [§ 3.3.2.1] or a transition table [§ 3.3.2.2] to make decisions, without requiring $\log_2(n)$ Boolean variables to represent each continuous variable the domain of which has been arbitrarily discretized into n values. But how to obtain an FIA representing a given decision policy or transition table? Before using them in practice, we have to examine the *compilation step* [§ 1.6.1].

To this end, we examine in this chapter different ways of building focusing interval automata. We first discuss the input language, in which problems are initially represented, that of *continuous constraint networks* [§ 5.1]. Then, we detail bottom-up compilation [§ 5.2] and compilation by solver tracing [§ 5.3].

5.1 Continuous Constraint Networks

As we pointed out in Section 1.6.1, a natural way of representing Boolean functions is the *constraint network*: a model of a Boolean function is a *solution* to its corresponding constraint network. When applied to a constraint network on real variables, the constraint satisfaction problem is generally referred to as a *continuous CSP* [SF96]. We use a similar terminology; however, we do not allow any real variable, but only $\mathbb{R}$-tileable ones.

Definition 5.1.1. A *continuous constraint network (CCN)* is a constraint network $\Pi = \langle V, \mathscr{C} \rangle$ in which $V \subseteq \mathcal{T}$.

Transforming a continuous CN into an FIA is not a trivial task; in particular, it is not tractable, since if it were, the constraint satisfaction problem would be polynomial—and it is not [see e.g. RBW06]. Note that interval automata compilation has an important difference with OBDD compilation. In a constraint network on Boolean variables, each constraint is representable as an OBDD; this allows notably a *bottom-up* compilation. This is not the case for IAs, because IA is not a complete language [Proposition 4.1.6]. Simple CCN constraints such as $x < 1$ or $x = y$ cannot be represented exactly as IAs.

The solution we adopted is to use an *interval-based constraint solver*, that is, a tool designed to provide an approximation of the solution set of a CCN by recursively splitting variable domains in several parts. We used the interval solver RealPaver [GB06]. Its output consists of a list of *boxes*, which makes it directly compatible with interval automata.

RealPaver guarantees that no solution is lost, that is, every solution belongs to at least one of the returned boxes. Under some hypotheses, it is also able to check whether the solution set is *exactly* the union of the returned boxes: this is done by distinguishing two types of boxes, the *outer* ones, which may contain non-solution assignments, and the *inner* ones, which contain solutions only. We decide to ignore this distinction, since interval automata can only represent one type of set.

To sum up, the incompleteness of the interval automata language forces us to use approximations of the Boolean functions we want to compile. We rely on RealPaver to compute this approximation, and only compile interval automata representing the unions of boxes returned by RealPaver.

5.2 Bottom-up Compilation

Bottom-up compilation is the most straightforward compilation method [§ 1.6.1.1]. We examine in this section how it is applicable to interval automata.

5.2.1 Union of Boxes

The interval-based solver RealPaver approximates the Boolean function that corresponds to a continuous constraint network, and returns a Boolean function that is guaranteed to be representable as an IA. Indeed, it returns a list of boxes, and we know that the functions representable as IAs are precisely those having a "finite union of boxes" model set. But this is not only interesting for expressivity reasons: union of boxes are actually $\mathtt{DNF}_{\mathcal{T}}^{\mathbb{IR}}$-representations, that are known to be polynomially translatable into both IA and FIA [Proposition 4.3.9].

The first, simple method we used for the compilation of some continuous constraint network into FIA is to solve this CCN using RealPaver, then to translate the output into FIA by computing the disjunction of all boxes. This method simply uses RealPaver as a tool to approximate the function, and compile this approximation into an FIA in a bottom-up fashion.

Observe that it is possible to compile RealPaver's output *on the fly*, that is, to compile each box whenever it is provided and add it to the current compiled form. The final compiled form is identical to that of the post-solving compilation, and the time complexity is the same, but this variant avoids having to memorize the *entire* output before starting compilation.

5.2.2 Combining Constraints

Since we can use this method on single-constraint networks, we can now examine the possibility of compiling each constraint of some network, then combine the elementary automata into a general one. This technique can be used to compile IAs; however, since `FIA` does not satisfy $\wedge$**BC**, we cannot combine even two constraints without facing exponential losses in space. Considering that only `FIA` satisfies enough queries to be helpful online, we discard this method.

5.3 RealPaver with a Trace

Instead of using RealPaver as a "black box", by only compiling its output, we can also adapt the "DPLL with a trace" approach, that we described in Section 1.6.1.2. The idea is to build an IA corresponding exactly to the trace of a solving: each variable choice yields a node, each domain splitting yields a set of edges. We start off with a closer inspection of the search algorithm of RealPaver.

5.3.1 RealPaver's Search Algorithm

Algorithm 5.1 is a simplified version of RealPaver's generic branch-and-prune algorithm. For the sake of simplicity, we present it as a recursive algorithm, even if the actual implementation is iterative.

Roughly speaking, it works on the current box by first pruning it (removing values that are proven not to be solutions), then splitting the box into several parts, and calling itself for each of the computed sub-boxes. Of course, it does not continue indefinitely—it stops whenever the box is empty, is proven to contain only solutions, or is *small enough*. The latter condition is controlled by a precision parameter.

More precisely, the procedure takes as input a continuous CN II, of which we denote the scope as $\{x_1, \ldots, x_n\}$ (variable domains can only be intervals—but this is not a problem, see Section 5.3.5.3), a box $B = [a_1, b_1] \times \cdots \times [a_n, b_n]$, with each $[a_i, b_i]$ being a closed interval included in $\mathrm{Dom}(x_i)$, and a positive floating-point real number ε to control precision.

The pruning step is done by the internal function `Prune`, that we do not detail; it uses techniques from interval arithmetic, interval analysis, and constraint satisfaction [see GB06 for details]. It is also able to indicate whether the pruned box B^P is entirely included in the solution set; this information is used on line 4.

Algorithm 5.1 `RealPaver`(Π, B, ε): returns a union U of boxes not larger than ε, such that U is included in box B, and the solution set of the CCN Π is included in U.

```
1:  B^P := Prune(Π, B)
2:  if B^P = ∅ then
3:      return ∅
4:  if B^P is not included in the solution set of Π then
5:      if B^P is not more precise than ε then
6:          y := Sel_var(Π, B^P)
7:          S := Split(Π, B^P, y)
8:          U := ∅
9:          for each B^s ∈ S do
10:             U := U ∪ RealPaver(Π, B^s, ε)
11:         return U
12: return B^P
```

On line 5, the precision of the pruned box B^P is tested against the desired precision. This simply consists in computing the length of each $[a_i, b_i]$ in the pruned box: B^P is more precise than ε if and only if *all* these lengths are less than ε.

The splitting step is controlled by the internal functions `Sel_var` and `Split`. `Sel_var` just chooses the next variable to split on—this depends on some heuristics, but generally the variable with the largest "domain" in B^P is chosen. Then `Split` divides the box into a number of smaller boxes that cover the initial box (this is not a partition: since boxes are cartesian products of *closed* intervals, boundaries are overlapping). The number of sub-boxes can be 2 or 3; it depends on a parameter of RealPaver, but cannot be changed during the search.

5.3.2 Tracing RealPaver

We adapt to RealPaver the principle described by Huang and Darwiche [HD05a], designed to build Boolean compilation structures using the trace of a solver's search tree. Algorithm 5.2 is the "twin" of Algorithm 5.1, but instead of returning a union of boxes, it returns an interval automaton. The instructions needed to achieve this are indicated using frames, so that the original "RealPaver" parts and the modified "compilation" parts be easily distinguishable. This procedure is a simplification of our actual compiler, which is presented in Algorithm 5.3.

The compilation procedure is directly based on the search tree: every node of the search tree (that is, every recursive call to `RealPaver`) corresponds to a node in the resulting IA. Lines 2 and 14 correspond to *leaves* of the search tree: In the first case, we know there is no solution in the current box—we return the empty automaton. In the second case, the current box is not split anymore, either because it contains only solutions, or because it is precise enough. We then return the sink-only automaton.

Algorithm 5.2 `RP_to_IA`(Π, B, ε): returns an interval automaton that represents a superset of the set calculated by `RealPaver`(Π, B, ε).

```
1:  B^P := Prune(Π, B)
2:  if B^P = ∅ then
3:      return the empty automaton
4:  if B^P is not included in the solution set of Π then
5:      if B^P is not more precise than ε then
6:          y := Sel_var(Π, B^P)
7:          S := Split(Π, B^P, y)
8:          Ψ := ∅
9:          for each B^s ∈ S do
10:             let ψ_s := RP_to_IA(Π, B^s, ε)
11:             Ψ := Ψ ∪ { ⟨B^s_|y, ψ_s⟩ }
12:         let node N := Get_node(y, Ψ)
13:         return the IA rooted at N
14: let ψ be the sink-only interval automaton
15: return ψ
```

The remainder of the procedure treats "internal" search nodes: the current box is split, and the search continues for each of the computed sub-boxes. Splitting the current box along a variable y boils down to creating an y-node, with n outgoing edges corresponding to the n parts of the split; $B_{|y}$ is a closed interval obtained by *projecting* box B on variable y.

Creation of internal `IA` nodes is delegated to a `Get_node` function, that takes as input a variable x and a set of n couples "interval–IA". It builds a node labeled x with n outgoing edges, each one corresponding to one of the couples. That is to say, to each couple $\langle I, \varphi \rangle$ corresponds an edge labeled I and pointing to the root of φ. The advantage of using a `Get_node` function to do this work, is that it can be implemented to make reduction operations on the fly. Thus, if one of the couples $\langle I, \varphi \rangle$ is such that $I = \varnothing$ or φ is the empty automaton, the corresponding edge is not added. If there is no edge at all, no node is returned. It is also possible to test whether the new node is undecisive, but not whether it is stammering (since this property depends on parent nodes). However, and most importantly, it can check whether it is isomorphic to another node that already exists, by using the well-known technique of *unique node table*: each existing node is stored in a hash table, at a key depending on its edges and children. If the node built by `Get_node` has the same key as some node in the unique node table, `Get_node` discards the newly created node and returns the stored node instead.

Note that `IA` nodes are not created when search nodes are opened. Indeed, `Get_node` can only build "root" nodes—it does not create edges that point to nothing. It only creates an `IA` node when the search subtree rooted at the current search

node has been entirely explored. Hence, if the current box contains no solution, no `IA` node is created; the procedure simply returns the empty IA, without having to remove any node.

Consequently, the current IA is never bigger than the final one. The unique node table only needs one entry per node in the final IA; all in all, the amount of space needed by the algorithm is polynomial in the size of the output IA. However, note that this resulting IA is *not* reduced, since stammering nodes cannot be treated until the IA is complete.

The algorithm we presented is so far a direct adaptation of the "DPLL with a trace" procedure of Huang and Darwiche [HD05a]. In particular, we ignored RealPaver's pruning step, since there is no such step in the Boolean case. Indeed, whenever a value is removed from a Boolean variable's domain, this variable is never selected again, since there is no choice left. On the contrary, in RealPaver, a domain can be reduced several times during the search. For that reason, the interval automata returned by `RP_to_IA` do *not* represent the exact union of boxes returned by `RealPaver`, but only a superset of this union. For example, for the continous CN Π with a unique variable x of domain $\mathbb{R}$ and a unique constraint $x = 0$, the `RealPaver` procedure only prunes the box from $\mathbb{R}$ to $\{0\}$, finds out that this pruned box is included in the solution set of Π, and returns it. On the same input, `RP_to_IA` returns the sink-only automaton—of which any $\{x\}$-assignment is a model. To return an IA equivalent to RealPaver's output, our compiler must be adapted to take the pruning step of RealPaver into account.

5.3.3 Taking Pruning into Account

It is not possible to keep a perfect matching between RealPaver's search tree and our resulting interval automaton. Indeed, the pruning step removes values from the current box, and this must appear in the IA. A first idea is to modify the IA returned for each box (on lines 13 and 15 of Algorithm 5.2 [p. 131]), by adding nodes explicitly restricting the domain of each variable to non-pruned values. It multiplies the number of nodes by the size of the scope (since we add one node per variable and per IA node).

Obviously, adding one node per variable is useless; an explicit restriction of the current domain of a variable x is needed only if the domain of x has changed during the pruning step. A second idea is therefore to add nodes explicitly restricting the domain of a variable *if and only if* this domain has been modified by the pruning step. This is better, but there are still lots of useless nodes: consider a path in the search tree in which the domain of a variable y has been modified k times by the `Prune` function. The corresponding path in the resulting IA thus contains k "pruning y-nodes", each being included in the previous one. Since each of these nodes have only one outgoing edge, removing all of these nodes except for the last one would leave the model set of the path unchanged.

Therefore, it seems much better to add "pruning nodes" at the end of the search tree only. In order to do that, we need to remember the variables the domain of

which has been modified at least once by the `Prune` function: let us call L the set of such variables. When a search leaf is encountered, instead of returning the sink, the procedure returns an interval automaton representing the term $\bigwedge_{x \in L} \left[x \in B^P_{|x}\right]$, where B^P denotes the current pruned box. It is then guaranteed that any domain pruning is reflected in the resulting interval automaton, without increasing too much the number of nodes to be added.

The last improvement is made by remarking that since the `Split` function returns sub-boxes included in the current pruned box, it already reflects the fact that the domain of the variable being split along has been modified. Denoting this variable y, this means that a pruning y-node is added only if y's domain has been pruned *after* the last split along y.

Algorithm 5.3 `IA_builder`(Π, B, ε, L): returns an interval automaton that represents the set calculated by `RealPaver`(Π, B, ε). L is the set of variables that must be "restricted" at the end.

1: $B^P \coloneqq$ `Prune`(Π, B)
2: add to L all variables y such that $B^P_{|y} \neq B_{|y}$
3: **if** $B^P = \varnothing$ **then**
4: **return** the empty automaton
5: **if** B^P is not included in the solution set of Π **then**
6: **if** B^P is not more precise than ε **then**
7: $y \coloneqq$ `Sel_var`(Π, B^P)
8: $L \coloneqq L \setminus \{y\}$
9: $S \coloneqq$ `Split`(Π, B^P, y)
10: $\Psi \coloneqq \varnothing$
11: **for each** $B^s \in S$ **do**
12: let $\psi_s \coloneqq$ `IA_builder`$(\Pi, B^s, \varepsilon, L)$
13: $\Psi \coloneqq \Psi \cup \left\{\langle B^s_{|y}, \psi_s \rangle\right\}$
14: let node $N \coloneqq$ `Get_node`(y, Ψ)
15: **return** the IA rooted at N
16: let ψ be the sink-only interval automaton
17: **for each** $y \in L$ **do**
18: let node $N \coloneqq$ `Get_node` $\left(y, \left\{\langle B^P_{|y}, \psi \rangle\right\}\right)$
19: let ψ be the IA rooted at N
20: **return** ψ

The final procedure, that we call `IA_builder`, is formally described in Algorithm 5.3. Once again, the elements that are new in comparison to Algorithm 5.2 are indicated by frames. On line 2, the list L of variables needing a final pruning node is updated; the variable along which the current box is split is removed from L on line 8. At each search leaf, pruning nodes are added on top of the sink thanks to the loop on line 17.

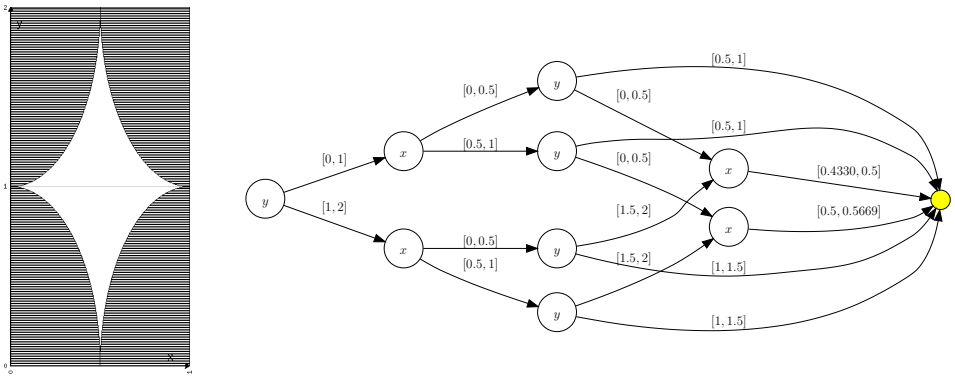

Figure 5.1: This figure illustrates Algorithm 5.3. The figure on the left is a graphical representation of problem Π defined in the text: the solution set is the non-hatched area. The IA on the right is the result of `IA_builder`.

5.3.4 Example

Let us illustrate Algorithm 5.3 on an example. Let Π be the CCN having two variables, x and y, with $\text{Dom}(x) = [0, 1]$ and $\text{Dom}(y) = [0, 2]$, and the following set of constraints:

$$\begin{cases} y \geqslant 2\sqrt{\max(0, 0.25 - x^2)}, \\ y \leqslant 2 - 2\sqrt{\max(0, 0.25 - x^2)}, \\ y \geqslant 2\sqrt{\max(0, 0.25 - (x-1)^2)}, \\ y \leqslant 2 - 2\sqrt{\max(0, 0.25 - (x-1)^2)}. \end{cases}$$

A graphical view of the solution set of Π is proposed in Fig. 5.1. The union of boxes returned by RealPaver for this problem (with $\varepsilon = 1$) is

$$\begin{array}{rcrcr} [0.5, 0.5669] \times [1.5, 2] & \cup & [0.5, 1] \times [1, 1.5] & \cup \\ [0.4330, 0.5] \times [1.5, 2] & \cup & [0, 0.5] \times [1, 1.5] & \cup \\ [0.5, 1] \times [0.5, 1] & \cup & [0.5, 0.5669] \times [0, 0.5] & \cup \\ [0, 0.5] \times [0.5, 1] & \cup & [0.4330, 0.5] \times [0, 0.5]. \end{array}$$

Figure 5.1 shows the corresponding IA obtained with Algorithm 5.3.

5.3.5 Properties of Compiled IAs

Structure

The interval automata obtained thanks to procedure `IA_builder` have the quite interesting property of being focusing. Indeed, all operations of RealPaver that lead to a node in the resulting automaton, namely pruning and splitting, are always restricting the current box. Thus, when calling `Get_node`, it is guaranteed that for each couple $\langle I, \varphi \rangle$, the label of each x-edge in φ is a subset of I. `IA_builder` is hence an `FIA` compiler, which was a requirement for it to be used for our application.

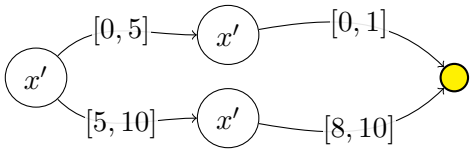

Figure 5.2: The FIA resulting from `IA_builder`, when given the continuous CN on variable $x' \in [0, 10]$ with a single constraint $x' \notin]1, 8[$.

Consistency

Note that if we obtain the empty automaton, it means that the problem is inconsistent: indeed, since the resulting IA represents exactly RealPaver's output, an empty automaton means that RealPaver proved that the initial box contains no solution.

However, obtaining a consistent IA does not imply that the problem is consistent: RealPaver is guaranteed to be complete, but not to be sound—it can return boxes even if the problem is inconsistent. The only thing that can be deduced from the fact that the compiled IA is consistent is simply that RealPaver has not proven the problem to be inconsistent. Let us recall that due to the incompleteness of `IA`, our compiler can only build an *approximation* of the actual Boolean function.

Discrete Variables

Variables in the `IA` framework have $\mathbb{R}$-tileable domains, meaning that they can be discrete. An IA can thus represent Boolean functions depending on both continuous and discrete variables. However, RealPaver only takes as input real variables of interval domain.[1] To represent a variable x of domain $[0, 1] \cup [8, 10]$ for example, it is necessary to declare a variable x' of domain $[0, 10]$, and add a constraint stating that $x \notin]1, 8[$.

Let us illustrate what happens for a trivial CCN using such a variable, viz., the CCN of scope $\{x\}$ with no constraint at all. RealPaver is forced to split the domain (provided that the given precision is lower than 10): it makes two boxes, say $[0, 5]$ and $[5, 10]$, then prunes each one to get $[0, 1]$ and $[8, 10]$. The resulting FIA is shown in Fig. 5.2: it contains 4 edges (2 if we remove stammering nodes), whereas it is possible to represent this function using the sink-only automaton. It is possible to improve `IA_builder` by considering the *actual* domains of the variables, and not rely on what was given as input to RealPaver. Indeed, replacing x' by x in the FIA leads to it being reduced to the sink-only automaton (by removal of stammering nodes, then contiguous edges, then undecisive node).

*
**

We implemented the two compilation techniques we described in this chapter (union of boxes and "RealPaver with a trace"). The next chapter provides experimental results about the compilation of FIAs using these methods, as well as about their operational efficiency.

[1] It actually also accepts integer variables of integer interval domain. However, it is only "syntactic sugar"—they are processed in a way that makes them behave exactly as we describe in this section.

CHAPTER

6

Experiments on Interval Automata

We have seen in Chapter 4 that interval automata, and in particular focusing interval automata, are theoretically suitable for compilation-based planning applications. In Chapter 5, we presented methods to compile continuous constraint networks into focusing interval automata. This chapter finally outlines our experimental work. First, we present our implementation of interval automata, of `IA` operations, and of a compiler [§ 6.1]. Then, experimental compilation results are provided [§ 6.2]. Finally, we give results about the experimental use of compiled forms [§ 6.3].

6.1 Implementation

6.1.1 Experimental Framework

We implemented a library for handling interval automata, written in Java. In the same way as classical packages for BDDs, such as CUDD [Som05], our program allows interval automata to be built in a bottom-up fashion. That is to say, nodes are incrementally added, from the sink to the root, using a `Get_node` function, as described in Section 5.3.2. Almost all reduction operations are thus made on the fly; in particular, the merging of isomorphic nodes relies on a unique node table.

6.1.2 Interval Automaton Compiler

Our library allows interval automata to be built "by hand", using the `Get_node` function; but since this approach is quite limited, we implemented two approaches to the compilation of continuous constraint networks into interval automata. As we

explained in the previous chapter, these two approaches make use of the RealPaver constraint solver.

The first approach is the compilation of RealPaver's output: we wrote a program that parses this output and builds an equivalent FIA. In practice, the two steps (solving and compilation) are separated, but nothing prevents them from being interlaced.

The second approach is "RealPaver with a trace": we implemented a prototype of this compiler, modifying RealPaver so that it outputs its search trace in a file, and using this file to guide compilation. Even if it requires the whole search trace to be memorized before compiling, our prototype gives insight about the feasibility of this approach, and the properties of compiled IAs.

6.1.3 Operations on Interval Automata

We implemented a number of operations on IAs and FIAs, restricting ourselves to polytime operations. Available queries include consistency (**CO**) on `FIA`, model checking (**MC**), model extraction (**MX**) on `FIA`, and context extraction (**CX**) on `FIA`. Available transformations include conditioning (**CD**), forgetting (**FO**) on `FIA`, disjunction ($\vee$**C**), conjunction ($\wedge$**C**) on `IA`, and conjunction with a term ($\wedge$**tC**).

However, as we will see in Section 6.3, these standard operations are too general to be efficient in practice. In particular, transformations are often space- and time-consuming. For that reason, we also implemented special operations, specifically designed for our planning applications. The main ones are **CDCO**, **CDMX**, and **CDFOMX**. Let us outline when they are useful and how they work.

For the *Satellite* benchmark, we need to condition the compiled form with respect to the current situation, then check whether the resulting FIA is consistent. We remarked that this task can be seen as a single query, instead of the combination of **CD** and **CO**. This "special" query, that we named **CDCO**, aims at answering the question "is this function consistent after having been conditioned by this assignment?". The advantage of using this special query is that it can be done with a simple traversal of the graph. The idea is to maintain a set of nodes that are "reachable from the root". Initially, this set contains only the root; we add all of its children that are accessible modulo the conditioning assignment; then we add all the children of these nodes, etc. Whenever the sink is added to this set, we know that there exists a model of the function that is consistent with the conditioning assignment—and hence the answer to the query is yes.

The second special query is **CDMX**: its goal is to extract a model from a conditioned FIA, without explicitly computing this conditioning. It is notably useful to handle decision policies (conditioning on the current state, and extraction of a decision) as well as transition relations (for example, conditioning on the current state and chosen decision, and extraction of a possible next state). It works in a similar way as **CDCO**: roughly speaking, we maintain a set of nodes that are reachable from the root, together with the incoming edge by means of which they were reached. Whenever the sink is reached, we can retrieve a consistent path in reverse order.

problem	time (ms)	#nodes	#edges	charac. size	filesize (octets)
*Drone*4-5-3	14 082	46 871	51 701	51 727	2 161 248
*Drone*4-10-3	20 317	97 722	104 576	104 602	4 237 494
*Drone*4-15-3	164 028	221 124	238 042	238 068	10 495 673
*Drone*4-20-3	146 504	223 984	245 078	245 104	10 868 910
*Drone*4-25-3	172 525	288 776	309 870	309 896	12 940 209
Satellite	25 391	141 096	145 016	145 059	8 476 904

Table 6.1: Results obtained when compiling RealPaver's output into `FIA`.

The last special query is a modified version of **CDMX**. It aims at extracting a model from an FIA in which some variables are conditioned and some other variables are forgotten. We call it **CDFOMX**: it can be achieved by applying **CD**, then **FO**, then **MX** —or by applying the specific traversal we described in the previous paragraph, with a few modifications to handle variables that have to be forgotten.

6.2 Compilation Tests

We compiled, on a standard laptop,[1] various instances of our benchmarks involving real-valued variables, namely *Drone* and *Satellite* [see Chapter 3], using the two approaches we described in the previous chapter. For each problem instance, we give the time necessary to compile the FIA (excluding the running time of RealPaver), the number of nodes and edges of the FIA, its characteristic size, and the size in octets of a file in which the FIA has been written. This filesize is meant to give a rough idea of the relative memory space taken by the instances; it is not an accurate measurement of the RAM size they take while handled.

Table 6.1 presents results obtained when compiling the output of RealPaver, while Table 6.2 presents results obtained using "RealPaver with a trace". There are five instances of the *Drone* benchmark, each one with four zones and three available marbles, the varying parameter being the allotted time.

Observe that compiled FIAs are relatively large, up to approximately 300 000 nodes and edges. Using graphs of such size in an autonomous system require a fair amount of memory; this is however not prohibitive for applications in which memory size is not crucial. On an opposite note, compilation is actually fast. Note that for *Drone* instances, it is faster to compile using "RealPaver with a trace", but this yields larger results. The longer time taken by the bottom-up approach comes from the fact that the hash key we use for the unique node table is not well adapted to nodes with lots of children, which is the case of the root node (the compiled form is a disjunction of hundreds of smaller FIAs).

[1] Mobile Turion 64 X2 TL-56, 1.80 GHz, 2 Go RAM.

problem	time (ms)	#nodes	#edges	charac. size	filesize (octets)
*Drone*4-5-3	17 924	56 766	61 596	61 611	2 784 515
*Drone*4-10-3	21 723	74 436	81 290	81 299	3 718 465
*Drone*4-15-3	58 013	252 995	269 913	269 961	11 985 073
*Drone*4-20-3	92 206	329 724	350 818	350 836	15 595 018
*Drone*4-25-3	64 754	333 390	354 772	354 798	15 760 686
Satellite	24 899	109 078	112 022	112 045	6 546 529

Table 6.2: Results obtained with "RealPaver with a trace".

problem	**CD** $\vec{s}$	**FO** $\mathcal{S}'$	**MX** $\hookrightarrow\vec{a}$	**CD** $\vec{a}$	**MX** $\hookrightarrow\vec{s}'$	**CDFOMX** $\langle\vec{s},\mathcal{S}'\rangle\hookrightarrow\vec{a}$	**CDMX** $\vec{a}\hookrightarrow\vec{s}'$
*Drone*4-5-3	1074	6	0	2	0	0	1
*Drone*4-10-3	3145	18	0	3	0	0	0
*Drone*4-15-3	12 414	9	0	3	2	1	1
*Drone*4-20-3	14 936	4	1	5	0	1	1
*Drone*4-25-3	16 216	5	0	6	0	3	3

Table 6.3: Results for Scenario 1 on several instances of the *Drone* transition relation, obtained using "RealPaver with a trace". All times are in milliseconds.

6.3 Application Tests

After having examined the compiled forms regarding the memory space they take, we study their efficiency in terms of operation duration. In this section, we give results regarding *simulations* of the use of compiled forms. That is, we consider a number of possible ways for the FIA to be handled online, depending on the problem it represents; we call *scenario* a possible use of the compiled form. Each scenario is divided into steps, each one corresponding to a query or a transformation. We provide results about the execution of these scenarios on the FIAs we compiled using "RealPaver with a trace".

6.3.1 Simulating Online Use of the *Drone* Transition Relation

Instances of the *Drone* benchmark are compiled *transition relations*, that is, relations linking a current state and an action to the corresponding next state: denoting $\mathcal{S}$ the set of state variables, $\mathcal{A}$ the set of action variables, and $\mathcal{S}'$ the set of next state variables, the compiled form is a Boolean function over $\mathcal{S} \cup \mathcal{A} \cup \mathcal{S}'$. We identify four scenarios that can be used on such structures. They are similar, in the sense that they are all based on **CD**, **FO**, and **MX**, but variables and operation order differ, which notably has an impact on the size of the compiled form the operations are applied on.

In Scenario 1, we observe a current state. We want to choose a possible decision, and then get a possible next state. This corresponds to the following operations:

- **CD** on the whole FIA, w.r.t. some $\mathcal{S}$-assignment $\vec{s}$;
- **FO** on the conditioned FIA, w.r.t. variables from $\mathcal{S}'$;
- **MX** on the resulting FIA, to get an action $\vec{a}$;
- **CD** on the conditioned FIA, w.r.t. $\vec{a}$;
- **MX** on the resulting FIA, to get a next state $\vec{s}'$.

Using the special queries we described in Section 6.1.3, it boils down to

- **CDFOMX** on the whole FIA, to get an action $\vec{a}$;
- **CDMX** on the whole FIA, to get a next state $\vec{s}'$.

We ran 20 simulations of Scenario 1, picking the initial current state at random among the possible states. Table 6.3 provides the average duration of each operation, in milliseconds. Without the special queries, this scenario suffers from the bottleneck of conditioning. This is because a lot of nodes are modified, and a modification of a node recursively impacts all its ancestors, because of the unique node table system. Once the first conditioning is over, the automaton is much smaller; the forgetting operation and the second conditioning are thus executed in a more reasonable time. Unsurprisingly, the model extraction query, which is roughly done through a direct descent in the graph, is done really fast—in less than one millisecond for all instances.

Using only standard queries and transformations, the whole scenario needs 17 s on average to be executed on the largest instance. It can be considered acceptable for some applications, depending for example on the time necessary to execute each action. But if the MAV must make decisions very quickly, for example if the zones are very close to each other, handling this transition table online to compute the next decision to make is not possible. However, using the specific queries **CDMX** and **CDFOMX**, the whole scenario can be done in 6 ms on average for the largest instance, which is much more reasonable.

In Scenario 2, we also observe a current state, but we want to find out a possible next state. This corresponds to the following operations:

- **CD** on the whole FIA, w.r.t. some $\mathcal{S}$-assignment $\vec{s}$;
- **FO** on the conditioned FIA, w.r.t. variables from $\mathcal{A}$;
- **MX** on the resulting FIA, to get a next state $\vec{s}'$.

It boils down to a single execution of **CDFOMX**. Results are presented in Table 6.4; the first column contains the same results as in Table 6.3, and the associated operation (conditioning of the whole transition relation) is once again the bottleneck

problem	**CD** $\vec{s}$	**FO** $\mathcal{A}$	**MX** $\hookrightarrow \vec{s}'$	**CDFOMX** $\langle \vec{s},\mathcal{A}\rangle \hookrightarrow \vec{s}'$
*Drone*4-5-3	1074	14	0	0
*Drone*4-10-3	3145	40	2	0
*Drone*4-15-3	12 414	18	2	1
*Drone*4-20-3	14 936	7	0	1
*Drone*4-25-3	16 216	8	1	5

Table 6.4: Results for Scenario 2 on several instances of the *Drone* transition relation, obtained using "RealPaver with a trace". All times are in milliseconds.

problem	**CD** $\vec{s}'$	**FO** $\mathcal{A}$	**MX** $\hookrightarrow \vec{s}$	**CDFOMX** $\langle \vec{s}',\mathcal{A}\rangle \hookrightarrow \vec{s}$	**CD** $\vec{s}'$	**MX** $\hookrightarrow \vec{s}.\vec{a}$	**CDMX** $\vec{s}' \hookrightarrow \vec{s}.\vec{a}$
*Drone*4-5-3	1001	44	8	6	1001	9	8
*Drone*4-10-3	2778	29	4	15	2778	5	16
*Drone*4-15-3	11 590	1205	203	11	11 590	250	14
*Drone*4-20-3	18 700	1801	296	13	18 700	364	12
*Drone*4-25-3	19 865	1982	373	18	19 865	378	19

Table 6.5: Results for Scenarios 3 and 4 on several instances of the *Drone* transition relation, obtained using "RealPaver with a trace". All times are in milliseconds.

of the scenario. Forgetting action variables instead of next state variables does not seem to make a difference: this bears witness to the fact that RealPaver does not follow a specific variable ordering.

Scenarios 3 and 4 take a reverse approach, actually going backwards in the state-transition system. The purpose is to find a state (or a state-action pair, respectively) that can lead to the current state. Finding such a state can be done using the following operations:

- **CD** on the whole FIA, w.r.t. some $\mathcal{S}'$-assignment $\vec{s}'$;
- **FO** on the conditioned FIA, w.r.t. variables from $\mathcal{A}$;
- **MX** on the resulting FIA, to get a state $\vec{s}$.

This corresponds to a single application of our specific query **CDFOMX**. To find a state-action pair, we do not need to forget action variables; the following operations are sufficient:

- **CD** on the whole FIA, w.r.t. some $\mathcal{S}'$-assignment $\vec{s}'$;
- **MX** on the resulting FIA, to get a state-action pair $\vec{s}\ .\ \vec{a}$.

This corresponds to a single execution of **CDMX**. Results for Scenarios 3 and 4 can be found in Table 6.5. They confirm our observations so far, aside from the fact

that operations following the first conditioning take more time than for the forward scenarios.

All in all, these results show that it seems possible for transition relations compiled as FIAs to be handled online, as long as one does not rely too much on standard, generic operations.

6.3.2 Simulating Online Use of the *Satellite* Subproblem

The *Satellite* benchmark represents a subproblem of a larger decision-making problem. To use it, we just have to condition the compiled form with respect to the current state (that is, the current attitude of the satellite), and check whether the result is consistent. If it is the case, it means that executing the sun-pointing maneuver is possible in the allotted time. We applied only this scenario to the *Satellite* benchmark; the operations are **CD** and **CO**, which boil down to a simple specific **CDCO** operation. On 20 randomly chosen states, the conditioning step took 13 173 ms and the consistency step less than 1 ms on average. This is too long for this application, in which time is a crucial parameter. However, the specific **CDCO** query also took less than 1 ms on average. Consequently, supposing that the embedded memory of the satellite is able to contain a compiled form of this size, it is likely that handling this compiled form online would be possible.

Part III

Set-labeled Diagrams

Back to Enumerated Variables

Remarks about Meshes and Discretization

Like GRDAG, the language of interval automata is not complete [Proposition 4.1.6]; that is to say, there exists Boolean functions on $\mathbb{R}$-tileable variables that cannot be represented as IAs. Indeed, each path in an IA represents a term $\left[x_1 \in I_1\right] \wedge \cdots \wedge \left[x_n \in I_n\right]$, the model set of which is simply a box $I_1 \times \cdots \times I_n$. Interval automata can thus only represent "unions of boxes", as presented in Figure 6.1.

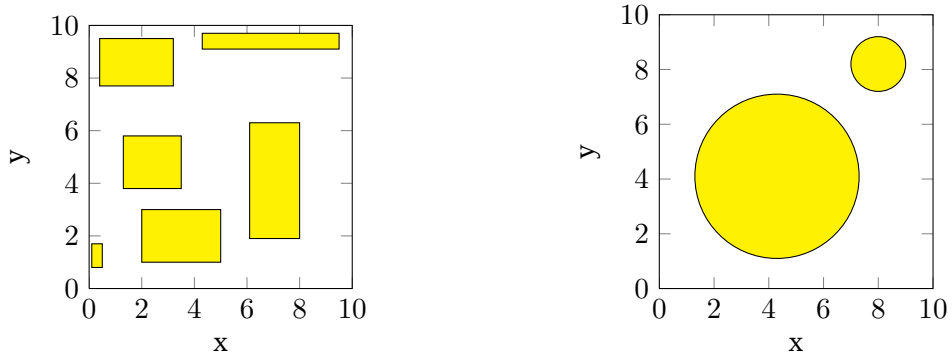

Figure 6.1: A model set representable as an IA (left) and another one, not representable as an IA (right).

The simple function $\left[x = y\right]$, for example, of which several representations are showed in Fig. 6.2, cannot be represented exactly using an IA: since IAs are finite, they cannot reach the precision needed. Figure 6.2 illustrates this: when two values in the domain of a variable are "too close" to each other, they cannot be distinguished by the IA.

It is possible to "quantify" the precision of a given IA. By looking at the grids pictured on each model set in Figure 6.2, it is obvious that the rightmost IA is much more precise than the leftmost one. This grid actually corresponds to what we earlier called *meshes* [Definition 4.3.7], that is, partitions of each variable's domain such that two values from a same bit are interchangeable. For a given interval automaton, this grid (and more generally, any mesh) constitutes a *discretization* of the domains.

Thus, replacing each continuous variable by a discrete one—associating with each part of the mesh a certain "meta-value"—and modifying edge labels accordingly, we obtain a discrete IA, that represents a *discretization* of the original interpretation. The interest of this discretization lies in the fact that it was not arbitrarily decided before the construction of the automaton, but is rather created *a posteriori*, based on the interval automaton. In particular, since the size of mesh elements can vary, the discretization is likely to be more adapted to the model set than an arbitrary, regular discretization.

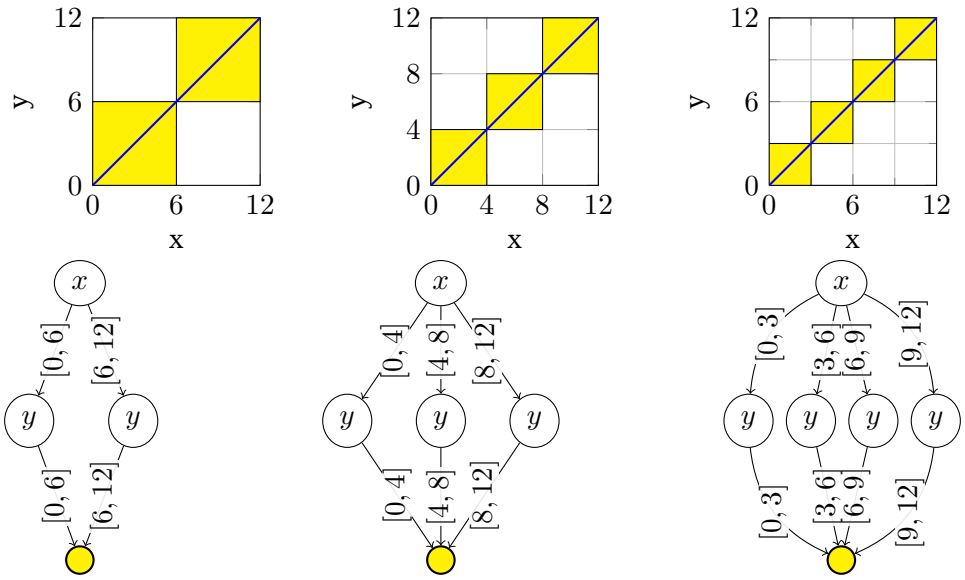

Figure 6.2: Three IAs representing Boolean function $[x = y]$ with increasing precision.

This is illustrated in Figure 6.3: the arbitrary discretization in the left figure leads to larger enumerated domains, while not being exact. On the contrary, the discretization in the right figure leads to enumerated domains of minimal size, without any approximation. This comes from the fact that it has been computed *after* the construction of the IA.

Discrete Interval Automata

In theory, from a "knowledge compilation map" point of view, using the original interval automaton or its discretized version is not different: their characteristic sizes are similar. In practice, however, this is not the case; real numbers take more memory space than integers. Since we aim at embarking compiled forms, being thereof constrained in memory space, discrete IAs seem better adapted. Of course, they would be embarked along with a "translation table", associating discrete meta-values with continuous mesh elements, to encode and decode the inputs and outputs of operations.

In this third part of the thesis, we study a Boolean representation language that is the discrete counterpart of interval automata, in the sense that the mesh-based discretization of any IA belongs to this new language. Since there is no constraint on the meta-values of the discretized variables, we choose to use the simplest alternative—variables with an integer interval domain. We also extend the expressivity of edge labels: they can now be any $\mathbb{Z}$-tileable set. This does not change the characteristic size of representations, but we show that it increases the generality of the focusingness property.

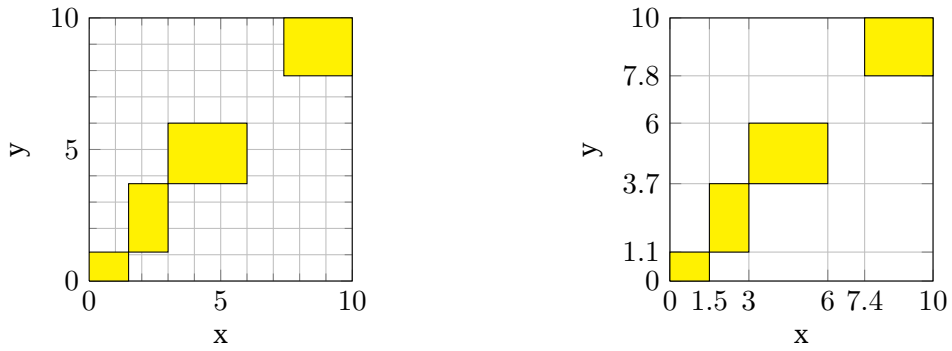

Figure 6.3: Some model set with two different discretizations, represented as grids. On the left, an arbitrary discretization, approximate and spatially costly. On the right, a discretization obtained *a posteriori* on an IA compiled using "RealPaver with a trace".

To emphasize the similarity of these new structures with the BDD family, we call them *set-labeled diagrams*. The present part deals with the study of this language and its sublanguages, and follows the same outline as Part II: we first formally define the languages and provide their knowledge compilation map [Chapter 7], then examine compilation techniques [Chapter 8] and present results about the experimental use of set-labeled diagrams for controlling autonomous systems [Chapter 9].

CHAPTER

7

Set-labeled Diagrams Framework

In this chapter, we define the language of *set-labeled diagrams*, which are decision diagrams on enumerated variables. We show how this language relates to interval automata and to the decision diagram family. We identify sublanguages having interesting properties—in particular, satisfying queries and transformations commonly used in a planning context.

The outline of the chapter is the following: we start with definitions about set-label diagrams [§ 7.1], then introduce various sublanguages and discuss their relations to state-of-the-art languages [§ 7.2]. The section that follows is devoted to the relationship between the interval automaton and set-labeled diagram families [§ 7.3]. Last, we draw up the knowledge compilation map of set-labeled diagrams [§ 7.4].

7.1 Language

7.1.1 Definition

Let us start with the formal definition of the variables we use in this part of the thesis, namely *integer variables*.

Definition 7.1.1 (Integer variables)**.** A variable $x \in \mathcal{V}$ is an *integer variable* if and only if its domain is an integer interval,viz., $\mathrm{Dom}(x) \in \mathbb{IZ}$.

We only consider integer variables with contiguous values, for the sake of simplicity; but any enumerated variable can be represented as an integer variable of this kind. Note that the domain of an integer variable needs not be finite—$\mathbb{Z}$, as a whole, is a closed interval of $\mathbb{Z}$. Elements of $\mathbb{IZ}$ are sets of the form $\{4, \ldots, 9\}$. To emphasize the fact that they are intervals, we use the following notation: $[a, \ldots, b]$ for $\{a, \ldots, b\}$, and $[a, \ldots, \infty]$ for $\mathbb{N} \setminus \{0, \ldots, a-1\}$. The dots between bounds

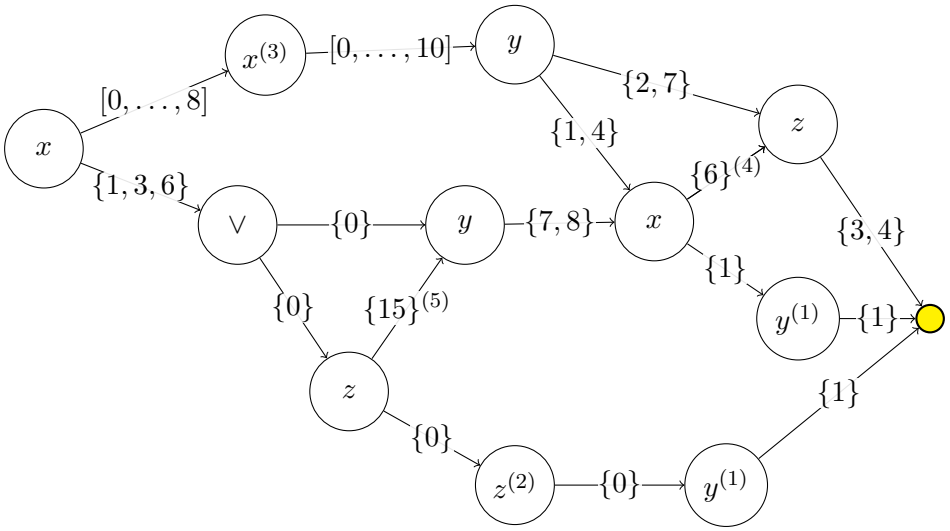

Figure 7.1: An example of a non-reduced SD. Variable domains are all $[0,\ldots,10]$. The two nodes marked $^{(1)}$ are isomorphic; node $^{(2)}$ is stammering; node $^{(3)}$ is undecisive; edges marked $^{(4)}$ are contiguous; edge $^{(5)}$ is dead.

distinguishes between real-valued and integer-valued intervals. We use $\mathcal{I}$ to denote the set of integer variables.

Let us now define the language of set-labeled diagrams, SD. We give the formal definition first, and detail each part afterwards.

Definition 7.1.2. The SD language is the restriction of $\mathtt{NNF}_{\mathcal{I}}^{\mathbb{T}\mathbb{Z}}$ to representations satisfying $\wedge$-simple decision [Def. 1.3.23].

Set $\mathbb{T}\mathbb{Z}$ contains all $\mathbb{Z}$-tileable sets [Definition 4.1.2], and $\wedge$-simple decision [Definition 1.3.23] means that we allow pure disjunctive nodes but not pure conjunctive nodes, just like interval automata. In other words, SD-representations are GRDAGs on integer variables, with literals of the form "x belongs to a union of intervals", and with a "decision diagram"-like structure. This definition is quite close to that of IA: variables and literal expressivity change, but the general structure is the same. For this reason, we will handle set-labeled diagrams (SDs) in the same "automaton form" that we used for IAs [§ 4.1.2], the only differences being variables and edge labels. Figure 7.1 gives an example of SD.

We use the notation introduced in the context of IAs [§ 4.1.2]: $\mathrm{Var}(N)$ denotes the label of a node N, etc. The only difference is that here, $\mathrm{Lbl}(E) \in \mathbb{T}\mathbb{Z}$, whereas for interval automata $\mathrm{Lbl}(E) \in \mathbb{I}\mathbb{R}$. The interest of using tileable sets rather than intervals on labels is that it groups more values: this impacts neither expressivity nor characteristic size, but is useful for focusingness. Indeed, intuitively, the more values an x-edge label contains, the more likely it is that it includes all subsequent x-edge labels. We give an example of this later (Fig. 7.4).

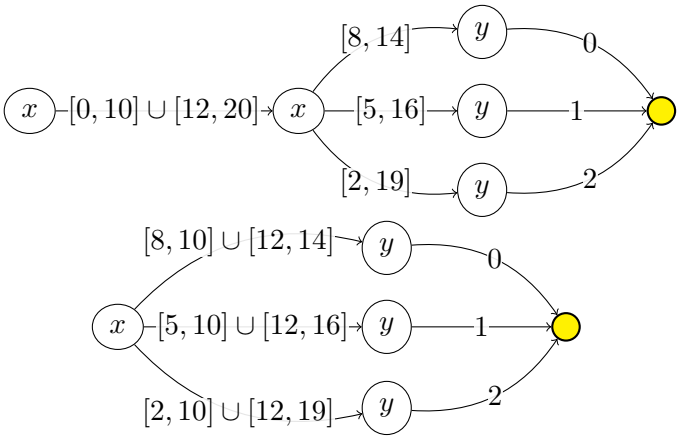

Figure 7.2: Illustration of the new definition of stammeringness. Fusing the two x-nodes in the top SD yields the bottom SD, that has a greater characteristic size. As a consequence, the middle x-node in the top SD cannot be considered as stammering.

7.1.2 Reduction

It is possible to reduce SDs using the same mechanisms as for IAs. Isomorphic nodes, dead edges, and undecisive nodes can be defined in the exact same way.

Definition 7.1.3. Let φ be an SD.

- Two nodes N_1 and N_2 of φ are *isomorphic* if and only if $\text{Var}(N_1) = \text{Var}(N_2)$ and there exists a bijection σ from $\text{Out}(N_1)$ to $\text{Out}(N_2)$, such that $\forall E \in \text{Out}(N_1), \text{Lbl}(E) = \text{Lbl}(\sigma(E))$ and $\text{Dest}(E) = \text{Dest}(\sigma(E))$.
- An edge E of φ is *dead* if and only if $\text{Lbl}(E) \cap \text{Dom}(\text{Var}(E)) = \varnothing$.
- A node N of φ is *undecisive* if and only if $|\text{Out}(N)| = 1$ and $E \in \text{Out}(N)$ is such that $\text{Dom}(\text{Var}(E)) \subseteq \text{Lbl}(E)$.

Nevertheless, the structural differences between SDs and IAs (scope and literal expressivity) raise the need for new definitions of contiguity and stammeringness.

Definition 7.1.4 (Contiguous edges)**.** Two edges E_1 and E_2 of an SD φ are *contiguous* if and only if $\text{Src}(E_1) = \text{Src}(E_2)$ and $\text{Dest}(E_1) = \text{Dest}(E_2)$.

The definition is much simpler than on IAs: there is indeed no condition on the labels. On SDs, the union of two labels is *always* a valid label, whereas it is not the case on IAs. On the contrary, the definition of a stammering node is more subtle (but also more general) for SDs than for IAs.

Definition 7.1.5 (Stammering node)**.** A non-root node N of an SD φ is *stammering* if and only if all parent nodes of N are labeled by $\text{Var}(N)$, and either $\sum_{E \in \text{Out}(N)} \|\text{Lbl}(E)\| = 1$ or $\sum_{E \in \text{In}(N)} \|\text{Lbl}(E)\| = 1$.

It is not sufficient that either $|\mathrm{Out}(N)| = 1$ or $|\mathrm{In}(N)| = 1$. If the edge in question is labeled by a union, the reduction procedure does remove it, but can increase the characteristic size of the others, as shown on Fig. 7.2. This is why we have to take the size of the label into account: we decide that it must be exactly one interval.

Definition 7.1.6 (Reduced form)**.** A set-labeled diagram φ is said to be *reduced* if and only if no node of φ is isomorphic to another, stammering, or undecisive, and no edge of φ is dead or contiguous to another.

As it is for interval automata, reduction can be achieved in time polynomial with respect to the size of the SD.

Proposition 7.1.7 (Reduction)**.** There exists a polytime algorithm that transforms any SD φ into an equivalent reduced SD φ' such that $\|\varphi'\| \leqslant \|\varphi\|$.

Detailed proof p. 167.

7.2 Sublanguages of Set-labeled Diagrams

7.2.1 The SD Family

We examine a number of fragments of SD.

Definition 7.2.1 (Fragments of SD)**.** FSD is the fragment of SD satisfying focusingness [Definition 4.2.3].

Let $<$ be a total strict order on $\mathcal{I}$; $\mathtt{OSD}_<$ is the restriction of SD to graphs ordered by $<$ [Definition 1.3.25]. OSD is the union of all $\mathtt{OSD}_<$, for every total strict order $<$.

SDD (resp. FSDD, $\mathtt{OSDD}_<$, OSDD) is the restriction of SD (resp. FSD, $\mathtt{OSD}_<$, OSD) to strong and exclusive decision [Definition 1.3.23].

The "DD" notation refers to *decision diagrams*; we consider that satisfying strong and exclusive decision, that is, containing no pure disjunctive or conjunctive nodes, and only decision nodes with disjoint outgoing edges, is constitutive of decision diagrams.

Note that formally, since $\veebar$ is not an actual variable, it is not meant to be concerned by the "exclusive decision" requirement. Furthermore, since this special variable represents pure GRDAG disjunction, an SD satisfies strong decision [Definition 1.3.23] if and only if it does not contain any $\veebar$-node. However, we can remark that if these $\veebar$-nodes satisfy exclusive decision, they are harmless: at most one of their outgoing edges is not dead. In the GRDAG format, they actually correspond to $\vee$-nodes with a single child—they can be harmlessly removed. All in all, the formal GRDAG requirement of "strong and exclusive decision" on SDs amounts to a simple requirement of "exclusive decision for all nodes (including $\veebar$-nodes)" when referring to SDs in a decision diagram form. In the following, "exclusive decision" always refer to this decision diagram version.

Let us now present the sublanguage hierarchy of the SD family using a graph.

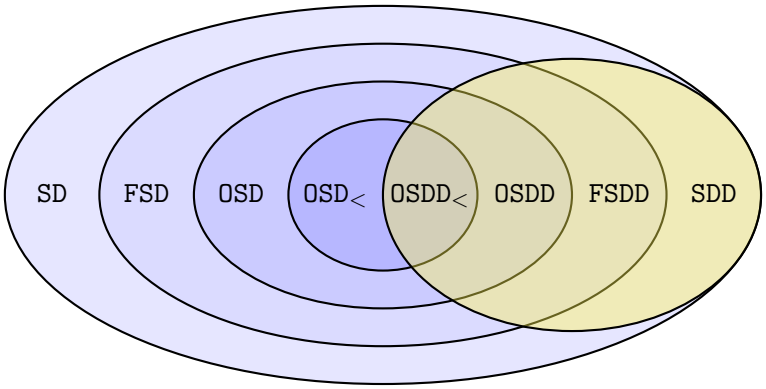

Figure 7.3: Language inclusion hierarchy of the SD family. Languages satisfying exclusive decision are inside the rightmost, yellow circle.

Proposition 7.2.2. The inclusion results of Figure 7.3 hold.

On all these fragments, we show that reduction works the same as for SD.

Proposition 7.2.3 (Reduction of SD fragments)**.** Let L be any fragment of SD from Definition 7.2.1. There exists a polytime algorithm that transforms any L-representation φ into an equivalent reduced L-representation φ' such that $\|\varphi'\| \leqslant \|\varphi\|$.

Detailed proof p. 168.

7.2.2 Relationship with the IA and BDD Families

Our purpose, when defining set-labeled diagrams, was to characterize the form taken by interval automata when they are discretized with respect to meshes. These two languages are hence very close to each other: this is obvious when considering that they are both defined as subsets of NNF satisfying $\wedge$-simple decision. However, because of their respective interpretation domain and literal expressivity, neither of the two languages is included in the other.

Proposition 7.2.4. It holds that SD $\not\subseteq$ IA and IA $\not\subseteq$ SD.

Proof. Let $x \in \mathcal{I}$ with $\mathrm{Dom}(x) = \mathbb{Z}$. Since $\mathbb{Z}$ is not $\mathbb{R}$-tileable (it would require an infinite union of singletons), there is at least one integer variable which is not $\mathbb{R}$-tileable: $\mathcal{I} \not\subseteq \mathcal{T}$. Hence $\mathfrak{D}_{\mathcal{I},\mathbb{B}} \not\subseteq \mathfrak{D}_{\mathcal{T},\mathbb{B}}$, which implies that SD is not a sublanguage of IA. Similarly, considering a variable with domain $\{\pi\}$ (or any other noninteger), we show that $\mathcal{T} \not\subseteq \mathcal{I}$, hence the second result. □

However, when restricting them to common variables (integer variables with a finite domain, which we defined in Section 1.2.1 as $\mathcal{E}$) and literals (the set of integer singletons $\mathbb{SZ}$), their similar structure becomes obvious.

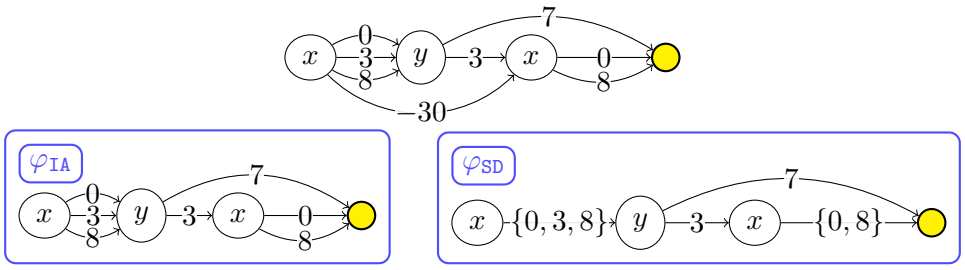

Figure 7.4: Top: An $\mathtt{SD}_{\mathcal{E}}^{\$\mathbb{Z}}$-representation φ, over variables x and y of domain $[0, \dots, 10]$. Bottom left: The structure $\varphi_{\mathtt{IA}}$ obtained when applying to φ the IA reduction procedure. Bottom right: The structure $\varphi_{\mathtt{SD}}$ obtained when applying to φ the SD reduction procedure. We see that $\varphi_{\mathtt{IA}}$ is still in $\mathtt{SD}_{\mathcal{E}}^{\$\mathbb{Z}}$, whereas $\varphi_{\mathtt{SD}}$ is not (some edges are not labeled with singletons). However, $\varphi_{\mathtt{SD}}$ is focusing, whereas $\varphi_{\mathtt{IA}}$ is not.

Proposition 7.2.5. It holds that $\mathtt{SD}_{\mathcal{E}}^{\$\mathbb{Z}} = \mathtt{IA}_{\mathcal{E}}^{\$\mathbb{Z}} = \mathtt{SD} \cap \mathtt{IA}$.

Detailed proof p. 168.

Note that it is also the case for focusing IAs and SDs: $\mathtt{FSD}_{\mathcal{E}}^{\$\mathbb{Z}} = \mathtt{FIA}_{\mathcal{E}}^{\$\mathbb{Z}} = \mathtt{FSD} \cap \mathtt{FIA}$. However, $\mathtt{SD}_{\mathcal{E}}^{\$\mathbb{Z}}$ and $\mathtt{FSD}_{\mathcal{E}}^{\$\mathbb{Z}}$ are not stable by the reduction procedure we defined on SDs. Figure 7.4 shows why, and incidentally illustrates why choosing to label edges with unions makes focusingness more general.

Interestingly enough, restricting SD to enumerated variables makes it a complete language.

Proposition 7.2.6. SD is incomplete, but $\mathtt{SD}_{\mathcal{E}}$ is complete.

Proof. Let $x \in \mathcal{I}$ be a variable of domain $\mathbb{Z}$. The Boolean function $\left[x/2 \in \mathbb{Z}\right]$, returning $\top$ for even numbers and $\bot$ for odd numbers, is not representable as an SD: we need an interval per even number, therefore representing the set of even numbers as a finite union of closed intervals is impossible. Hence, there exists a Boolean function on integer variables that has no representation as an SD: SD is incomplete.

However, when variable domains are required to be finite, this is no longer the case. Indeed, any function in $\mathfrak{D}_{\mathcal{E},\mathbb{B}}$ has a finite model set; it is thus always possible to build an SD representing the disjunction of all these models. □

Let us now examine the relationship of the SD and BDD families. Since SDs are close to interval automata, and we had $\mathtt{BDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \mathtt{IA}$ and $\mathtt{FBDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \mathtt{FIA}$, we expect to find similar results on SDs.

Proposition 7.2.7. The following properties hold:

$$\mathtt{BDD}_{\mathcal{I}}^{\mathbb{TZ}} = \mathtt{SDD},$$
$$\mathtt{FBDD}_{\mathcal{I}}^{\mathbb{TZ}} \subsetneq \mathtt{FSDD},$$

$$\mathtt{OBDD}_{\mathcal{I}}^{\mathbb{T}\mathbb{Z}} = \mathtt{OSDD},$$
$$\mathtt{OBDD}_{<,\mathcal{I}}^{\mathbb{T}\mathbb{Z}} = \mathtt{OSDD}_{<}.$$

Detailed proof p. 168.

This is the reason why we used "DD" to name set-labeled diagrams satisfying exclusive decision: SDD exactly corresponds to the general BDD language, restricted to integer variables and $\mathbb{Z}$-tileable labels. In particular, SDD contains BDDs on Boolean variables, as stated in the following proposition.

Proposition 7.2.8. It holds that

$$\mathtt{SDD}_{\mathcal{B}}^{\$\mathbb{B}} = \mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}},$$
$$\mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} = \mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}},$$
$$\mathtt{OSDD}_{<,\mathcal{B}}^{\$\mathbb{B}} = \mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}}.$$

Proof. These come directly from Proposition 7.2.7 and Proposition 1.2.7. □

We also get that $\mathtt{MDD} = \mathtt{OSDD}_{\mathcal{E}}^{\$\mathbb{Z}}$, since $\mathtt{MDD} = \mathtt{OBDD}_{\mathcal{E}}^{\$\mathbb{Z}}$. However, we can remark that $\mathtt{MDD} \subsetneq \mathtt{OSDD}_{\mathcal{E}}$, because of the literal expressivity of OSDD. We can even easily show that $\mathtt{OSDD}_{\mathcal{E}}$ is strictly more succinct than MDD.

Proposition 7.2.9. It holds that $\mathtt{MDD} \not\leqslant_s \mathtt{OSDD}_{\mathcal{E}}$.

Proof. This comes from the fact that edges in an MDD are labeled with singletons. Consider $n \in \mathbb{N}^*$, and a variable $x_n \in \mathcal{E}$ of domain $[0, \dots, 2^n]$. The function $\llbracket x_n \neq 0 \rrbracket$ has an $\mathtt{OSDD}_{\mathcal{E}}$-representation the size of which is 1: one x_n-node with one outgoing edge labeled $[1, \dots, 2^n]$ and pointing to the sink. The size of all its MDD-representations is exponential in n, since they need 2^n edges—one for each consistent value of x_n. □

To sum up, the SD family is closely related to the other decision diagram languages, but it generalizes binary and multivalued decision diagrams by several aspects:

- more general variables than $\mathcal{B}$ and $\mathcal{E}$;
- relaxed literal expressivity, allowing exponential gains in space;
- some languages relax the exclusive decision requirement, notably allowing pure disjunction nodes.

7.3 From IAs to SDs

We have seen, in the previous section, that interval automata are not particular set-labeled diagrams—and *vice-versa*. However, we can transform IAs into SDs, by discretizing them, as explained in the introduction of Part III. In this section, we formally present this discretization, and prove the "equivalence" of the input IA

and its discretization. Since `IA` and `SD` do not have the same interpretation domain, we cannot prove this equivalence within our framework. We must first decide what it means.

7.3.1 A Formal Definition of Discretization

Discretizing a continuous function is basically finding a discrete function that has "the same behavior". A discretization is often an approximation of the original function. However, functions represented by IAs have a specific property, as explained in the introduction of Part III: their model set can only be a finite union of boxes. This makes an exact discretization possible, as we will see.

The first thing we need to formally define is the discretization of a variable. Basically, this discretization is based on some partition of its domain, and consists of a discrete variable, together with some functions associating continuous values with discrete ones.

Definition 7.3.1 (Discretization of a variable)**.** Let x be an $\mathbb{R}$-tileable variable, and p a *finite* partition of $\mathrm{Dom}(x)$. The *discretization of x with respect to p* is the triple $\delta = \langle d, \sigma, \pi \rangle$ where

- $d \in \mathcal{E}$ is an enumerated variable, the domain of which verifies $|\mathrm{Dom}(d)| = |p|$;
- σ is a bijection from p to $\mathrm{Dom}(d)$;
- $\pi\colon \mathrm{Dom}(x) \to p$ is the *partition function*, associating each value $\omega \in \mathrm{Dom}(x)$ with the unique element of p containing ω.

Using this definition, each value ω in $\mathrm{Dom}(x)$ corresponds to a unique value ω_δ in $\mathrm{Dom}(d)$, given by

$$\omega_\delta = \sigma(\pi(\omega)),$$

and each dicrete value ω_δ in $\mathrm{Dom}(d)$ corresponds to a set of "interchangeable" values ω in $\mathrm{Dom}(x)$, verifying

$$\omega \in \sigma^{-1}(\omega_\delta).$$

Remark that any $\mathbb{R}$-tileable variable x has at least one (degenerate) discretization: indeed, $\{\mathrm{Dom}(x)\}$ is a finite partition of $\mathrm{Dom}(x)$.

We can extend this definition to a set of variables, joining together the functions σ and π of each variable.

Definition 7.3.2 (Discretization of a scope)**.** Let $X \subseteq \mathcal{T}$ be a finite set of $\mathbb{R}$-tileable variables, denoted as $X = \{x_1, \ldots, x_n\}$. For each $x_i \in X$, we consider a finite partition p_i of $\mathrm{Dom}(x_i)$, and $\delta_i = \langle d_i, \sigma_i, \pi_i \rangle$ the discretization of x_i with respect to p_i.

A *discretization of X* is a triple $\Delta = \langle D, \Sigma, \Pi \rangle$, with

- $D = \{d_1, \ldots, d_n\}$ the set of all d_i;

- $\Sigma\colon p_1 \times \cdots \times p_n \to \mathrm{Dom}(D)$, the function defined as the concatenation of functions $\sigma_1, \ldots, \sigma_n$;
- $\Pi\colon \mathrm{Dom}(X) \to p_1 \times \cdots \times p_n$ the partition function, defined as $\Pi(\vec{x}) = \langle \pi_1(\vec{x}_{|x_1}), \ldots, \pi_n(\vec{x}_{|x_n}) \rangle$.

In the same way as for single variables, we get that each X-assignment $\vec{x}$ corresponds to a unique D-assignment $\vec{d} = \Sigma(\Pi(\vec{x}))$, and that a given D-assignment $\vec{d}$ is associated with a set of X-assignments $\Sigma^{-1}(\vec{d})$.

We now have all the elements necessary to define the discretization of a Boolean function. Since we want an exact discretization, the original function applied on some assignment must return the same output as the discrete function applied on the corresponding discrete assignment.

Definition 7.3.3 (Discretization of a function)**.** Let $f \in \mathfrak{D}_{\mathcal{T},\mathbb{B}}$ be a Boolean function on $\mathbb{R}$-tileable variables, and X its scope. Let $\Delta = \langle D, \Sigma, \Pi \rangle$ be a discretization of X. A *discretization of f with respect to* Δ is a Boolean function $f_\Delta \in \mathfrak{D}_{\mathcal{E},\mathbb{B}}$, of scope D, such that

$$f \equiv f_\Delta \circ \Sigma \circ \Pi.$$

Note that there does not always exist a discretization for a given function. Indeed, we want the discretization to be *equivalent* to the continuous function. Hence, since we require domain partitions to be finite, functions that are not representable as a union of boxes have no discretization. Consider once again the function $\llbracket x = y \rrbracket$ for example: for its discretization to be equivalent, infinite partitions of $\mathrm{Dom}(x)$ and $\mathrm{Dom}(y)$ would be needed (such as $\mathbb{SR}$). Using finite partitions only, it is always possible to find two $\{x, y\}$-assignments yielding different results with f, but corresponding to the same $\{d_x, d_y\}$-assignment—and thus yielding the same result with f_Δ.

However, there exists a discretization for any function representable as an IA, or equivalently, Expr(`IA`) [Definition 1.2.11] is included in the set of discretizable functions. This is a corollary of the fact that any IA can be discretized—which we prove in the next section.

7.3.2 Transforming IAs into SDs

Algorithm 7.1 implements the procedure described informally in the introduction of Part III. It builds an SD corresponding to an input IA, using meshes obtained as described in the proof of Lemma 4.4.4, and *indexing sequences*—that is, for each mesh $\mathcal{M} = \{M_1, \ldots, M_n\}$, a sequence of values Indexes $= \langle m_1, \ldots, m_n \rangle$ such that $m_i \in M_i$ for all $i \in [1, \ldots, n]$. The SD built has the same characteristic size as the IA (it is actually even smaller, since variable domains, being intervals, are of characteristic size 1 in the `SD` framework), and fits our formal definition of a discretization. We use this procedure to prove the following proposition.

Algorithm 7.1 Given an IA φ and an indexing sequence Indexes^i for every variable x_i in Scope(φ), builds an SD ψ representing a discretization of $[\![\varphi]\!]$.

```
let ψ be the empty SD
for each node N in φ, ordered from the sink to the root do
    let x_i ≔ Var(N)
    add to ψ a node N′ labeled by d_i
    for each E ∈ Out(N) do
        let S ≔ ∅
        for each m_j ∈ Indexes^i do
            if m_j ∈ Lbl(E) then
                S ≔ S ∪ {j}
        let N′_E be the node in ψ corresponding to Dest(E)
        add to ψ an edge from N′ to N′_E, labeled S
```

Proposition 7.3.4. There exists a polytime algorithm that maps any IA φ to an SD ψ, of size $\|\psi\| \leqslant \|\varphi\|$, and such that $[\![\psi]\!]$ is a discretization of $[\![\varphi]\!]$.

Detailed proof p. 168.

This means that using this discretization, we lose no memory space and no information. Embarking an SD instead of an IA (along with tables representing each bijection σ_i) is thus completely harmless from a semantic point of view, and saves memory space: only $\sum_{x_i \in \text{Scope}(\varphi)} 2n_i$ real numbers need to be embarked for representing the σ bijections (two for each interval of each mesh). By construction of the mesh, this number is always smaller than the number of real numbers necessary to represent φ—and it can potentially be much inferior, for example when a same bound is used a lot in the graph.

Of course, this would not be interesting if the procedure did not maintain focusingness, but it actually does.

Proposition 7.3.5. There exists a polytime algorithm that maps any FIA φ to an FSD ψ, of size $\|\psi\| \leqslant \|\varphi\|$, and such that $[\![\psi]\!]$ is a discretization of $[\![\varphi]\!]$.

Detailed proof p. 170.

This section aimed at providing a formal motivation for the study of the SD family. Now that we know FIAs can be efficiently represented as FSDs, we examine the knowledge compilation properties of languages from this family.

7.4 The Knowledge Compilation Map of SD

This section contains the knowledge compilation map of the SD family. Most of the proofs are gathered in the end of the chapter [§ 7.5], for the sake of readability. We

start by pointing out some fundamental properties, that entail many results in the map.

7.4.1 Preliminaries

Combining SDs

In a similar fashion as for interval automata and binary decision diagrams, there exists simple procedures for combining SDs, using pure disjunctive nodes or making chains of graphs.

Proposition 7.4.1 (Disjunction)**.** Let $<$ be a total strict order on $\mathcal{I}$. SD, FSD, and $\mathtt{OSD}_<$ satisfy $\vee$**C** and $\vee$**BC**.

Detailed proof p. 170.

Proposition 7.4.2 (Conjunction)**.** SD and SDD satisfy $\wedge$**C** and $\wedge$**BC**.

Detailed proof p. 171.

Importantly enough, terms and clauses can be represented in polytime using any of the languages in the SD family.

Proposition 7.4.3 (Terms and clauses)**.** Let L be any of the SD fragments we defined; it holds that $\mathtt{L} \leqslant_p \mathtt{term}_{\mathcal{I}}^{\mathbb{TZ}}$ and $\mathtt{L} \leqslant_p \mathtt{clause}_{\mathcal{I}}^{\mathbb{TZ}}$.

Detailed proof p. 171.

From these three propositions, we trivially get fundamental properties about the relationship between the SD languages and the DNF and CNF ones.

Proposition 7.4.4. Let $<$ be a total strict order on $\mathcal{I}$. The following relations hold:

$$\mathtt{OSD}_< \leqslant_p \mathtt{DNF}_{\mathcal{I}}^{\mathbb{TZ}},$$
$$\mathtt{SDD} \leqslant_p \mathtt{CNF}_{\mathcal{I}}^{\mathbb{TZ}}.$$

Proof. Any term can be expressed as an $\mathtt{OSD}_<$-representation in polytime; and since $\mathtt{OSD}_<$ satisfies $\vee$**C**, obtaining the disjunction of a set of terms is also polytime.

Similarly, any clause can be expressed as an SDD-representation in polytime, and SDD satisfies $\wedge$**C**, hence the second result. □

In particular, we get that $\mathtt{OSD}_< \leqslant_p \mathtt{DNF}_{\mathcal{B}}^{\mathbb{SB}}$ and $\mathtt{SDD} \leqslant_p \mathtt{CNF}_{\mathcal{B}}^{\mathbb{SB}}$, which brings important results, for succinctness, queries, and transformations.

Validity of FSDDs

Proposition 7.4.5. FSDD satisfies **VA**.

Detailed proof p. 172.

The proof relies on Algorithm 7.2, which roughly checks whether "what enters a node is equal to what goes out of it". As checking validity on DNF is untractable, this notably allows to prove that FSDD does not satisfy $\vee$**C**.

Algorithm 7.2 Given an FSDD φ, checks whether φ is valid.

for each $x \in \text{Scope}(\varphi)$ **do**
  for each node N in φ **do**
    let $S_{x,N} \coloneqq \varnothing$
  $S_{x,\text{Root}(\varphi)} \coloneqq \text{Dom}(x)$
for each node N in φ, ordered from the root to the sink **do**
  let $x \coloneqq \text{Var}(N)$
  if $\bigcup_{E \in \text{Out}(N)} \text{Lbl}(E) \not\supseteq S_{x,N}$ **then**
    return false
  for each $E \in \text{Out}(N)$ **do**
    $S_{x,\text{Dest}(E)} \coloneqq S_{x,\text{Dest}(E)} \cup S_{x,N} \cup \text{Lbl}(E)$
    for each $y \in \text{Scope}(\varphi) \setminus \{x\}$ **do**
      $S_{y,\text{Dest}(E)} \coloneqq S_{y,\text{Dest}(E)} \cup S_{y,N}$
return true

Disjoining FSDDs with clauses

However, any FSDD can be tractably disjoined with any clause.

| **Proposition 7.4.6.** FSDD satisfies $\vee$**clC**.

Detailed proof p. 173.

Together with validity, this property is the key allowing us to prove that FSDD satisfies **IM**, which has important consequences on succinctness, much like **CE**; see Lemma 7.5.6 (p. 174).

Negation of SDDs

Negating an SDD can be performed by using Algorithm 7.3, that recursively "complements" each node, starting from the sink.

| **Proposition 7.4.7** (Negation)**.** SDD, OSDD, and $\text{OSDD}_{<}$ satisfy $\neg$**C**.

Detailed proof p. 177.

However, this procedure does not maintain focusingness, and it is actually unknown whether FSDD satisfies $\neg$**C**.

7.4.2 Succinctness

| **Theorem 7.4.8** (Succinctness of the SD family)**.** The results of Table 7.1 hold.

Detailed proof p. 184.

Figure 7.5 sums up the succinctness results of Theorem 7.4.8, as well as some other results about the succinctness of the SD family as compared with BDD (see § 7.2.2). Roughly speaking, we can see that imposing focusingness, exclusivity or

Algorithm 7.3 Computes the negation of an SDD φ.

if φ is empty **then**
    return the sink-only graph
else if φ is the sink-only graph **then**
    return the empty graph
let ψ be the sink-only graph
let $T \coloneqq \varnothing$ be a set of couples of nodes, associating each node in φ with a node in ψ
for each node N in φ, ordered from the sink to the root, excluding the sink **do**
    create a node N' labeled by the same variable x as N
    let $U \coloneqq \mathrm{Dom}(x) \setminus \bigcup_{E \in \mathrm{Out}(N)} \mathrm{Lbl}(E)$
    if $U \neq \varnothing$ **then**
        add to N' an outgoing edge E_{compl} labeled by U and pointing to the sink of ψ
    for each $E \in \mathrm{Out}(N)$ **do**
        let $D \coloneqq \mathrm{Dest}(E)$
        if there exists a couple $\langle D, D' \rangle \in T$ **then** *// D has a corresponding node D' in ψ*
            add to N' an outgoing edge E' labeled by $\mathrm{Lbl}(E)$ and pointing to D'
    if N' has at least one outgoing edge **then**
        add N' to ψ
        add $\langle N, N' \rangle$ to T
let $R \coloneqq \mathrm{Root}(\varphi)$
if there exists a couple $\langle R, R' \rangle \in T$ **then** *// the root has a corresponding node*
    return ψ
else
    return the empty graph

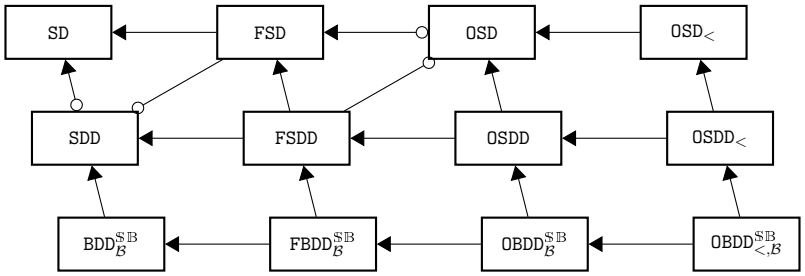

Figure 7.5: Relative succinctness of the SD and $\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$ families. On an edge linking $\mathtt{L}_1$ and $\mathtt{L}_2$, if there is an arrow pointing towards $\mathtt{L}_1$, it means that $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$. If there is no symbol on $\mathtt{L}_1$'s side (neither an arrow nor a circle), it means that $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$. If there is a circle on $\mathtt{L}_1$'s side, it means that it is unknown whether $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$ or $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$ holds. Relations deducible by transitivity are not represented, which means that two fragments not being ancestors to each other are incomparable with respect to succinctness.

L	SD	SDD	FSD	FSDD	OSD	OSDD	OSD$_<$	OSDD$_<$
SD	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$
SDD	?	$\leqslant_s$	?	$\leqslant_s$	?	$\leqslant_s$	?	$\leqslant_s$
FSD	$\not\leqslant_s^*$	$\not\leqslant_s^*$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$
FSDD	$\not\leqslant_s^*$	$\not\leqslant_s^*$	$\not\leqslant_s^*$	$\leqslant_s$	$\not\leqslant_s^*$	$\leqslant_s$	$\not\leqslant_s^*$	$\leqslant_s$
OSD	$\not\leqslant_s$	$\not\leqslant_s$	?	?	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$
OSDD	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\leqslant_s$	$\not\leqslant_s$	$\leqslant_s$
OSD$_<$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\leqslant_s$	$\leqslant_s$
OSDD$_<$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\leqslant_s$

Table 7.1: Results about succinctness. A star (*) indicates a result that holds unless the polynomial hierarchy PH collapses.

ordering may lead to an exponential increase in space. Let us give a few indications about why the most crucial results hold.

Results holding modulo the collapse of the polynomial hierarchy all come from Lemma 4.3.10, that states that no polysize compilation function can make clausal entailment on a CNF tractable.

The proof of $\mathtt{OSD}_< \not\leqslant_s \mathtt{OSDD}$ comes from the classical $\bigwedge_{i=1}^{n} [y_i = z_i]$ family of functions, that has polynomial representations for some variable orders and only exponential representations for some others—which proves that in the worst case, one cannot impose an order without requiring exponential space.

To prove that $\mathtt{OSDD} \not\leqslant_s \mathtt{FSDD}$, we use the function representing the n-coloring problem of a star graph with n vertices. Figure 7.6 is an illustration of what happens when $n = 3$: putting the center variable at the root gives polynomial OSDDs, while putting it in last position can give exponential OSDDs. We use this to build an FSDD that has no equivalent OSDD of polynomial size.

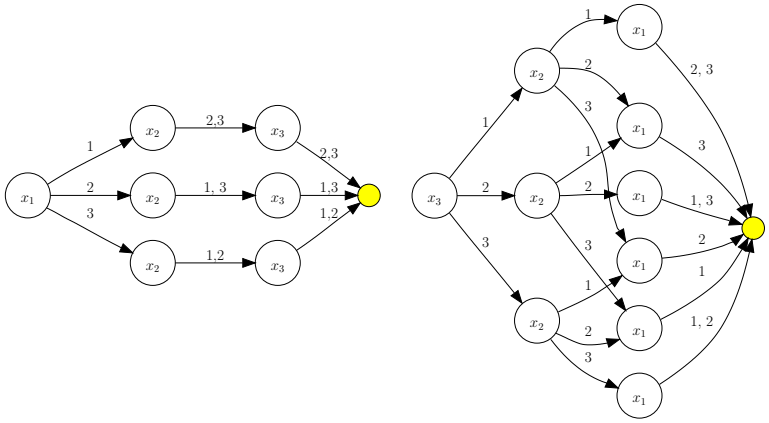

Figure 7.6: The coloring problem on a star graph with 3 variables and 3 colors, that is, $x_2 \neq x_1 \neq x_3$. The OSDD on the right is ordered with the center variable x_1 in last position, whereas the OSDD on the left is ordered with x_1 in first position.

Finally, $\texttt{OSDD} \not\leqslant_s \texttt{OSD}_<$ is implied by the Boolean-variable succinctness map [Theorem 1.4.17]: if it were false, we would get $\texttt{OBDD}^{\$\mathbb{B}}_{\mathcal{B}} \leqslant_s \texttt{DNF}^{\$\mathbb{B}}_{\mathcal{B}}$, which is impossible.

7.4.3 Queries and Transformations

Theorem 7.4.9. The results in Tables 7.2 and 7.3 hold.

Detailed proof p. 196.

Some remarks are in order. First, as it is usually the case in knowledge compilation, languages satisfying many queries do not satisfy many transformations, and *vice-versa*. As in the IA knowledge compilation map, there is a clear similarity between focusing SD languages and decomposable NNF ones [Theorem 1.4.18]: FSD and DNNF satisfy *the same set of queries and transformations*, as well as FSDD and d-DNNF —whereas focusing SDs are *not* decomposable structures.

We can also see that OSDD and $\texttt{OSDD}_<$, being direct extensions of $\texttt{OBDD}^{\$\mathbb{B}}_{\mathcal{B}}$ and $\texttt{OBDD}^{\$\mathbb{B}}_{<,\mathcal{B}}$, have unsurprisingly very similar properties. There is however an important difference: while Boolean OBDDs support **SFO** in polytime, integer ones do not support it. This is because the size of the variable domains are known in the $\texttt{BDD}^{\$\mathbb{B}}_{\mathcal{B}}$ framework: forgetting a variable amounts to a binary disjunction using Shannon's decomposition. On the contrary, in the case of SDs, domains are unbounded, so to apply the same procedure, unbounded disjunction is necessary—yet it is not satisfied by OSDD or $\texttt{OSDD}_<$.

Note also the similarity between OSD and FSD. They satisfy the exact same set of queries and transformations—except for $\vee$**C** and $\vee$**BC**, of which we do not know whether they are satisfied by OSD.

L	CO	VA	MC	CE	IM	EQ	SE	MX	CX	CT	ME
SD	○	○	√	○	○	○	○	○	○	○	○
SDD	○	○	√	○	○	○	○	○	○	○	○
FSD	√	○	√	√	○	○	○	√	√	○	√
FSDD	√	√	√	√	√	?	○	√	√	?	√
OSD	√	○	√	√	○	○	○	√	√	○	√
OSDD	√	√	√	√	√	√	○	√	√	√	√
$\mathtt{OSD}_<$	√	○	√	√	○	○	○	√	√	○	√
$\mathtt{OSDD}_<$	√	√	√	√	√	√	√	√	√	√	√

Table 7.2: Results about queries; √ means "satisfies", and ○ means "does not satisfy, unless $\mathsf{P} = \mathsf{NP}$".

L	CD	TR	FO	SFO	EN	SEN	$\vee$C	$\vee$BC	$\vee$clC	$\wedge$C	$\wedge$BC	$\wedge$tC	$\neg$C
SD	√	○	○	√	○	√	√	√	√	√	√	√	?
SDD	√	○	○	√	○	√	√	√	√	√	√	√	√
FSD	√	√	√	√	○	○	√	√	√	○	○	√	○
FSDD	√	○	○	○	○	○	○	○	√	○	○	√	?
OSD	√	√	√	√	○	○	?	?	√	○	○	√	○
OSDD	√	•	•	•	•	•	•	○	√	•	○	√	√
$\mathtt{OSD}_<$	√	√	√	√	○	○	√	√	√	○	√	√	○
$\mathtt{OSDD}_<$	√	•	•	•	•	•	•	√	√	•	√	√	√

Table 7.3: Results about transformations; √ means "satisfies", • means "does not satisfy", and ○ means "does not satisfy, unless $\mathsf{P} = \mathsf{NP}$".

From a more practical point of view, when one needs to execute queries on compiled forms, but no transformation except conditioning (e.g., in configuration applications), FSDDs are more interesting than OSDDs—they can be much more compact, while satisfying almost the same queries. Imposing a variable order (that is, going from FSDD to OSDD) is worthwhile whenever one of the $\neg$**C**, **SEN**, or **SFO** transformations is required. Relaxing the requirement of exclusive decision, we get the FSD fragment, which moreover satisfies **FO** and $\vee$**C**; FSD particularly suits applications like planning, where one needs to often check consistency, forget variables, and extract models.

*
**

After these theoretical considerations about SDs, Chapter 8 (which starts on page 199, after the proofs) takes a more practical point of view, examining a way to *compile* set-labeled diagrams.

7.5 Chapter Proofs

7.5.1 Reduction

Proof of Proposition 7.1.7 [p. 154]. The proof of Proposition 4.1.15 still works here. Checking contiguity is simpler, and the "stammering" operation does not create more edges than it deletes: $\|\mathrm{Lbl}(E_{\text{in}}) \cap \mathrm{Lbl}(E_{\text{out}})\| < \|\mathrm{Lbl}(E_{\text{in}})\| + \|\mathrm{Lbl}(E_{\text{out}})\|$ since either $\|\mathrm{Lbl}(E_{\text{in}})\|$ or $\|\mathrm{Lbl}(E_{\text{out}})\|$ equals 1, which means that the label is a single interval. □

We also prove Proposition 7.1.7 thanks to Algorithm 4.1 [p. 102], using the following lemma.

Lemma 7.5.1. Algorithm 4.1 [p. 102] maintains the exclusive decision property on SDs.

Proof. Let φ be an SD satisfying the exclusive decision property.

- "Stammering" operation: parent labels are modified only by removal of values, so they are still exclusive after the operation.
- "Dead" operation: suppressing edges does not have any influence on exclusivity.
- "Contiguous" operation: the two contiguous edges were disjoint with the other edges, so their union also is.
- "Undecisive" operation: no edge is modified or added.
- "Isomorphism" operation: no edge is modified or added.

Hence, the result of Algorithm 4.1 still satisfies the exclusive decision property. □

Proof of Proposition 7.2.3 [p. 155]. The argumentation developed in the proof of Proposition 4.2.4 is still valid for SDs. Algorithm 4.1 maintains the focusingness property on SDs, which proves the result for FSD. It is obvious that it also maintains the ordering property (the order of nodes is not modified; nodes can be removed, but this does not affect the ordering property). The proposition thus holds for $\mathtt{OSD}_<$ and OSD. Last, since the procedure moreover maintains strong decision (it never adds $\veebar$-nodes) and exclusive decision [Lemma 7.5.1], we get the result for SDD, FSDD, $\mathtt{OSDD}_<$, and OSDD. □

7.5.2 Relationship with Other Languages

Proof of Proposition 7.2.5 [p. 156]. An integer interval is $\mathbb{R}$-tileable if and only if it is finite, so we get $\mathcal{E} = \mathcal{I} \cap \mathcal{T}$. We also get that the intersection of $\mathbb{IR}$ and $\mathbb{TZ}$ is the set of singletons of $\mathbb{Z}$.

Now, let φ be an $\mathtt{SD}_{\mathcal{E}}^{\mathbb{SZ}}$-representation. Since it is an $\mathtt{NNF}_{\mathcal{E}}^{\mathbb{SZ}}$-representation satisfying $\wedge$-simple decision, it belongs to $\mathtt{IA}_{\mathcal{E}}^{\mathbb{SZ}}$, by definition. Hence, $\mathtt{SD}_{\mathcal{E}}^{\mathbb{SZ}} \subseteq \mathtt{IA}_{\mathcal{E}}^{\mathbb{SZ}}$, and $\mathtt{SD}_{\mathcal{E}}^{\mathbb{SZ}} \subseteq \mathtt{SD} \cap \mathtt{IA}$ (indeed, $\mathtt{SD}_{\mathcal{E}}^{\mathbb{SZ}} \subseteq \mathtt{SD}$ and $\mathtt{IA}_{\mathcal{E}}^{\mathbb{SZ}} \subseteq \mathtt{IA}$).

We only have to prove that $\mathtt{SD} \cap \mathtt{IA} \subseteq \mathtt{SD}_{\mathcal{E}}^{\mathbb{SZ}}$ to complete the proof. Let φ be an $\mathtt{SD} \cap \mathtt{IA}$-representation. Its scope has to be included in both $\mathcal{I}$ and $\mathcal{T}$, and therefore in $\mathcal{E}$; similarly, its literals belong to both $\mathbb{IR}$ and $\mathbb{TZ}$—and thus to $\mathbb{SZ}$. All in all, φ is an $\mathtt{NNF}_{\mathcal{E}}^{\mathbb{SZ}}$-representation satisfying $\wedge$-simple decision: it is an element of $\mathtt{SD}_{\mathcal{E}}^{\mathbb{SZ}}$. □

Proof of Proposition 7.2.7 [p. 156]. SDD is the restriction of $\mathtt{NNF}_{\mathcal{I}}^{\mathbb{TZ}}$ to strong and exclusive decision, therefore it is equal to $\mathtt{NNF}_{\mathcal{I}}^{\mathbb{TZ}} \cap \mathtt{BDD}$. Now, we show that $\mathtt{BDD}_{\mathcal{I}}^{\mathbb{TZ}}$ is also equal to $\mathtt{NNF}_{\mathcal{I}}^{\mathbb{TZ}} \cap \mathtt{BDD}$.

Indeed, since $\mathtt{BDD}_{\mathcal{I}}^{\mathbb{TZ}}$ is the restriction of BDD to variables from $\mathcal{I}$ and to literal expressivity $\mathbb{TZ}$, it is therefore included in $\mathtt{NNF}_{\mathcal{I}}^{\mathbb{TZ}} \cap \mathtt{BDD}$. Now, let $\varphi \in \mathcal{R}_{\mathtt{NNF}_{\mathcal{I}}^{\mathbb{TZ}} \cap \mathtt{BDD}}$; since it is a BDD-representation, of literal expressivity $\mathbb{TZ}$ and of scope included in $\mathcal{I}$, by definition of literal expressivity and thanks to Proposition 1.2.5, we get that it is included in $\mathtt{BDD}_{\mathcal{I}}^{\mathbb{TZ}}$. Thus $\mathtt{BDD}_{\mathcal{I}}^{\mathbb{TZ}} = \mathtt{NNF}_{\mathcal{I}}^{\mathbb{TZ}} \cap \mathtt{BDD} = \mathtt{SDD}$. The proof is similar for the two other equalities.

Now, $\mathtt{FBDD}_{\mathcal{I}}^{\mathbb{TZ}} \subseteq \mathtt{BDD}_{\mathcal{I}}^{\mathbb{TZ}}$, so $\mathtt{FBDD}_{\mathcal{I}}^{\mathbb{TZ}} \subseteq \mathtt{SDD}$. Since variables are never repeated along a path of an FBDD, any $\mathtt{FBDD}_{\mathcal{I}}^{\mathbb{TZ}}$-representation is trivially focusing. Hence $\mathtt{FBDD}_{\mathcal{I}}^{\mathbb{TZ}} \subseteq \mathtt{FSDD}$. Since not all focusing SDDs are read-once, we get the result. □

7.5.3 Discretization of IAs

Proof of Proposition 7.3.4 [p. 160]. Let φ be an IA. For each variable $x_i \in \mathrm{Scope}(\varphi)$, we consider a number of elements. First, let $\mathcal{M}^i = \{M_1^i, \ldots, M_{n_i}^i\}$ be a mesh of x_i in φ; it can be built with a simple traversal of φ, as shown in the

proof of Lemma 4.4.4. Second, we consider an *indexing sequence* Indexes^i associated with $\mathcal{M}^i$, that is, Indexes$^i = \langle m_1, \ldots, m_{n_i} \rangle$ such that for all $j \in [1, \ldots, n_i]$, $m_j \in M^i_j$. We impose the harmless condition that the m_j are in ascending order ($\forall j \in [1, \ldots, n_i - 1], m_j < m_{j+1}$). Last, we consider a discrete variable $d_i \in \mathcal{E}$, of domain $[1, \ldots, n_i]$.

Algorithm 7.1 [p. 160] uses these elements to build an SD ψ. Let us first show that $\|\psi\|$ is less than $\|\varphi\|$. Each node in φ corresponds to one node in ψ, each edge in φ corresponds to one edge in ψ. This is not enough to prove our claim, since while `IA` edges are of size 1, the size of `SD` edges depends on their label. However, we prove that they are all of size 1 too.

Indeed, suppose there is an edge E in ψ such that $\|E\| \geqslant 2$. This means it is labeled with a union of disjoint integer intervals: denoting $x_i = \text{Var}(E)$, there exists three integers a, b, and c in $[1, \ldots, n_i]$ such that $a < b < c$ and $a \in \text{Lbl}(E)$, $c \in \text{Lbl}(E)$, but $b \notin \text{Lbl}(E)$. Then the label of the edge E' in φ corresponding to E contains m_a and m_c, but not m_b. Since the m_j are in ascending order by hypothesis, this means $\text{Lbl}(E')$ is not an interval, which is absurd. Hence all edges in ψ are of size 1.

Now, for each i, $\|\text{Dom}(d_i)\| = 1$ since $\text{Dom}(d_i)$ is an integer interval. Yet $\|\text{Dom}(x_i)\| \geqslant 1$, as it is an $\mathbb{R}$-tileable set. All in all, φ and ψ have the same "edge size", and the variables of φ all have greater size; hence $\|\psi\| \leqslant \|\varphi\|$.

We now prove that $[\![\psi]\!]$ is a discretization of $[\![\varphi]\!]$. Let us consider, for each x_i, the discretization $\delta_i = \langle d_i, \sigma_i, \pi_i \rangle$ of x_i with respect to $\mathcal{M}^i$. Bijection σ_i simply associates with any element M^i_j of $\mathcal{M}^i$ its index j. Function π_i associates with any value in $\text{Dom}(x_i)$ the unique element of $\mathcal{M}^i$ to which it belongs. Let $\Delta = \langle D, \Sigma, \Pi \rangle$ be the discretization of $\text{Scope}(\varphi)$ defined using all the δ_i. We show that $[\![\psi]\!]$ is a discretization of $[\![\varphi]\!]$ with respect to Δ. To this end, we prove that $[\![\varphi]\!] \equiv [\![\psi]\!] \circ \Sigma \circ \Pi$.

We consider $\vec{x} \in \text{Dom}(\text{Scope}(\varphi))$.

($\Rightarrow$) Let us suppose that $[\![\varphi]\!](\vec{x}) = \top$. Then there exists a path p in φ that is compatible with $\vec{x}$. We consider the path p' in ψ corresponding to p. Let E' be an edge along p' and E its corresponding edge in p. Let $x_i = \text{Var}(E)$ and $\omega = \vec{x}_{|x_i}$. Since E is compatible with $\vec{x}$, $\omega \in \text{Lbl}(E)$. Denoting M^i_j the element of mesh $\mathcal{M}^i$ containing ω, we know that $M^i_j \subseteq \text{Lbl}(E)$, by definition of a mesh. Hence the indexing value $m_j \in \text{Indexes}^i$ is in $\text{Lbl}(E)$ too, therefore $j \in \text{Lbl}(E')$ by construction.

Now, by definition of the discretization, $\Sigma(\Pi(\vec{x}))_{|d_i} = j$. Therefore the D-assignment $\Sigma(\Pi(\vec{x}))$ is compatible with E'. Since this holds for any E', it is compatible with the whole path p', which proves that it is a model of ψ: it holds that $[\![\psi]\!](\Sigma(\Pi(\vec{x}))) = \top$, and hence $([\![\psi]\!] \circ \Sigma \circ \Pi)(\vec{x}) = \top$.

($\Leftarrow$) Let us suppose that $([\![\psi]\!] \circ \Sigma \circ \Pi)(\vec{x}) = \top$. Let $\vec{d} = \Sigma(\Pi(\vec{x}))$; $\vec{d}$ is a model of ψ. Let us consider a path p in ψ compatible with $\vec{d}$, and its corresponding path p' in φ. Each edge E along p verifies $\vec{d}_{|\,\text{Var}(E)} \in \text{Lbl}(E)$.

Denoting $d_i = \mathrm{Var}(E)$ and $j = \vec{d}_{|d_i}$, we can infer that the indexing value $m_j \in \mathrm{Indexes}^i$ belongs to the label of the edge E' corresponding to E in φ. Therefore, $M^i_j \cap \mathrm{Lbl}(E') \neq \varnothing$, and this implies, by definition of a mesh, that $M^i_j \subseteq \mathrm{Lbl}(E)$.

Now, $\Sigma(\Pi(\vec{x}))_{|d_i} = j$ means that $\sigma_i(\pi_i(\vec{x}_{|x_i}))$ and thus $\pi_i(\vec{x}_{|x_i}) = \sigma_i^{-1}(j)$, from which we get $\vec{x}_{|x_i} \in M^i_j$. Hence $\vec{x}_{|x_i} \in \mathrm{Lbl}(E')$: E' is compatible with $\vec{x}$, as are all edges along p', so $\vec{x}$ is a model of φ. This proves that $[\![\varphi]\!](\vec{x}) = \top$. □

Proof of Proposition 7.3.5 [p. 160]. Algorithm 7.1 [p. 160] maintains focusingness. Indeed, if an edge (on some variable x) is not focusing in ψ, it means that there exists an integer j in its label that does not belong to one of its ancestor's label. Considering the corresponding edges and value in φ, this implies that there is a mesh index m_j and two x-edges E and E' such that $m_j \notin \mathrm{Lbl}(E)$, $m_j \in \mathrm{Lbl}(E')$, and yet E is an ancestor of E', which means φ is not focusing.

Therefore, by contrapositive, if φ is focusing then ψ is focusing. Our procedure fits all the requirements of the proposition (as proven in Proposition 7.3.4) and moreover maintains focusingness, which proves the claim. □

7.5.4 Preliminaries to the Map

Combining SDs

Lemma 7.5.2. There exists a linear algorithm mapping every finite set Φ of SDs to an SD ψ such that $[\![\psi]\!] \equiv \bigvee_{\varphi\in\Phi} [\![\varphi]\!]$. In addition, if all SDs in Φ are focusing (resp. ordered by some total strict order $<$), ψ is also focusing (resp. ordered by $<$).

Proof. The proof is similar to that of Proposition 4.3.3: we copy all SDs of Φ into ψ, fuse their sinks, and create a new root node for ψ, labeled by $\veebar$, with $|\Phi|$ outgoing edges, each one labeled by $\{0\}$ and pointing to the root of a different $\varphi \in \Phi$. Since neither node order nor edge labels are modified in each φ, the procedure maintains focusingness and ordering (provided that all $\varphi \in \Phi$ be ordered with respect to the same $<$). The algorithm is obviously linear in $\sum_{\varphi\in\Phi} \|\varphi\|$. □

Proof of Proposition 7.4.1 [p. 161]. As a direct corollary of Lemma 7.5.2, we get that SD, FSD, and $\mathtt{OSD}_<$ satisfy $\vee$**C**, and thus $\vee$**BC**. □

Lemma 7.5.3. There exists a linear algorithm mapping every finite set Φ of SDs to an SD ψ such that $[\![\psi]\!] \equiv \bigwedge_{\varphi\in\Phi} [\![\varphi]\!]$. In addition, if all SDs in Φ are deterministic, ψ is also deterministic.

Proof. The proof is similar to that of Proposition 4.3.4: we build ψ by replacing the sink of each SD by the root of the following SD, the root of ψ ending up being the root of the first SD, and its sink the sink of the last SD. The procedure is obviously linear in $\sum_{\varphi\in\Phi} \|\varphi\|$, and since no label is modified, ψ is deterministic if every $\varphi \in \Phi$ is deterministic. □

Proof of Proposition 7.4.2 [p. 161]. As a direct corollary of Lemma 7.5.3, we get that SD and SDD satisfy $\wedge$**C**, and thus $\wedge$**BC**. □

Algorithm 7.4 Given a $\mathtt{term}_{\mathcal{I}}^{\mathbb{TZ}}$-representation γ, and a total strict order $<$ on Scope(γ), builds an $\mathtt{OSDD}_{<}$-representation ψ of $[\![\gamma]\!]$.

if γ is a constant ($\top$ or $\bot$) **then**
    return γ
let ψ be the sink-only SD
for each $x \in$ Scope(γ), ordered in descending $<$ order **do**
    let $S \coloneqq$ Dom(x)
    for each literal $\langle x, A\rangle$ in γ **do**
        $S \coloneqq S \cap A$
    add to φ an x-node N with one outgoing edge, labeled S and pointing to Root(φ)
    let N be the new root of φ

Algorithm 7.5 Given a $\mathtt{clause}_{\mathcal{I}}^{\mathbb{TZ}}$-representation γ, and a total strict order $<$ on Scope(γ), builds an $\mathtt{OSDD}_{<}$-representation ψ of $[\![\gamma]\!]$.

if γ is a constant ($\top$ or $\bot$) **then**
    return γ
let ψ be the sink-only SD
for each $x \in$ Scope(γ), ordered in descending $<$ order **do**
    let $S \coloneqq \varnothing$
    for each literal $\langle x, A\rangle$ in γ **do**
        $S \coloneqq S \cup A$
    add to φ an x-node N
    add to N an outgoing edge, labeled S and pointing to Sink(φ)
    add to N an outgoing edge, labeled Dom(x) $\setminus S$ and pointing to Root(φ)
    let N be the new root of φ

Lemma 7.5.4. Let $<$ be a total strict order on $\mathcal{I}$. It holds that $\mathtt{OSDD}_{<} \leqslant_p \mathtt{term}_{\mathcal{I}}^{\mathbb{TZ}}$ and $\mathtt{OSDD}_{<} \leqslant_p \mathtt{clause}_{\mathcal{I}}^{\mathbb{TZ}}$.

Proof. We use Algorithms 7.4 and 7.5. Both procedures are trivially polynomial in $\|\gamma\|$, and they return an SD that is exclusive and ordered with respect to $<$.

□

Proof of Proposition 7.4.3 [p. 161]. As a direct consequence of Lemma 7.5.4, we get that any language L such that $\mathtt{L} \leqslant_p \mathtt{OSDD}_{<}$ verifies both $\mathtt{L} \leqslant_p \mathtt{term}_{\mathcal{I}}^{\mathbb{TZ}}$ and $\mathtt{L} \leqslant_p \mathtt{clause}_{\mathcal{I}}^{\mathbb{TZ}}$. This is in particular the case for languages L of which $\mathtt{OSDD}_{<}$ is a sublanguage—hence the result, thanks to Proposition 7.2.2. □

Validity of FSDDs

Proof of Proposition 7.4.5 [p. 161]. We prove that FSDD satisfies **VA**.

Algorithm 7.2 [p. 162], applied to an FSDD φ, returns true if and only if φ is valid. In this algorithm, we consider all labels to be included in the domain of their corresponding variable; this is not the case in general, nor is it enforced by reduction, but it is trivial to transform any FSDD to fit that condition. Basically, make the intersection of each label with the domain of its corresponding domain; this can be done in time linear in the size of the FSDD.

The idea of the algorithm is to check whether what comes in each node is included into what comes out, i.e., no value is "lost" (what comes in the root being the whole $\text{Dom}(\text{Scope}(\varphi))$).

Now, our algorithm checks whether the following property, that we denote $\text{InEqOut}(N)$, holds for each node N:

- if there exists a path from the root to N that contains no other x-node than N, then

$$\forall \omega \in \text{Dom}(\text{Var}(N)), \exists E_\omega \in \text{Out}(N), \omega \in \text{Lbl}(E_\omega);$$

- if not, then for every ancestor edge E of N

$$\forall \omega \in \text{Lbl}(E), \exists E_\omega \in \text{Out}(N), \omega \in \text{Lbl}(E_\omega).$$

Our algorithm does this by gathering at each node the last label encountered for each variable (line 10), and making at each node, for each variable, the union of gathered labels of all incoming paths (Lines 10 and 12). It checks on line 7 whether the union of outgoing labels contains all the computed "label set" for the node's variable. The root is treated as a special case, allowing us to check the first item of $\text{InEqOut}(N)$'s definition.

Now, let us prove that

$$\forall N \in \mathscr{N}_\varphi, \text{InEqOut}(N) \quad \Leftrightarrow \quad \top \models \varphi.$$

($\Rightarrow$) We show that for any assignment $\vec{y} \in \text{Dom}(\text{Scope}(\varphi))$, there exists a path in the graph that is compatible with $\vec{y}$. Let us consider $\vec{y} \in \text{Dom}(\text{Scope}(\varphi))$. At each node N labeled by any variable x, we are assured to have the possibility to choose an outgoing edge compatible with $\vec{y}_{|x}$. Indeed, either this is the first x-node we encounter, and thus there exists an outgoing edge compatible with any value in $\text{Dom}(x)$ (first item of $\text{InEqOut}(N)$), or we already encountered at least one x-node, and chosen a value ω for x; then the second item of $\text{InEqOut}(N)$'s definition ensures that there exists an outgoing edge compatible with ω.

As it is true for each node, there always exists a compatible edge that can be taken, therefore we are assured to reach the sink; overall, there exists a path

from the root to the sink of φ that is compatible with $\vec{y}$. This proves that $\vec{y}$ is a model of φ.

As it is true for any assignment $\vec{y} \in \text{Dom}(\text{Scope}(\varphi))$, φ is valid.

($\Leftarrow$) We prove this by contraposition. Suppose there exists a node N that does not verify $\text{InEqOut}(N)$. Denoting $\text{Var}(N) = x$, this means that

- either (a) there exists a path p_R from the root to N that contains no other x-node than N, and yet there exists a value $\omega \in \text{Dom}(x)$ such that $\forall E \in \text{Out}(N), \omega \notin \text{Lbl}(E)$;
- or (b) there is no such path, and there exists an ancestor edge E_{ancestor} of N and a value $\omega \in \text{Lbl}(E_{\text{ancestor}})$ such that $\forall E\, \text{Out}(N), \omega \notin \text{Lbl}(E)$.

Let us consider a path p from the root to n, the one denoted by p_R in case (a), or any path containing E_{ancestor} in case (b). Let $\vec{y} \in \text{Dom}(\text{Scope}(\varphi))$ be an assignment compatible with p, and such that $\vec{y}_{|x} = \omega$. Such an assignment exists, because (i) as φ is focusing, all paths from the root to N are compatible with at least one model (there is no contradiction between edges—as long as φ is reduced, of course), and (ii) p has been defined in both cases (a) and (b) to be compatible with value ω for x.

Let us suppose $\vec{y} \in \text{Mod}(\varphi)$. There exists a path p', from the root to the sink of φ, that is compatible with $\vec{y}$. Determinism imposes that p and p' be equal from the root to N; indeed, only one acceptable choice is possible at each node. Since p' is compatible with $\vec{y}$, this means there is an edge E going out of N and such that $\omega \in \text{Lbl}(E)$, which has been supposed to be false. By contradiction, $\vec{y} \notin \text{Mod}(\varphi)$, hence φ is not valid.

Overall, our polytime algorithm checks that $\text{InEqOut}(N)$ holds for all N, and we showed that it is equivalent to checking validity. As a consequence, FSDD satisfies **VA**. □

Disjoining FSDDs with clauses, and other succinctness preliminaries

Proof of Proposition 7.4.6 [p. 162]. Let γ be a clause, and φ an FSDD. First, we note that $[\![\varphi]\!] \vee [\![\gamma]\!] \equiv \big([\![\varphi]\!] \wedge \neg[\![\gamma]\!]\big) \vee [\![\gamma]\!]$.

Let us suppose that $\gamma = \big[x_1 \in A_1\big] \vee \cdots \vee \big[x_k \in A_k\big]$. Then the term $\delta = \big[x_1 \in \text{Dom}(x_1) \setminus A_1\big] \wedge \cdots \wedge \big[x_k \in \text{Dom}(x_k) \setminus A_k\big]$ is equivalent to $\neg[\![\gamma]\!]$, and can be obtained in polytime [Prop. 7.4.3]. Since FSDD satisfies $\wedge$**tC** [Lemma 7.5.7], we can thus obtain in polytime an FSDD ψ_t representing $[\![\varphi]\!] \wedge \neg[\![\gamma]\!]$. This FSDD moreover has an important property: for all $x \in \text{Scope}(\gamma)$, and all literal $\langle x, A\rangle$ in γ, the label of each x-edge of ψ_t is included in $\text{Dom}(x) \setminus A$.

This is the key that will allow us to disjoin ψ_t with γ while maintaining focusingness. We can use a procedure similar to Algorithm 7.5 [p. 171], with the difference that on line 3, instead of using the sink-only SD, we use the SD ψ_t we just built. Obviously enough, the resulting SD represents $[\![\psi_t]\!] \vee \gamma$, and is focusing,

since the new x-edges pointing to the root of ψ_t are labeled $\mathrm{Dom}(x) \setminus A$—a superset of all x-edge labels in ψ_t. As the procedure is polytime, we get that FSDD satisfies $\vee$**clC**. □

Lemma 7.5.5. FSDD satisfies **IM**.

Proof. Let γ be a term. Checking whether $\gamma \models \varphi$ is equivalent to checking whether $[\![\varphi]\!] \vee \neg[\![\gamma]\!]$ is valid.

Let us suppose that $\gamma = \big[x_1 \in A_1\big] \wedge \cdots \wedge \big[x_k \in A_k\big]$. Then the clause $\delta = \big[x_1 \in \mathrm{Dom}(x_1) \setminus A_1\big] \vee \cdots \vee \big[x_k \in \mathrm{Dom}(x_k) \setminus A_k\big]$ is equivalent to $\neg[\![\gamma]\!]$, and can be obtained in polytime [Prop. 7.4.3]. Since FSDD satisfies $\vee$**clC** [Proposition 7.4.6] and **VA** [Prop. 7.4.5], checking whether $[\![\varphi]\!] \vee [\![\delta]\!]$ is valid can be done in polytime, hence FSD satisfies **IM**. □

The following lemma is another corollary of Lemma 4.3.10, detailing the consequences of the fact that FSDD satisfies **IM**.

Lemma 7.5.6. Let L be a Boolean representation language satisfying **IM**. It holds that $\mathtt{L} \not\leqslant_s \mathtt{DNF}^{\mathbb{S}\mathbb{B}}_{\mathcal{B}}$ unless PH collapses at the second level.

Proof. The proof is close to that of d-DNNF $\not\leqslant_s$ DNF [DM02]. Suppose that $\mathtt{L} \leqslant_s \mathtt{DNF}^{\mathbb{S}\mathbb{B}}_{\mathcal{B}}$. Let us consider the compilation function comp that associates with any Boolean CNF Σ, an L-representation of $\neg\Sigma$. By De Morgan's laws, the negation of Σ is a Boolean DNF, so comp is polysize, by hypothesis. We show that we can use $\mathrm{comp}(\Sigma)$ to check in polytime whether $\Sigma \models \gamma$, for any Boolean clause γ.

Checking whether $\Sigma \models \gamma$ is equivalent to checking whether $\neg\Sigma \vee \gamma$ is valid, which is in turn equivalent to checking whether $\neg\gamma \models \neg\Sigma$. Since γ is a clause, $\neg\gamma$ is a term, applying once again the laws of De Morgan.

Since L satisfies **IM**, it is possible to check in polytime whether a given term entails a given L-representation. It is hence possible to check in polytime whether $\neg\gamma \models \mathrm{comp}(\Sigma)$ holds. By construction, the answer is the same as when checking whether $\Sigma \models \gamma$.

All in all, comp is a polysize compilation function allowing us to check in polytime whether a given propositional CNF entails any propositional clause. Since this is impossible unless the polynomial hierarchy collapses at the second level [Lemma 4.3.10], we get that our hypothesis was false modulo the collapse of PH. □

We also include here the proof of the satisfaction of **CE** by FSD, which has similar consequences on succinctness. We need to show support of the conjunction with a term transformation and of the consistency checkinq query beforehand.

Lemma 7.5.7. FSD and FSDD satisfy $\wedge$**tC**.

Proof. We can use Algorithm 4.6 [p. 124], adapted to SDs, to compute the conjunction of an FSD with a term. The procedure is still polynomial, since each intersection of labels is. Furthermore, it maintains exclusive decision: the added top

nodes have only one outgoing edge each, and no value is added in the labels of other edges. □

Lemma 7.5.8. FSD satisfies **MX**.

Proof. We can use Algorithm 4.3 [p. 106], adapted to SDs. The only operation made on labels is the selection of an arbitrary value in a label, so the fact that labels are now tileable integer sets instead of real intervals does not change the complexity. □

Corollary 7.5.9. FSD, FSDD, OSD, OSDD, $\text{OSD}_<$, and $\text{OSDD}_<$ satisfy **MX** and **CO**.

Proof. Since **MX** implies **CO** [Proposition 3.3.2], FSD satisfies **CO**. Thanks to Proposition 1.2.18, all sublanguages of FSD satisfy **MX** and **CO**, hence the result. □

Lemma 7.5.10. FSD satisfies **CE**.

Proof. Let γ be a clause. Checking whether $\varphi \models \gamma$ is equivalent to checking whether $[\![\varphi]\!] \wedge \neg[\![\gamma]\!]$ is inconsistent.

Let us suppose that $\gamma = \big[x_1 \in A_1\big] \vee \cdots \vee \big[x_k \in A_k\big]$. Then the term $\delta = \big[x_1 \in \text{Dom}(x_1) \setminus A_1\big] \wedge \cdots \wedge \big[x_k \in \text{Dom}(x_k) \setminus A_k\big]$ is equivalent to $\neg[\![\gamma]\!]$, and can be obtained in polytime [Prop. 7.4.3]. Since FSD satisfies $\wedge$**tC** [Lem. 7.5.7] and **CO** [Cor. 7.5.9], checking whether $[\![\varphi]\!] \wedge [\![\delta]\!]$ is consistent can be done in polytime; hence FSD satisfies **CE**. □

Corollary 7.5.11. FSDD, OSD, OSDD, $\text{OSD}_<$, and $\text{OSDD}_<$ satisfy **CE**.

Proof. FSD satisfies **CE**, thanks to Lemma 7.5.10. Now, Proposition 1.2.18 implies that all sublanguages of FSD satisfy **CE**, hence the result. □

Negation of SDDs

Lemma 7.5.12. Algorithm 7.3 [p. 163] takes an SDD φ as input. It has the following properties:

- it returns an SDD (that is, it maintains exclusive decision);
- the SDD it returns is equivalent to the negation of $[\![\varphi]\!]$;
- the size of the SDD it returns is linear in $\|\varphi\|$;
- it runs in time polynomial in $\|\varphi\|$;
- it preserves ordering.

Proof. Let φ be an SDD. We denote as $\text{compl}(\varphi)$ the SD returned by the application of Algorithm 7.3 to φ.

- **Exclusive decision:** $\text{compl}(\varphi)$ is obviously exclusive, as we do not add nodes, and only add edges that are disjoint with the other edges coming out of their respective source node.

- **Negation:** We prove this by induction on the number of nodes of φ. For $n \in \mathbb{N}$, let $\mathcal{P}(n)$ be the following proposition: "For any SDD φ containing n nodes, $[\![\text{compl}(\varphi)]\!] \equiv \neg[\![\varphi]\!]$". Obviously enough, thanks to lines 1–4, $\mathcal{P}(n)$ holds for $n \leqslant 1$. Let $n \geqslant 2$, and suppose that $\mathcal{P}(k)$ is true for all $k < n$.

  We consider an SDD φ containing n nodes. Let us denote by R the root of φ, and $\text{Var}(R) = x$. We prove that $[\![\text{compl}(\varphi)]\!] \equiv \neg[\![\varphi]\!]$.

  ($\Rightarrow$) Let $\overrightarrow{y} \in \text{Mod}(\varphi)$; we show that $\overrightarrow{y} \notin \text{Mod}(\text{compl}\,\varphi)$. There must exist an edge $E \in \text{Out}(R)$ such that $\overrightarrow{y}_{|x} \in \text{Lbl}(E)$. Now, we can meet two cases: either R has a corresponding node in $\text{compl}(\varphi)$, or it has not. In the latter case, $\text{compl}(\varphi)$ is the empty SD by construction (line 23), so it has no model. In the former case, let R' be the corresponding node in question. If edge E has no corresponding edge E' among the outgoing edges of R', then clearly, $\overrightarrow{y}$ cannot be a model of $\text{compl}(\varphi)$, since (i) $\text{compl}(\varphi)$ is exclusive, (ii) by construction, $\overrightarrow{y}_{|x} \notin \text{Lbl}(E_{\text{compl}})$, and (iii) no other edge is created. If there is an E', then let φ_D be the subgraph rooted at $D = \text{Dest}(E)$.

  By construction, the subgraph $\text{compl}(\varphi)_{D'}$ rooted at the destination node D' of E' is $\text{compl}(\varphi_D)$. Our induction hypothesis allows us to infer that $\text{compl}(\varphi)_{D'} \equiv \neg[\![\varphi_D]\!]$; now since φ is exclusive, the path compatible with $\overrightarrow{y}$ is unique, so $\overrightarrow{y}$ is a model of φ_D, therefore it is not a model of $\text{compl}(\varphi)_{D'}$. All in all, no path in $\text{compl}(\varphi)$ is compatible with $\overrightarrow{y}$, therefore $\overrightarrow{y} \notin \text{Mod}(\text{compl}(\varphi))$.

  ($\Leftarrow$) Let $\overrightarrow{y} \in \text{Mod}(\text{compl}(\varphi))$; we show that $\overrightarrow{y} \notin \text{Mod}(\varphi)$. Node R' corresponding to R must exist (otherwise $\text{compl}(\varphi)$ would be empty), and there must exist an edge $E' \in \text{Out}(R')$ such that $\overrightarrow{y}_{|x} \in \text{Lbl}(E')$. If E' is the special edge E_{compl}, then by construction, all edges $E \in \text{Out}(R)$ verify $\overrightarrow{y}_{|x} \notin \text{Lbl}(E)$, hence $\overrightarrow{y}$ cannot be a model of φ. Otherwise, E' corresponds to an edge $E \in \text{Out}(R)$, with $\overrightarrow{y}_{|x} \in \text{Lbl}(E)$.

  Let us denote $D = \text{Dest}(E)$ and $D' = \text{Dest}(E')$; by construction, the subgraph $\text{compl}(\varphi)_{D'}$ rooted at D' is $\text{compl}(\varphi_D)$, with φ_D the subgraph rooted at D. Using our induction hypothesis, $\text{compl}(\varphi)_{D'} \equiv \neg[\![\varphi_D]\!]$. Now, since $\text{compl}(\varphi)$ is exclusive, the path compatible with $\overrightarrow{y}$ is unique, and therefore $\overrightarrow{y} \in \text{Mod}(\text{compl}(\varphi)_{D'})$. Consequently, $\overrightarrow{y} \notin \text{Mod}(\varphi_D)$, and since φ is also exclusive, there can be no path in φ compatible with $\overrightarrow{y}$ and leading to the sink: $\overrightarrow{y} \notin \text{Mod}(\varphi)$.

  In all cases, $\overrightarrow{y} \in \text{Mod}(\varphi) \Leftrightarrow \overrightarrow{y} \notin \text{Mod}(\text{compl}(\varphi))$ holds, which implies that $[\![\text{compl}(\varphi)]\!] \equiv \neg[\![\varphi]\!]$. Since both the basis and the inductive step are proven, it shows by induction that $\mathcal{P}(n)$ holds for all $n \in \mathbb{N}$.

- **Space complexity:** For a given node N in φ, let us define its size $\|N\|$ as the sum of the sizes of its outgoing edges: $\|N\| = \sum_{E \in \text{Out}(N)} \|\text{Lbl}(E)\|$. In the worst case (when there is, in the complement, an interval between each couple of intervals from the outgoing edges of N), $\|U\| = 1 + \|N\|$. So the size of E_{compl} for N' is at most $1 + \|N\|$. In the worst case, there is an E_{compl} edge for each node of $\text{compl}(\varphi)$; the total size of these added edges is thus bounded by

$$\sum_{N \in \mathscr{N}_\varphi} \left(1 + \sum_{E \in \text{Out}(N)} \|\text{Lbl}(E)\| \right) = |\mathscr{N}_\varphi| + \sum_{E \in \mathscr{E}_\varphi} \|\text{Lbl}(E)\|,$$

  which is itself bounded by $2\|\varphi\|$. Hence the size of $\text{compl}(\varphi)$ is always linear in $\|\varphi\|$.

- **Time complexity:** Each node is encountered once. To compute U, the hardest step is the sorting of the bounds of intervals labeling edges, hence a complexity in $\text{O}(n \log(n))$ for each node. The whole algorithm is thus polytime.

- **Ordering:** If φ is ordered with respect to $<$, then $\text{compl}(\varphi)$ also is, since no node is added and node labels are not modified. □

Proof of Proposition 7.4.7 [p. 162]. The fact that SDD, OSDD, and $\text{OSDD}_<$ satisfy $\neg$**C** is a direct consequence of Lemma 7.5.12. □

Conditioning

One of the succinctness proofs relies on conditioning being polynomial, so let us prove this here.

Lemma 7.5.13. It is possible to adapt Algorithm 4.2 [p. 103] for it to take an SD φ as input, rather than an IA. It then has the following properties:

- the SD it returns is a representation of the conditioning of φ by $\vec{x}$;
- it is linear in $\|\varphi\|$;
- it preserves exclusive decision;
- it preserves focusingness;
- it preserves ordering.

Proof. Let φ be an SD, $Y \subseteq \mathbb{IZ}$ a set of integer variables, and $\vec{y}$ a Y-assignment; we denote by $\varphi_{|\vec{y}}$ the SD obtained by applying the modified Algorithm 4.2.

- **Conditioning:** By definition of the conditioning, $\vec{z} \in \text{Dom}(\text{Scope}(\varphi) \setminus Y)$ is a model of $[\![\varphi]\!]_{|\vec{y}}$ if and only if $\vec{y} \,.\, \vec{z}$ is a model of φ.

Suppose that $\overrightarrow{y} \cdot \overrightarrow{z}$ is a model of φ. Then there is in φ a path p compatible with $\overrightarrow{z} \cdot \overrightarrow{y}$. By construction, a copy p' of p exists in $\varphi_{|\overrightarrow{y}}$, and $\overrightarrow{z}$ is compatible with p': $\overrightarrow{z}$ is a model of $\varphi_{|\overrightarrow{y}}$.

Suppose that $\overrightarrow{y} \cdot \overrightarrow{z}$ is a not model of φ: any path p in φ contains an edge E such that $(\overrightarrow{y} \cdot \overrightarrow{z})_{|\operatorname{Var}(E)} \notin \operatorname{Lbl}(E)$. Recall that the paths in $\varphi_{|\overrightarrow{y}}$ are the same as those in φ, and let p' be the one corresponding to p. If $\operatorname{Var}(E) \in Y$, the fact that $(\overrightarrow{y} \cdot \overrightarrow{z})_{|\operatorname{Var}(E)} \notin \operatorname{Lbl}(E)$ has led the algorithm to label the corresponding edge in $\varphi_{|\overrightarrow{y}}$ by $\varnothing$. Consequently, $\overrightarrow{z}$ cannot be compatible with this path. If $\operatorname{Var}(E) \notin Y$, $(\overrightarrow{y} \cdot \overrightarrow{z})_{|\operatorname{Var}(E)} \notin \operatorname{Lbl}(E)$ means that $\overrightarrow{z}_{|\operatorname{Var}(E)} \notin \operatorname{Lbl}(E)$: because such edges remain unchanged in $\varphi_{|\overrightarrow{y}}$, $\overrightarrow{z}$ cannot be compatible with p'. Therefore $\overrightarrow{z}$ is not compatible with any path in $\varphi_{|\overrightarrow{y}}$, so $\overrightarrow{z}$ is not a model of $\varphi_{|\overrightarrow{y}}$.

Hence $[\![\varphi_{|\overrightarrow{y}}]\!] = [\![\varphi]\!]_{|\overrightarrow{y}}$.

- **Linearity:** It is trivial, since each node and each edge is scanned at most once.

- **Exclusive decision:** Supposing that φ satisfies exclusive decision, we show that $\varphi_{|\overrightarrow{y}}$ also does. Let us consider a node N in φ, and denote by $N_{|\overrightarrow{y}}$ its corresponding node in $\varphi_{|\overrightarrow{y}}$. If the two nodes are the same, $N_{|\overrightarrow{y}}$ is obviously exclusive. Suppose it has been modified: it is now labeled by $\veebar$. Since the outgoing edges of N are labeled by disjoint sets, at most one of them can include the value $\overrightarrow{y}(\operatorname{Var}(N))$. Consequently, at most one of $N_{|\overrightarrow{y}}$'s outgoing edges can be labeled by a non-empty set. Thus, all nodes of $\varphi_{|\overrightarrow{y}}$, including $\veebar$-nodes, are exclusive: the algorithm maintains exclusive decision.

- **Focusingness:** Supposing that φ is an FSD, we show that $\varphi_{|\overrightarrow{y}}$ is also focusing. The only edges that are modified by the algorithm are those that, in φ, are associated with a variable from Y. In $\varphi_{|\overrightarrow{y}}$, they are all associated with $\veebar$: they hence do not compromise focusingness. As for the other edges in $\varphi_{|\overrightarrow{y}}$, since they *all* remain unchanged, they are still focusing.

  Thus, clearly enough, $\varphi_{|\overrightarrow{y}}$ is focusing with respect to all variables in its scope $\operatorname{Scope}(\varphi) \setminus Y$; hence it is focusing, by definition.

- **Ordering:** Supposing that φ is an $\text{OSD}_<$, it is trivial that $\varphi_{|\overrightarrow{y}}$ is also ordered by $<$, since the order of the variables is not modified (no node is added or moved, and $\veebar$-nodes do not interfere in the ordering). □

Corollary 7.5.14. All the fragments of SD we defined (SD, SDD, FSD, FSDD, OSD, OSDD, $\text{OSD}_<$, $\text{OSDD}_<$) satisfy **CD**.

More Lemmas

The following lemmas are useful in a few proofs.

Lemma 7.5.15. The only reduced SDs φ such that $\text{Scope}(\varphi) = \varnothing$ are the empty and the sink-only SDs.

Proof. The proof is the same as for IAs [see proof of Prop. 4.1.16, p. 113]: if $\text{Scope}(\varphi)$ is empty, there are only $\veebar$-nodes in φ, and they are removed by reduction. □

Lemma 7.5.16. Let $\mathtt{L}$ be any of the sublanguages of $\mathtt{SDD}$ we defined; $\mathtt{L}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{L}_{\mathcal{B}}$ holds.

Proof. Replacing each edge label by its intersection with $\mathbb{B}$ is easy and does not change the interpretation of the SDD. Then, a simple reduction operation removes all edges with an empty label (they are dead) or with a non-singleton label (their parent node is undecisive, given that it cannot have any other outgoing edge since the SDD is exclusive). Thus we obtain in polytime an $\mathtt{SDD}_{\mathcal{B}}^{\$\mathbb{B}}$-representation. Moreover, the procedure maintains focusingness and ordering, hence the result. □

7.5.5 Succinctness

Lemma 7.5.17. Let $<$ be a total strict order on $\mathcal{I}$; it holds that $\mathtt{OSD}_{<} \not\leqslant_s \mathtt{OSDD}$.

Proof. Let X be a set of $2n$ Boolean variables, and let $<^b$ be a total strict order on X. We are going to show that there exists a family Γ_n of Boolean functions over X, and a total strict order $<^a$ on X, such that

- Γ_n has $\mathtt{OSDD}_{<^a}$-representations of size polynomial in n;
- all $\mathtt{OSD}_{<^b}$-representations of Γ_n are of size exponential in n.

Let us consider that X is partitioned into two sets, $Y = \{y_1, \ldots, y_n\}$ and $Z = \{z_1, \ldots, z_n\}$, such that the total strict order $<^b$ on X verifies

$$y_1 <^b y_2 <^b \ldots <^b y_n \qquad <^b \qquad z_1 <^b z_2 <^b \ldots <^b z_n.$$

Let Γ_n be the Boolean function $\bigwedge_{i=1}^{n} \big[y_i = z_i\big]$. We consider the total strict order $<^a$ on X, defined as

$$y_1 <^a z_1 \quad <^a \quad y_2 <^a z_2 \quad <^a \qquad \ldots \qquad <^a \quad y_n <^a z_n.$$

Γ_n has an $\mathtt{OSDD}_{<^a}$-representation the size of which is polynomial in n: each constraint $x_i = y_i$ can be represented as an OSDD with only 3 nodes and 4 edges, and since they do not share variables, they can be combined into a simple read-once and ordered SD. We denote the resulting $\mathtt{OSDD}_{<^a}$-representation as φ_n^a.

Now, we show that the size of any of the $\mathtt{OSD}_{<^b}$-representations of Γ_n is exponential in n. Let φ_n^b be a (reduced) $\mathtt{OSD}_{<^b}$ representation of Γ_n. Consider an edge E in φ_n^b, representing the assignment of some variable y_i (that is, an edge in the first half of the graph). We denote as ω the label of E, as N its source node, and as N' its destination node.

Let us consider a path p from the root the sink of φ_n^b including E. On this path, the value of z_i must be ω. Therefore, a path p' obtained from p by replacing E by another edge between N and N' labeled differently cannot belong to φ_n^b.

For the same reason, any two edges entering a Y-node must be labeled with the same value, and any two paths from the root to N' are either completely disjoint or represent the same assignment of $y_1, \ldots, y_i$. Each of the possible assignments of variables from Y is thus represented by (at least) one path pointing to a distinct z_1-node. Since there are 2^n assignments of variables from Y, there are at least 2^n nodes labeled z_1. Hence $\|\varphi_n^b\| \geqslant 2^n$: all $\mathtt{OSD}_{<^b}$-representations of Γ_n are of size exponential in n.

All in all, for any total strict order $<$ on $\mathcal{I}$, there exists a family Γ_n of Boolean functions that have polynomial OSDD-representations but no polynomial representation in $\mathtt{OSD}_<$: $\mathtt{OSD}_< \not\leqslant_s \mathtt{OSDD}$. □

Corollary 7.5.18. Let $<$ be a total strict order on $\mathcal{I}$. For all languages L among SD, SDD, FSD, FSDD, OSD, and OSDD, it holds that $\mathtt{OSD}_< \not\leqslant_s \mathtt{L}$ and $\mathtt{OSDD}_< \not\leqslant_s \mathtt{L}$.

Proof. This comes from the fact that OSDD is a sublanguage of all languages L [Prop. 7.2.2]. If there existed L such that $\mathtt{OSD}_< \leqslant_s \mathtt{L}$, we could infer $\mathtt{OSD}_< \leqslant_s \mathtt{L} \leqslant_s \mathtt{OSDD}$, and thus $\mathtt{OSD}_< \leqslant_s \mathtt{OSDD}$, that we proved false. The second result is in turn a corollary of the first one, since $\mathtt{OSDD}_< \subseteq \mathtt{OSD}_<$ [Prop. 7.2.2]. □

Lemma 7.5.19. Let us consider an integer $n \geqslant 2$, and $X = \{y, x_1, \ldots, x_n\}$ a set of $n+1$ variables of domain $[1, \ldots, n]$.

There exists a family Σ_n of Boolean functions over X, such that:

- Σ_n has an FSDD-representation of size polynomial in n;
- for all total strict order $<$ on X ending with y, all $\mathtt{OSDD}_<$-representations of Σ_n are of size exponential in n.

Proof. Let Σ_n be the Boolean function $\bigwedge_{i=1}^n \big[y \neq x_i\big]$. Building an FSDD representing Σ_n is easy: the root node is an y-node, with n outgoing edges, each one labeled with a different value from $[1, \ldots, n]$. The k-labeled edge points to a subgraph φ_k, that represents the Boolean function $\bigwedge_{i=1}^n [x_i \neq k]$. This function is equivalent to

$$\bigwedge_{i=1}^{n} \Big[x_i \in [1, \ldots, k-1] \cup [k+1, \ldots, n]\Big],$$

which is a term, representable as a chain of nodes, the order not being important. This automaton is clearly an FSDD (it is indeed read-once), and its size is bounded by $n \cdot (1 + 2n)$; it is thus polynomial in n.

Now, let $<$ be a total strict order on X that ends with y. Let us consider an $\mathtt{OSDD}_<$-representation ψ of Σ_n. In all paths of ψ, the last node is labeled y (there cannot be a path not mentioning y). We show that there must be at least $2^n - 2$ nodes labeled y.

Let S be a strict, nonempty subset of $[1, \ldots, n]$. Let $\vec{x}_S$ be an assignment of the variables from $X \setminus \{y\}$, such that no variable is assigned to a value outside of S, and each value in S is assigned to at least one variable. Clearly enough, for any $\{y\}$-assignment $\vec{y}$ such that $\vec{y} \notin S$, $\vec{x}_S \,.\, \vec{y}$ is a model of Σ_n.

This means that there exists a least one path p from the root to the sink of ψ, such that p is compatible with $\vec{x}_S$. We show that this path is unique up to the last edge (the one pertaining to y). Let p' be a path compatible with $\vec{x}_S$, such that p and p' are identical up to an x-node N. Since ψ satisfies exclusive decision, the edges going out of N have disjoint labels. So if one of these edges, E, belongs to p, the others cannot be compatible with $\vec{x}_S$. Hence E belongs to p and p'. By induction, p and p' are identical up to the first y-node (recall that $y \notin \text{Scope}(\vec{x}_S)$).

Thus, all paths compatible with $\vec{x}_S$ go through a same y-node, that we call N. Thus, the edges going out of N must cover all values of y that are consistent with $\vec{x}_S$: if such a value ω misses, there can be no path compatible with $\vec{x}_S \,.\, \vec{y}$ with $\vec{y}_{|y} = \omega$, so $\vec{x}_S \,.\, \vec{y}$ is not a model of ψ, which is absurd. Hence the union of the labels of all edges going out of N must be exactly $[1, \ldots, n] \setminus S$.

This reasoning is true for any of the $2^n - 2$ possible sets S: we need a different y-node for each of these sets. Consequently, we need at least $2^n - 2$ different y-nodes. Hence $\|\psi\| \geqslant 2^n - 2$: any $\texttt{OSDD}_<$-representation of Σ_n is of size exponential in n. □

Lemma 7.5.20. $\texttt{OSDD} \not\leqslant_s \texttt{FSDD}$ holds.

Proof. Let $n \in [2, \ldots, \infty]$. Let Z be a set of $n+1$ variables $\{z_0, z_1, \ldots, z_n\}$ of domain $[1, \ldots, n]$. For each $k \in [1, \ldots, n]$, we denote as Σ_n^k, the Boolean function defined in Lemma 7.5.19 with z_k playing the role of y and the other variables from Z playing the role of the x_i. For all k, let φ_k be an FSDD-representation of Σ_n^k such that $\|\varphi_k\|$ is polynomial in n (we know there exists one, thanks to the lemma).

Let t be another integer variable of domain $[1, \ldots, n]$. We consider the Boolean function τ_n defined on $Z \cup \{t\}$ by the following formula:

$$\tau_n \equiv \bigvee_{k=1}^{n} \left([t = k] \wedge \Sigma_n^k \right).$$

τ_n has an FSDD-representation Φ of size polynomial in n. We just have to fuse the sinks of the φ_k, and add a root node, labeled t, with n outgoing edges, each one labeled with a different $k \in [1, \ldots, n]$ and pointing to the root of φ_k. This automaton Φ clearly represents τ_n. Furthermore, Φ is an FSDD: each φ_k is an FSDD, t is not mentioned in any of the φ_k and its outgoing edges are disjoint, and no $\veebar$-node is added. Last, $\|\Phi\|$ is polynomial in n: it consists of a root node with n outgoing edges of size 1, and of n subgraphs of size polynomial in n.

Now, we prove that τ_n has only exponential OSDD-representations. Let us suppose that there exists a total strict order $<$ of $Z \cup \{t\}$, and an $\texttt{OSDD}_<$-representation ψ of τ_n such that $\|\psi\|$ is polynomial in n. Let z_k be the greatest Z variable in $<$. Since $\texttt{OSDD}_<$ supports **CD** [Cor. 7.5.14], we can obtain a polysize $\texttt{OSDD}_<$-representation

ψ_k of $\tau_{n|t=k}$, which is equivalent to Σ_n^k. Thus there exists a total strict order $<$ of Z, ending with z_k, for which there exists an $\mathtt{OSDD}_<$-representation of Σ_n^k of size polynomial in n. However, Lemma 7.5.19 states that it is impossible. Hence, there can be no total strict order $<$ of $Z \cup \{t\}$ such that τ_n has an $\mathtt{OSDD}_<$-representation of size polynomial in n. This proves that τ_n has no polynomial OSDD-representation. □

Corollary 7.5.21. It holds that

$$\mathtt{OSDD} \not\leqslant_s \begin{matrix} \mathtt{SD}, \\ \mathtt{SDD}, \\ \mathtt{FSD}. \end{matrix}$$

Proof. This comes from the fact that FSDD is a sublanguage of SD, SDD, and FSD [Prop. 7.2.2]. □

Lemma 7.5.22. Let $<$ be a total strict order on $\mathcal{I}$; it holds that $\mathtt{OSDD} \not\leqslant_s \mathtt{OSD}_<$.

Proof. Assuming $\mathtt{OSDD} \leqslant_s \mathtt{OSD}_<$, we can infer $\mathtt{OSDD} \leqslant_s \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$ [direct consequence of 7.4.4]. Then $\mathtt{OSDD}_{\mathcal{B}} \leqslant_s \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$, thanks to Proposition 1.2.16. Because of Lemma 7.5.16, we get $\mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{OSDD}_{\mathcal{B}}$, and finally $\mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_s \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$.

Now, $\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_s \mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}}$ [Prop. 7.2.8]. Hence $\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_s \mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_s \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$, and thus $\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_s \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$—which is false [Th. 1.4.17]. This proves by contradiction that $\mathtt{OSDD} \not\leqslant_s \mathtt{OSD}_<$. □

Corollary 7.5.23. Let $<$ be a total strict order on $\mathcal{I}$. It holds that $\mathtt{OSDD} \not\leqslant_s \mathtt{OSD}$ and that $\mathtt{OSDD}_< \not\leqslant_s \mathtt{OSD}_<$.

Proof. It respectively comes from the facts that $\mathtt{OSD}_< \subseteq \mathtt{OSD}$ and that $\mathtt{OSDD}_< \subseteq \mathtt{OSDD}$ [Proposition 7.2.2]. □

Lemma 7.5.24. $\mathtt{OSD} \not\leqslant_s \mathtt{SDD}$ holds.

Proof. Let $n \in \mathbb{N}^*$. Let Z_n be a set of n variables $\{z_1, \ldots, z_n\}$ of domain $[1, \ldots, n]$. Let Σ_n be the Boolean function defined as $\Sigma_n \equiv \text{alldiff}(z_1, \ldots, z_n)$, or equivalently:

$$\Sigma_n \equiv \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{i-1} \big[z_i \neq z_j\big].$$

There exists an SDD of size polynomial in n representing Σ_n. Indeed, each constraint $\big[z_i \neq z_j\big]$ can be represented as an SDD of $n(n-1)$ singleton-labeled edges (n edges for z_i, and for each one, a z_j-node with $n-1$ outgoing edges). They can be combined into a polysize SDD, since SDD satisfies $\wedge\mathbf{C}$ [Prop. 7.4.2].

Now, we prove that all OSDs representing Σ_n are of size exponential in n. Let φ be an OSD-representation of Σ_n; we consider, without loss of generality, that its variable order $<$ verifies $z_1 < \cdots < z_n$.

Let $i \in [1, \ldots, n]$; consider two distinct subsets S and S' of $[1, \ldots, n]$ such that $|S| = |S'| = i$. Let $\omega \in [1, \ldots, n]$ such that $\omega \in S$ and $\omega \notin S'$. Let $\vec{z}_S$ and $\vec{z}_{S'}$ be two $\{z_1, \ldots, z_i\}$-assignments that cover all values in S and S', respectively. We consider two paths p_S and $p_{S'}$ from the root to the sink of φ, compatible with $\vec{z}_S$ and $\vec{z}_{S'}$, respectively. Suppose they go through a same z_i-node N; p_S assigns ω to some variable before encountering N, and $p_{S'}$ assigns ω to some variable after having encountered N. Then, the path obtained by joining the first part of p_S (from the root to N) and the second part of $p_{S'}$ (from N to the sink) is a consistent path assigning ω to two different variables. Since such a path violates the "alldifferent" constraint, it is impossible, so p_S and $p_{S'}$ must go through two different z_i-nodes.

Hence, there are at least one z_i-node for each $S \subseteq [1, \ldots, n]$ of cardinal i; since there are $\binom{n}{i}$ such subsets, there are at least $\binom{n}{i}$ z_i-nodes. The number of nodes in φ is thus greater than $\sum_{i=1}^{n} \binom{n}{i} = 2^n$.

All in all, family Σ_n has polynomial SDD-representations, but only exponential OSD-representations, which proves $\texttt{OSD} \not\leqslant_s \texttt{SDD}$. □

Corollary 7.5.25. $\texttt{OSD} \not\leqslant_s \texttt{SD}$ holds.

Proof. Straight from the fact that SDD is a sublanguage of SD [Prop. 7.2.2]. □

Lemma 7.5.26. Let $<$ be a total strict order on $\mathcal{I}$. It holds that $\texttt{FSDD} \not\leqslant_s \texttt{OSD}_<$, unless PH collapses at the second level.

Proof. We know that FSDD supports **IM** [Lemma 7.5.5]. Thanks to Lemma 7.5.6, we get that FSDD cannot be at least as succinct as $\texttt{DNF}_{\mathcal{B}}^{\mathbb{SB}}$ unless PH collapses at the second level. However, $\texttt{OSD}_< \leqslant_s \texttt{DNF}_{\mathcal{B}}^{\mathbb{SB}}$ [direct consequence of 7.4.4]. Therefore, $\texttt{FSDD} \not\leqslant_s \texttt{OSD}_<$ holds unless PH collapses. □

Corollary 7.5.27. The following propositions hold, unless PH collapses at the second level:

$$\texttt{FSDD} \not\leqslant_s \begin{array}{l} \texttt{OSD}, \\ \texttt{FSD}, \\ \texttt{SD}. \end{array}$$

Proof. This is inferred from the fact that $\texttt{OSD}_<$ is a sublanguage of OSD, FSD, and SD [Proposition 7.2.2]. □

Lemma 7.5.28. $\texttt{FSD} \not\leqslant_s \texttt{SDD}$ holds, unless PH collapses at the second level.

Proof. We know that FSD supports **CE** [Lemma 7.5.10]. Thanks to Lemma 4.3.11, we get that FSD cannot be at least as succinct as $\texttt{CNF}_{\mathcal{B}}^{\mathbb{SB}}$ unless PH collapses at the second level.

Now, we also know that $\texttt{SDD} \leqslant_s \texttt{CNF}_{\mathcal{B}}^{\mathbb{SB}}$ holds [direct consequence of 7.4.4]. Hence, if $\texttt{FSD} \leqslant_s \texttt{SDD}$ held, it would be true that $\texttt{FSD} \leqslant_s \texttt{CNF}_{\mathcal{B}}^{\mathbb{SB}}$; yet we just proved that it was impossible unless PH collapses, so we get the result. □

L	SD	SDD	FSD	FSDD
SD	$\leqslant_s$	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]
SDD	?	$\leqslant_s$	?	$\leqslant_s$ [Prop. 7.2.2]
FSD	$\not\leqslant_s^*$ [Cor. 7.5.29]	$\not\leqslant_s^*$ [Lem. 7.5.28]	$\leqslant_s$	$\leqslant_s$ [Prop. 7.2.2]
FSDD	$\not\leqslant_s^*$ [Cor. 7.5.27]	$\not\leqslant_s^*$ [Cor. 7.5.29]	$\not\leqslant_s^*$ [Cor. 7.5.27]	$\leqslant_s$
OSD	$\not\leqslant_s$ [Cor. 7.5.25]	$\not\leqslant_s$ [Lem. 7.5.24]	?	?
OSDD	$\not\leqslant_s$ [Cor. 7.5.21]	$\not\leqslant_s$ [Cor. 7.5.21]	$\not\leqslant_s$ [Cor. 7.5.21]	$\not\leqslant_s$ [Lem. 7.5.20]
$\text{OSD}_<$	$\not\leqslant_s$ [Cor. 7.5.18]	$\not\leqslant_s$ [Cor. 7.5.18]	$\not\leqslant_s$ [Cor. 7.5.18]	$\not\leqslant_s$ [Cor. 7.5.18]
$\text{OSDD}_<$	$\not\leqslant_s$ [Cor. 7.5.18]	$\not\leqslant_s$ [Cor. 7.5.18]	$\not\leqslant_s$ [Cor. 7.5.18]	$\not\leqslant_s$ [Cor. 7.5.18]

L	OSD	OSDD	$\text{OSD}_<$	$\text{OSDD}_<$
SD	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]
SDD	?	$\leqslant_s$ [Prop. 7.2.2]	?	$\leqslant_s$ [Prop. 7.2.2]
FSD	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]
FSDD	$\not\leqslant_s^*$ [Cor. 7.5.27]	$\leqslant_s$ [Prop. 7.2.2]	$\not\leqslant_s^*$ [Lem. 7.5.26]	$\leqslant_s$ [Prop. 7.2.2]
OSD	$\leqslant_s$	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]	$\leqslant_s$ [Prop. 7.2.2]
OSDD	$\not\leqslant_s$ [Cor. 7.5.23]	$\leqslant_s$	$\not\leqslant_s$ [Lem. 7.5.22]	$\leqslant_s$ [Prop. 7.2.2]
$\text{OSD}_<$	$\not\leqslant_s$ [Cor. 7.5.18]	$\not\leqslant_s$ [Lem. 7.5.17]	$\leqslant_s$	$\leqslant_s$ [Prop. 7.2.2]
$\text{OSDD}_<$	$\not\leqslant_s$ [Cor. 7.5.18]	$\not\leqslant_s$ [Cor. 7.5.18]	$\not\leqslant_s$ [Cor. 7.5.23]	$\leqslant_s$

Table 7.4: Results about succinctness, with indication of the proposition corresponding to each result. A star (*) indicates a result that holds unless the polynomial hierarchy PH collapses.

Corollary 7.5.29. Unless PH collapses at the second level, it holds that FSDD $\not\leqslant_s$ SDD and FSD $\not\leqslant_s$ SD.

Proof. It comes from the facts that FSDD is a sublanguage of FSD, and that SDD is a sublanguage of SD, respectively [Prop. 7.2.2]. (We already proved that FSDD $\not\leqslant_s$ SD in Corollary 7.5.27.) □

Proof of Theorem 7.4.8 [p. 162]. All results of this theorem come from proposition, lemmas, and corollaries. Table 7.4 associates with each claim its corresponding proposition. □

7.5.6 Queries and Transformations

Various Queries

In this section, we prove most of the remaining results about queries. Results about **SE** and **EQ** are proven in Section 7.5.6.5 [p. 191].

Lemma 7.5.30. SD satisfies **MC**.

Proof. First, condition the SD by the X-assignment that is to be checked, say $\vec{x}$. The resulting SD contains only $\veebar$-nodes, so the reduction procedure removes them all [Lemma 7.5.15]. We get either the empty SD (then $\vec{x}$ is not a model) or the sink-only SD (then $\vec{x}$ is a model). Since conditioning [Cor. 7.5.14] and reduction [Prop. 7.2.3] are polytime, SD satisfies **MC**. □

Corollary 7.5.31. SDD, FSD, FSDD, OSD, $\text{OSD}_<$, OSDD, and $\text{OSDD}_<$ satisfy **MC**.

Proof. All of them are fragments of SD, and as such, satisfy the same set of queries [Prop. 1.2.18]. □

Lemma 7.5.32. SD and SDD do not satisfy **CO**, **VA**, **CE**, **IM**, **EQ**, **SE**, **MX**, **CX**, **CT**, or **ME**, unless $\mathsf{P} = \mathsf{NP}$.

Proof. We know that $\text{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$ does not satisfy **CO**, **VA**, **CE**, **IM**, **EQ**, **SE**, **CT**, or **ME**, unless $\mathsf{P} = \mathsf{NP}$ [Th. 1.4.18]. Since **MX** and **CX** imply **CO** [Proposition 3.3.2], $\text{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$ does not satisfy any of the queries mentioned in the lemma unless $\mathsf{P} = \mathsf{NP}$.

Now, $\text{BDD}_{\mathcal{B}}^{\$\mathbb{B}} = \text{SDD}_{\mathcal{B}}^{\$\mathbb{B}}$ [Proposition 7.2.8], so $\text{BDD}_{\mathcal{B}}^{\$\mathbb{B}} \subseteq \text{SDD}$, and $\text{BDD}_{\mathcal{B}}^{\$\mathbb{B}} \subseteq \text{SD}$, so thanks to Proposition 1.2.18, we know that SD and SDD cannot support any query that $\text{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$ does not support. As a result, we get the lemma. □

Lemma 7.5.33. $\text{OSD}_<$ does not satisfy **VA**, **IM**, **EQ**, **SE**, or **CT**, unless $\mathsf{P} = \mathsf{NP}$.

Proof. This is a direct consequence of Proposition 7.4.4, Theorem 1.4.18, and Proposition 1.2.18: $\text{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$ does not satisfy these queries unless $\mathsf{P} = \mathsf{NP}$, and $\text{OSD}_< \leqslant_p \text{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$. □

Corollary 7.5.34. OSD and FSD do not satisfy **VA**, **IM**, **EQ**, **SE**, or **CT**, unless $\mathsf{P} = \mathsf{NP}$.

Proof. We apply once again the property of sublanguages and queries provided in Proposition 1.2.18, given that $\text{OSD}_< \subseteq \text{OSD} \subseteq \text{FSD}$ [Prop. 7.2.2]. □

Lemma 7.5.35. OSDD and $\text{OSDD}_<$ satisfy **CT**.

Proof. Since OSDDs are read-once and exclusive, given an OSDD φ of variable order $<$, we simply have to associate with the sink the number $n_{\text{Sink}(\varphi)} = 1$, then traverse the graph from the sink to the root, associating with each edge E the number (possibly infinite)

$$n_E = |\text{Lbl}(E)| \cdot \prod_{x_s < x < x_d} |\text{Dom}(x)|,$$

where $x_s = \text{Var}(\text{Src}(E))$ and $x_d = \text{Var}(\text{Dest}(E))$, and with each internal node N the number

$$n_N = \sum_{E \in \text{Out}(N)} n_E.$$

The number of models is then $n_{\text{Root}(\varphi)}$. This process being polynomial in $\|\varphi\|$, OSDD and $\text{OSDD}_<$ satisfy **CT**. □

Lemma 7.5.36. Let $<$ be a total strict order on $\mathcal{I}$; OSDD and $\text{OSDD}_<$ satisfy **VA** and **IM**.

Proof. FSDD satisfies **VA** [Prop. 7.4.5] and **IM** [Lem. 7.5.5]. Now, applying Proposition 1.2.18, all sublanguages of FSDD must satisfy **VA** and **IM**, and we know that $\text{OSDD}_< \subseteq \text{OSDD} \subseteq \text{FSDD}$ [Prop. 7.2.2]. □

Lemma 7.5.37. FSD satisfies **CX**.

Proof. We use Algorithm 4.4 [p. 107], adapted to SDs. Instead of making unions of intervals, it now makes unions of unions of intervals; the overall complexity is the same (recall that the size of the SD depends on the number of intervals in its labels). □

Corollary 7.5.38. FSDD, OSD, OSDD, $\text{OSD}_<$, and $\text{OSDD}_<$ satisfy **CX**.

Proof. Thanks to Proposition 1.2.18, all sublanguages of FSD satisfy **CX**, hence the result. □

Lemma 7.5.39. Let φ be an SD, and $x \in \text{Scope}(\varphi)$. We can obtain in polytime a mesh of x in φ.

Proof. The proof is exactly the same as for IAs [Lemma 4.4.4]. We just have to gather the bounds of all intervals in φ pertaining to x, and create a partition of $\text{Dom}(x)$ based on this set of bounds. □

The fact that the labels are union of intervals does not change the properties of meshes. In particular, it still holds that for any two values m and m' in a given element of a mesh of x in φ,

$$[\![\varphi]\!]_{|x=m} = [\![\varphi]\!]_{|x=m'},$$

as stated in Lemma 4.4.5.

Lemma 7.5.40. FSD satisfies **ME**.

Proof. We use the same mechanism as for **ME** on FIAs. We have to use a mesh, because domain cardinals are not taken into account in the size of an SD: integer variables have an integer interval domain, and these structures have a characteristic size of 1, whatever their cardinal may be. Hence, we cannot branch on all values of each variable—the tree would be potentially exponential in the size of the SD. □

Corollary 7.5.41. FSDD, OSD, OSDD, $\text{OSD}_<$, and $\text{OSDD}_<$ satisfy **ME**.

Proof. Thanks to Proposition 1.2.18, all sublanguages of FSD satisfy **ME**, hence the result. □

Negation

Lemma 7.5.42. Let L be a representation language; if $\mathtt{L} \leqslant_p \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$ and L satisfies **CO**, L cannot satisfy $\neg$**C** unless $\mathsf{P} = \mathsf{NP}$.

Proof. If L satisfied $\neg$**C**, $\mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ would satisfy **CO**. Indeed, by negating a Boolean CNF, one obtains a DNF, which is, by hypothesis, polynomially translatable into an L-representation. Computing its negation, we would obtain an L-representation equivalent to the original CNF—and we can decide in polytime whether it is consistent, since by hypothesis L satisfies **CO**. Hence, since $\mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ does not satisfy **CO** unless $\mathsf{P} = \mathsf{NP}$ [Th. 1.4.18], L does not satisfy $\neg$**C** unless $\mathsf{P} = \mathsf{NP}$. □

Corollary 7.5.43. $\mathtt{OSD}_<$, OSD, and FSD do not satisfy $\neg$**C** unless $\mathsf{P} = \mathsf{NP}$.

Proof. We know that $\mathtt{OSD}_< \leqslant_p \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$ [Prop. 7.4.4], $\mathtt{OSD}_< \subseteq \mathtt{OSD} \subseteq \mathtt{FSD}$ [Prop. 7.2.2], and these three languages satisfy **CO** [Cor. 7.5.9]. □

Conjunction

Lemma 7.5.44. OSDD and $\mathtt{OSDD}_<$ do not satisfy $\wedge$**C**.

Proof. If OSDD satisfied $\wedge$**C**, we could transform in polytime any CNF into an OSDD, since clauses can be turned into OSDDs in polytime [Prop. 7.4.3]. Thus, we would have $\mathtt{OSDD} \leqslant_p \mathtt{CNF}_{\mathcal{I}}^{\mathbb{T}^{\mathbb{Z}}}$. Using Proposition 1.2.16, this would lead to $\mathtt{OSDD}_{\mathcal{B}} \leqslant_s \mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$. Using Lemma 7.5.16, $\mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{OSDD}_{\mathcal{B}}$. All in all, we get that if OSDD satisfied $\wedge$**C**, this would imply that $\mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$. Now, since $\mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \sim_p \mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$, this would imply that $\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$; yet it is not true [Th. 1.4.18].

The same proof can be used for $\mathtt{OSDD}_<$. □

Lemma 7.5.45. FSD, FSDD, OSD, and OSDD do not satisfy $\wedge$**BC** unless $\mathsf{P} = \mathsf{NP}$.

Proof. $\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ is polynomially translatable into any of these languages [consequence of Prop 7.2.8], that all satisfy **CO** [Cor. 7.5.9]; because of Lemma 4.4.3, they cannot satisfy $\wedge$**BC** unless $\mathsf{P} = \mathsf{NP}$. □

Corollary 7.5.46. FSD, FSDD, and OSD do not satisfy $\wedge$**C** unless $\mathsf{P} = \mathsf{NP}$. (We already have a stronger result on OSDD, see Lemma 7.5.44.)

Lemma 7.5.47. $\mathtt{OSD}_<$ does not satisfy $\wedge$**C** unless $\mathsf{P} = \mathsf{NP}$.

Proof. If $\mathtt{OSD}_<$ satisfied $\wedge$**C**, we could transform in polytime any Boolean CNF into an $\mathtt{OSD}_<$-representation, since clauses can be turned into OSDs in polytime [Prop. 7.4.3]. Yet, $\mathtt{OSD}_<$ supports **CO** [Cor. 7.5.9], and $\mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ does not support **CO** unless $\mathsf{P} = \mathsf{NP}$ [Th. 1.4.18]. Hence, $\mathtt{OSD}_<$ cannot satisfy $\wedge$**C** unless $\mathsf{P} = \mathsf{NP}$. □

Lemma 7.5.48. $\mathtt{OSD}_<$ and $\mathtt{OSDD}_<$ satisfy $\wedge$**BC**.

Proof. We can use Algorithm 7.6, adapted from the classical OBDD one [Bry86]. It applies on non-empty OSDs of a same variable order (if one of the OSDs is empty, it is trivial to compute the conjunction). A cache is maintained to avoid computing twice the same couple of nodes, thus `conjunct_step` is not called more than $|\mathscr{N}_{\varphi_1}| \cdot |\mathscr{N}_{\varphi_2}|$ times.

Before `conjunct_step` is called on the roots of φ_1 and φ_2, the cache is initialized, and a mesh of both φ_1 and φ_2 is computed for each variable $x \in \text{Scope}(\varphi_1) \cup \text{Scope}(\varphi_2)$: it is denoted as $\mathcal{M}^x = \{M_1^x, \ldots, M_n^x\}$. Each element M_i^x of $\mathcal{M}^x$ is associated with an "index value" $m_i \in M_i^x$. The set of index values for x is denoted as $\text{Indexes}(x)$.

For each execution of `conjunct_step`, if variables are different, each outgoing edge of the top node is explored once, and if variables are equal, each index value is treated once—and the number of index values is linear in $\|\varphi_1\| + \|\varphi_2\|$.

Hence, the overall procedure is polynomial in $\|\varphi_1\|$ and $\|\varphi_2\|$. When both φ_1 and φ_2 satisfy exclusive decision, the resulting SD ψ also satisfies exclusive decision: the "new" nodes that are added when variables are equal (line 17) have only exclusive outgoing edges (they are all labeled by different elements of the same mesh). □

Lemma 7.5.49. SD, SDD, $\text{OSD}_<$, and $\text{OSDD}_<$ satisfy $\wedge$**tC**.

Proof. This is trivial, since they all satisfy $\wedge$**BC** [Prop. 7.4.2 and Lemma 7.5.48], and translating a term into any of these languages is polynomial [Prop. 7.4.3]. □

Lemma 7.5.50. OSD and OSDD satisfy $\wedge$**tC**.

Proof. Let φ be an OSD, and γ a term. Obviously, there exists a total strict order $<$ such that φ is an $\text{OSD}_<$-representation. Since $\text{OSD}_<$ satisfies $\wedge$**tC** [Lemma 7.5.49], we can obtain in polytime an OSD representing the conjunction of $[\![\varphi]\!]$ and $[\![\gamma]\!]$. Hence, OSD satisfies $\wedge$**tC**.

The proof is similar for OSDD since $\text{OSDD}_<$ also satisfies $\wedge$**tC**. □

Disjunction

Lemma 7.5.51. SDD satisfies $\vee$**C** and $\vee$**BC**.

Proof. This holds since SDD satisfies $\wedge$**C** and $\neg$**C**. Indeed by De Morgan's laws, to obtain the disjunction of $\varphi_1, \ldots, \varphi_n$, we can compute the negation of each disjunct, compute their conjunction, then compute the negation of the result. SDD satisfies $\neg$**C** [Prop. 7.4.7]: there exists a polynomial P such that $\forall \varphi_i, \|\neg\varphi_i\| \leqslant P(\|\varphi_i\|)$ (denoting $\neg\varphi_i$ the SDD representing $\neg[\![\varphi_i]\!]$). SDD satisfies $\wedge$**C** [Prop. 7.4.2]: there exists a polynomial P' and an SDD-representation ψ of $\bigwedge_{i=1}^n [\![\neg\varphi_i]\!]$ such that $\|\psi\| \leqslant P'(\sum_{i=1}^n P(\|\varphi_i\|))$. The size of the disjunction is thus bounded by the number $P(P'(\sum_{i=1}^n P(\|\varphi_i\|)))$, which is polynomial in each of the $\|\varphi_i\|$. □

Algorithm 7.6 `conjunct_step`(φ_1, φ_2): when given two (nonempty) $\texttt{OSD}_<$-representations φ_1 and φ_2, returns an $\texttt{OSD}_<$-representation of $[\![\varphi_1]\!] \wedge [\![\varphi_2]\!]$.

1: let $N_1 \coloneqq \text{Root}(\varphi_1)$
2: let $N_2 \coloneqq \text{Root}(\varphi_2)$
3: **if** the cache contains the key $\langle N_1, N_2 \rangle$ **then**
4: **return** the OSD corresponding to this key in the cache
5: **if** $\text{Out}(N_1) = \varnothing$ **then** *// the sink of φ_1 is reached*
6: let $\psi \coloneqq \varphi_2$
7: **else if** $\text{Out}(N_2) = \varnothing$ **then** *// the sink of φ_2 is reached*
8: let $\psi \coloneqq \varphi_1$
9: **else if** $\text{Var}(N_1) \neq \text{Var}(N_2)$ **then**
10: $N_i = \text{Argmin}_<(\text{Var}(N_1), \text{Var}(N_2))$,
11: $N_j = \text{Argmax}_<(\text{Var}(N_1), \text{Var}(N_2))$
12: create a node N_i' labeled $\text{Var}(N_i)$
13: **for each** $E \in \text{Out}(N_i)$ **do**
14: let $\psi_E \coloneqq \texttt{conjunct\_step}(\text{Dest}(E), N_j)$
15: add an edge going out of N_i', labeled $\text{Lbl}(e)$ and pointing to the root of ψ_E
16: let ψ be the OSD rooted at N_i'
17: **else** *// $\text{Var}(N_1) = \text{Var}(N_2)$*
18: let $x \coloneqq \text{Var}(N_1)$
19: create a node N' labeled by x
20: **for each** $m_i \in \text{Indexes}(x)$, $E_1 \in \text{Out}(N_1)$, $E_2 \in \text{Out}(N_2)$ **do**
21: **if** $m_i \in \text{Lbl}(E_1)$ **and** $m_i \in \text{Lbl}(E_2)$ **then**
22: let $\psi_i \coloneqq \texttt{conjunct\_step}(\text{Dest}(E_1), \text{Dest}(E_2))$
23: add an edge going out of N', labeled by M_i^x and pointing to the root of ψ_i
24: let ψ be the OSD rooted at N'
25: add ψ to the cache, at key $\langle N_1, N_2 \rangle$
26: **return** ψ

Lemma 7.5.52. OSDD and $\texttt{OSDD}_<$ do not satisfy $\vee$**C**.

Proof. Since $\texttt{OSDD} \leqslant_p \texttt{term}_{\mathcal{I}}^{\mathbb{TZ}}$ [Prop. 7.4.3], if OSDD satisfied $\vee$**C**, we could transform in polytime any DNF into an OSDD, and in particular we would get $\texttt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \texttt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$ [Prop. 1.2.16 and Lemma 7.5.16]. Now, since $\texttt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \sim_p \texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ [Prop. 7.2.8], this would imply that $\texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \texttt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$; yet it is not true [Th. 1.4.18].

The same proof can be used for $\texttt{OSDD}_<$. □

Lemma 7.5.53. Let L be a Boolean representation language; if $\texttt{L} \leqslant_p \texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ and L satisfies **VA**, it cannot satisfy $\vee$**BC** unless $\mathsf{P} = \mathsf{NP}$.

Proof. The proof is uses the same mechanism as the one of Lemma 4.4.3, and is also adapted from Darwiche and Marquis [DM02]. Let φ_1 and φ_2 be two $\texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$-representations (not necessarily of the same order); checking whether their conjunction is consistent is equivalent to checking whether $\neg[\![\varphi_1]\!] \vee \neg[\![\varphi_2]\!]$ is valid. $\texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ satisfies $\neg$**C** [Th. 1.4.18], so we can obtain in polytime two OBDDs representing $\neg[\![\varphi]\!]_1$ and $\neg[\![\varphi]\!]_2$, that we can translate in polytime into L-representations, since $\texttt{L} \leqslant_p \texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$. If L satisfied $\vee$**BC**, we could then obtain in polytime an L-representation of $\neg[\![\varphi_1]\!] \vee \neg[\![\varphi_2]\!]$, of which we could easily check the validity (since L satisfies **VA**).

All in all, we would have a polytime algorithm deciding whether the conjunction of two OBDDs (the variable orderings being possibly different in each OBDD) is consistent; yet, this problem is NP-complete, as shown by Meinel and Theobald [MT98, Lemma 8.14]. Therefore L does not support $\vee$**BC** unless $\mathsf{P} = \mathsf{NP}$. □

Corollary 7.5.54. FSDD and OSDD do not satisfy $\vee$**BC** unless $\mathsf{P} = \mathsf{NP}$. FSDD does not satisfy $\vee$**C** unless $\mathsf{P} = \mathsf{NP}$.

Proof. $\texttt{FSDD} \leqslant_p \texttt{OSDD} \leqslant_p \texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ [Props 7.2.2 and 7.2.7], and both FSDD and OSDD satisfy **VA** [Prop. 7.4.5 and Lemma 7.5.36], so they cannot satisfy $\vee$**BC**, and thus $\vee$**C**, unless $\mathsf{P} = \mathsf{NP}$. (We already have a stronger result for $\vee$**C** on OSDD, see Lemma 7.5.52.) □

Lemma 7.5.55. Let $<$ be a total strict order on $\mathcal{I}$. $\texttt{OSDD}_<$ satisfies $\vee$**BC**.

Proof. Similarly to the proof of Lemma 7.5.51, this holds because $\texttt{OSDD}_<$ satisfies both $\wedge$**BC** and $\neg$**C**. Let f and g be two Boolean functions; by De Morgan laws, $f \vee g \equiv \neg(\neg f \wedge \neg g)$.

Let φ_1 and φ_2 be two $\texttt{OSDD}_<$-representations. We can obtain an $\text{OSDD}_<$ representing $[\![\varphi_1]\!] \vee [\![\varphi_2]\!]$ by computing their negation, which is polynomial [Prop. 7.4.7], then computing the conjunction of the two $\texttt{OSDD}_<$-representations we obtain, which is also polynomial [Lemma 7.5.48], then negating the result. □

Lemma 7.5.56. Let $<$ be a total strict order on $\mathcal{I}$. $\mathtt{SD}$, $\mathtt{SDD}$, $\mathtt{FSD}$, $\mathtt{OSD}_<$, and $\mathtt{OSDD}_<$ satisfy $\vee$**clC**.

Proof. This is trivial, since clauses can be represented in any of these languages in polytime [Prop. 7.4.3], and they all satisfy $\vee$**BC** [Prop. 7.4.1 and Lemmas 7.5.51 and 7.5.55]. □

Corollary 7.5.57. $\mathtt{OSD}$ and $\mathtt{OSDD}$ satisfy $\vee$**clC**.

Proof. Let φ be an OSD. There exists a total strict order $<$ on $\mathcal{I}$ such that φ is ordered by $<$, therefore φ is an $\mathtt{OSD}_<$-representation. $\mathtt{OSD}_<$ satisfies $\vee$**clC**, so obtaining the disjunction of φ with a clause γ can be done in polytime. Hence $\mathtt{OSD}$ satisfies $\vee$**clC**.

A similar reasoning shows that $\mathtt{OSDD}$ also satisfies $\vee$**clC**. □

Sentential Entailment and Equivalence

Lemma 7.5.58. $\mathtt{OSDD}$ and $\mathtt{FSDD}$ do not satisfy **SE** unless $\mathsf{P} = \mathsf{NP}$.

Proof. We use once again the same mechanism as in the proof of Lemma 4.4.3. Let φ_1 and φ_2 be two $\mathtt{OBDD}^{\$\mathbb{B}}_{\mathcal{B}}$-representations (not necessarily of the same order); checking whether their conjunction is consistent is equivalent to checking whether $[\![\varphi_1]\!]$ does not entail $\neg[\![\varphi_2]\!]$. Since $\mathtt{OSDD} \leqslant_p \mathtt{OBDD}^{\$\mathbb{B}}_{\mathcal{B}}$ [Prop. 7.2.7], and $\mathtt{OSDD}$ satisfies $\neg$**C** [Prop. 7.4.7], if it satisfied **SE** we could check in polytime whether the conjunction of two OBDDs (the variable orderings being possibly different in each OBDD) is consistent; yet, this problem is NP-complete, as shown by Meinel and Theobald [MT98, Lemma 8.14]. Hence $\mathtt{OSDD}$ does not support **SE** unless $\mathsf{P} = \mathsf{NP}$, and neither does $\mathtt{FSDD}$, by application of Proposition 1.2.18. □

Lemma 7.5.59. Let $<$ be a total strict order on $\mathcal{I}$. $\mathtt{OSDD}_<$ satisfies **SE**.

Proof. Let f and g be two Boolean functions. Checking whether $f \models g$ is equivalent to checking whether $\neg f \vee g$ is valid.

Now, $\mathtt{OSDD}_<$ satisfies $\neg$**C** [Prop. 7.4.7] and $\vee$**BC** [Lemma 7.5.55], so given two $\mathtt{OSDD}_<$-representations φ and ψ, we can build in polytime an OSDD representing $\neg[\![\varphi]\!] \vee [\![\psi]\!]$. Moreover, $\mathtt{OSDD}_<$ satisfies **VA** [Lemma 7.5.36], so we can check in polytime whether this construct is valid. Consequently, $\mathtt{OSDD}_<$ satisfies **SE**. □

Lemma 7.5.60. Let $<$ be a total strict order on $\mathcal{I}$. $\mathtt{OSDD}_<$ satisfies **EQ**.

Proof. Checking whether $\varphi \equiv \psi$ is equivalent to checking whether both $\varphi \models \psi$ and $\psi \models \varphi$ hold. Since $\mathtt{OSDD}_<$ satisfies **SE** [Lemma 7.5.59], it also satisfies **EQ**. □

Lemma 7.5.61. $\mathtt{OSDD}$ satisfies **EQ**.

Proof. The procedure described in Algorithm 7.7 is adapted from Meinel and Theobald [MT98, Th. 8.11].

Before the procedure is called, a mesh of both Φ and Ψ is computed for each variable $x \in \mathrm{Scope}(\Phi) \cup \mathrm{Scope}(\Psi)$: it is denoted as $\mathcal{M}^x = \{M^x_1, \ldots, M^x_n\}$. Each

Algorithm 7.7 Given two OSDDs Φ and Ψ, checks whether $\Phi \equiv \Psi$ holds.

```
1:  let L := {⟨Φ, Ψ⟩}
2:  for each node N in Φ, ordered from the root to the sink, except the sink do
3:      let φ be the subgraph rooted at N
4:      let ψ be one of the SDs such that ⟨φ, ψ⟩ ∈ L
5:      for each ψ' such that ⟨φ, ψ'⟩ ∈ L do
6:          if ψ' ≢ ψ then
7:              return false
8:          remove ⟨φ, ψ'⟩ from L
9:      let x := Var(N)
10:     for each m_i ∈ Indexes(x) do
11:         if there exists E ∈ Out(N) such that m_i ∈ Lbl(E) then
12:             let φ_E be the SD rooted at Dest(E)
13:         else
14:             let φ_E be the empty SD
15:         add the couple ⟨φ_E, ψ_{|x=m_i}⟩ to L
16:     remove the couple ⟨φ, ψ⟩ from L
17: for each couples ⟨φ, ψ⟩ ∈ L do
18:     if φ ≢ ψ then
19:         return false
20: return true
```

element M_i^x of $\mathcal{M}^x$ is associated with an "index value" $m_i \in M_i^x$. The set of index values for x is denoted as $\text{Indexes}(x)$. We suppose that Φ and Ψ are reduced: since they are exclusive, there remains no $\veebar$-node in both graphs.

The procedure is based on the following equation, holding for any variable $x \in \text{Scope}(\Phi) \cup \text{Scope}(\Psi)$:

$$\Phi \equiv \Psi \quad \iff \quad \forall \omega \in \text{Dom}(x), \Phi_{|x=\omega} \equiv \Psi_{|x=\omega}.$$

Using the usual properties of meshes, we can transform the quantification on domain values into a quantification on mesh indexes:

$$\Phi \equiv \Psi \quad \iff \quad \forall m \in \text{Indexes}(x), \Phi_{|x=m} \equiv \Psi_{|x=m}. \tag{7.1}$$

Now, the procedure keeps a list L of couples of subgraphs from Φ and Ψ respectively. We show that this list has the following property:

$$\Phi \equiv \Psi \quad \iff \quad \forall \langle \varphi, \psi \rangle \in L, \varphi \equiv \psi. \tag{7.2}$$

L is initialized with $\langle \Phi, \Psi \rangle$, so this is trivially true at the beginning. On line 8, we remove only redundant couples. On line 16, we have replaced the last couple $\langle \varphi, \psi \rangle$ by a set of couples according to the decomposition scheme using the mesh (7.1). Hence, while the procedure runs, we know that $\Phi \equiv \Psi$ holds if and only if $\forall \langle \varphi, \psi \rangle \in L, \varphi \equiv \psi$, q.e.d. (7.2).

We use this equivalence to show that the algorithm is sound and complete. On line 6, we encounter two inconsistent couples: it is impossible that φ be equivalent to both ψ and ψ'. Thus there exists a couple in L such that $\varphi \not\equiv \psi$, and hence $\Phi \not\equiv \Psi$. We can return false. At the end of the algorithm, we have tested the equivalence of all couples in L (lines 17–19), we can thus return true.

Let us now show that the algorithm is polynomial. Each node of Φ is treated once. For a given node, we add exactly n couples to L (line 15), with n being the size of the mesh on x, which is bounded by $2(\|\Phi\| + \|\Psi\|)$. We also remove one couple (line 16). Hence L contains at most $n \cdot |\mathscr{N}_\Phi|$ couples.

For each node, we make at most $n \cdot |\mathscr{N}_\Phi|$ equivalence tests at line 6; equivalence tests being on OSDDs of the same variable order, they can be done in polytime [Lemma 7.5.60]. The traversal of the graph (lines 2–16) is thus polynomial.

Once we have traversed the entire Φ, we know by construction that the only couples $\langle \varphi, \psi \rangle$ left in L are such that φ is either the sink-only or the empty SD, so φ is an OSDD of the same variable order as Ψ: all the equivalence tests of line 18 (there can be at most $n \cdot |\mathscr{N}_\Phi|$) can be done in polytime. □

Term Restriction and Quantification

Lemma 7.5.62. Algorithm 4.5 [p. 117] can be easily adapted to take as input an FSD φ and a $\mathtt{term}_{\mathcal{I}}^{\mathbb{T}\mathbb{Z}}$-representatio γ. It has then the following properties:

- it is polynomial in $\|\varphi\| \cdot \|\gamma\|$;
- it preserves focusingness;
- it preserves ordering.

Proof.

- Each edge in φ is processed once; for each one, every literal from γ is examined once. The intersection of two elements of $\mathbb{T}\mathbb{Z}$ can be done in time polynomial in their size (roughly speaking, it only requires an ordering of the bounds).
- The procedure preserves focusingness as in the case of FIA: for each variable x in Scope(γ), all x-nodes are replaced by $\veebar$-nodes, that are not required to be focusing.
- It preserves ordering for the same reason: the order of the nodes is not modified. Some nodes are replaced by $\veebar$-nodes, but this does not interfere with the ordering property. □

Corollary 7.5.63. FSD, OSD, and $\mathtt{OSD}_<$ satisfy **TR**, **FO**, and **SFO**.

Proof. FSD, OSD, and $\mathtt{OSD}_<$ satisfy **TR**, as a consequence of Lemma 7.5.62. Since forgetting a set of variables $\{x_1, \ldots, x_k\}$ boils down to computing the restriction to the valid term $\left[x_1 \in \mathbb{Z}\right] \wedge \cdots \wedge \left[x_k \in \mathbb{Z}\right]$, FSD, OSD, and $\mathtt{OSD}_<$ satisfy **FO**, and thus **SFO**. □

Lemma 7.5.64. Let $n \in \mathbb{N}^*$, $\varphi_1, \ldots, \varphi_n$ be n SDs, and $\psi = \text{Join}(\varphi_1, \ldots, \varphi_n)$ be the result of the application of Algorithm 7.8 on the φ_i. The following properties hold:

- Algorithm 7.8 runs in time linear in $\sum_{i=1}^{n} \|\varphi_i\|$;
- if all φ_i are focusing, ψ is also focusing;
- if all φ_i are ordered by some total strict order $<$, ψ is also ordered by $<$;
- if all φ_i are exclusive, ψ is also exclusive;
- $\exists x.[\![\psi]\!] \equiv \bigvee_{i=1}^{n} [\![\varphi_i]\!]$;
- $\forall x.[\![\psi]\!] \equiv \bigwedge_{i=1}^{n} [\![\varphi_i]\!]$.

Proof.

- **Complexity:** This is trivial, since the φ_i are just copied into ψ, and only n edges of size 1 are added.
- **Focusingness:** x does not appear in any of the φ_i. Therefore if they are all focusing, ψ also is.
- **Ordering:** Considering (without loss of generality) that x is the smallest variable of ψ with respect to $<$, the result is obvious.
- **Exclusive decision:** The added root node is exclusive, so if all the φ_i satisfy exclusive decision, ψ also does.
- **Conjunction and disjunction:** By construction, for all $i \in [1, \ldots, n]$, we have $[\![\varphi_i]\!] \equiv [\![\psi_{|x=i}]\!]$. Hence $\bigvee_{i=1}^{n} [\![\psi]\!]_{|x=i} \equiv \bigvee_{i=1}^{n} [\![\varphi_i]\!]$ and $\bigwedge_{i=1}^{n} [\![\psi]\!]_{|x=i} \equiv \bigwedge_{i=1}^{n} [\![\varphi_i]\!]$. By Prop. 1.4.12, we get that $\exists x.[\![\psi]\!] \equiv \bigvee_{i=1}^{n} [\![\varphi_i]\!]$, and $\forall x.[\![\psi]\!] \equiv \bigwedge_{i=1}^{n} [\![\varphi_i]\!]$. □

Algorithm 7.8 Given n SDs $\varphi_1, \ldots, \varphi_n$, builds an SD $\text{Join}(\varphi_1, \ldots, \varphi_n)$.

```
let x ∈ 𝒳 with Dom(x) = [1, ..., n], such that ∀φ_i, x ∉ Scope(φ_i)
fuse the sinks of all φ_i
create a root node N, labeled x
for i from 1 to n do
    create an edge between N and Root(φ_i), labeled {i}
```

Lemma 7.5.65. OSDD and $\text{OSDD}_<$ do not satisfy **SFO**. FSDD does not satisfy **SFO** unless P = NP.

Proof. Let L be a language among OSDD, $\text{OSDD}_<$, and FSDD.

Let Σ be a $\text{DNF}_{\mathcal{I}}^{\mathbb{T}\mathbb{Z}}$-representation. Each of its terms can be transformed in an L-representation φ_i in polytime [Prop. 7.4.3]. Let us apply Algorithm 7.8 on the φ_i.

Thanks to Lemma 7.5.64, we know that we obtain in polytime an $\mathtt{L}$-representation ψ such that

$$\exists x.[\![\psi]\!] \equiv \bigvee_{i=1}^{n} [\![\varphi_i]\!].$$

Suppose $\mathtt{L}$ satisfies **SFO**; then we can obtain an $\mathtt{L}$-representation of $\bigvee_{i=1}^{n} [\![\varphi_i]\!]$, and thus Σ, in polytime. All in all, this means that if $\mathtt{L}$ satisfies **SFO**, then $\mathtt{L} \leqslant_p \mathtt{DNF}_{\mathcal{I}}^{\mathbb{TZ}}$, and in particular $\mathtt{L}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$, by Proposition 1.2.16 and Lemma 7.5.16.

However, we know that $\mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \sim_p \mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ and that $\mathtt{OSDD}_{<,\mathcal{B}}^{\$\mathbb{B}} \sim_p \mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}}$ [Prop. 7.2.8]; hence, if $\mathtt{OSDD}$ or $\mathtt{OSDD}_<$ satisfied **SFO**, we would get $\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$ or $\mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$, respectively. Since it is impossible [Th. 1.4.17], this means that $\mathtt{OSDD}$ and $\mathtt{OSDD}_<$ do not satisfy **SFO**.

As for $\mathtt{FSDD}$, it satisfies **VA** [Prop. 7.4.5], so having $\mathtt{FSDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$ would imply that we could check the validity of any Boolean DNF in polytime, which is impossible unless $\mathsf{P} = \mathsf{NP}$ [Th. 1.4.18]. Hence, $\mathtt{FSDD}$ does not satisfy **SFO** unless $\mathsf{P} = \mathsf{NP}$. □

Corollary 7.5.66. $\mathtt{OSDD}$ and $\mathtt{OSDD}_<$ do not satisfy **FO** or **TR**. $\mathtt{FSDD}$ does not satisfy **FO** or **TR** unless $\mathsf{P} = \mathsf{NP}$.

Proof. The results hold because **SFO** is implied by **FO**, and **FO** by **TR** (forgetting a set of variables $\{x_1, \ldots, x_k\}$ boils down to computing the restriction to the valid term $\big[x_1 \in \mathbb{Z}\big] \wedge \cdots \wedge \big[x_k \in \mathbb{Z}\big]$). □

Lemma 7.5.67. $\mathtt{OSDD}$ and $\mathtt{OSDD}_<$ do not satisfy **SEN**. $\mathtt{FSD}$, $\mathtt{FSDD}$, $\mathtt{OSD}$, and $\mathtt{OSD}_<$ do not satisfy **SEN** unless $\mathsf{P} = \mathsf{NP}$.

Proof. Let $\mathtt{L}$ be a language among $\mathtt{FSD}$, $\mathtt{FSDD}$, $\mathtt{OSD}$, $\mathtt{OSDD}$, $\mathtt{OSD}_<$, and $\mathtt{OSDD}_<$. We use a similar technique as in the proof of Lemma 7.5.65.

Let Σ be a $\mathtt{CNF}_{\mathcal{I}}^{\mathbb{TZ}}$-representation. Each of its clauses can be transformed in an $\mathtt{L}$-representation φ_i in polytime [Prop. 7.4.3]. Let us apply Algorithm 7.8 on the φ_i. Thanks to Lemma 7.5.64, we know that we obtain in polytime an $\mathtt{L}$-representation ψ such that

$$\forall x.[\![\psi]\!] \equiv \bigwedge_{i=1}^{n} [\![\varphi_i]\!].$$

Suppose $\mathtt{L}$ satisfies **SEN**; then we can obtain an $\mathtt{L}$-representation of $\bigwedge_{i=1}^{n} [\![\varphi_i]\!]$, and thus Σ, in polytime. All in all, this means that if $\mathtt{L}$ satisfies **SEN**, then $\mathtt{L} \leqslant_p \mathtt{CNF}_{\mathcal{I}}^{\mathbb{TZ}}$, and in particular $\mathtt{L}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$, by Proposition 1.2.16 and Lemma 7.5.16..

However, we know that $\mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} \sim_p \mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$ and that $\mathtt{OSDD}_{<,\mathcal{B}}^{\$\mathbb{B}} \sim_p \mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}}$ [Prop. 7.2.8]; hence, if $\mathtt{OSDD}$ or $\mathtt{OSDD}_<$ satisfied **SEN**, we would get $\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ or $\mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$, respectively. Since it is impossible [Th. 1.4.17], this means that $\mathtt{OSDD}$ and $\mathtt{OSDD}_<$ do not satisfy **SEN**.

As for the other languages, they all satisfy **CO** [Cor. 7.5.9], so having $\mathtt{L}_{\mathcal{B}}^{\$\mathbb{B}} \leqslant_p \mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$ would imply that we could check the consistency of any Boolean CNF in

polytime, which is impossible unless $\mathsf{P} = \mathsf{NP}$ [Th. 1.4.18]. Hence, FSD, FSDD, OSD, and $\mathtt{OSD}_<$ do not satisfy **SEN** unless $\mathsf{P} = \mathsf{NP}$. □

Corollary 7.5.68. OSDD and $\mathtt{OSDD}_<$ do not satisfy **EN**. FSD, FSDD, OSD, and $\mathtt{OSD}_<$ do not satisfy **EN** unless $\mathsf{P} = \mathsf{NP}$.

Lemma 7.5.69. SD and SDD do not satisfy **EN** unless $\mathsf{P} = \mathsf{NP}$.

Proof. The proof is similar to that of Lemma 4.4.11: we show that if these languages satisfied **EN**, they would also satisfy **VA**. Indeed, if they satisfied **EN**, we could obtain in polytime, for any φ, an SD ψ equivalent to $\forall V.[\![\varphi]\!]$, with $V = \text{Scope}(\varphi)$. By definition of the quantification [Def. 1.4.9], $\text{Scope}(\psi) = \varnothing$; after reducing ψ, we thus obtain either the empty or the sink-only SD [Lemma 7.5.15]. It is hence easy to check whether ψ is consistent; and thanks to Proposition 1.4.10, we know that φ is valid if and only if ψ is consistent. We could thus check in polytime the validity of any SD or SDD; yet neither SD nor SDD satisfies **VA** unless $\mathsf{P} = \mathsf{NP}$ [Lemma 7.5.32]. □

Lemma 7.5.70. SD and SDD do not satisfy **FO** or **TR** unless $\mathsf{P} = \mathsf{NP}$.

Proof. The proof is similar to that of Lemma 4.4.12. We show that if one of these languages satisfied **FO**, they would also satisfy **CO**.

Indeed, we could in this case obtain in polytime, for any φ, an SD ψ equivalent to $\exists V.[\![\varphi]\!]$, with $V = \text{Scope}(\varphi)$. By definition of the quantification, [Def. 1.4.9] $\text{Scope}(\psi) = \varnothing$; after reducing ψ, we thus obtain either the empty or the sink-only SD [Lemma 7.5.15]. It is hence easy to check whether ψ is consistent; and thanks to Proposition 1.4.10, we know that φ is consistent if and only if ψ is consistent. We could thus check in polytime the consistency of any SD or SDD; yet it is impossible unless $\mathsf{P} = \mathsf{NP}$ [Lemma 7.5.32].

Hence, neither SD nor SDD satisfies **FO** unless $\mathsf{P} = \mathsf{NP}$; and since **TR** implies **FO** (forgetting a set of variables $\{x_1, \ldots, x_k\}$ boils down to computing the restriction to the valid term $\left[x_1 \in \mathbb{Z}\right] \wedge \cdots \wedge \left[x_k \in \mathbb{Z}\right]$), SD and SDD do not satisfy **TR** unless $\mathsf{P} = \mathsf{NP}$. □

Lemma 7.5.71. SD and SDD satisfy **SFO** and **SEN**.

Proof. We use the Shannon decomposition [Prop. 1.4.12] using a mesh, just as we did on FIAs [see proof of Prop. 4.3.8, p. 120]. Any sublanguage of SD that satisfies both **CD** and $\vee$**C** satisfies **SFO**; any sublanguage of SD that satisfies both **CD** and $\wedge$**C** satisfies **SEN**.

Since SD and SDD satisfy **CD** [Lemma 7.5.13], $\vee$**C** [respectively, Prop. 7.4.1 and Lemma 7.5.51], and $\wedge$**C** [Prop. 7.4.2], they satisfy both **SFO** and **SEN**. □

Final Proof

Proof of Theorem 7.4.9 [p. 165]. All the results of this theorem come from propositions and lemmas. Tables 7.5 and 7.6 associate with each claim its corresponding proposition. □

Query	SD	SDD	FSD	FSDD	OSD	OSDD	OSD$_<$	OSDD$_<$
CO	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	√ [Cor. 7.5.9]	√ [Cor. 7.5.9]	√ [Cor. 7.5.9]	√ [Cor. 7.5.9]	√ [Cor. 7.5.9]	√ [Cor. 7.5.9]
VA	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	○ [Cor. 7.5.34]	√ [Prop. 7.4.5]	○ [Cor. 7.5.34]	√ [Lem. 7.5.36]	○ [Lem. 7.5.33]	√ [Lem. 7.5.36]
MC	√ [Lem. 7.5.30]	√ [Cor. 7.5.31]	√ [Cor. 7.5.31]	√ [Cor. 7.5.31]	√ [Cor. 7.5.31]	√ [Cor. 7.5.31]	√ [Cor. 7.5.31]	√ [Cor. 7.5.31]
CE	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	√ [Lem. 7.5.10]	√ [Cor. 7.5.11]	√ [Cor. 7.5.11]	√ [Cor. 7.5.11]	√ [Cor. 7.5.11]	√ [Cor. 7.5.11]
IM	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	○ [Cor. 7.5.34]	√ [Lem. 7.5.5]	○ [Cor. 7.5.34]	√ [Lem. 7.5.36]	○ [Lem. 7.5.33]	√ [Lem. 7.5.36]
EQ	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	○ [Cor. 7.5.34]	?	○ [Cor. 7.5.34]	√ [Lem. 7.5.61]	○ [Lem. 7.5.33]	√ [Lem. 7.5.60]
SE	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	○ [Cor. 7.5.34]	○ [Lem. 7.5.58]	○ [Cor. 7.5.34]	○ [Lem. 7.5.58]	○ [Lem. 7.5.33]	√ [Lem. 7.5.59]
MX	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	√ [Lem. 7.5.8]	√ [Cor. 7.5.9]	√ [Cor. 7.5.9]	√ [Cor. 7.5.9]	√ [Cor. 7.5.9]	√ [Cor. 7.5.9]
CX	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	√ [Lem. 7.5.37]	√ [Cor. 7.5.38]	√ [Cor. 7.5.38]	√ [Cor. 7.5.38]	√ [Cor. 7.5.38]	√ [Cor. 7.5.38]
CT	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	○ [Cor. 7.5.34]	?	○ [Cor. 7.5.34]	√ [Lem. 7.5.35]	○ [Lem. 7.5.33]	√ [Lem. 7.5.35]
ME	○ [Lem. 7.5.32]	○ [Lem. 7.5.32]	√ [Lem. 7.5.40]	√ [Cor. 7.5.41]	√ [Cor. 7.5.41]	√ [Cor. 7.5.41]	√ [Cor. 7.5.41]	√ [Cor. 7.5.41]

Table 7.5: Results about queries, and proposition corresponding to each result. √ means "satisfies" and ○ means "does not satisfy, unless P = NP".

Transfo.	SD	SDD	FSD	FSDD	OSD	OSDD	OSD$_<$	OSDD$_<$
CD	$\surd$ [Cor. 7.5.14]	$\surd$ [Cor. 7.5.14]	$\surd$ [Cor. 7.5.14]	$\surd$ [Cor. 7.5.14]	$\surd$ [Cor. 7.5.14]	$\surd$ [Cor. 7.5.14]	$\surd$ [Cor. 7.5.14]	$\surd$ [Cor. 7.5.14]
TR	○ [Lem. 7.5.70]	○ [Lem. 7.5.70]	$\surd$ [Cor. 7.5.63]	○ [Cor. 7.5.66]	$\surd$ [Cor. 7.5.63]	● [Cor. 7.5.66]	$\surd$ [Cor. 7.5.63]	● [Cor. 7.5.66]
FO	○ [Lem. 7.5.70]	○ [Lem. 7.5.70]	$\surd$ [Cor. 7.5.63]	○ [Cor. 7.5.66]	$\surd$ [Cor. 7.5.63]	● [Cor. 7.5.66]	$\surd$ [Cor. 7.5.63]	● [Cor. 7.5.66]
SFO	$\surd$ [Lem. 7.5.71]	$\surd$ [Lem. 7.5.71]	$\surd$ [Cor. 7.5.63]	○ [Lem. 7.5.65]	$\surd$ [Cor. 7.5.63]	● [Lem. 7.5.65]	$\surd$ [Cor. 7.5.63]	● [Lem. 7.5.65]
EN	○ [Lem. 7.5.69]	○ [Lem. 7.5.69]	○ [Cor. 7.5.68]	○ [Cor. 7.5.68]	○ [Cor. 7.5.68]	● [Cor. 7.5.68]	○ [Cor. 7.5.68]	● [Cor. 7.5.68]
SEN	$\surd$ [Lem. 7.5.71]	$\surd$ [Lem. 7.5.71]	○ [Lem. 7.5.67]	○ [Lem. 7.5.67]	○ [Lem. 7.5.67]	● [Lem. 7.5.67]	○ [Lem. 7.5.67]	● [Lem. 7.5.67]
$\vee$**C**	$\surd$ [Prop. 7.4.1]	$\surd$ [Lem. 7.5.51]	$\surd$ [Prop. 7.4.1]	○ [Cor. 7.5.54]	?	● [Lem. 7.5.52]	$\surd$ [Prop. 7.4.1]	● [Lem. 7.5.52]
$\vee$**BC**	$\surd$ [Prop. 7.4.1]	$\surd$ [Lem. 7.5.51]	$\surd$ [Prop. 7.4.1]	○ [Cor. 7.5.54]	?	○ [Cor. 7.5.54]	$\surd$ [Prop. 7.4.1]	$\surd$ [Lem. 7.5.55]
$\vee$**clC**	$\surd$ [Lem. 7.5.56]	$\surd$ [Lem. 7.5.56]	$\surd$ [Lem. 7.5.56]	$\surd$ [Prop. 7.4.6]	$\surd$ [Cor. 7.5.57]	$\surd$ [Cor. 7.5.57]	$\surd$ [Lem. 7.5.56]	$\surd$ [Lem. 7.5.56]
$\wedge$**C**	$\surd$ [Prop. 7.4.2]	$\surd$ [Prop. 7.4.2]	○ [Cor. 7.5.46]	○ [Cor. 7.5.46]	○ [Cor. 7.5.46]	● [Lem. 7.5.44]	○ [Lem. 7.5.47]	● [Lem. 7.5.44]
$\wedge$**BC**	$\surd$ [Prop. 7.4.2]	$\surd$ [Prop. 7.4.2]	○ [Lem. 7.5.45]	○ [Lem. 7.5.45]	○ [Lem. 7.5.45]	○ [Lem. 7.5.45]	$\surd$ [Lem. 7.5.48]	$\surd$ [Lem. 7.5.48]
$\wedge$**tC**	$\surd$ [Lem. 7.5.49]	$\surd$ [Lem. 7.5.49]	$\surd$ [Lem. 7.5.7]	$\surd$ [Lem. 7.5.7]	$\surd$ [Lem. 7.5.50]	$\surd$ [Lem. 7.5.50]	$\surd$ [Lem. 7.5.49]	$\surd$ [Lem. 7.5.49]
$\neg$**C**	?	$\surd$ [Prop. 7.4.7]	○ [Cor. 7.5.43]	?	○ [Cor. 7.5.43]	$\surd$ [Prop. 7.4.7]	○ [Cor. 7.5.43]	$\surd$ [Prop. 7.4.7]

Table 7.6: Results about transformations, and proposition corresponding to each result. $\surd$ means "satisfies", ● means "does not satisfy", and ○ means "does not satisfy, unless $\mathsf{P} = \mathsf{NP}$".

CHAPTER

8

Building Set-labeled Diagrams

Chapter 7 identified a number of sublanguages of set-labeled diagrams suiting our application, namely, the family of focusing set-labeled diagrams. We saw that FIAs can be transformed into FSDs in polytime without loss of information. In this chapter, we study how to compile set-labeled diagrams directly from a constraint network over discrete variables. We begin by formally defining this input language [§ 8.1], then quickly survey bottom-up compilation [§ 8.2], and lastly present our "CHOCO with a trace" compiler [§ 8.3].

8.1 Discrete Constraint Networks

As we have seen in Section 1.6.1, the constraint network [Definition 1.6.1] is a natural, generic way of representing a knowledge base. Deciding whether a given constraint network over enumerated variables is consistent is generally called a *discrete constraint satisfaction problem (CSP)*. We use a similar name here.

Definition 8.1.1. A *discrete constraint network (DCN)* is a constraint network $\Pi = \langle V, \mathscr{C} \rangle$ in which $V \subseteq \mathcal{E}$.

Note that DCNs' variables are *not* strictly the same as SDs': only *finite* intervals of integers are considered here. This distinction has two advantages: first, it reflects the most widespread definition of discrete CSPs, and second, it allows the compilation of any DCN into SD, since $\mathtt{SD}_{\mathcal{E}}$ is complete [Proposition 7.2.6]. Hence, in order to compile the solution set of a discrete CN as an SD, no approximation is necessary, contrary to the compilation of continuous CNs as IAs [§ 5.1].

Constraints are simply defined as sets of assignments. However, they are not always *given* in this form; two cases are generally distinguished.

- A constraint can be given *in extension*, that is, using a list of *admissible* tuples of values. This directly corresponds to the definition.

- A constraint can be given *in intension*, that is, using a *formula* over its scope variables: the admissible tuples are those that are consistent with the formula.

Constraints given in extension can be seen as DNF formulæ—and can as such be transformed into SDs in polytime [Proposition 7.4.4]. On the contrary, constraints given in intension are not *directly* translatable into SDs: consider for example constraints $x < y$, $x \times y = z$, or alldiff$(x_1, \ldots, x_n)$.

To compile a DCN into SD, we choose to transform constraints given in intension into DNF formulæ. For that purpose, we use a *discrete CN solver* as a tool to help us build SDs. Contrary to the continuous CN solver RealPaver that we used to build interval automata, discrete CN solvers return the *exact* solution set of an input constraint network. There are no concerns about approximation in this chapter. The particular solver we used is CHOCO [CHO10]: it is an open-source solver, written in Java, that implements a lot of state-of-the-art constraint satisfaction techniques.

8.2 Bottom-up Compilation

Aside from the fact that the resolution step outputs the exact solution set rather than an approximation, using CHOCO's output to compile SDs is very similar to using RealPaver's output to compile IAs [§ 5.2]. CHOCO outputs the *enumerated* solution set of its input DCN, that is to say, it outputs a list of all solutions. There is no box in this context, only assignments—but assignments are just specific boxes, and are consequently polynomially translatable into any sublanguage of SD, as shown in Proposition 7.4.3. However, disjoining these SDs is necessary to obtain an SD representing the same Boolean function as the original DCN. To compile ordered SDs, care must be taken that all assignment SDs have the same variable ordering. Note also that ordered SDDs are likely to be larger than nonexclusive ordered SDs, since they only support *bounded* disjunction in polytime.

Using the output of CHOCO is the simplest way to compile a DCN into an SD. It is also possible, following Amilhastre [Ami99], to compile an SD for each constraint in the DCN, then combine the elementary SDs into a larger one representing the conjunction of all constraints and thus the complete DCN. This was not possible on FIA, since this language does not support conjunction in polytime, even bounded; but it is possible on $\mathtt{OSD}_<$ and $\mathtt{OSDD}_<$, provided that the same variable order is used to compile all constraints.

As we explained in Section 1.6.1.1, the drawback of this kind of method is that it generates intermediate data structures, which are space consuming and can even be exponentially larger than the final decision diagram. This is why we examine a different method, based on CHOCO's *trace*, as we did with RealPaver [§ 5.3].

8.3 CHOCO with a Trace

We apply the principles of "DPLL with a trace" [HD05a] to the CHOCO solver. The idea of using a solver trace for compilation has been adapted by Hadzic et al. [HH$^+$08], who build approximate MDDs (i.e., MDDs the model set of which is an approximation of the solution set of the input constraint network) tracing a depth-first search algorithm. A similar technique is used by Mateescu, Dechter, and Marinescu [MDM08], where AOMDDs (MDDs with "and" nodes) are built following the trace of an AND/OR search.

These approaches use a predetermined variable order (a tree order in the case of AOMDDs), and variables cannot be repeated along a path. We relax these assumptions here: the choice of the next variable to branch on can be done *dynamically*, depending on a heuristics, and domains need not be split into singletons, which means that a variable can be branched on several times along a search path. Indeed, one does not always need ordered SDs; focusing SDs are sufficient for applications needing only conditioning and model extraction, such as the exploitation of a decision policy.

8.3.1 CHOCO's Search Algorithm

Algorithm 8.1 is a simplified version of CHOCO's search procedure. It is a generic depth-first search, with constraint propagation at each search node. We present it as a recursive, high-level procedure to clarify the way it works.

It takes as input a constraint network $\Pi = \langle X, \mathcal{C} \rangle$, defined by a set of constraints $\mathcal{C}$ over a set of variables X [Definition 1.6.1], as well as a set of assigned variables (this set being of course empty at the top-level call). It outputs the enumerated model set of Π, i.e., each model is explicitly returned.

First, Algorithm 8.1 calls the internal function `Prune` on the current DCN; this is the *propagation* step, where values are removed from variable domains when they are proven incompatible with some constraints. The `Prune` function uses several state-of-the-art constraint propagation techniques, that we do not detail here [see CHOCO's user guide: CHO10]. We retrieve a modified DCN (line 1), which is guaranteed to have the same solution set as input Π.

Then, the procedure checks whether it has encountered a leaf in the search tree: this is the case either (i) if the current DCN is inconsistent (that is, one variable has an empty domain), in which case the empty set is returned, or (ii) if all variables have been assigned, in which case the corresponding assignment is a solution.

The remainder of the procedure takes care of new nodes in the search tree. A variable x is chosen, using function `Sel_var`; the choice depends on a heuristics selected by the user, but only unassigned variables can be selected. Then the current domain of the chosen variable is partitioned, using the `Split` function. The user can control the number of elements in the partition and the size of each element. Then, for each element r in the partition, the procedure recursively calls itself on the

Algorithm 8.1 `CHOCO`(Π, X_a): returns the set of all solutions of the discrete CN Π. X_a is the current set of assigned variables.

```
1:  Prune(Π)
2:  if Π is proven inconsistent then
3:      return ∅
4:  if X_a = X then   // all variables in Π are assigned
5:      let x⃗ be the corresponding assignment
6:      return {x⃗}
7:  x := Sel_var(Π)
8:  R := Split(Π, x)
9:  S := ∅
10: for each r ∈ R do
11:     let X'_a := X_a
12:     if r is reduced to a singleton then   // x is now assigned
13:         X'_a := X'_a ∪ {x}
14:     S_r := CHOCO(Π_|x∈r, X'_a)
15:     S := S ∪ S_r
16: return S
```

DCN Π in which $\mathrm{Dom}(x)$ has been restricted to r. If r is a singleton, variable x is marked as assigned: it will not be modified anymore in the current search subgraph.

The described procedure is not specific to CHOCO. Note in particular that the actual implementation of internal functions `Prune`, `Split`, and `Sel_var` does not matter as long as they fulfill the requirements.

8.3.2 Tracing CHOCO

Our approach consists in following Algorithm 8.1 to build set-labeled diagrams, instead of simply returning an enumeration of the solution set. Similarly to what has been done in Section 5.3.2, we present the modified Algorithm 8.2 using frames to indicate changes.

The set-labeled diagram built by Algorithm 8.2 exactly reflects the search tree. It does not need to handle "pruning nodes", like Algorithm 5.3 [p. 133], because there is no precision parameter: the search goes on until every variable is explicitly assigned.

Whenever the procedure reaches a leaf node in the search tree, this leaf node corresponds either to a solution or to a dead-end; in the former case, the algorithm returns a sink-only SD (line 5), to account for the fact that any assignment is a solution of the current problem; in the latter case, it returns an empty SD (line 3).

In the case of an internal search node, an unassigned variable x is chosen, and its current domain is split. The exploration of the search tree associated with each partition element r returns an SD ψ_r that represents the solution set of the subproblem $\Pi_{|x \in r}$ obtained by reducing the domain of x to r. The solution set of the current

Algorithm 8.2 `CHOCO_to_SD`(Π, X_a): returns a set-labeled diagram that represents the solution set of the discrete CN Π. X_a is the set of currently assigned variables.

```
 1: Prune(Π)
 2: if Π is proven inconsistent then
 3:     return the empty SD
 4: if X_a = X then   // all variables in Π are assigned
 5:     return the sink-only SD
 6: x := Sel_var(Π)
 7: R := Split(Π, x)
 8: Ψ := ∅
 9: for each r ∈ R do
10:     let X'_a := X_a
11:     if r is reduced to a singleton then   // x is now assigned
12:         X'_a := X'_a ∪ {x}
13:     let ψ_r := CHOCO_to_SD(Π_{|x∈r}, X'_a)
14:     add the couple ⟨r, ψ_r⟩ to Ψ
15: let node N := Get_node(x, Ψ)
16: let ψ be the graph rooted at N
17: return ψ
```

constraint network Π is then an SD ψ, rooted at an x-node with an r-labeled outgoing edge pointing to ψ_r for each r in the domain partition. This node is obtained on line 15 thanks to an internal function `Get_node`, which works in a fashion similar to the `Get_node` from Section 5.3.2. In particular, it does not add an r-edge if the relative SD ψ_r is empty, and makes as many reduction operations as possible on the fly. The compiled SD is, for instance, guaranteed to contain no isomorphic nodes, thanks to the *unique node table* maintained by `Get_node`.

Like for Algorithm 5.2 [p. 131], at any time during compilation, the current SD is always smaller than the final one (but note that the resulting SD is generally not reduced, since stammering nodes cannot be treated by `Get_node`). The procedure is therefore polyspace with respect to the size of its output, which is not the case for the bottom-up approach.

8.3.3 Caching Subproblems

Still following Huang and Darwiche [HD05a], we try to achieve a time complexity closer to the size of the output by using *caching*.

Principle and Algorithm

The purpose is to avoid *equivalent subproblems* from being re-explored. Changes are shown in Algorithm 8.3. More precisely, each time a subproblem Π is solved, it is stored in a hash table at key k, which depends on the current variable domains

Algorithm 8.3 `SD_builder`(Π, X_a): returns a set-labeled diagram that represents the solution set of the discrete CN Π. X_a is the set of currently assigned variables.

```
 1: Prune(Π)
 2: k ≔ Compute_key(Π, X_a)
 3: if there is an entry for the key k in the cache then
 4:     return the SD corresponding to key k in the cache
 5: if Π is proven inconsistent then
 6:     return the empty SD
 7: if X_a = X then   // all variables in Π are assigned
 8:     return the sink-only SD
 9: x ≔ Sel_var(Π)
10: R ≔ Split(Π, x)
11: Ψ ≔ ∅
12: for each r ∈ R do
13:     let X'_a ≔ X_a
14:     if r is reduced to a singleton then   // x is now assigned
15:         X'_a ≔ X'_a ∪ {x}
16:     let ψ_r ≔ SD_builder(Π_|x∈r, X'_a)
17:     add the couple ⟨r, ψ_r⟩ to Ψ
18: let node N ≔ Get_node(x, Ψ)
19: let ψ be the graph rooted at N
20: store ψ at key k in the cache
21: return ψ
```

(line 20). Moreover, prior to processing any subproblem Π', the compiler computes its key k' (line 2), checks whether it is already present in the hash table, and returns the corresponding SD if it is the case (line 4).

The idea is that since the subproblems in question are equivalent, exploring the search subtree for the second subproblem is useless: the obtained SD would be equivalent to the cached one anyway. In a way, the use of a cache transforms the search tree into a search *graph*. Note that this caching concerns *problems*—it is unrelated to the unique node table used to avoid creating isomorphic nodes.

Computation of the Cache Key

Let Π be the current constraint network at a given search node, and X_a be the set of variables already assigned in Π. Exploring the search subtree rooted at the current node returns an SD, that involves only variables in $X \setminus X_a$, and represents the solution set of Π restricted to $X \setminus X_a$. The current constraint network Π' at a different search node is equivalent to Π if and only if the corresponding SDs are equivalent—that is to say, if and only if their solution sets restricted to $X \setminus X_a$ are equal.

Since we use caching to avoid having to explore equivalent subproblems multiple times, cache keys must be designed so that they group as many equivalent

subproblems as possible. Ideally, *all* equivalent subproblems are associated with a same key, so that the algorithm *never* computes a subgraph equivalent to one it already returned. However, for the cache lookup to be efficient, the key must be as short and easy to compute as possible. A compromise is therefore necessary.

Intuitively, the current subproblem could be represented by a key listing the current domain of all variables. It is possible to reduce the size of the key, by using the exact same technique as Lecoutre et al. [LS$^+$07]. The idea is to remove some of the variables from the key; to be removed, a variable x must be assigned ($x \in X_a$) and involved only in constraints proven to be *necessarily satisfied* in the current subproblem (such constraints are sometimes called *universal* or *entailed* constraints). Lecoutre et al. [LS$^+$07] have proven that this mechanism preserves the solution set.

Advantages of Caching

Caching is interesting for the reasons described by Huang and Darwiche [HD05a]—it allows the time complexity of the compiler to be polynomial in the output. This is not especially related to the compilation aspect of the procedure: for example, Lecoutre et al. [LS$^+$07] apply this idea to constraint satisfaction procedures, in which the goal is to find only one solution. Of course, they do not cache consistent subproblems, since their algorithm stops as soon as a solution is found.

There is also another advantage to caching, which is related to compilation only. If the current subproblem is consistent, storing its SD may allow the size of the final SD to decrease. Indeed, if caching were not used, equivalent subproblems would be explored independently, possibly using different variable orders, especially when variable selection heuristics and domain partitioning involve randomness. This may lead to several subgraphs representing the same Boolean function without being isomorphic, hence the increase in the resulting graph size.

Cache Minimization

Keeping track of every single subproblem can lead to a huge cache. In order to minimize memory space, we choose to limit cache size to an arbitrary value. That is to say, each time the cache is full, some entries have to be removed—the goal being to keep only the most interesting ones.

We use the following heuristics: first, discard entries that have been used the least, then the oldest entries, and then entries with the longest keys (which correspond to small subproblems).

This cache minimization technique reduces the amount of memory space necessary for the compiler to run, and also speeds up cache operations. This allows larger problems to be compiled, but does *not* improve the resulting SD; it is actually the opposite, since, as we showed in the previous subsection, using a cache can reduce the size of the output.

8.3.4 Properties of Compiled SDs

Structure

For the same reasons as for "RealPaver with a trace" [see § 5.3.5.1], the set-labeled diagrams returned by Algorithm 8.3 always satisfy the focusing property. Indeed, variable domains can only be reduced, either by domain splitting or by constraint propagation. Contrary to the FIAs obtained in Chapter 5, however, the returned SDs always satisfy exclusive decision: indeed, function `Split` computes a *partition* of the variable's current domain. Since partitions contain only *disjoint* subsets, Algorithm 8.3 always outputs FSDDs.

However, depending on the behavior of the `Sel_var` and `Split` functions, the structure of these FSDDs can be more specific.

- If `Split` branches on *singletons*, each variable is always assigned the first time it is chosen; the resulting set-labeled diagrams thus satisfy the read-once property.
- If `Sel_var` moreover follows a static ordering $<$ on the variables, the result is an $\texttt{OSDD}_{<}$-representation.

Variables

Note that being based on CHOCO, our compiler is inherently restricted to the interpretation domain of CHOCO's input. That is to say, it handles *enumerated* variables only, whereas SDs can potentially use variables with an unbounded domain. "CHOCO with a trace" thus provides $\texttt{SD}_{\mathcal{E}}$-representations only. Since $\texttt{MDD} = \texttt{OSDD}_{\mathcal{E}}^{\$\mathbb{Z}}$ [see Section 7.2.2], it could be expected that by branching on singletons and following a static variable ordering, "CHOCO with a trace" would provide MDDs. This is actually not the case in general, because the contiguity reduction operation applied by the `Get_node` function can merge some edge labels, changing the literal expressivity from $\$\mathbb{Z}$ to $\mathbb{T}\mathbb{Z}$. Using this specific tuning, the compiled form is generally an $\texttt{OSDD}_{\mathcal{E}}$-representation, but may not be an $\texttt{OSDD}_{\mathcal{E}}^{\$\mathbb{Z}}$-representation, that is, an MDD.

*
**

We implemented "CHOCO with a trace" by adding the elements presented in this chapter to the source code of CHOCO. This compiler allowed us to obtain experimental results for our applications; this is the object of the next chapter.

CHAPTER

9

Experiments with Set-labeled Diagrams

Set-labeled diagrams can be used to discretize interval automata. We showed in Chapter 7 that the language of focusing SDs and its sublanguages support the major queries and transformations necessary for compilation-based planning applications. In particular, this implies that discretizing IAs is harmless: SDs can be handled online as efficiently as IAs. It is also interesting for problems involving only enumerated variables, such as our *ObsToMem* or *Telecom* benchmarks. For that reason, we studied in Chapter 8 ways to compile discrete constraint neworks into SDs.

In the present chapter, we outline our experimental manipulation of set-labeled diagrams. We give some details about the framework we implemented [§ 9.1], then present our experimental compilation results [§ 9.2], and finally provide results about the practical use of SDs [§ 9.3].

9.1 Implementation

9.1.1 Experimental Framework

We implemented a library, written in Java, allowing a user to handle set-labeled diagrams. It works in a manner similar to our `IA` library [§ 6.1], relying on a `Get_node` function to incrementally build SDs while applying reduction operations on the fly. Our library is able to handle the eight `SD` sublanguages we defined in Chapter 7. This is done using a system of *graph properties*: the program is able, for example, to recognize that an SD is focusing—and to choose an implementation of the queries and tranformations that fits this specificity. Consider for example an SD

on which a user needs to forget some variables: the basic way to do this is by computing a disjunction of conditioned graphs, but if the SD is focusing, there exists a much more efficient method, which our program can choose automatically.

9.1.2 Set-labeled Diagrams Compiler

There are currently three ways to build an SD within our framework: "by hand", with the `Get_node` function; by discretizing an interval automaton; and by using"CHOCO with a trace". The discretization procedure works as described in Section 7.3.2, computing meshes of the given IA and associating some discrete value with each element of each mesh.

We modified the code of the constraint solver CHOCO for it to build an FSDD while performing a search, as detailed in Section 8.3. In particular, we implemented suproblem caching, that impacts CHOCO's search procedure by preventing it from exploring equivalent subproblems multiple times. We also implemented a number of heuristics guiding the choice of the next variable to branch on. In particular, we implemented the heuristics described by Amilhastre [Ami99], namely **HBW**, **HSBW**, and **MCSInv**, that choose the next variable depending on the variables that have already been chosen and on the variables they are linked to. These three heuristics are *static*, that is, the variable order is computed once and for all, before the start of the compilation. We modified them to make them *dynamic*, that is to say, they choose the next variable depending on the *current* constraint network.

9.1.3 Operations on Set-labeled Diagrams

We implemented a number of operations on SDs, mainly those of polynomial complexity. The available queries are the following: consistency (**CO**) on `FSD`, model checking (**MC**) on all languages, equivalence (**EQ**) on all languages, model extraction (**MX**) on all languages, context extraction (**CX**) on `FSD`, model counting (**CT**) on all languages, and model enumeration (**ME**) on all languages. Available transformations include conditioning (**CD**) on all languages, forgetting (**FO**) on `FSD` and `OSD`, disjunction ($\vee$**C**) on `SD`, `FSD`, and $\texttt{OSD}_<$, conjunction ($\wedge$**C**) on `SD` and `SDD`, binary conjunction ($\wedge$**BC**) on $\texttt{OSD}_<$ and $\texttt{OSDD}_<$, and conjunction with a term ($\wedge$**tC**) on all languages.

In addition, we implemented the "special" queries we defined in Section 6.1.3, namely **CDCO** (consistency once conditioned), **CDMX** (model extraction once conditioned), and **CDFOMX** (model extraction once conditioned and projected). A fourth special query has been added, the purpose of which is to extract the context of a given set of variables in an SD that has been restricted to a given term. We call this query **TRCX**. It can be used for example on a decision policy, to check the values of action variables that remain available after an uncertain observation. It can also be especially useful in configuration applications, in which extracting contexts is especially important.

9.2 Compilation Tests

problem	time (ms)	#nodes	#edges	charac. size	filesize (octets)
*Drone*4-30-3	3397	217	315	337	10 901
*Drone*7-30-3	11 146	625	929	953	28 957
*Drone*10-30-3	69 092	2145	3145	3239	97 941
*Drone*13-30-3	1 574 836	10 222	14 787	14 824	472 074
*ObsToMem*1-2-4-1-2	1796	308	383	416	13 895
*ObsToMem*1-2-4-2-4	2246	279	358	389	12 954
*ObsToMem*2-3-6-2-3	15 843	9515	11 415	11 893	385 461
*ObsToMem*2-3-6-3-6	23 002	4580	5612	5817	188 010
*ObsToMem*3-4-8-1-2	63 790	92 778	104 505	106 389	3 836 096
*ObsToMem*3-4-8-1-3	180 379	125 361	144 563	148 927	5 303 145
*ObsToMem*3-4-8-1-4	438 697	126 015	147 562	153 035	5 377 446
*ObsToMem*3-4-8-1-6	504 718	60 078	73 664	76 173	2 587 457
*ObsToMem*3-4-8-2-4	555 211	128 317	152 098	158 695	5 517 311
*Telecom*3-5-5-32	19 047	1631	2234	2256	68 021
*Telecom*3-5-6-40	34 087	2721	3865	3888	115 796
*Telecom*3-5-6-64	107 933	4661	7390	7413	209 898
*Telecom*4-5-4-33	37 207	2115	3010	3035	89 883
*Telecom*4-5-5-43	119 785	3906	5855	5881	170 737
*Telecom*4-6-5-56	715 574	8405	13 671	13 698	384 139
*Telecom*4-6-6-63	1 324 677	13 763	21 592	21 620	624 958

Table 9.1: Compilation results using “CHOCO with a trace”.

Using “CHOCO with a trace”, we compiled on a standard laptop a number of instances of the *Drone* (in its discrete form), *ObsToMem*, and *Telecom* benchmarks. We used the dynamic **HBW** heuristics and a splitting function that enumerates the current domain; we thus compile free-ordered read-once SDDs in practice. The results can be found in Table 9.1; they include compilation time, number of nodes and edges, characteristic size, and size of a file containing a text version of the SD. All *Drone* instances have a fixed allotted time of 30 units, and 3 balls; the varying parameter is the number of zones. *ObsToMem* instances are labeled with the number of detector lines, the number of COMs, and the number of memory banks; the last two parameters are the maximum number of failures of COMs and memory banks, respectively. The *Telecom* instances are labeled with the number of input channels, the number of amplifiers, the number of output channels, and the number of available paths.

The properties of compiled forms are satisfactory; they are relatively small, and except for the largest instances, could probably be embarked in autonomous systems. However, compilation is longer than expected, given the size of the compiled

problem	**CD** $\vec{s}$	**FO** $\mathcal{S}'$	**MX** $\hookrightarrow\vec{a}$	**CD** $\vec{a}$	**MX** $\hookrightarrow\vec{s}'$	**CDFOMX** $\langle\vec{s},\mathcal{S}'\rangle\hookrightarrow\vec{a}$	**CDMX** $\vec{a}\hookrightarrow\vec{s}'$
*Drone*4-3-30	28	16	0	14	1	1	1
*Drone*7-3-30	99	39	2	21	1	1	2
*Drone*10-3-30	72	11	0	9	0	4	5
*Drone*13-3-30	553	34	1	22	1	74	59
*OTM*1-2-4-1-2	37	10	1	4	0	1	1
*OTM*1-2-4-2-4	110	60	13	24	0	1	1
*OTM*2-3-6-2-3	429	26	2	6	0	12	4
*OTM*2-3-6-3-6	270	78	17	26	0	33	2
*OTM*3-4-8-1-2	5619	71	39	26	0	235	11
*OTM*3-4-8-1-3	10 516	463	74	171	0	127	19
*OTM*3-4-8-1-4	5148	1258	63	156	0	30	2
*OTM*3-4-8-1-6	3160	509	96	1329	0	25	4
*OTM*3-4-8-2-4	6891	68	13	17	0	32	3

Table 9.2: Results for Scenario 1 on several instances of the *Drone* and *ObsToMem* transition relations. All times are in milliseconds.

forms. In Chapter 6, using "RealPaver with a trace", we compiled larger graphs in less time. This issue may come from our implementation of subproblem caching. Our results nevertheless show that this approach is worth being further studied.

9.3 Application Tests

In this section, we provide results about the online use of compiled forms. Similar to what we did in Section 6.3, we simulate realistic sequences of operations corresponding to the nature of the compiled problem.

9.3.1 Simulating Online Use of a Transition Relation

Let us begin by the compiled transition relations, namely the *Drone* and *ObsToMem* instances. We applied the four scenarios we described in Section 6.3.1. The first scenario aims at choosing an action that is executable in the current state, and retrieve one state among those it can lead to. Results are presented in Table 9.2. Scenario 2 is similar, but no action is chosen: we just want to find out a possible next state. Table 9.3 contains results for this simulation. Scenario 3 aims at returning a predecessor state of the current state; finally, the purpose of Scenario 4 is to find a state-action pair that can lead to the current state. Results for Scenarios 3 and 4 can be found in Table 9.4.

Unsurprisingly, the general profile of each operation is similar to what we noticed in Section 6.3.1: the hardest operation is the first conditioning, followed by

problem	**CD** $\vec{s}$	**FO** $\mathcal{A}$	**MX** $\hookrightarrow \vec{s}'$	**CDFOMX** $\langle \vec{s}, \mathcal{A} \rangle \hookrightarrow \vec{s}'$
*Drone*4-3-30	28	11	1	0
*Drone*7-3-30	99	40	4	1
*Drone*10-3-30	72	14	1	5
*Drone*13-3-30	553	50	5	67
*ObsToMem*1-2-4-1-2	37	6	0	1
*ObsToMem*1-2-4-2-4	110	54	1	1
*ObsToMem*2-3-6-2-3	429	27	2	22
*ObsToMem*2-3-6-3-6	270	54	1	10
*ObsToMem*3-4-8-1-2	5619	146	1	117
*ObsToMem*3-4-8-1-3	10 516	336	1	134
*ObsToMem*3-4-8-1-4	5148	285	1	38
*ObsToMem*3-4-8-1-6	3160	345	1	19
*ObsToMem*3-4-8-2-4	6891	43	1	32

Table 9.3: Results for Scenario 2 on several instances of the *Drone* and *ObsToMem* transition relations. All times are in milliseconds.

problem	**CD** $\vec{s}'$	**FO** $\mathcal{A}$	**MX** $\hookrightarrow \vec{s}$	**CDFOMX** $\langle \vec{s}', \mathcal{A} \rangle \hookrightarrow \vec{s}$	**CD** $\vec{s}'$	**MX** $\hookrightarrow \vec{s}.\vec{a}$	**CDMX** $\vec{s}' \hookrightarrow \vec{s}.\vec{a}$
*Drone*4-3-30	24	6	1	4	24	1	1
*Drone*7-3-30	90	27	4	2	90	7	1
*Drone*10-3-30	122	38	3	1	122	7	1
*Drone*13-3-30	1001	476	57	7	1001	64	4
*OTM*1-2-4-1-2	43	22	2	0	43	3	0
*OTM*1-2-4-2-4	137	100	5	1	137	17	3
*OTM*2-3-6-2-3	773	378	29	1	773	87	1
*OTM*2-3-6-3-6	394	196	15	1	394	54	1
*OTM*3-4-8-1-2	8223	2728	175	1	8223	877	2
*OTM*3-4-8-1-3	15 413	10 455	512	2	15 413	3577	3
*OTM*3-4-8-1-4	9908	6782	319	0	9908	1791	1
*OTM*3-4-8-1-6	6567	4376	261	2	6567	1346	1
*OTM*3-4-8-2-4	9739	3263	309	0	9739	1144	1

Table 9.4: Results for Scenarios 3 and 4 on several instances of the *Drone* and *ObsToMem* transition relations. All times are in milliseconds.

problem	**CD** $\vec{s}$	**MX** $\hookrightarrow c$	**CDMX** $\vec{s}\hookrightarrow c$	**CT** $\hookrightarrow \#c$
*Telecom*3-5-5-32	68	0	3	31
*Telecom*3-5-6-40	105	0	7	57
*Telecom*3-5-6-64	408	1	15	170
*Telecom*4-5-4-33	110	1	1	37
*Telecom*4-5-5-43	324	1	7	145
*Telecom*4-6-5-56	489	3	8	321
*Telecom*4-6-6-63	273	0	7	186

Table 9.5: Results for the online use of several instances of the *Telecom* problem. All times are in milliseconds.

the forgetting in Scenario 3. Model extraction is fast, independantly from the size of the graph. The special queries **CDMX** and **CDFOMX** perform well. Note however that while they are slower than model extraction in Scenarios 1 and 2 (forward lookahead in the state-transition system), it is the opposite in Scenarios 3 and 4 (backward lookahead in the state-transition system).

All in all, basic handling of transition relations compiled as set-labeled diagrams seems possible in practice; it would be interesting to examine whether more complex uses remain workable, such as the use of `FSDD` to build a decision policy using a planning as model-checking approach [GT99].

9.3.2 Simulating Online Use of the *Telecom* Benchmark

The problem underlying the *Telecom* benchmark is similar to that of *ObsToMem*, but the latter is encoded as a transition relation ("if this COM fails, it cannot be used in the next state"), whereas the former is encoded as a set of valid configurations. An online use of this constraint network is to specify an observed state of the system—that is, such input channel is active or not, such amplifier is out of order—and to extract one valid configuration, that is, a path linking each active input channel to an available amplifier and output channel. Once again, this scenario combines a conditioning and a model extraction. On this benchmark, it can also be interesting to count the possible configurations; this can be used for example as an indicator of the overall health state of the system. This boils down to a standard model counting query (**CT**). We present experimental results for these scenarios in Table 9.5.

Like for the transition relations in the previous section, simulations for the first scenario are satisfactory. Using such a compiled form to indicate valid configurations to an online controller seems a viable idea, especially if it uses the dedicated **CDMX** query, which is orders of magnitude faster than the combination of **CD** and **MX**. Counting valid configurations also turns out to be a tractable online task.

Conclusion

In this thesis, we studied the application of knowledge compilation to the control of aeronautical and spatial autonomous systems. Looking into the literature, we found no work addressing the compilation of problems involving continuous variables, whereas such problems are common when dealing with realistic autonomous systems. Our approach thus consisted in defining and studying target compilation languages:

- allowing the representation of Boolean functions on both continuous and discrete variables;
- supporting in polytime the queries and transformations we identified as major for compilation-based planning;
- being as compact as possible—and therefore as general as possible.

This conclusion sums up the contributions and limitations of our work, and outlines some perspectives for future work.

Contributions

A State-of-the-Art Knowledge Compilation Map

We extended the existing knowledge compilation map framework [GK$^+$95, DM02, FM07, FM09] so that it includes representation languages for any kind of function. In particular, we defined a general language that encompasses all graph-based languages representing Boolean functions, and allows us to easily extend existing languages from Boolean variables to discrete or continuous variables—defining for example MDD as a particular sublanguage of OBDD. We also proposed an extension to non-Boolean variables of the classical queries and transformations on Boolean languages. Casting known results in our extended framework, we drew up the knowledge compilation map of Boolean graph-based languages; it is however not complete yet.

Knowledge Compilation for Planning

We studied the literature about the application of knowledge compilation to planning problems, and provided an overview of what has already been done. We proposed new queries and transformations that are useful in planning, but also in diagnostic and configuration applications. We identified a number of important queries and transformations that a target compilation language should satisfy to be used online by an autonomous system. We found no study about the compilation of problems involving continuous variables, and decided to examine new target languages allowing this.

A New Target Language for Real Variables

We defined the language of *interval automata (IAs)*, that generalizes binary decision diagrams in multiple ways:

- variables can be discrete or continuous;
- edge labels are intervals, not singletons;
- decision nodes are not exclusive, that is, edge labels are not necessarily disjoint;
- pure disjunctive nodes are allowed.

This generalization implied a modification of the classical *reduction* procedure of binary decision diagrams.

We also identified a sublanguage of `IA`, namely that of *focusing interval automata*, that satisfies several important queries and transformations, among which **CO**, **MX**, **CD**, and **FO**, making it a good target language for online planning.

After having drawn up the knowledge compilation map of IAs and FIAs, including succinctness results, satisfaction of queries, and satisfaction of transformations, we described methods to compile problems into `FIA`, notably by *tracing* the search tree of an interval-based constraint solver, à la "DPLL with a trace" [HD05a].

Experimenting FIAs

We implemented a framework for handling interval automata, including a unique node table to process isomorphic nodes, and various queries and transformations, notably those that are used for the control of autonomous systems. We also implemented a prototype of the compiler "RealPaver with a trace". Our first experiments showed that the practical use of FIAs for controlling autonomous systems is conceivable: compiled forms for our benchmarks have a reasonable size, allowing them to be embarked; and execution of operations, even if they are not optimized, is fast enough for an online use to be possible.

However, the memory space that is gained when compared to binary decision diagrams—since no discretization is needed—is most likely compensated by real

numbers taking more memory space than integers. Nevertheless, we remarked that interval automata define an intrinsic discretization of the variable domains. This means that it is actually possible to translate IAs into a target language over *discrete* variables, without losing information, or increasing the size of the compiled form. We thus decided to study the family of languages corresponding to discrete IAs.

A New Target Language for Discrete Variables

We defined the language of *set-labeled diagrams (SDs)*, that generalize binary decision diagrams and multivalued decision diagrams in multiple ways:

- variables are discrete, but can have an unbounded domain;
- edge labels are unions of intervals, not singletons;
- decision nodes are not exclusive, that is, edge labels are not necessarily disjoint;
- pure disjunctive nodes are allowed.

Once again, we adapted the reduction procedure to these new specifications, and identified sublanguages of SD, among which those of *focusing* set-labeled diagrams (FSD), of *ordered* ones (OSD), and their counterparts satisfying *exclusive decision*: SDD, FSDD, and OSDD.

After having formally shown that FIAs can be transformed into FSDs of similar size in polytime, we drew up the knowledge compilation map of all languages in the SD family, with their relative succinctness, and the queries and transformations they support in polytime. In particular, FSD satisfies the queries and transformations satisfied by FIA, which is not surprising but ensures that handling FIAs transformed into FSDs online is not harder than directly handling FIAs. FSDD and OSDD can also be interesting for some applications, depending on the set of queries and transformations required to be tractable.

We described a method to compile problems into FSDD or OSDD, adapted from the "DPLL with a trace" approach [HD05a], which consists in using the trace of the search of a constraint solver. Using various parameters for variable selection and domain splitting of the solver, we can obtain different compiled forms.

Experimenting FSDs

We implemented a framework allowing one to handle all languages in the SD family, including most tractable operations in the knowledge compilation map. We implemented a "CHOCO with a trace" compiler, which is able to output FSDDs or OSDDs, depending on the user's choice. Our first experiments are promising: going from FIAs to FSDs decreases memory space without increasing operation duration. Moreover, compiling discrete problems into FSDDs or OSDDs seems possible in practice, and handling the resulting compiled forms online is not an issue. However, compilation time can be problematic for large problems. In addition,

we only explored a little part of the possibilities of “CHOCO with a trace”; for instance, we did not show the impact of variable choice or domain splitting heuristics on the size of compiled forms.

Perspectives

A lot of work remains to be done about “CHOCO with a trace”. The framework allows one to build FSDDs with variable repetition and no static order; using heuristics for variable choice or domain splitting could possibly give rise to better results. A possibility is to try to maximize the use of the cache: it can be done for example by looking one step ahead and choosing the variable that minimizes the number of new search nodes. As a side note, the question of the compilation of pure, non-deterministic FSDs is left open—as is the case for pure DNNFs.

Further experimentation about planning with IAs and SDs includes the actual building of a policy (e.g., using a strong planning as model-checking algorithm [GT99]) and comparisons with more state-of-the-art languages.

From a theoretical point of view, our work also raised questions about representations: as we showed, any FIA can be transformed in polytime into an “equivalent” FSD. To state this fact, we had to introduce a specific notion of equivalence, which is different from the classical equivalence of Boolean functions. Indeed, variables in the `FIA` and `FSD` framework are completely different; in a way, the equivalence we defined is an “equivalence modulo some variable change”. Future work includes studies about such relations between different representation languages.

APPENDIX

A

Benchmark Specifications

A.1 *Drone* problem

This problem deals with a competition drone having to achieve different goals on a number of zones.

There are three different kinds of goal:

- identifying the target of a given zone, by doing “eight”-shaped flying maneuvers above it
- localizing the target in a given zone, by scanning it
- dropping a ball on the target of a given zone

Each zone contains at most one goal (thus, no target has to be both identified and localized, for example). There is a special “home” zone where the drone takes off and lands; it cannot land anywhere else without losing the competition.

Data

The following data define an instance of the problem. Let us begin with the integer constants:

- an integer *nbZones*, the total number of zones;
- an integer *nbZonesId*, the total number of zones containing a target to identify;
- an integer *nbZonesLoc*, the total number of zones containing a target to localize;

- an integer *nbZonesDrop*, the total number of zones containing a target to touch;
- an integer *nbBallsMax*, the maximal number of balls that the drone can carry;
- an integer *allottedTime*, the number of minutes allotted to the mission;
- a sequence of integers $\langle \mathit{zoneId}_n \rangle_{1 \leqslant n \leqslant \mathit{nbZonesId}}$, all different and between 0 and $\mathit{nbZones} - 1$, representing the zones containing a target to identify;
- a sequence of integers $\langle \mathit{zoneLoc}_n \rangle_{1 \leqslant n \leqslant \mathit{nbZonesLoc}}$, all different and between 0 and $\mathit{nbZones} - 1$, representing the zones containing a target to localize;
- a sequence of integers $\langle \mathit{zoneDrop}_n \rangle_{1 \leqslant n \leqslant \mathit{nbZonesDrop}}$, all different and between 0 and $\mathit{nbZones} - 1$, representing the zones containing a target to touch;

Since each zone contains at most one target, it is obvious that $\mathit{nbZonesId} + \mathit{nbZonesLoc} + \mathit{nbZonesDrop} \leqslant \mathit{nbZones}$, and that one integer cannot belong to two distinct sequences. The next data are constants; their type (real or integer) depends on the version of the benchmark—continuous or enumerated. They are defined as follows:

- a value *idDuration*, the number of minutes necessary to identify a target;
- a value *locDuration*, the number of minutes necessary to localize a target;
- a value *dropDuration*, the number of minutes necessary to touch a target;
- a value *toffDuration*, the number of minutes necessary to take off;
- a value *landDuration*, the number of minutes necessary to land;
- a table of values $\langle \mathit{gotoDuration}_{i,j} \rangle_{1 \leqslant i \leqslant \mathit{nbZones}, 1 \leqslant j \leqslant \mathit{nbZones}}$, indicating the time (in minutes) necessary for the drone to go from zone i to zone j.

State Variables

State variables are the following:

- a Boolean *flying* indicating whether the drone is flying;
- an integer *zone* representing the zone it is in;
- an integer *nbBalls*, the number of remaining balls;
- a sequence of Boolean values $\langle \mathit{goalAch}_n \rangle_{0 \leqslant n < \mathit{nbZones}}$, indicating for each zone whether its corresponding goal has been achieved.

There is also a variable corresponding to the current number of remaining minutes, *remTime*. This variable is real-valued or integer-valued depending on the version of the problem.

Action Variables

Action variables are the following:

- a Boolean value *TOFF*, true if the drone takes off;
- a Boolean value *LAND*, true if the drone lands;
- a Boolean value *EIGHT*, true if the drone identifies a target (making an "eight" above);
- a Boolean value *SCAN*, true if the drone localizes a target (scanning the zone);
- a Boolean value *DROP*, true if the drone drops a ball;
- a Boolean value *GOTO*, true if the drone goes from a zone to another;
- an integer *zoneGOTO*, representing the zone to which the drone heads if *GOTO* is true.

Preconditions

The set $P(S, A)$ contains constraints deciding which decisions are possible in the current state.

- The following constraint ($\oplus$ being the "xor" operator),

$$\mathit{TOFF} \oplus \mathit{LAND} \oplus \mathit{EIGHT} \oplus \mathit{SCAN} \oplus \mathit{DROP} \oplus \mathit{GOTO},$$

  forbids that more than one decision be made at the same time.

- There cannot remain more time than the allotted time:

$$\left[\mathit{remTime} \leqslant \mathit{allottedTime}\right].$$

- There cannot remain more balls than the maximum number of balls:

$$\left[\mathit{nbBalls} \leqslant \mathit{nbBallsMax}\right].$$

- Precondition to the takeoff: the drone must be landed at home, and have enough time to take off, i.e.,

$$\mathit{TOFF} \rightarrow \left(\left[\mathit{zone} = \mathit{home}\right] \wedge \neg \mathit{flying} \wedge \left[\mathit{remTime} \geqslant \mathit{toffDuration}\right]\right).$$

- Precondition to the landing: the drone must be flying at home, and have enough time to land, i.e.,

$$\mathit{LAND} \rightarrow \left(\left[\mathit{zone} = \mathit{home}\right] \wedge \mathit{flying} \wedge \left[\mathit{remTime} \geqslant \mathit{landDuration}\right]\right).$$

- Precondition to the identification: the drone must be flying and have enough time, i.e.,

$$\mathit{EIGHT} \rightarrow \left(\mathit{flying} \wedge \left[\mathit{remTime} \geqslant \mathit{idDuration}\right]\right),$$

  and it must be on a zone containing a target to identify, i.e.,

$$\mathit{EIGHT} \rightarrow \left[\mathit{zone} \in \{\mathit{zoneId}_1, \ldots, \mathit{zoneId}_{\mathit{nbZonesId}}\}\right].$$

- Precondition to the localization: the drone must be flying and have enough time, i.e.,

$$\mathit{SCAN} \rightarrow \left(\mathit{flying} \wedge \left[\mathit{remTime} \geqslant \mathit{locDuration}\right]\right),$$

  and it must be on a zone containing a target to localize, i.e.,

$$\mathit{SCAN} \rightarrow \left[\mathit{zone} \in \{\mathit{zoneLoc}_1, \ldots, \mathit{zoneLoc}_{\mathit{nbZonesLoc}}\}\right].$$

- Precondition to the dropping: the drone must be flying, have enough time, and have enough balls, i.e.,

$$\mathit{DROP} \rightarrow \left(\mathit{flying} \wedge \left[\mathit{remTime} \geqslant \mathit{dropDuration}\right] \wedge \left[\mathit{nbBalls} > 0\right]\right),$$

  and it must be on a zone containing a target to touch, i.e.,

$$\mathit{DROP} \rightarrow \left[\mathit{zone} \in \{\mathit{zoneDrop}_1, \ldots, \mathit{zoneDrop}_{\mathit{nbZonesDrop}}\}\right].$$

- Precondition to the moving: the drone must be flying and have enough time to go to the specified zone, i.e.,

$$\mathit{GOTO} \rightarrow \left(\mathit{flying} \wedge \left[\mathit{remTime} \geqslant \mathit{gotoDuration}_{\mathit{zone},\mathit{zoneGOTO}}\right]\right).$$

Effects

The set $E(S, A, S')$ contains constraints indicating the resulting state, according to the previous state and the action made. First, the following constraints describe how the state changes when a given action is made.

- Effects of a takeoff: the drone is flying, and the duration of a takeoff has been removed from the remaining time, i.e.,

$$\mathit{TOFF} \rightarrow \left(\mathit{flying}' \wedge \left[\mathit{remTime}' = \mathit{remTime} - \mathit{toffDuration}\right]\right).$$

- Effects of a landing: the drone is landed, and the duration of a landing has been removed from the remaining time, i.e.,

$$\mathit{LAND} \rightarrow \left(\neg\mathit{flying}' \wedge \left[\mathit{remTime}' = \mathit{remTime} - \mathit{landDuration}\right]\right).$$

- Effects of an identification: the duration of an identification has been removed from the remaining time, i.e.,

$$EIGHT \rightarrow \left[remTime' = remTime - idDuration\right],$$

and the goal corresponding to this zone has been achieved, i.e.,

$$EIGHT \rightarrow goalAch'_{zone}.$$

- Effects of an localization: the duration of a localization has been removed from the remaining time, i.e.,

$$SCAN \rightarrow \left[remTime' = remTime - locDuration\right],$$

and the goal corresponding to this zone has been achieved, i.e.,

$$SCAN \rightarrow goalAch'_{zone}.$$

- Effects of a dropping: the drone has lost a ball, and the duration of a dropping has been removed from the remaining time, i.e.,

$$DROP \rightarrow \left(\left[nbBalls' = nbBalls - 1\right] \wedge \left[remTime' = remTime - dropDuration\right]\right),$$

and goal corresponding to this zone has been achieved, i.e.,

$$DROP \rightarrow goalAch'_{zone}.$$

- Effects of a move: the drone is in the specified zone, and the duration of the journey has been removed from the remaining time, i.e.,

$$GOTO \rightarrow \left(\left[zone' = zoneGOTO\right] \wedge \left[remTime' = remTime - gotoDuration_{zone,zoneGOTO}\right]\right).$$

Conditions of a State Change

These constraints are also in $E(S, A, S')$, but they specify which decisions can modify a given state variable.

- The flying state can only change when the drone takes off or lands:

$$\neg\left(flying' \leftrightarrow flying\right) \rightarrow (TOFF \vee LAND).$$

- The zone can only change when the drone moves:

$$\neg\left[zone' = zone\right] \rightarrow GOTO.$$

- The number of balls can only change when the drone drops one:

$$\neg\left[nbBalls' = nbBalls\right] \rightarrow DROP.$$

- A goal can only be achieved if the corresponding decision has been made and the drone is in the corresponding zone:

$$\begin{aligned}\neg\left(goalAch'_{zone} \leftrightarrow goalAch_{zone}\right) \rightarrow \\ \Big(\left(EIGHT \wedge \left[zone \in \{zoneId_1, \ldots, zoneId_{nbZonesId}\}\right]\right) \\ \vee \left(SCAN \wedge \left[zone \in \{zoneLoc_1, \ldots, zoneLoc_{nbZonesLoc}\}\right]\right) \\ \vee \left(DROP \wedge \left[zone \in \{zoneDrop_1, \ldots, zoneDrop_{nbZonesDrop}\}\right]\right)\Big).\end{aligned}$$

- In zones containing no target, the goal is considered as achieved from the beginning:

$$\begin{aligned}\Big(\left[zone \in \{zoneId_1, \ldots, zoneId_{nbZonesId}\}\right] \\ \wedge \left[zone \in \{zoneLoc_1, \ldots, zoneLoc_{nbZonesLoc}\}\right] \\ \wedge \left[zone \in \{zoneDrop_1, \ldots, zoneDrop_{nbZonesDrop}\}\right]\Big) \\ \rightarrow \left(goalAch_{zone} \wedge goalAch'_{zone}\right).\end{aligned}$$

Goal of the Mission

The goal of the mission is given by this constraint:

$$\neg flying' \wedge \left[zone' = home\right] \wedge \bigwedge_{k=0}^{nbZones-1} goalAch'_k.$$

At the end, the drone must be landed at home, and having achieved all zone goals.

A.2 Telecom

Data

The problem is defined by

- a set of reception channels, numbered from 1 to N_R;
- a set of signal amplifiers, numbered from 1 to N_A;
- a set of emission channels, numbered from 1 to N_E;

- a set of paths numbered from 1 to N_P; each path p links a reception channel $R(p) \in [1, \ldots, N_R]$ to an emission channel $E(p) \in [1, \ldots, N_E]$ through an amplifier $A(p) \in [1, \ldots, N_A]$;
- a list *Incomp* of couples of distinct paths $\langle p, p' \rangle \in [1, \ldots, N_P] \times [1, \ldots, N_P]$ that are incompatible (that is, they use the same amplifier or the same emission channel).

State Variables

The state variables are the following:

- for each reception channel $r \in [1, \ldots, N_R]$, a Boolean variable *active*(r) indicating whether the channel is active;
- for each amplifier $a \in [1, \ldots, N_A]$, a Boolean variable *afail*(a) indicating whether the amplifier is out of order;
- for each emission channel $e \in [1, \ldots, N_E]$, a Boolean variable *efail*(a) indicating whether the channel is out of order.

Decision Variables

The decision variables are as follows: to each reception channel $r \in [1, \ldots, N_R]$ corresponds a variable *path*(r) of domain $[0, \ldots, N_P]$, indicating the number of the path associated with r (value 0 for channels associated with no path, such as non-active channels).

Constraints

The path is 0 by default for non-active reception chanels, and each active reception channel must be associated with a path:

$$\forall r \in [1, \ldots, N_R], \quad \neg \mathit{active}(r) \leftrightarrow \big[\mathit{path}(r) = 0\big].$$

The path used by an active reception channel r must start from r:

$$\forall r \in [1, \ldots, N_R], \quad \mathit{active}(r) \rightarrow \big[R(\mathit{path}(r)) = r\big].$$

Every active reception channel must be connected to a working emission channel, via a working amplifier:

$$\forall r \in [1, \ldots, N_R], \quad \mathit{active}(r) \rightarrow \neg \mathit{afail}(A(\mathit{path}(r))),$$
$$\forall r \in [1, \ldots, N_R], \quad \mathit{active}(r) \rightarrow \neg \mathit{efail}(E(\mathit{path}(r))).$$

Incompatible paths are not used at the same time:

$$\forall \langle r, r' \rangle \in [1, \ldots, N_R]^2, \quad \langle \mathit{path}(r), \mathit{path}(r') \rangle \notin \mathit{Incomp}.$$

Bibliography

[Ake78] Sheldon B. Akers. “Binary Decision Diagrams”. In: *IEEE Transactions on Computers* 27.6 (1978), pp. 509–516.

[Ami99] Jérôme Amilhastre. “Représentation par automate d’ensemble de solutions de problèmes de satisfaction de contraintes”. French. PhD thesis. Université Montpellier II, 1999.

[AFM02] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. “Consistency Restoration and Explanations in Dynamic CSPs — Application to Configuration”. In: *Artificial Intelligence Journal* 135.1–2 (2002), pp. 199–234.

[AH97] Henrik Reif Andersen and Henrik Hulgaard. “Boolean Expression Diagrams”. In: *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS), Warsaw, Poland*. 1997, pp. 88–98.

[SHB00] Robert St-Aubin, Jesse Hoey, and Craig Boutilier. “APRICODD: Approximate Policy Construction Using Decision Diagrams”. In: *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS)*. 2000, pp. 1089–1095.

[BF$^+$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”. In: *Formal Methods in System Design* 10.2/3 (1997), pp. 171–206.

[Bar03] Anthony Barrett. “Domain Compilation for Embedded Real-Time Planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 2003.

[Bar77] Jon Barwise, ed. *Handbook of Mathematical Logic*. North-Holland, 1977.

[BVC07] Grégory Beaumet, Gérard Verfaillie, and Marie-Claire Charmeau. “Estimation of the Minimal Duration of an Attitude Change for an Autonomous Agile Earth-Observing Satellite”. In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2007, pp. 3–17.

[Bel57] Richard Bellman. “A Markovian Decision Process”. In: *Indiana University Mathematics Journal* 6 (4 1957), pp. 679–684. ISSN: 0022-2518.

[BYD07] Salem Benferhat, Safa Yahi, and Habiba Drias. “On the Compilation of Stratified Belief Bases under Linear and Possibilistic Logic Policies”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2007, pp. 2425–2430.

[Bey08] Dirk Beyer. *CrocoPat, a Tool for Simple and Efficient Relational Programming*. 2008. URL: http://www.cs.sfu.ca/~dbeyer/CrocoPat/.

[Bie00] Armin Biere. *ABCD : Armin Biere's Compact Decision Diagram BDD library, release 0.3*. 2000. URL: http://fmv.jku.at/abcd/.

[BG01] Blai Bonet and Hector Geffner. “Planning as Heuristic Search”. In: *Artificial Intelligence Journal* 129.1–2 (2001), pp. 5–33.

[BG06] Blai Bonet and Hector Geffner. “Heuristics for Planning with Penalties and Rewards using Compiled Knowledge”. In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 2006, pp. 452–462.

[Bry86] Randall E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* 35.8 (1986), pp. 677–691.

[BC+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. “Symbolic Model Checking: 10^{20} States and Beyond”. In: *Information and Computation* 98.2 (1992), pp. 142–170.

[CD97] Marco Cadoli and Francesco M. Donini. “A Survey on Knowledge Compilation”. In: *AI Communications* 10.3–4 (1997), pp. 137–150.

[CD+96] Marco Cadoli, Francesco M. Donini, Paolo Liberatore, and Marco Schaerf. “Feasibility and Unfeasibility of Off-Line Processing”. In: *Proceedings of the Israel Symposium on Theory of Computing Systems (ISTCS)*. 1996, pp. 100–109.

[CD+00] Marco Cadoli, Francesco M. Donini, Paolo Liberatore, and Marco Schaerf. “Space Efficiency of Propositional Knowledge Representation Formalisms”. In: *Journal of Artificial Intelligence Research (JAIR)* 13 (2000), pp. 1–31.

[CGT03] Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. “SAT-based Planning in Complex Domains: Concurrency, Constraints and Nondeterminism”. In: *Artificial Intelligence Journal* 147.1–2 (2003), pp. 85–117.

[CHO10] CHOCO Team. *CHOCO: An Open Source Java Constraint Programming Library*. Research report 10-02-INFO. École des Mines de Nantes, 2010. URL: http://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf.

[CC+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. “NuSMV 2: An OpenSource Tool for Symbolic Model Checking”. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2002, pp. 359–364.

[CC+99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. “NUSMV: A New Symbolic Model Verifier”. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 1999, pp. 495–499.

[CG+97] Alessandro Cimatti, Fausto Giunchiglia, Enrico Giunchiglia, and Paolo Traverso. “Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$”. In: *Proceedings of the European Conference on Planning (ECP)*. 1997, pp. 130–142.

[CRT98a] Alessandro Cimatti, Marco Roveri, and Paolo Traverso. “Automatic OBDD-Based Generation of Universal Plans in Non-Deterministic Domains”. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 1998, pp. 875–881.

[CRT98b] Alessandro Cimatti, Marco Roveri, and Paolo Traverso. “Strong Planning in Non-Deterministic Domains Via Model Checking”. In: *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS)*. 1998, pp. 36–43.

[CF+93] Edmund M. Clarke, Masahiro Fujita, Patrick C. McGeer, Kenneth L. McMillan, Jerry Chih-Yuan Yang, and Xudong Zhao. “Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation”. In: *Proceedings of the International Workshop on Logic Synthesis*. 1993, 6a:1–15.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999. ISBN: 9780262032704.

[CL+01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001. ISBN: 0-262-03293-7, 0-07-013151-1.

[CK+07] William Cushing, Subbarao Kambhampati, Mausam, and Daniel S. Weld. “When is Temporal Planning Really Temporal?” In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2007, pp. 1852–1859.

[DLPT02] Ugo Dal Lago, Marco Pistore, and Paolo Traverso. “Planning with a Language for Extended Goals”. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 2002, pp. 447–454.

[Dar98] Adnan Darwiche. “Model-Based Diagnosis Using Structured System Descriptions”. In: *Journal of Artificial Intelligence Research (JAIR)* 8 (1998), pp. 165–222.

[Dar01a] Adnan Darwiche. “Decomposable Negation Normal Form”. In: *Journal of the ACM* 48.4 (2001), pp. 608–647. ISSN: 0004-5411.

[Dar01b] Adnan Darwiche. “On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision”. In: *Journal of Applied Non-Classical Logics* 11.1–2 (2001), pp. 11–34.

[Dar03] Adnan Darwiche. “A Differential Approach to Inference in Bayesian Networks”. In: *Journal of the ACM* 50.3 (2003), pp. 280–305.

[Dar04] Adnan Darwiche. *C2D: CNF to d-DNNF Compiler*. 2004. URL: http://reasoning.cs.ucla.edu/c2d/.

[DC07] Adnan Darwiche and Mark Chavira. *ACE, an Arithmetic Circuit Compiler*. 2007. URL: http://reasoning.cs.ucla.edu/ace/.

[DM02] Adnan Darwiche and Pierre Marquis. “A Knowledge Compilation Map”. In: *Journal of Artificial Intelligence Research (JAIR)* 17 (2002), pp. 229–264.

[DM04] Adnan Darwiche and Pierre Marquis. “Compiling Propositional Weighted Bases”. In: *Artificial Intelligence Journal* 157.1–2 (2004), pp. 81–113.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. “A Machine Program for Theorem-Proving”. In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[De 90] Johan De Kleer. “Compiling Devices and Processes”. In: *Proceedings of the 4th International Workshop on Qualitative Physics*. 1990.

[DHW94] Denise Draper, Steve Hanks, and Daniel S. Weld. “Probabilistic Planning with Information Gathering and Contingent Execution”. In: *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS)*. 1994, pp. 31–36.

[Eme90] E. Allen Emerson. “Temporal and Modal Logic”. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. 1990, pp. 995–1072.

[EHN96] Kutluhan Erol, James A. Hendler, and Dana S. Nau. “Complexity Results for HTN Planning”. In: *Annals of Mathematics and Artificial Intelligence* 18.1 (1996), pp. 69–93.

[FM08a] Hélène Fargier and Pierre Marquis. “Extending the Knowledge Compilation Map: Closure Principles”. In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. 2008, pp. 50–54.

[FM08b] Hélène Fargier and Pierre Marquis. “Extending the Knowledge Compilation Map: Krom, Horn, Affine and Beyond”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2008, pp. 442–447.

[FM09] Hélène Fargier and Pierre Marquis. “Knowledge Compilation Properties of Trees-of-BDDs, Revisited”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2009, pp. 772–777.

[FV04] Hélène Fargier and Marie-Catherine Vilarem. “Compiling CSPs into Tree-Driven Automata for Interactive Solving”. In: *Constraints* 9.4 (2004), pp. 263–287.

[FM06] Hélène Fargier and Pierre Marquis. “On the Use of Partially Ordered Decision Graphs in Knowledge Compilation and Quantified Boolean Formulae”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2006.

[FM07] Hélène Fargier and Pierre Marquis. “On Valued Negation Normal Form Formulas”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2007, pp. 360–365.

[FG00] Paolo Ferraris and Enrico Giunchiglia. “Planning as Satisfiability in Nondeterministic Domains”. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 2000, pp. 748–753.

[FN71] Richard Fikes and Nils J. Nilsson. “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1971, pp. 608–620.

[FL06] Maria Fox and Derek Long. “Modelling Mixed Discrete-Continuous Domains for Planning”. In: *Journal of Artificial Intelligence Research (JAIR)* 27 (2006), pp. 235–297.

[Gef11] Hector Geffner. *Advanced Introduction to Planning: Models and Methods*. Tutorial at the International Joint Conference on Artificial Intelligence (IJCAI). 2011. URL: http://www.dtic.upf.edu/~hgeffner/tutorial-planning-ijcai-2011.html.

[GM94] Jordan Gergov and Christoph Meinel. “Efficient Boolean Manipulation with OBDD’s Can Be Extended to FBDD’s”. In: *IEEE Transactions on Computers* 43.10 (1994), pp. 1197–1209.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558608567.

[Gib85] Alan M. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985. ISBN: 9780521288811.

[GMS98] Enrico Giunchiglia, Alessandro Massarotto, and Roberto Sebastiani. "Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability". In: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 1998, pp. 948–953.

[GT99] Fausto Giunchiglia and Paolo Traverso. "Planning as Model Checking". In: *Proceedings of the European Conference on Planning (ECP)*. 1999, pp. 1–20.

[GK+95] Goran Gogic, Henry A. Kautz, Christos H. Papadimitriou, and Bart Selman. "The Comparative Linguistics of Knowledge Representation". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1995, pp. 862–869.

[GB06] Laurent Granvilliers and Frédéric Benhamou. "RealPaver: an Interval Solver Using Constraint Satisfaction Techniques". In: *ACM Transactions on Mathematical Software (TOMS)* 32.1 (2006), pp. 138–156.

[GV08] Orna Grumberg and Helmut Veith, eds. *25 Years of Model Checking—History, Achievements, Perspectives*. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008. ISBN: 978-3-540-69849-4.

[HH+08] Tarik Hadzic, John N. Hooker, Barry O'Sullivan, and Peter Tiedemann. "Approximate Compilation of Constraints into Multivalued Decision Diagrams". In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2008, pp. 448–462.

[HJA07] Tarik Hadzic, Rune Moller Jensen, and Henrik Reif Andersen. "Calculating Valid Domains for BDD-Based Interactive Configuration". In: *The Computing Research Repository (CoRR)* abs/0704.1394 (2007).

[HCK92] Walter Hamscher, Luca Console, and Johan de Kleer, eds. *Readings in Model-Based Diagnosis*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1-55860-249-6.

[HG00] Patrik Haslum and Hector Geffner. "Admissible Heuristics for Optimal Planning". In: *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS)*. 2000, pp. 140–149.

[HS+99] Jesse Hoey, Robert St-Aubin, Alan J. Hu, and Craig Boutilier. "SPUDD: Stochastic Planning Using Decision Diagrams". In: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*. 1999, pp. 279–288.

[HN01] Jörg Hoffmann and Bernhard Nebel. “The FF Planning System: Fast Plan Generation Through Heuristic Search”. In: *Journal of Artificial Intelligence Research (JAIR)* 14 (2001), pp. 253–302.

[HD04] Jinbo Huang and Adnan Darwiche. “Using DPLL for Efficient OBDD Construction”. In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2004, pp. 157–172.

[HD05a] Jinbo Huang and Adnan Darwiche. “DPLL with a Trace: From SAT to Knowledge Compilation”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2005, pp. 156–162.

[HD05b] Jinbo Huang and Adnan Darwiche. “On Compiling System Models for Faster and More Scalable Diagnosis”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2005, pp. 300–306.

[JKK09] Saket Joshi, Kristian Kersting, and Roni Khardon. “Generalized First Order Decision Diagrams for First Order Markov Decision Processes”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2009, pp. 1916–1921.

[KMS96] Henry A. Kautz, David A. McAllester, and Bart Selman. “Encoding Plans in Propositional Logic”. In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 1996, pp. 374–384.

[KS92a] Henry A. Kautz and Bart Selman. “Forming Concepts for Fast Inference”. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. San Jose (CA), 1992, pp. 786–793.

[KS92b] Henry A. Kautz and Bart Selman. “Planning as Satisfiability”. In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. 1992, pp. 359–363.

[Kov11] Dániel L. Kovács. *Complete BNF Description of PDDL 3.1*. 2011. URL: http://www.plg.inf.uc3m.es/ipc2011-deterministic/Resources?action=AttachFile&do=get&target=kovacs-pddl-3.1-2011.pdf.

[LS92] Yung-Te Lai and Sarma Sastry. “Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification”. In: *Proceedings of the 29th Design Automation Conference (DAC)*. 1992, pp. 608–613.

[LS+07] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. “Transposition Tables for Constraint Satisfaction”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2007, pp. 243–248.

[LS06] Christophe Lecoutre and Radoslaw Szymanek. “Generalized Arc Consistency for Positive Table Constraints”. In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2006, pp. 284–298.

[Lee59] C. Y. Lee. “Representation of Switching Circuits by Binary Decision Programs”. In: *Bell Systems Technical Journal* 38 (1959), pp. 985–999.

[Lib98] Paolo Liberatore. “On the Compilability of Diagnosis, Planning, Reasoning about Actions, Belief Revision, etc.” In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 1998, pp. 144–155.

[Lin02] Jorn Lind-Nielsen. *BuDDy : Binary Decision Diagrams Library Package, release 2.4*. 2002. URL: http://sourceforge.net/projects/buddy/.

[Mar08] Pierre Marquis. *Knowledge Compilation: A Sightseeing Tour*. Tutorial at the European Conference on Artificial Intelligence. 2008. URL: http://www.cril.univ-artois.fr/~marquis/tutorialNotes-ECAI08-PMarquis.pdf.

[MD06] Robert Mateescu and Rina Dechter. “Compiling Constraint Networks into AND/OR Multi-valued Decision Diagrams (AOMDDs)”. In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2006, pp. 329–343.

[MD08] Robert Mateescu and Rina Dechter. “AND/OR Multi-valued Decision Diagrams for Constraint Networks”. In: *Concurrency, Graphs and Models*. 2008, pp. 238–257.

[MDM08] Robert Mateescu, Rina Dechter, and Radu Marinescu. “AND/OR Multi-Valued Decision Diagrams (AOMDDs) for Graphical Models”. In: *Journal of Artificial Intelligence Research (JAIR)* 33 (2008), pp. 465–519.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer, 1993, pp. I–XV, 1–194. ISBN: 978-0-7923-9380-1.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications*. Springer, 1998. ISBN: 3-540-64486-5.

[Mer01] Stephan Merz. “Model Checking: A Tutorial Overview”. In: *Modeling and Verification of Parallel Processes*. Ed. by Franck et al. Cassez. Vol. 2067. Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2001, pp. 3–38.

[OP06] Barry O’Sullivan and Gregory M. Provan. “Approximate Compilation for Embedded Model-Based Reasoning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2006.

[PB+05] Héctor Palacios, Blai Bonet, Adnan Darwiche, and Hector Geffner. “Pruning Conformant Plans by Counting Models on Compiled d-DNNF Representations”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 2005, pp. 141–150.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994, pp. I–XV, 1–523. ISBN: 978-0-201-53082-7.

[Par03] Bernard Pargamin. “Extending Cluster Tree Compilation with Non-Boolean Variables in Product Configuration: A Tractable Approach to Preference-based Configuration”. In: *Proceedings of the Workshop on Configuration at IJCAI*. 2003.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988. ISBN: 0-934613-73-7.

[Ped89] Edwin P. D. Pednault. “ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus”. In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 1989, pp. 324–332.

[PT01] Marco Pistore and Paolo Traverso. “Planning as Model Checking for Extended Goals in Non-deterministic Domains”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2001, pp. 479–486.

[PV+10] Cédric Pralet, Gérard Verfaillie, Michel Lemaître, and Guillaume Infantès. “Controller Synthesis for Autonomous Systems: A Constraint-Based Approach”. In: *Proceedings of the 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS 2010)*. Ed. by Takashi Kubota. 2010.

[Rin11] Jussi Rintanen. *Algorithms for Classical Planning*. Tutorial at the International Joint Conference of Artificial Intelligence (IJCAI). 2011. URL: http://users.cecs.anu.edu.au/~jussi/ijcai11tutorial/.

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, eds. *Handbook of Constraint Programming*. Elsevier, 2006.

[SW98a] Daniel Sabin and Rainer Weigel. “Product Configuration Frameworks — A Survey”. In: *IEEE Intelligent Systems* 13.4 (1998), pp. 42–49.

[Sac75] Earl D. Sacerdoti. “The Nonlinear Nature of Plans”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1975, pp. 206–214.

[SF96] Djamila Sam-Haroud and Boi Faltings. “Consistency Techniques for Continuous Constraints”. In: *Constraints* 1.1/2 (1996), pp. 85–118.

[SM05] Scott Sanner and David A. McAllester. “Affine Algebraic Decision Diagrams (AADDs) and their Application to Structured Probabilistic Inference”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2005, pp. 1384–1390.

[Seb07] Roberto Sebastiani. “Lazy Satisability Modulo Theories”. In: *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)* 3.3–4 (2007), pp. 141–224.

[Sin02] Carsten Sinz. “Knowledge Compilation for Product Configuration”. In: *Proceedings of the Workshop on Configuration at ECAI*. Lyon, France, July 2002, pp. 23–26.

[SFJ00] David E. Smith, Jeremy Frank, and Ari K. Jónsson. “Bridging the Gap between Planning and Scheduling”. In: *The Knowledge Engineering Review* 15 (1 Mar. 2000), pp. 47–83. ISSN: 0269-8889. DOI: 10.1017/S0269888900001089. URL: http://dl.acm.org/citation.cfm?id=975748.975752.

[SW98b] David E. Smith and Daniel S. Weld. “Conformant Graphplan”. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 1998, pp. 889–896.

[Som05] Fabio Somenzi. *CUDD : Colorado University Decision Diagram package, release 2.4.1*. 2005. URL: http://vlsi.colorado.edu/~fabio/CUDD/.

[SK+90] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. “Algorithms for Discrete Function Manipulation”. In: *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. Nov. 1990, pp. 92–95.

[ST98] Karsten Strehl and Lothar Thiele. “Symbolic Model Checking of Process Networks Using Interval Diagram Techniques”. In: *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. 1998, pp. 686–692.

[Stu97] Markus Stumptner. “An Overview of Knowledge-Based Configuration”. In: *AI Communications* 10.2 (1997), pp. 111–125.

[SM96] Janos Sztipanovits and Amit Misra. “Diagnosis of Discrete Event Systems Using Ordered Binary Decision Diagrams”. In: *Proceedings of the Seventh International Workshop on Principles of Diagnosis*. Val Morin, Québec, Oct. 1996.

[TP97] Paul Tafertshofer and Massoud Pedram. “Factored Edge-Valued Binary Decision Diagrams”. In: *Formal Methods in System Design* 10.2/3 (1997), pp. 243–270.

[TT03] Pietro Torasso and Gianluca Torta. “Computing Minimum-Cardinality Diagnoses Using OBDDs”. In: *Proceedings of the German Conference on Artificial Intelligence (KI, Künstliche Intelligenz)*. 2003, pp. 224–238.

[Vah03] Arash Vahidi. *JDD, a pure Java BDD and Z-BDD library*. 2003. URL: http://javaddlib.sourceforge.net/jdd/.

[Vem92] Nageshwara Rao Vempaty. “Solving Constraint Satisfaction Problems Using Finite State Automata”. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 1992, pp. 453–458.

[VP08a] Alberto Venturini and Gregory Provan. “Incremental Algorithms for Approximate Compilation”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2008, pp. 1495–1499.

[VP08b] Gérard Verfaillie and Cédric Pralet. “Utiliser des chronogrammes pour modéliser des problèmes de planification et d’ordonnancement”. French. In: *Proceedings of the French Conference on Planning, Decision, and Learning (JFPDA, Journées Francophones de Planification, Décision et Apprentissage)*. 2008.

[WH06] Michael Wachter and Rolf Haenni. “Propositional DAGs: A New Graph-Based Language for Representing Boolean Functions”. In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 2006, pp. 277–285.

[Wal00] Toby Walsh. “SAT $\vee$ CSP”. In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2000, pp. 441–456.

[Wha07] John Whaley. *JavaBDD, Java Library for Manipulating BDDs*. 2007. URL: http://javabdd.sourceforge.net/.

[WFD07] Robert Wille, Görschwin Fey, and Rolf Drechsler. “Building Free Binary Decision Diagrams Using SAT Solvers”. In: *Facta Universitatis, Series: Electronics and Energetics* 20.3 (2007), pp. 381–394. ISSN: 03533670.

[Wil05] Nic Wilson. “Decision Diagrams for the Computation of Semiring Valuations”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2005, pp. 331–336.

Index of Symbols