

Introduction and Basic Concepts

While the focus of this book is on algorithms rather than mathematical logic, the two points of view are inevitably mixed: one cannot truly understand why a given algorithm is correct without understanding the logic behind it. This does not mean, however, that logic is a prerequisite, or that without understanding the fundamentals of logic, it is hard to learn and use these algorithms. It is similar, perhaps, to a motorcyclist who has the choice of whether to learn how his or her bike works.

He or she can ride a long way without such knowledge, but at certain points, when things go wrong or if the bike has to be tuned for a particular ride, understanding how and why things work comes in handy. And then again, suppose our motorcyclist does decide to learn mechanics: where should he or she stop? Is the physics of combustion engines important? Is the “why” important at all, or just the “how”? Or an even more fundamental question: should one first learn how to ride a motorcycle and then refer to the basics when necessary, or learn things “bottom-up”, from principles to mechanics – from science to engineering – and then to the rules of driving?

The reality is that different people have different needs, tendencies, and backgrounds, and there is no right way to write a motorcyclist’s manual that fits all. And things can get messier when one is trying to write a book about decision procedures which is targeted, on one hand, at practitioners – programmers who need to know about algorithms that solve their particular problems – and, on the other hand, at students and researchers who need to see how these algorithms can be defined in the theoretical framework that they are accustomed to, namely logic.

This first chapter has been written with both types of reader in mind. It is a combination of a reference for later chapters and a general introduction. Section 1.1 describes the two most common approaches to formal reasoning, namely deduction and enumeration, and demonstrates them with propositional logic. Section 1.2 serves as a reference for basic terminology such as *validity*, *satisfiability*, *soundness* and *completeness*. More basic terminology is described in Sect. 1.3, which is dedicated to *normal forms* and some of their

properties. Up to that point in the chapter, there is no new material. As of Sect. 1.5, the chapter is dedicated to more advanced issues that are necessary as a general introduction to the book. Section 1.4 positions the subject which this book is dedicated to in the theoretical framework in which it is typically discussed in the literature. This is important mainly for the second type of reader: those who are interested in entering this field as researchers, and, more generally, those who are trained to some extent in mathematical logic. This section also includes a description of the types of problem that we are concerned with in this book, and the standard form in which they are presented in the following chapters. Section 1.5 describes the trade-off between expressiveness and decidability. In Sect. 1.6, we conclude the chapter by discussing the need for reasoning about formulas with a Boolean structure.

What about the rest of the book? Each chapter is dedicated to a different first-order **theory**. We have not yet explained what a theory is, and specifically what a **first-order theory** is – this is the role of Sect. 1.4 – but some examples are still in order, as some intuition as to what theories are is required before we reach that section in order to understand the direction in which we are proceeding.

Informally, one may think of a theory as a finite or an infinite set of formulas, which are characterized by common grammatical rules, allowed functions and predicates, and a domain of values. The fact that they are called “first-order” means only that there is a restriction on the quantifiers (only variables, rather than sets of variables, can be quantified), but this is mostly irrelevant to us, because, in all chapters but one, we restrict the discussion to quantifier-free formulas. The table below lists some of the first-order theories that are covered in this book.¹

Theory name	Example formula	Chapter
Propositional logic	$x_1 \wedge (x_2 \vee \neg x_3)$	2
Equality	$y_1 = y_2 \wedge \neg(y_1 = y_3) \implies \neg(y_1 = y_3)$	3, 4
Linear arithmetic	$(2z_1 + 3z_2 \leq 5) \vee (z_2 + 5z_2 - 10z_3 \geq 6)$	5
Bit vectors	$((a \gg b) \& c) < c$	6
Arrays	$(i = j \wedge a[j] = 1) \implies a[i] = 1$	7
Pointer logic	$p = q \wedge *p = 5 \implies *q = 5$	8
Combined theories	$(i \leq j \wedge a[j] = 1) \implies a[i] < 2$	10

In the next few sections, we use propositional logic, which we assume the reader is familiar with, in order to demonstrate various concepts that apply equally to other first-order theories.

¹ Here we consider propositional logic as a first-order theory, which is technically correct, although not common.

1.1 Two Approaches to Formal Reasoning

The primary problem that we are concerned with is that of the validity (or satisfiability) of a given formula. Two fundamental strategies for solving this problem are the following:

- The **model-theoretic** approach is to enumerate possible solutions from a finite number of candidates.
- The **proof-theoretic** approach is to use a deductive mechanism of reasoning, based on **axioms** and **inference rules**, which together are called an **inference system**.

These two directions – enumeration and deduction – are apparent as early as the first lessons on propositional logic. We dedicate this section to demonstrating them.

Consider the following three contradicting claims:

1. If x is a prime number greater than 2, then x is odd.
2. It is not the case that x is not a prime number greater than 2.
3. x is not odd.

Denote the statement “ x is a prime number greater than 2” by A and the statement “ x is odd” by B . These claims translate into the following propositional formulas:

$$\begin{aligned} A &\implies B . \\ \neg\neg A &. \\ \neg B &. \end{aligned} \tag{1.1}$$

We would now like to prove that this set of formulas is indeed inconsistent.

1.1.1 Proof by Deduction

The first approach is to derive conclusions by using an inference system. Inference rules relate **antecedents** to their **consequents**. For example, the following are two inference rules, called modus ponens (M.P.) and CONTRADICTION:

$$\frac{\varphi_1 \implies \varphi_2 \quad \varphi_1}{\varphi_2} \text{ (M.P.)} , \tag{1.2}$$

$$\frac{\varphi \quad \neg\varphi}{\text{FALSE}} \text{ (CONTRADICTION)} . \tag{1.3}$$

The rule M.P. can be read as follows: from $\varphi_1 \implies \varphi_2$ and φ_1 being TRUE, deduce that φ_2 is TRUE. The formula φ_2 is the consequent of the rule M.P. Axioms are inference rules without antecedents:

$$\frac{}{\neg\neg\varphi \iff \varphi} \text{ (DOUBLE-NEGATION-AX)} . \tag{1.4}$$

(Axioms are typically written without the separating line above them.) We can also write a similar inference rule:

$$\frac{\neg\neg\varphi}{\varphi} \text{ (DOUBLE-NEGATION) .} \quad (1.5)$$

(DOUBLE-NEGATION-AX and DOUBLE-NEGATION are not the same, because the latter is not symmetric.) Many times, however, axioms and inference rules are interchangeable, so there is not always a sharp distinction between them.

The inference rules and axioms above are expressed with the help of arbitrary formula symbols (such as φ_1 and φ_2 in (1.2)). In order to use them for proving a particular theorem, they need to be *instantiated*, which means that these arbitrary symbols are replaced with specific variables and formulas that are relevant to the theorem that we wish to prove. For example, the inference rules (1.2), (1.3), and (1.5) can be instantiated such that FALSE, i.e., a contradiction, can be derived from the set of formulas in (1.1):

$$\begin{array}{ll} (1) A \implies B & \text{(premise)} \\ (2) \neg\neg A & \text{(premise)} \\ (3) A & (2; \text{DOUBLE-NEGATION}) \\ (4) \neg B & \text{(premise)} \\ (5) B & (1, 3; \text{M.P.}) \\ (6) \text{FALSE} & (4, 5; \text{CONTRADICTION}) . \end{array} \quad (1.6)$$

Here, in step (3), φ in the rule DOUBLE-NEGATION is instantiated with A . The antecedent φ_1 in the rule M.P. is instantiated with A , and φ_2 is instantiated with B .

More complicated theorems may require more complicated inference systems. This raises the question of whether everything that can be proven with a given inference system is indeed valid (in this case the system is called **sound**), and whether there exists a proof of validity using the inference system for every valid formula (in this case it is called **complete**). These questions are fundamental for every deduction system; we delay further discussion of this subject and a more precise definition of these terms to Sect. 1.2.

While deductive methods are very general, they are not always the most convenient or the most efficient way to know whether a given formula is valid.

1.1.2 Proof by Enumeration

The second approach is relevant if the problem of checking whether a formula is satisfiable can be reduced to a problem of *searching* for a satisfying assignment within a finite set of options. This is the case, for example, if the variables range over a finite domain,² such as in propositional logic. In the case of propositional logic, enumerating solutions can be done using **truth tables**, as demonstrated by the following example:

² A finite domain is a sufficient but not a necessary condition. In many cases, even if the domain is infinite, it is possible to find a bound such that if there exists a satisfying assignment, then there exists one within this bound. Theories that have this property are said to have the **small-model property**.

A	B	$(A \implies B)$	$(A \implies B) \wedge A$	$(A \implies B) \wedge A \wedge \neg B$
1	1	1	1	0
1	0	0	0	0
0	1	1	0	0
0	0	1	0	0

The rightmost column, which represents the formula in our example (see (1.1)), is not satisfied by any one of the four possible assignments, as expected.

1.1.3 Deduction and Enumeration

The two basic approaches demonstrated above, deduction and enumeration, go a long way, and in fact are major subjects in the study of logic. In practice, many decision procedures are not based on explicit use of either enumeration or deduction. Yet, typically their actions can be understood as performing one or the other (or both) implicitly, which is particularly helpful when arguing for their correctness.

1.2 Basic Definitions

We begin with several basic definitions that are used throughout the book. Some of the definitions that follow do not fully coincide with those that are common in the study of mathematical logic. The reason for these gaps is that we focus on quantifier-free formulas, which enables us to simplify various definitions. We discuss these issues further in Sect. 1.4.

Definition 1.1 (assignment). *Given a formula φ , an assignment of φ from a domain D is a function mapping φ 's variables to elements of D . An assignment to φ is full if all of φ 's variables are assigned, and partial otherwise.*

In this definition, we assume that there is a single domain for all variables. The definition can be trivially extended to the case in which different variables have different domains.

Definition 1.2 (satisfiability, validity and contradiction). *A formula is satisfiable if there exists an assignment of its variables under which the formula evaluates to TRUE. A formula is a contradiction if it is not satisfiable. A formula is valid (also called a tautology) if it evaluates to TRUE under all assignments.*

What does it mean that a formula “evaluates to TRUE” under an assignment? To evaluate a formula, one needs a definition of the **semantics** of the various functions and predicates in the formula. In propositional logic, for example, the semantics of the propositional connectives is given by truth tables, as presented above. Indeed, given an assignment of all variables in a propositional

formula, a truth table can be used for checking whether it satisfies a given formula, or, in other words, whether the given formula evaluates to TRUE under this assignment.

It is not hard to see that a formula φ is valid if and only if $\neg\varphi$ is a contradiction. Although somewhat trivial, this is a very useful observation, because it means that we can check whether a formula is valid by checking instead whether its negation is a contradiction, i.e., not satisfiable.

Example 1.3. The propositional formula

$$A \wedge B \tag{1.7}$$

is satisfiable because there exists an assignment, namely $\{A \mapsto \text{TRUE}, B \mapsto \text{TRUE}\}$, which makes the formula evaluate to TRUE. The formula

$$(A \implies B) \wedge A \wedge \neg B \tag{1.8}$$

is a contradiction, as we saw earlier: no assignment satisfies it. On the other hand, the negation of this formula, i.e.,

$$\neg((A \implies B) \wedge A \wedge \neg B), \tag{1.9}$$

is valid: every assignment satisfies it. ▀

Given a formula φ and an assignment α of its variables, we write $\alpha \models \varphi$ to denote that α satisfies φ . If a formula φ is valid (and hence, all assignments satisfy it), we write $\models \varphi$.³

Definition 1.4 (the decision problem for formulas). *The decision problem for a given formula φ is to determine whether φ is valid.*

Given a theory T , we are interested in a procedure⁴ that terminates with a correct answer to the decision problem, for every formula of the theory T .⁵

This can be formalized with a generalization of the notions of “soundness” and “completeness” that we saw earlier in the context of inference systems. These terms can be defined for the more general case of procedures as follows:

³ Recall that the discussion here refers to propositional logic. In the more general case, we are not talking about assignments, rather about structures that may or may not satisfy a formula. In that case, the notation $\models \varphi$ means that all structures satisfy φ . These terms are explained later in Sect. 1.4.

⁴ We follow the convention by which a **procedure** does not necessarily terminate, whereas an **algorithm** terminates. This may cause confusion, because a “decision procedure” is by definition terminating, and thus should actually be called a “decision algorithm”. This confusion is rooted in the literature, and we follow it here.

⁵ Every theory is defined over a set of symbols (e.g., linear arithmetic is defined over symbols such as “+” and “≥”). By saying “every formula of the theory” we mean every formula that is restricted to the symbols of the theory. This will be explained in more detail in Sect. 1.4.

Definition 1.5 (soundness of a procedure). *A procedure for the decision problem is sound if when it returns “Valid”, the input formula is valid.*

Definition 1.6 (completeness of a procedure). *A procedure for the decision problem is complete if*

- *it always terminates, and*
- *it returns “Valid” when the input formula is valid.*

Definition 1.7 (decision procedure). *A procedure is called a decision procedure for T if it is sound and complete with respect to every formula of T .*

Definition 1.8 (decidability of a theory). *A theory is decidable if and only if there is a decision procedure for it.*

Given these definitions, we are able to classify procedures according to whether they are sound and complete or only sound. It is rarely the case that unsound procedures are of interest. Ideally, we would always like to have a decision procedure, as defined above. However, sometimes either this is not possible (if the problem is undecidable) or the problem is easier to solve with an incomplete procedure. Some incomplete procedures are categorized as such because they do not *always* terminate (or they terminate with a “don’t know” answer). However, in many practical cases, they do terminate. Thus, completeness can also be thought of as a quantitative property rather than a binary one.

All the theories that we consider in this book are decidable. Once a theory is decidable, the next question is how difficult it is to decide it. A common measure is that of the worst-case or average-case complexity, parameterized by certain characteristics of the input formula, for example its size. One should distinguish between the complexity of a problem and the complexity of an algorithm. For example, most of the decision problems that we consider in this book are in the same complexity class, namely they are NP-complete, but we present different algorithms with different worst-case complexities to solve them. Moreover, since the worst-case complexities of alternative algorithms are frequently the same, we take a pragmatic point of view: is a given decision procedure faster than its alternatives *on a significant set of real benchmark formulas*?

Comparing decision procedures with the same worst-case complexity is problematic: it is rare that one procedure dominates another. The common practice is to consider a decision procedure relevant if it is able to perform faster than others on some significant subset of public benchmarks, or on some well-defined subclass of problems. When there is no way to predict the relative performance of procedures without actually running them, they can be run in parallel, with a “first-to-end kills all others” policy. This is a common practice in industry.

1.3 Normal Forms and Some of Their Properties

The term **normal form**, in the context of formulas, is commonly used to indicate that a formula has certain syntactic properties. In this chapter, we introduce normal forms that refer to the Boolean structure of the formula. It is common to begin the process of deciding whether a given formula is satisfiable by transforming it to some normal form that the decision procedure is designed to work with. In order to argue that the overall procedure is correct, we need to show that the transformation preserves satisfiability. The relevant term for describing this relation is the following.

Definition 1.9 (equisatisfiability). *Two formulas are equisatisfiable if they are both satisfiable or they are both unsatisfiable.*

The basic blocks of a first-order formula are its predicates, also called the **atoms** of the formula. For example, Boolean variables are the atoms of propositional logic, whereas equalities of the form $x_i = x_j$ are the atoms of the theory of equality that is studied in Chap. 4.

Definition 1.10 (negation normal form (NNF)). *A formula is in negation normal form (NNF) if negation is allowed only over atoms, and \wedge, \vee, \neg are the only allowed Boolean connectives.*

For example, $\neg(x_1 \vee x_2)$ is *not* an NNF formula, because the negation is applied to a subformula which is not an atom.

Every quantifier-free formula with a Boolean structure can be transformed in linear time to NNF, by rewriting \implies ,

$$(a \implies b) \equiv (\neg a \vee b), \quad (1.10)$$

and applying repeatedly what are known as **De Morgan's rules**,

$$\begin{aligned} \neg(a \vee b) &\equiv (\neg a \wedge \neg b), \\ \neg(a \wedge b) &\equiv (\neg a \vee \neg b). \end{aligned} \quad (1.11)$$

In the case of the formula above, this results in $\neg x_1 \wedge \neg x_2$.

Definition 1.11 (literal). *A literal is either an atom or its negation. We say that a literal is negative if it is a negated atom, and positive otherwise.*

For example, in the propositional-logic formula

$$(a \vee \neg b) \wedge \neg c, \quad (1.12)$$

the set of literals is $\{a, \neg b, \neg c\}$, where the last two are negative. In the theory of equality, where the atoms are equality predicates, a set of literals can be $\{x_1 = x_2, \neg(x_1 = x_3), \neg(x_2 = x_1)\}$.

Literals are syntactic objects. The set of literals of a given formula changes if we transform it by applying De Morgan's rules. Formula (1.12), for example, can be written as $\neg(\neg a \wedge b) \wedge \neg c$, which changes its set of literals.

Definition 1.12 (state of a literal under an assignment). A positive literal is satisfied if its atom is assigned TRUE. Similarly, a negative literal is satisfied if its atom is assigned FALSE.

Definition 1.13 (pure literal). A literal is called pure in a formula φ , if all occurrences of its variable have the same sign.

In many cases, it is necessary to refer to the set of a formula's literals as if this formula were in NNF. In such cases, either it is assumed that the input formula is in NNF (or transformed to NNF as a first step), or the set of literals in this form is computed indirectly. This can be done by simply counting the number of negations that nest each atom instance: it is negative if and only if this number is odd.

For example, $\neg x_1$ is a literal in the NNF of

$$\varphi := \neg(\neg x_1 \implies x_2), \quad (1.13)$$

because there is an occurrence of x_1 in φ that is nested in three negations (the fact that x_1 is on the left-hand side of an implication is counted as a negation). It is common in this case to say that the **polarity** (also called the **phase**) of this occurrence is negative.

Theorem 1.14 (monotonicity of NNF). Let φ be a formula in NNF and let α be an assignment of its variables. Let the positive set of α with respect to φ , denoted $\text{pos}(\alpha, \varphi)$, be the literals that are satisfied by α . For every assignment α' to φ 's variables such that $\text{pos}(\alpha, \varphi) \subseteq \text{pos}(\alpha', \varphi)$, $\alpha \models \varphi \implies \alpha' \models \varphi$.

pos

Figure 1.1 illustrates this theorem: increasing the set of literals satisfied by an assignment maintains satisfiability. It does *not* maintain unsatisfiability, however: it can turn an unsatisfying assignment into a satisfying one.

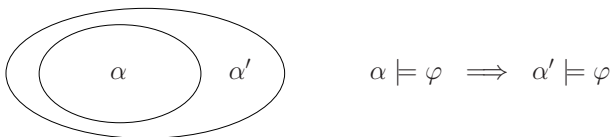


Fig. 1.1. Illustration of Theorem 1.14. The ellipses correspond to the sets of literals satisfied by α and α' , respectively

The proof of this theorem is left as an exercise (Problem 1.3).

Example 1.15. Let

$$\varphi := (\neg x \wedge y) \vee z \quad (1.14)$$

be an NNF formula. Consider the following assignments and their corresponding positive sets with respect to φ :

$$\begin{aligned} \alpha &:= \{x \mapsto 0, y \mapsto 1, z \mapsto 0\} & pos(\alpha, \varphi) &:= \{\neg x, y\} \\ \alpha' &:= \{x \mapsto 0, y \mapsto 1, z \mapsto 1\} & pos(\alpha', \varphi) &:= \{\neg x, y, z\}. \end{aligned} \quad (1.15)$$

By Theorem 1.14, since $\alpha \models \varphi$ and $pos(\alpha, \varphi) \subseteq pos(\alpha', \varphi)$, then $\alpha' \models \varphi$. Indeed, $\alpha' \models \varphi$. \blacksquare

We now describe two very useful restrictions of NNF: disjunctive normal form (DNF) and conjunctive normal form (CNF).

Definition 1.16 (disjunctive normal form (DNF)). *A formula is in disjunctive normal form if it is a disjunction of conjunctions of literals, i.e., a formula of the form*

$$\bigvee_i \left(\bigwedge_j l_{ij} \right), \quad (1.16)$$

where l_{ij} is the j -th literal in the i -th **term** (a term is a conjunction of literals).

Example 1.17. In propositional logic, l is a Boolean literal, i.e., a Boolean variable or its negation. Thus the following formula over Boolean variables a , b , c , and d is in DNF:

$$\begin{aligned} (a \wedge c \wedge \neg b) &\vee \\ (\neg a \wedge d) &\vee \\ (b \wedge \neg c \wedge \neg d) &\vee \\ &\vdots \end{aligned} \quad (1.17)$$

In the theory of equality, the atoms are equality predicates. Thus, the following formula is in DNF:

$$\begin{aligned} ((x_1 = x_2) \wedge \neg(x_2 = x_3) \wedge \neg(x_3 = x_1)) &\vee \\ (\neg(x_1 = x_4) \wedge (x_4 = x_2)) &\vee \\ ((x_2 = x_3) \wedge \neg(x_3 = x_4) \wedge \neg(x_4 = x_1)) &\vee \\ &\vdots \end{aligned} \quad (1.18)$$

\blacksquare

Every formula with a Boolean structure can be transformed into DNF, while potentially increasing the size of the formula exponentially. The following example demonstrates this exponential ratio.

Example 1.18. The following formula is of length linear in n :

$$(x_1 \vee x_2) \wedge \cdots \wedge (x_{2n-1} \vee x_{2n}). \quad (1.19)$$

The length of the equivalent DNF, however, is exponential in n , since every new binary clause (a disjunction of two literals) doubles the number of terms in the equivalent DNF, resulting, overall, in 2^n terms:

$$\begin{aligned}
& (x_1 \wedge x_3 \wedge \cdots \wedge x_{2n-3} \wedge x_{2n-1}) \vee \\
& (x_1 \wedge x_3 \wedge \cdots \wedge x_{2n-3} \wedge x_{2n}) \vee \\
& (x_1 \wedge x_3 \wedge \cdots \wedge x_{2n-2} \wedge x_{2n}) \vee \\
& \vdots
\end{aligned} \tag{1.20}$$

■

Although transforming a formula to DNF can be too costly in terms of computation time, it is a very natural way to decide formulas with an arbitrary Boolean structure.

Suppose we are given a disjunctive linear arithmetic formula, that is, a Boolean structure in which the atoms are linear inequalities over the reals. We know how to decide whether a *conjunction* of such literals is satisfiable: there is a known method called simplex that can give us this answer. In order to use the simplex method to solve the more general case in which there are also disjunctions in the formula, we can perform **syntactic case-splitting**. This means that the formula is transformed into DNF, and then each term is solved separately. Each such term contains a conjunction of literals, a form which we know how to solve. The overall formula is satisfiable, of course, if any one of the terms is satisfiable. **Semantic case-splitting**, on the other hand, refers to techniques that split the search space, in the case where the variables are finite (“first the case in which $x = 0$, then the case in which $x = 1 \dots$ ”).

The term **case-splitting** (without being prefixed with “syntactic”) usually refers in the literature to either syntactic case-splitting or a “smart” implementation thereof. Indeed, many of the cases that are generated in syntactic case-splitting are redundant, i.e., they share a common subset of conjuncts that contradict each other. Efficient decision procedures should somehow avoid replicating the process of deducing this inconsistency, or, in other words, they should be able to **learn**, as demonstrated in the following example.

Example 1.19. Consider the following formula:

$$\varphi := (a = 1 \vee a = 2) \wedge a \geq 3 \wedge (b \geq 4 \vee b \leq 0) . \tag{1.21}$$

The DNF of φ consists of four terms:

$$\begin{aligned}
& (a = 1 \wedge a \geq 3 \wedge b \geq 4) \vee \\
& (a = 2 \wedge a \geq 3 \wedge b \geq 4) \vee \\
& (a = 1 \wedge a \geq 3 \wedge b \leq 0) \vee \\
& (a = 2 \wedge a \geq 3 \wedge b \leq 0) .
\end{aligned} \tag{1.22}$$

These four cases can each be discharged separately, by using a decision procedure for linear arithmetic (Chap. 5). However, observe that the first and the third case share the two conjuncts $a = 1$ and $a \geq 3$, which already makes the case unsatisfiable. Similarly, the second and the fourth case share the conjuncts $a = 2$ and $a \geq 3$. Thus, with the right learning mechanism, two of the

four calls to the decision procedure can be avoided. This is still case-splitting, but more efficient than a plain transformation to DNF. \blacksquare

The problem of reasoning about formulas with a general Boolean structure is a common thread throughout this book.

Definition 1.20 (conjunctive normal form (CNF)). *A formula is in conjunctive normal form if it is a conjunction of disjunctions of literals, i.e., it has the form*

$$\bigwedge_i \left(\bigvee_j l_{ij} \right), \quad (1.23)$$

where l_{ij} is the j -th literal in the i -th **clause** (a clause is a disjunction of literals).

Every formula with a Boolean structure can be transformed into an equivalent CNF formula, while potentially increasing the size of the formula exponentially. Yet, any propositional formula can also be transformed into an equisatisfiable CNF formula with only a *linear* increase in the size of the formula. The price to be paid is n new Boolean variables, where n is the number of **logical gates** in the formula. This transformation is done via **Tseitin's encoding** [195].

Tseitin suggested that one new variable should be added for every *logical gate* in the original formula, and several clauses to constrain the value of this variable to be equal to the gate it represents, in terms of the inputs to this gate. The original formula is satisfiable if and only if the conjunction of these clauses together with the new variable associated with the topmost operator is satisfiable. This is best illustrated with an example.

Example 1.21. Given a propositional formula

$$x_1 \implies (x_2 \wedge x_3), \quad (1.24)$$

with Tseitin's encoding we assign a new variable to each subexpression, or, in other words, to each logical gate, for example AND (\wedge), OR (\vee), and NOT (\neg). For this example, let us assign the variable a_2 to the AND gate (corresponding to the subexpression $x_2 \wedge x_3$) and a_1 to the IMPLICATION gate (corresponding to $x_1 \implies a_2$), which is also the topmost operator of this formula. Figure 1.2 illustrates the **derivation tree** of our formula, together with these auxiliary variables in square brackets.

We need to satisfy a_1 , together with two equivalences,

$$\begin{aligned} a_1 &\iff (x_1 \implies a_2), \\ a_2 &\iff (x_2 \wedge x_3). \end{aligned} \quad (1.25)$$

The first equivalence can be rewritten in CNF as

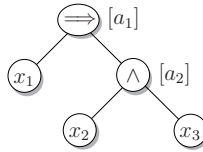


Fig. 1.2. Tseitin's encoding. Assigning an auxiliary variable to each logical gate (shown here in square brackets) enables us to translate each propositional formula to CNF, while increasing the size of the formula only linearly

$$\begin{aligned}
 (a_1 \vee x_1) & \quad \wedge \\
 (a_1 \vee \neg a_2) & \quad \wedge \\
 (\neg a_1 \vee \neg x_1 \vee a_2) , &
 \end{aligned} \tag{1.26}$$

and the second equivalence can be rewritten in CNF as

$$\begin{aligned}
 (\neg a_2 \vee x_2) & \quad \wedge \\
 (\neg a_2 \vee x_3) & \quad \wedge \\
 (a_2 \vee \neg x_2 \vee \neg x_3) . &
 \end{aligned} \tag{1.27}$$

Thus, the overall CNF formula is the conjunction of (1.26), (1.27), and the unit clause

$$(a_1) , \tag{1.28}$$

which represents the topmost operator. ▀

There are various optimizations that can be performed in order to reduce the size of the resulting formula and the number of additional variables. For example, consider the following formula:

$$x_1 \vee (x_2 \wedge x_3 \wedge x_4 \wedge x_5) . \tag{1.29}$$

With Tseitin's encoding, we need to introduce four auxiliary variables. The encoding of the clause on the right-hand side, however, can be optimized to use just a single variable, say a_2 :

$$a_2 \iff (x_2 \wedge x_3 \wedge x_4 \wedge x_5) . \tag{1.30}$$

In CNF,

$$\begin{aligned}
 (\neg a_2 \vee x_2) & \quad \wedge \\
 (\neg a_2 \vee x_3) & \quad \wedge \\
 (\neg a_2 \vee x_4) & \quad \wedge \\
 (\neg a_2 \vee x_5) & \quad \wedge \\
 (a_2 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4 \vee \neg x_5) . &
 \end{aligned} \tag{1.31}$$

In general, we can encode a conjunction of n literals with a single variable and $n + 1$ clauses, which is an improvement over the original encoding, requiring $n - 1$ auxiliary variables and $3(n - 1)$ clauses.

Such savings are also possible for a series of disjunctions (see Problem 1.1). Another popular optimization is that of **subsumption**: given two clauses such that the set of literals in one of the clauses subsumes the set of literals in the other clause, the longer clause can be discarded without affecting the satisfiability of the formula.

Finally, if the original formula is in NNF, the number of clauses can be reduced substantially, as was shown by Plaisted and Greenbaum in [152]. Tseitin's encoding is based on constraints of the form

$$\text{auxiliary variable} \iff \text{formula}, \quad (1.32)$$

but only the left-to-right implication is necessary. The proof that this improvement is correct is left as an exercise (Problem 1.4). In practice, experiments show that owing to the requirement of transforming the formula to NNF first, this reduction has a relatively small (positive) effect on the run time of modern SAT solvers compared with Tseitin's encoding.

Example 1.22. Consider a gate $x_1 \wedge x_2$, which we encode with a new auxiliary variable a . Three clauses are necessary to encode the constraint $a \iff (x_1 \wedge x_2)$, as was demonstrated in (1.27). The constraint $a \Leftarrow (x_1 \wedge x_2)$ (equivalently, $(a \vee \neg x_1 \vee \neg x_2)$) is redundant, however, which means that only two out of the three constraints are necessary. \blacksquare

A conversion algorithm with similar results to [152], in which the elimination of the negations is built in (rather than the formula being converted to NNF a priori), has been given by Wilson [201].

1.4 The Theoretical Point of View

While we take the algorithmic point of view in this book, it is important to understand also the theoretical context, especially for readers who are also interested in following the literature in this field or are more used to the terminology of formal logic. It is also necessary for understanding Chaps. 10 and 11. We must assume in this subsection that the reader is familiar to some extent with first-order logic – a reasonable exposition of this subject is beyond the scope of this book. See [30, 91] for a more organized study of these matters. Let us recall some of the terms that are directly relevant to our topic.

First-order logic (also called **predicate logic**) is based on the following elements:

1. *Variables*: a set of *variables*.
2. *Logical symbols*: the standard Boolean connectives (e.g., “ \wedge ”, “ \neg ”, and “ \vee ”), quantifiers (“ \exists ” and “ \forall ”) and parentheses.
3. *Nonlogical symbols*: function, predicate, and constant symbols.
4. *Syntax*: rules for constructing formulas. Formulas adhering to these rules are said to be **well-formed**.

Essentially, first-order logic extends propositional logic with quantifiers and the nonlogical symbols. The syntax of first-order logic extends the syntax of propositional logic naturally. Two examples of such formulas are

- $\exists y \in \mathbb{Z}. \forall x \in \mathbb{Z}. x > y$,
- $\forall n \in \mathbb{N}. \exists p \in \mathbb{N}. n > 1 \implies (isprime(p) \wedge n < p < 2n)$,

where “ $>$ ”, “ $<$ ”, and “*isprime*” are nonlogical binary predicate symbols.

The elements listed above only refer to symbols and syntax – they still do not tell us how to evaluate whether a given formula is true or false. This separation between symbols and their interpretation – between syntax and semantics – is an important principle in the study of logic. We shall explain this separation with an example. Let Σ denote the set of symbols $\{0, 1, +, =\}$, where “0” and “1” are constant symbols, “+” is a binary function symbol, and “=” is a binary predicate symbol. Consider the following formula over Σ :

$$\varphi := \exists x. x + 0 = 1. \quad (1.33)$$

Now, is φ true in \mathbb{N}_0 ? (\mathbb{N}_0 denotes the naturals, including 0.)

What seems like a trivial question is not so simple in the world of formal logic. A logician would say that the answer depends, among other things, on the **interpretation** of the symbols in Σ . What does the “+” symbol mean? Which elements in the domain do “0” and “1” refer to? From a formal perspective, whether φ is true can only be answered with respect to a given **structure**. A structure is a tuple consisting of

- a domain;
- an interpretation of the nonlogical symbols, in the form of a mapping between each function and predicate symbol to a function and a predicate, respectively, and an assignment of a domain element to each of the constant symbols;
- an assignment of a domain element to each of the free (unquantified) variables.

For example, if we choose to interpret the “+” symbol as the *multiplication* function, the answer is that φ in (1.33) is false.

The formula φ is **satisfiable** if and only if there *exists* a structure under which the formula is true. Indeed, in this case there exists such a domain and interpretation – namely, \mathbb{N}_0 and the common interpretation of “+”, “=”, “0” and “1” – and, hence, the formula is satisfiable.

First-order logic can be thought of as a framework giving a generic syntax and the building blocks for defining specific restrictions thereof, called **theories**. The restrictions defined by a theory are on the nonlogical symbols that can be used and the interpretation that we can give them. Indeed, in a practical setting we would not want to consider an arbitrary interpretation of the symbols as above (where “+” is multiplication); rather we consider only specific ones.

A set of nonlogical symbols is called a **signature**. Given a signature Σ , a Σ -**formula** is a formula that uses only nonlogical symbols from Σ (possibly in addition to logical symbols). A variable is **free** if it is not bound by a quantifier. A **sentence** is a formula without free variables. A first-order Σ -**theory** T consists of a set of Σ -sentences. For a given Σ -theory T , a Σ -formula φ is **T -satisfiable** if there exists a structure that satisfies both the formula and the sentences of T . Similarly, a Σ -formula φ is **T -valid** if all structures that satisfy the sentences of T , also satisfy φ .

The set of sentences that are required is sometimes large or even infinite. It is therefore common to define theories via a set of axioms, which implicitly represent all the sentences that can be inferred from them, using some sound and complete inference system for the logical symbols.

Example 1.23. Consider a simple signature Σ consisting only of the predicate symbol “=”.⁶ Let T be a Σ -theory. An example of a well-formed Σ -formula is

$$\forall x, y, z. ((x = y) \wedge \neg(y = z)) \implies \neg(x = z) . \quad (1.34)$$

If we wish T to restrict the interpretation of “=” to the equality predicate, the following three axioms are sufficient:

$$\begin{aligned} \forall x. x = x & \quad (\text{REFLEXIVITY}) , \\ \forall x, y. x = y \implies y = x & \quad (\text{SYMMETRY}) , \\ \forall x, y, z. x = y \wedge y = z \implies x = z & \quad (\text{TRANSITIVITY}) . \end{aligned} \quad (1.35)$$

Since every domain and interpretation that satisfy these axioms also satisfy (1.34), then (1.34) is T -valid. ▀

As said above, a theory restricts only the nonlogical symbols. If we want to restrict the set of logical symbols or the grammar, we need to speak about **fragments** of the logic. For example, we can speak about the **quantifier-free fragment** of T as defined in the example above. This fragment, called **equality logic**, happens to be the subject of Chap. 4. Most of the chapters, in fact, are concerned with quantifier-free fragments of theories. Another useful fragment is called the **conjunctive fragment**, which means that the only Boolean connective that is allowed is conjunction. What about restricting the interpretation of the logical symbols? The axioms that restrict the interpretation of the logical symbols, called the **logical axioms**, are assumed to be “built in”, i.e., they are common to all first-order theories.

Numerous theories have been considered over the years, corresponding to various problems of interest. Many of them lead to decidability, and, frequently to efficient decision procedures. The theory of **Presburger arithmetic**, for example, is defined with a signature $\Sigma = \{0, 1, +, =\}$ and is still decidable. In

⁶ It is frequently the case in the literature that the equality sign is considered as a logical symbol, and then the theory defined here has an empty signature. We do not follow this convention here, however.

contrast, the theory of **Peano arithmetic**, which is defined over a signature $\Sigma = \{0, 1, +, \cdot, =\}$, is undecidable. Thus, the addition of the multiplication symbol and the corresponding axioms that define it make the decision problem undecidable. Other famous theories include the theory of equality, the theory of reals, the theory of integers, the theory of arrays, the theory of recursive data structures and the theory of sets (“set theory”). Many of the decidable ones that are in practical use are covered in this book.

1.4.1 The Problem We Solve

Unless otherwise stated, we are concerned with

the satisfiability problem of the quantifier-free fragment of various first-order theories.

Formulas in such fragments are called **ground formulas**, as they only contain free (unquantified, also called ground) variables and constants. Exceptions are Chap. 9, which is concerned with quantified formulas, and a small part of Chap. 7, which is concerned with quantified array logic.

There is a subtle difference between the satisfiability problem of ground formulas and the satisfiability problem of existentially quantified formulas. It is, of course, trivial that a ground formula φ over variables x_1, \dots, x_n is satisfiable if and only if

$$\exists x_1, \dots, x_n. \varphi \tag{1.36}$$

is satisfiable. Thus, the decision procedures for both problems can be similar. The reason we use the former definition is that this entails, from a formal perspective, that the satisfying structure includes an assignment of the variables, because they are all free. In many practical applications, such an assignment is necessary. In fact, the former problem can be seen as an instance of the **constraint satisfaction problem (CSP)**, which is all about finding an assignment that satisfies a set of unquantified constraints.⁷

We assume that the input formulas are given in negation normal form, or that they are implicitly transformed to this form as a first step of any of the algorithms described later. As explained in Sect. 1.10, every formula can be transformed to this form in linear time. The reason that this assumption is important is that it simplifies the algorithms and the arguments for their correctness.

1.4.2 Our Presentation of Theories

Our presentation of theories in the chapters to come is not as defined above. In an attempt to make the presentation more accessible and the chapters more self-contained, we make the following changes:

⁷ The emphasis and terminology are somewhat different. Most of the research in the CSP community is concerned with finite, discrete domains, in contrast to the problems considered in this book.

1. Rather than specifying theories through their set of symbols and sentences, we give the domain explicitly, and fix the interpretations of symbols in accordance with their common use. Hence, “+” is always the addition function, “0” is the 0 element in the given domain, and so forth.
2. Rather than specifying the theory fragment we are concerned with by referring to the generic grammar of first-order logic as a starting point, we give an explicit, self-contained definition of the grammar.

From a formal-logic point of view, fixing the interpretation means only that we have the sentences implicitly; the satisfiability problem remains the same. From the *algorithmic* point of view, however, the satisfiability problem now amounts to searching for a satisfying assignment of variables from the predefined domain. Whether a given assignment satisfies the formula can be determined according to the commonly used meanings of the various symbols.

This form of presentation is in line with our focus on the algorithmic point of view: when designing a decision procedure for a theory, the interpretation of the symbols has to be predefined. In other words, changing the domain or interpretation of symbols changes the algorithm.

1.5 Expressiveness vs. Decidability

There is an important trade-off between what a theory can express and how hard it is to decide, that is, how hard it is to determine whether a given formula allowed by the theory is valid or not. This is the reason for defining many different theories: otherwise, we would define and use only a single theory sufficiently expressive for all perceivable decision problems.

A theory can be seen as a tool for defining **languages**. Every formula in the theory defines a language, which is the set of “words” (the assignments, in the case of quantifier-free formulas) that satisfy it. We now define what it means that one theory is more expressive than another.

Definition 1.24 (expressiveness). *Theory A is more expressive than theory B if every language that can be defined by a B -formula can also be defined by an A -formula, and there exists at least one language definable by an A -formula that cannot be defined by a B -formula. We denote the fact that theory B is less expressive than theory A by $B \prec A$.*

For example, propositional logic is more expressive than what is known as “2-CNF”, i.e., CNF in which each clause has at most two literals. In propositional logic, we can define the formula

$$x_1 \vee x_2 \vee x_3 , \tag{1.37}$$

which defines a language that we cannot define with 2-CNF: it accepts all truth assignments to x_1, x_2, x_3 except $\{x_1 \mapsto \text{FALSE}, x_2 \mapsto \text{FALSE}, x_3 \mapsto \text{FALSE}\}$. How can we prove this?

Well, assume that there exists a 2-CNF representation of this formula using the same set of variables, and consider one of its binary clauses. Such a clause contradicts two of the eight possible assignments. For example, a clause $(x_1 \vee x_2)$ contradicts $\{x_1 \mapsto \text{FALSE}, x_2 \mapsto \text{FALSE}, x_3 \mapsto \text{FALSE}\}$ and $\{x_1 \mapsto \text{FALSE}, x_2 \mapsto \text{FALSE}, x_3 \mapsto \text{TRUE}\}$. Any additional clause can only contradict more assignments. Hence, we can never create a 2-CNF formula such that exactly one of the eight assignments does not satisfy it.

On the other hand, 2-CNF is a restriction of propositional logic; hence, obviously, any 2-CNF formula can be expressed in propositional logic. Thus, we have

$$\text{2-CNF} \prec \text{propositional logic} . \quad (1.38)$$

This example also demonstrates the influence of expressiveness on computational hardness: while propositional logic is NP-complete, 2-CNF can be solved in polynomial time.

In order to illustrate the trade-off between how expressive a theory is and how hard it is to decide formulas in that theory, consider a theory T defined by some syntax. Let T_1, \dots, T_n denote a list of fragments of T , defined by various restrictions on the syntax of T (similarly to the way we restricted propositional logic to 2-CNF above), for which $T_1 \prec T_2 \prec \dots \prec T_n \prec T$. Technically, this means that we have imposed a **total order** on these fragments in terms of their expressive power. Under such assumptions, Fig. 1.3 illustrates the trade-off between expressiveness and computational hardness: the less expressive the theory is (the more restrictions we put on it), the easier it is to decide it. Assume our imaginary theory T is undecidable. After some threshold is crossed (from right to left in the figure), the theory fragments can become decidable. After enough restrictions have been added, the theory becomes solvable in polynomial time. The decidable but nonpolynomially decidable fragments pose a *computational challenge*. This is one of the challenges we focus on in this book.

This view is simplistic, however, because there is no total order on the expressive power of theories, only a partial order. This means that there can be two theories, A and B , neither of which is more expressive than the other, yet their expressive power is different. In other words, there are languages that can be defined by A and not by B , and there are languages that can be defined by B and not by A .

1.6 Boolean Structure in Decision Problems

We conclude this chapter by demonstrating the need for reasoning about formulas with a Boolean structure.

Many decision procedures assume that the decision problem is given by a conjunction of constraints. The simplex algorithm and the Omega test, both of which are described in Chap. 5, are examples of such procedures.

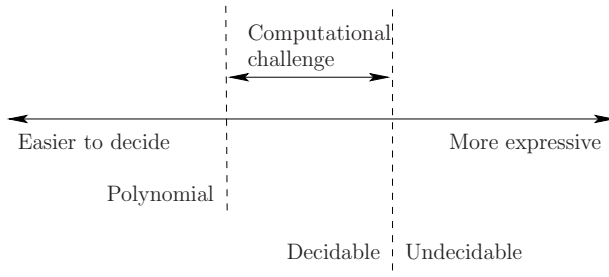


Fig. 1.3. The trade-off between expressiveness of theories and the hardness of deciding them, illustrated for an imaginary series of theories T_1, \dots, T_n, T for which each T_i , $i \in \{1, \dots, n\}$, is less expressive than its successor

Many applications, however, require a more complex Boolean structure. In program analysis and verification, for example, disjunctions may appear in the program to be verified, either explicitly (e.g., $x = y \mid\mid z$) or implicitly through constructs such as `if` and `switch` statements. Any reasoning system about such programs, therefore, must be able to deal with disjunctions. For example, **verification conditions** that arise in program verification (e.g., using **Hoare logic**), often have the form of an implication.

The following example focuses on a technique for reasoning about programs, that demonstrates how program structure, including `if` statements, is evident in the underlying verification conditions that need to be checked.

Example 1.25. Bounded model checking (BMC) of programs is a technique for verifying that a given property (typically given as an assertion by the user) holds for a program in which the number of loop iterations and recursive calls is bounded by a given number k . The states that the program can reach within this bound are represented symbolically by a formula, together with the negation of the property. If the combined formula is satisfiable, then there exists a path in the program that violates the property.

Consider the program in the left part of Fig. 1.4. The number of paths through this program is exponential in N , as each of the $a[i]$ elements can be either zero or nonzero. Despite the exponential number of paths through the program, its states can be encoded with a formula of size linear in N , as demonstrated in the right part of the figure.

The formula on the right of Fig. 1.4 encodes the states of the program on its left, using the **static-single-assignment (SSA)** form. Briefly, this means that in each assignment of the form $x = \text{exp}$;, the left-hand side variable x is replaced with a new variable, say x_1 , and any reference to x after this line and before x is assigned again is replaced with x_1 . Such a replacement is possible because there are no loops (recall that this is done in the context of BMC). After this transformation, the statements are conjoined. The resulting equation represents the states of the original program.

<pre> int a[N]; unsigned c; ... c = 0; for(i = 0; i < N; i++) if(a[i] == 0) c++; </pre>	$ \begin{aligned} c_1 &= 0 \wedge \\ c_2 &= (a[0] = 0) ? c_1 + 1 : c_1 \wedge \\ c_3 &= (a[1] = 0) ? c_2 + 1 : c_2 \wedge \\ &\dots \\ c_{N+1} &= (a[N-1] = 0) ? c_N + 1 : c_N \end{aligned} $
--	--

Fig. 1.4. A simple program with an exponential number of paths (*left*), and a static-single-assignment (SSA) form of this program after unwinding its `for` loop (*right*)

The ternary operator $c ? x : y$ in the equation on the right of Fig. 1.4 can be rewritten using a disjunction, as illustrated in (1.39). These disjunctions lead to an exponential number of clauses once the formula is converted to DNF.

$$\begin{aligned}
&c_1 = 0 \wedge \\
&((a[0] = 0 \wedge c_2 = c_1 + 1) \vee (a[0] \neq 0 \wedge c_2 = c_1)) \wedge \\
&((a[1] = 0 \wedge c_3 = c_2 + 1) \vee (a[1] \neq 0 \wedge c_3 = c_2)) \wedge \\
&\dots \\
&((a[N-1] = 0 \wedge c_{N+1} = c_N + 1) \vee (a[N-1] \neq 0 \wedge c_{N+1} = c_N)) .
\end{aligned} \tag{1.39}$$

In order to verify that some assertion holds at a specific location in the program, it is sufficient to add a constraint corresponding to the negation of this assertion, and check whether the resulting formula is satisfiable. For example, to prove that at the end of the program $c \leq N$, we need to conjoin (1.39) with $(c_{N+1} > N)$. ▀

To summarize this section, there is a need to reason about formulas with disjunctions, as illustrated in the example above. The simple solution of going through DNF does not scale, and better solutions are needed. Solutions that perform better in practice (the worst case remains exponential, of course) indeed exist, and are covered extensively in this book.

1.7 Problems

Problem 1.1 (improving Tseitin's encoding).

- (a) Using Tseitin's encoding, transform the following formula φ to CNF. How many clauses are needed?

$$\varphi := \neg(x_1 \wedge (x_2 \vee \dots \vee x_n)) . \tag{1.40}$$

- (b) Consider a clause $(x_1 \vee \dots \vee x_n)$, $n > 2$, in a non-CNF formula. How many auxiliary variables are necessary for encoding it with Tseitin's encoding? Suggest an alternative way to encode it, using a single auxiliary variable. How many clauses are needed?

Problem 1.2 (expressiveness and complexity).

- (a) Let T_1 and T_2 be two theories whose satisfiability problem is decidable and in the same complexity class. Is the satisfiability problem of a T_1 -formula reducible to a satisfiability problem of a T_2 -formula?
- (b) Let T_1 and T_2 be two theories whose satisfiability problems are reducible to one another. Are T_1 and T_2 in the same complexity class?

Problem 1.3 (monotonicity of NNF with respect to satisfiability).

Prove Theorem 1.14.

Problem 1.4 (one-sided Tseitin encoding). Let φ be an NNF formula (see Definition 1.10). Let $\vec{\varphi}$ be a formula derived from φ as in Tseitin's encoding (see Sect. 1.3), but where the CNF constraints are derived from implications from left to right rather than equivalences. For example, given a formula

$$a_1 \wedge (a_2 \vee \neg a_3) ,$$

the new encoding is the CNF equivalent of the following formula,

$$\begin{array}{l} x_0 \qquad \qquad \qquad \wedge \\ (x_0 \implies a_1 \wedge x_1) \wedge \\ (x_1 \implies a_2 \vee x_2) \wedge \\ (x_2 \implies \neg a_3) , \end{array}$$

where x_0, x_1, x_2 are new auxiliary variables. Note that Tseitin's encoding to CNF starts with the same formula, except that the " \implies " symbol is replaced with " \iff ".

1. Prove that $\vec{\varphi}$ is satisfiable if and only if φ is.
2. Let l, m, n be the number of AND, OR, and NOT gates, respectively, in φ . Derive a formula parameterized by l, m and n that expresses the ratio of the number of CNF clauses in Tseitin's encoding to that in the one-sided encoding suggested here.

1.8 Glossary

The following symbols were used in this chapter:

Symbol	Refers to ...	First used on page ...
$\alpha \models \varphi$	An assignment α satisfies a formula φ	6
$\models \varphi$	A formula φ is valid (in the case of quantifier-free formulas, this means that it is satisfied by all assignments from the domain)	6
T	A theory	6
$pos(\alpha, \varphi)$	Set of literals of φ satisfied by an assignment α	9
$B \prec A$	Theory B is less expressive than theory A	18



<http://www.springer.com/978-3-540-74104-6>

Decision Procedures

An Algorithmic Point of View

Kroening, D.; Strichman, O.

2008, XVI, 306 p., Hardcover

ISBN: 978-3-540-74104-6