

Logic and Applications

Jussi Rintanen
Department of Computer Science
Aalto University
Helsinki, Finland

October 30, 2019

Contents

Table of Contents	i
1 Introduction	1
2 Definition	2
2.1 Formulas	2
2.2 Equivalences	4
2.3 Truth-tables	5
2.4 Logical Consequence, Satisfiability, Validity	5
3 Automated Reasoning in the Propositional Logic	7
3.1 Truth-Tables	7
3.2 Tree-Search Algorithms	8
3.2.1 Unit Resolution	9
3.2.2 Subsumption	10
3.2.3 Unit Propagation	10
3.2.4 The Davis-Putnam Procedure	10
4 Normal Forms and Logic Data Structures	12
4.1 Complexity of Validity and Satisfiability for DNF and CNF	14
4.2 Formulas as a Data Structure	14
4.3 Binary Decision Diagrams	15
4.3.1 Properties of OBDDs	18
4.3.2 Model Counting	19
4.3.3 Restriction	20
4.3.4 Clausal consequences	20
4.3.5 Abstraction	20
4.3.6 Variable Ordering	21
4.4 Normal Forms DNNF and d-DNNF	22
4.5 Normal Forms Summary	23
5 Sets and Relations in the Propositional Logic	24
5.1 Representation of Sets	24
5.1.1 Set Operations as Logical Operations	24
5.2 Relations as Formulas	25
5.2.1 Relational Operations in Logic	27
6 Transition Systems	30
6.1 Basic Models	30
6.2 Procedural Representations	31
6.2.1 Deterministic Unconditional Transitions	31
6.2.2 General Definition	32
6.3 Higher Order Representations	33
7 Symbolic Reachability Testing	35
7.1 OBDD-Based Symbolic Reachability	35
7.2 SAT-Based Symbolic Reachability	37

7.2.1	Parallel Actions	38
7.2.2	Alternative Meanings of Parallelism	39
8	Modal Logics	42
8.1	Modal Logics with One Modality	42
8.1.1	Validity in classes of frames	44
8.1.2	Properties of the accessibility relation	44
8.1.3	Different uni-modal logics	45
8.1.4	Undefinable properties	46
8.1.5	Logical consequence in modal logics	47
8.2	Multi-Modal Logics	47
8.3	Linear Temporal Logic LTL	48
8.4	Branching Time Temporal Logics CTL and CTL*	49
8.4.1	The language of CTL*	49
8.4.2	The semantics of CTL*	49
8.4.3	Computation tree logic CTL	50
8.4.4	Relations between CTL, LTL and CTL*	50
8.5	Dynamic Logic	50
8.5.1	Propositional dynamic logic PDL	50
8.5.2	The semantics of PDL	51
8.5.3	Axiom system for PDL	52
8.5.4	Relations to other modal logics and Hoare logic	52
	Index	55

Chapter 1

Introduction

The classical propositional logic is the most basic and most widely used logic. It is a notation for Boolean functions, together with several proof and reasoning methods.

The use of the propositional logic as a computational tool has dramatically increased since the development of powerful search algorithms and implementation methods since the late 1990ies. Today it enjoys extensive use in several areas of computer science, especially in Computer-Aided Verification and Artificial Intelligence. Its uses in AI include planning, problem-solving, intelligent control, and diagnosis, and in Computer-Aided Verification it is used in Model-Checking, Reachability Analysis, Deadlock Detection, and many other problems.

The reason why logics are used is their ability to precisely express data and information, in particular when it is *partial* or *incomplete*, and some of the *implicit* consequences of the information must be *inferred* to make them *explicit*.

The propositional logic, as the first known NP-complete problem [Coo71], is used for representing many types of co-NP-complete and NP-complete *combinatorial* search problems. Such problems are prevalent in artificial intelligence as a part of *decision-making*, *problem-solving*, and *planning*.

For many applications natural choices would be various more expressive logics, including the *predicate logic* or various *modal logics*. These logics, however, lack the kind of efficient and scalable algorithms that are available for the classical propositional logic. The existence of high performance algorithms for reasoning with propositional logic is the main reason for its wide use.

Chapter 2

Definition

2.1 Formulas

Propositional logic is about Boolean functions, which are mappings from *truth-values* 0 and 1 (*false* and *true*) to truth-values. Arbitrary Boolean functions can be represented as *formulas* formed with *connectives* such as \wedge , \vee and \neg , and it can be shown that any Boolean function can be represented by such a formula.

Boolean functions and formulas have important applications in several areas of Computer Science.

- Digital electronics and the construction of digital computation devices such as microprocessors, is based on Boolean functions.
- The theory of computation and complexity uses Boolean functions for representing abstract models of computation (for example Boolean circuits) and investigating their properties (for example the theory of Computational Complexity, where some of the fundamental results like NP-completeness [Coo71, GJ79] were first established with Boolean functions).
- Parts of Artificial Intelligence use Boolean functions and formulas for representing models of the world and the reasoning processes of intelligent systems.

Some of the best known and most primitive Boolean functions are represented by the connectives \vee , \wedge , \neg , \rightarrow , \leftrightarrow , and \oplus as follows, with the columns marked with α and β giving the input values and the third column the value of the indicated Boolean function with these inputs.

α	β	$\alpha \vee \beta$
0	0	0
0	1	1
1	0	1
1	1	1

α	β	$\alpha \wedge \beta$
0	0	0
0	1	0
1	0	0
1	1	1

α	$\neg\alpha$
0	1
1	0

α	β	$\alpha \rightarrow \beta$
0	0	1
0	1	1
1	0	0
1	1	1

α	β	$\alpha \leftrightarrow \beta$
0	0	1
0	1	0
1	0	0
1	1	1

α	β	$\alpha \oplus \beta$
0	0	0
0	1	1
1	0	1
1	1	0

The connectives are used for forming complex Boolean functions from primitive ones.

Often only some of the connectives, typically \neg and one or both of \wedge and \vee , are viewed as *primitive connectives*, and other connectives are defined in terms of the primitive connectives. For example *implication* \rightarrow and *equivalence* \leftrightarrow connectives are defined as follows: $\alpha \rightarrow \beta$ as $\neg\alpha \vee \beta$, and $\alpha \leftrightarrow \beta$ as $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$.

There is a close connection between the Boolean connectives \vee , \wedge and \neg and the operations \cup , \cap and complementation in Set Theory. Observe the similarity between the truth-tables for the three connectives and the analogous tables for the set-theoretic operations.

α	β	$\alpha \cup \beta$
\emptyset	\emptyset	\emptyset
\emptyset	$\{1\}$	$\{1\}$
$\{1\}$	\emptyset	$\{1\}$
$\{1\}$	$\{1\}$	$\{1\}$

α	β	$\alpha \cap \beta$
\emptyset	\emptyset	\emptyset
\emptyset	$\{1\}$	\emptyset
$\{1\}$	\emptyset	\emptyset
$\{1\}$	$\{1\}$	$\{1\}$

α	$\bar{\alpha}$
\emptyset	$\{1\}$
$\{1\}$	\emptyset

Formulas in logic are formed analogously to arithmetic expressions in mathematics, with parentheses, variables (atomic propositions) and binary connectives analogous to the binary arithmetic operations like addition and multiplication.

Definition 2.1 (Propositional Formulas) *Formulas are recursively defined as follows.*

- Atomic propositions $x \in X$ are formulas.
- Symbols \top and \perp are formulas.
- If α and β are formulas, then so are
 1. $\neg\alpha$
 2. $(\alpha \wedge \beta)$
 3. $(\alpha \vee \beta)$
 4. $(\alpha \rightarrow \beta)$
 5. $(\alpha \leftrightarrow \beta)$

Nothing else is a formula.

Parentheses (and) do not always need to be written if precedences between the connectives are observed. The highest precedence is with \neg , followed by \wedge and \vee , then \rightarrow , and finally \leftrightarrow . So, $\neg a \vee b$ means the same as $(\neg a) \vee b$.

Associativity rules for \vee and \wedge are usually not paid much attention to, as these connectives are associative: for example, $a \vee (b \vee c)$ is equivalent to $(a \vee b) \vee c$. For implication, one can define $\phi_1 \rightarrow \phi_2 \rightarrow \phi_3$ to mean $\phi_1 \rightarrow (\phi_2 \rightarrow \phi_3)$, but this expression would practically always be parenthesized anyway, so adopting this precedence rule is not important.

Definition 2.2 (Valuation of atomic propositions) *A valuation $v : X \rightarrow \{0, 1\}$ is a mapping from atomic propositions $X = \{x_1, \dots, x_n\}$ to truth-values 0 and 1.*

Valuations can be extended to formulas $\phi \in \mathcal{L}(X)$, i.e. $v : \mathcal{L}(X) \rightarrow \{0, 1\}$.

Definition 2.3 (Valuation of propositional formulas) *A given valuation $v : X \rightarrow \{0, 1\}$ of atomic propositions can be extended to a valuation of arbitrary propositional formulas over X .*

$$\begin{aligned}
 v(\neg\alpha) &= 1 \text{ iff } v(\alpha) = 0 \\
 v(\top) &= 1 \\
 v(\perp) &= 0 \\
 v(\alpha \wedge \beta) &= 1 \text{ iff } v(\alpha) = 1 \text{ and } v(\beta) = 1 \\
 v(\alpha \vee \beta) &= 1 \text{ iff } v(\alpha) = 1 \text{ or } v(\beta) = 1 \\
 v(\alpha \rightarrow \beta) &= 1 \text{ iff } v(\alpha) = 0 \text{ or } v(\beta) = 1 \quad (\text{or, equivalently, } v(\alpha) \leq v(\beta)) \\
 v(\alpha \leftrightarrow \beta) &= 1 \text{ iff } v(\alpha) = v(\beta)
 \end{aligned}$$

Example 2.4 Let $v(a) = 1, v(b) = 1, v(c) = 1, v(d) = 1$. Then

$$\begin{aligned}
 v(a \vee b \vee c) &= 1, \\
 v(\neg a \rightarrow b) &= 1, \\
 v(a \rightarrow \neg b) &= 0.
 \end{aligned}$$

■

Definition 2.5 (Subformulas and immediate subformulas) *We recursively define immediate subformulas $isfma(\phi)$ of a formula ϕ as follows.*

double negation	$\neg\neg\alpha$	\equiv	α
associativity \vee	$\alpha \vee (\beta \vee \gamma)$	\equiv	$(\alpha \vee \beta) \vee \gamma$
associativity \wedge	$\alpha \wedge (\beta \wedge \gamma)$	\equiv	$(\alpha \wedge \beta) \wedge \gamma$
commutativity \vee	$\alpha \vee \beta$	\equiv	$\beta \vee \alpha$
commutativity \wedge	$\alpha \wedge \beta$	\equiv	$\beta \wedge \alpha$
distributivity $\wedge \vee$	$\alpha \wedge (\beta \vee \gamma)$	\equiv	$(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$
distributivity $\vee \wedge$	$\alpha \vee (\beta \wedge \gamma)$	\equiv	$(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
idempotence \vee	$\alpha \vee \alpha$	\equiv	α
idempotence \wedge	$\alpha \wedge \alpha$	\equiv	α
absorption 1	$\alpha \wedge (\alpha \vee \beta)$	\equiv	α
absorption 2	$\alpha \vee (\alpha \wedge \beta)$	\equiv	α
De Morgan's law 1	$\neg(\alpha \vee \beta)$	\equiv	$(\neg\alpha) \wedge (\neg\beta)$
De Morgan's law 2	$\neg(\alpha \wedge \beta)$	\equiv	$(\neg\alpha) \vee (\neg\beta)$
contraposition	$\alpha \rightarrow \beta$	\equiv	$\neg\beta \rightarrow \neg\alpha$
negation \top	$\neg\top$	\equiv	\perp
negation \perp	$\neg\perp$	\equiv	\top
constant \perp	$\alpha \wedge \neg\alpha$	\equiv	\perp
constant \top	$\alpha \vee \neg\alpha$	\equiv	\top
elimination $\top \vee$	$\top \vee \alpha$	\equiv	\top
elimination $\top \wedge$	$\top \wedge \alpha$	\equiv	α
elimination $\perp \vee$	$\perp \vee \alpha$	\equiv	α
elimination $\perp \wedge$	$\perp \wedge \alpha$	\equiv	\perp
elimination $\perp \rightarrow$	$\perp \rightarrow \alpha$	\equiv	\top
elimination $\perp \rightarrow$	$\alpha \rightarrow \perp$	\equiv	$\neg\alpha$
elimination $\top \rightarrow$	$\top \rightarrow \alpha$	\equiv	α
elimination $\top \rightarrow$	$\alpha \rightarrow \top$	\equiv	\top
commutativity \leftrightarrow	$\alpha \leftrightarrow \beta$	\equiv	$\beta \leftrightarrow \alpha$
elimination $\top \leftrightarrow$	$\top \leftrightarrow \alpha$	\equiv	α
elimination $\perp \leftrightarrow$	$\perp \leftrightarrow \alpha$	\equiv	$\neg\alpha$

Table 2.1: Propositional Equivalences

1. If $\phi = x$ where x is an atomic proposition, then $isfma(\phi) = \emptyset$.
2. $isfma(\alpha \wedge \beta) = \{\alpha, \beta\}$
3. $isfma(\alpha \vee \beta) = \{\alpha, \beta\}$
4. $isfma(\alpha \rightarrow \beta) = \{\alpha, \beta\}$
5. $isfma(\alpha \leftrightarrow \beta) = \{\alpha, \beta\}$
6. $isfma(\neg\alpha) = \{\alpha\}$

Subformulas of a formula ϕ are defined by $sfma(\phi) = \{\phi\} \cup \bigcup_{\psi \in isfma(\phi)} sfma(\psi)$.

2.2 Equivalences

Table 2.1 lists equivalences that hold in the propositional logic. Replacing one side of any of these equivalences by the other - in any formula - does not change the Boolean function represented by the formula.

These equivalences have several applications, including translating formulas into *normal forms* (Section 4), and simplifying formulas. For example, all occurrences \top and \perp of the constant symbols (except one, if the whole formula reduces to \top or \perp) can be eliminated with the equivalences containing these symbols.

2.3 Truth-tables

All valuations relevant to a formula are often tabulated as *truth-tables* so that each row corresponds to a valuation. Truth-tables are used for analyzing the most basic properties of formulas.

Valuations for formulas containing exactly one connective are as follows.

α	$\neg\alpha$	α	β	$\alpha \wedge \beta$	α	β	$\alpha \vee \beta$	α	β	$\alpha \rightarrow \beta$	α	β	$\alpha \leftrightarrow \beta$
0	1	0	0	0	0	0	0	0	0	1	0	0	1
1	0	0	1	0	0	1	1	0	1	1	0	1	0
		1	0	0	1	0	1	1	0	0	1	0	0
		1	1	1	1	1	1	1	1	1	1	1	1

Truth-tables for more complex formulas ϕ are constructed as follows.

1. Create columns for all n atomic propositions that occur in ϕ .
2. Create columns for all subformulas $\text{sfma}(\phi)$, with columns for shorter formulas left of the bigger formulas.
3. Create 2^n rows corresponding to all the valuations of the atomic propositions.
4. Fill in truth-values in the remaining cells from left to right: every row in the column for a formula ψ is filled in by looking at the contents of the immediate subformulas $\text{isfma}(\psi)$ in the same row (residing to the left), based on the truth table for the outermost connective in ψ .

Example 2.6 The truth-table for $\neg B \wedge (A \rightarrow B)$ is the following.

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

■

2.4 Logical Consequence, Satisfiability, Validity

Definition 2.7 (Logical Consequence) A formula ϕ is a logical consequence of $\Sigma = \{\phi_1, \dots, \phi_n\}$, denoted by $\Sigma \models \phi$, if and only if for all valuations v , if $v \models \Sigma$ then $v \models \phi$.

Logical consequences ϕ of Σ are what can be inferred with certainty from Σ : if Σ holds, then ϕ is guaranteed to hold too.

Example 2.8 From “if it is a weekday today then most people are working today”, and “it is a weekday today” it follows that “most people are working today”.

$$\{w \rightarrow p, w\} \models p$$

■

Example 2.9 $\{a, a \rightarrow b, b \rightarrow c\} \models c$

■

Example 2.10 $\{m \vee t, m \rightarrow w, t \rightarrow w\} \models w$

■

Definition 2.11 (Validity) A formula ϕ is valid if and only if $v(\phi) = 1$ for all valuations v .

Example 2.12 $x \vee \neg x$ is valid.

$x \rightarrow (x \vee y)$ is valid.

$x \vee y$ is not valid.

■

	$\alpha \wedge \beta$	\models	α
	α	\models	$\alpha \vee \beta$
	α	\models	$\beta \rightarrow \alpha$
	$\neg \alpha$	\models	$\alpha \rightarrow \beta$
Modus Ponens	$\alpha \wedge (\alpha \rightarrow \beta)$	\models	β
	$\neg \beta \wedge (\alpha \rightarrow \beta)$	\models	$\neg \alpha$

Table 2.2: Useful Logical Consequences

Definition 2.13 (Satisfiability) A formula ϕ is satisfiable if and only if there is at least one valuation v such that $v(\phi) = 1$.

A set $\{\phi_1, \dots, \phi_n\}$ is satisfiable if there is at least one valuation v such that $v(\phi_i) = 1$ for all $i \in \{1, \dots, n\}$.

The best known result in computational complexity theory is that the propositional satisfiability problem SAT is NP-complete. SAT is defined as decision problem of deciding whether a given formula ϕ is satisfiable, often stated as $\phi \in \text{SAT}$.

Example 2.14 The following formulas are satisfiable: x , $x \wedge y$, and $x \vee \neg x$. ■

Example 2.15 The following formulas are not satisfiable: $x \wedge \neg x$, \perp , $a \wedge (a \rightarrow b) \wedge \neg b$. ■

A satisfiable formula is also said to be *consistent*. A formula that is not satisfiable is *unsatisfiable*, *inconsistent*, or *contradictory*.

Satisfiability means *logically possible*: it is possible that the formula is true (if the truth-values of the atomic propositions in the formula are chosen right.)

Importantly, there are close connections between satisfiability, validity, and logical consequence. These connections allow *reducing* questions concerning these concepts to each other, which allows choosing one of these concepts as the most basic one (for example implemented in algorithms for reasoning in the propositional logic), and reducing the other concepts to the basic one.

Theorem 2.16 (Validity vs. Logical Consequence 1) The formula ϕ is valid if and only if $\emptyset \models \phi$.

Theorem 2.17 (Validity vs. Logical Consequence 2) $\{\phi_1, \dots, \phi_n\} \models \phi$ if and only if $(\phi_1 \wedge \dots \wedge \phi_n) \rightarrow \phi$ is valid.

Theorem 2.18 (Satisfiability vs. Validity) ϕ is satisfiable if and only if $\neg \phi$ is not valid.

Theorem 2.19 (Logical Consequence vs. Satisfiability) $\Sigma \models \phi$ if and only if $\Sigma \cup \{\neg \phi\}$ is not satisfiable.

In practice, algorithms for the propositional logic implement *satisfiability*. Other reasoning tasks are reduced to satisfiability testing.

Chapter 3

Automated Reasoning in the Propositional Logic

In this chapter we discuss the main methods for automating inference in the propositional logic.

The simplest method is based on truth-tables (Section 3.1), which enumerate all valuations of the relevant atomic propositions, and questions about satisfiability and logical consequence can be answered by evaluating the truth-values of the relevant formulas for every valuation.

Truth-tables are easy to implement as a program, but are impractical when the number of atomic propositions is higher than 20 or 30. The algorithms used in practice (Section 3.2) are based on searching a tree in which each path starting from the root node represents a (partial) valuation of the atomic propositions. The size of this tree is in practice orders of magnitudes smaller than corresponding truth-tables, and for finding one satisfying valuation not the whole tree needs to be traversed. Finally, only one path of the tree, from the root node to the current leaf node, needs to be kept in the memory at a time, which reduces the memory requirements substantially. This enables the use of these algorithms for formulas of the size encountered in solving important problems in AI and computer science in general, with up to hundreds of thousands of atomic propositions and millions of clauses.

3.1 Truth-Tables

The most basic method for testing satisfiability of a formula ϕ is to construct the truth-table for ϕ , representing all valuations of atomic propositions occurring in ϕ , and then check whether the column for ϕ contains at least one 1: ϕ is satisfiable if and only if the column for ϕ contains at least one 1. Obviously, this is because all possible valuations are represented in the truth-table, and a formula is satisfiable if and only if it is true in at least one valuation.

Example 3.1 The truth-table for $\neg B \wedge (A \rightarrow B)$:

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

$\neg B \wedge (A \rightarrow B)$ is satisfiable because it is true when both A and B are false, corresponding to the first row. ■

The second important problem is testing for logical consequence.

Algorithm 3.2 To test whether ϕ a logical consequence of Σ do the following.

1. Construct the truth-table for ϕ and Σ .
2. Mark every row in which the columns for all members of Σ contain 1.
3. Check that there is 1 in the column for ϕ for every marked row.

This algorithm tests whether ϕ is true in every valuation in which Σ is true.

If there is marked row with ϕ false, then we have a *counterexample* to $\Sigma \models \phi$: Σ is true but ϕ is false.

Example 3.3 Test $\{a \rightarrow b, b \rightarrow c\} \models a \rightarrow c$:

a	b	c	$a \rightarrow b$	$b \rightarrow c$	$a \rightarrow c$
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	1	1

$\{a \rightarrow b, b \rightarrow c\}$ true on 4 rows, and we need to confirm that $a \rightarrow c$ is true on those rows. ■

Truth-tables are a practical method to be used by hand only up to a couple of atomic propositions. With 6 atomic propositions there are 64 rows, which barely fits on a sheet of paper. With computer it is possible to use the truth-table method for up to 20 or 25 variables. After that memory consumption and computation times will be impractically high. In many applications the number of atomic propositions is in the hundreds, thousands, or even hundreds of thousands, and completely different types of algorithms are necessary.

The exponentiality in the truth-table is inherent in all algorithms for testing satisfiability or logical consequence, similarly to many of challenging problems in AI. For testing satisfiability, and many other problems, this exponentiality is an indication of NP-completeness [Coo71, GJ79]. The exponentiality is known as the *combinatorial explosion*, and is caused by the exponential number 2^n of combinations of values of n atomic propositions. NP-completeness of the satisfiability problem suggests that all algorithms for the problem necessarily need to do an exponential amount of computation, in the worst case, and the problem is *inherently hard* in this sense. Work on practically useful algorithms for the satisfiability problem try to avoid the exponential worst-case by using methods for pruning the search tree, and by using heuristics for choosing the traversal order and a tree structure that minimizes the time it takes to find a satisfying valuation or to determine that none exist.

3.2 Tree-Search Algorithms

More practical algorithms for solving the satisfiability problem do not explicitly go through all possible valuations of the atomic propositions, and do not store all of them in a big table that has to be kept in the memory.

The simplest algorithm for testing satisfiability that does not construct a truth-table does a depth-first traversal of a binary tree that represents all valuations. This is illustrated in Figure 3.1 for atomic propositions $X = \{a, b, c\}$.

During the traversal, this algorithm only needs to store in the memory the (partial) valuation on the path from the root node to the current node, and hence its memory consumption is linear in the number of atomic propositions.

The runtime of the algorithm, however, is still exponential in the number of atomic propositions for all unsatisfiable formulas, because of the size of the tree. For satisfiable formulas the whole tree does not need to be traversed, but since the first satisfying valuation may be encountered late in the traversal, in practice this algorithm is also usually exponential.

The improvements to this basic tree-search algorithm are based on the observation that many of the (partial) valuations in the tree make the formula false, and this is easy to detect during the traversal. Hence large parts of the search tree can be pruned.

In the following, we assume that the formula has been translated into the conjunctive normal form (Section 4). Practically all leading algorithms assume the input to be in CNF.

The fundamental observation in pruning the search tree is the following. Let v be a valuation and $l \vee l_1 \vee l_2 \vee \dots \vee l_n$ a clause such that $v(l_1) = 0, \dots, v(l_n) = 0$,

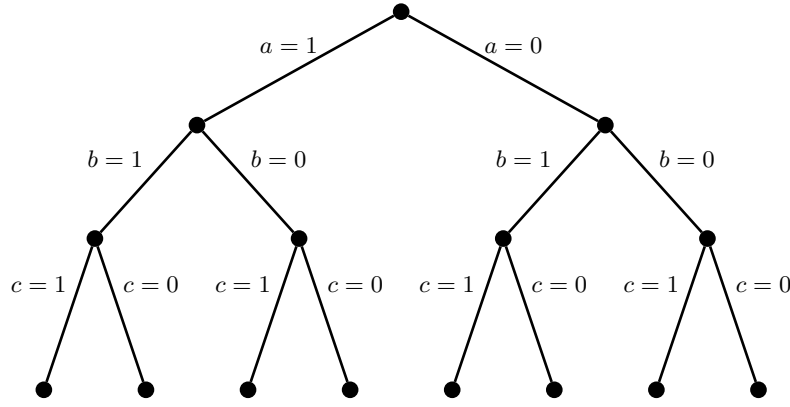


Figure 3.1: All valuations as a binary tree

Since we are trying to find a valuation that makes our clause set *true*, this clause has to be *true* (similarly to all other clauses), and it can only be *true* if the literal l is *true*.

Hence if the partial valuation corresponding to the current node in the search tree makes, in some clause, all literals *false* except one (call it l) that still does not have a truth-value, then l has to be made true. If l is an atomic proposition x , then the current valuation has to assign $v(x) = 1$, and if $l = \neg x$ for some atomic proposition x , then we have to assign $v(x) = 0$. We call these *forced assignments*.

Consider the search tree in Figure 3.1 and the clauses $\neg a \vee b$ and $\neg b \vee c$. The leftmost child node of the root node, which is reached by following the arc $a = 1$, corresponds to the partial valuation $v = \{(a, 1)\}$ that assigns $v(a) = 1$ and leaves the truth-values of the remaining atomic propositions open. Now with clause $\neg a \vee b$ we have the forced assignment $v(b) = 1$. Now, with clause $\neg b \vee c$ also $v(c) = 1$ is a forced assignment.

Hence assigning $a = 1$ forces the assignments of both of the remaining variables. This means that the leftmost subtree of the root node essentially only consists of one path that leads to the leftmost leaf node. The other leaf nodes in the subtree, corresponding to valuations $\{a = 1, b = 1, c = 0\}$, $\{a = 1, b = 0, c = 0\}$, and $\{a = 1, b = 0, c = 1\}$, would not be visited, because it is obvious that they falsify at least one of the clauses. Since we are doing the tree search in order to determine the satisfiability of our clause set, the computation in the subtree with $a = 1$ can be stopped here, continuing from the root node with $a = 0$ to the rightmost subtree.

3.2.1 Unit Resolution

What we described as forced assignments is often formalized as the inference rule Unit Resolution.

Theorem 3.4 (Unit Resolution) *In*

$$\frac{l \quad \bar{l} \vee l_1 \vee l_2 \vee \cdots \vee l_n}{l_1 \vee l_2 \vee \cdots \vee l_n}$$

the formula below the line is a logical consequence of the formulas above the line.

Here the *complementation* \bar{l} of a literal l is defined by $\bar{x} = \neg x$ and $\overline{\neg x} = x$.

Example 3.5 From $\{A \vee B \vee C, \neg B, \neg C\}$ one can derive A by two applications of the Unit Resolution rule. ■

A special case of the Unit Resolution rule is when both clauses are unit clauses (consisting of one literal only.) Since the $n = 0$ in this case, the formula below the line has 0 disjuncts. We identify a chain-disjunction with 0 literals with the constant false \perp , corresponding to the *empty clause*.

$$\frac{l \quad \bar{l}}{\perp}$$

Any clause set with the empty clause is *unsatisfiable*.

The Unit Resolution rule is a special case of the Resolution rule (which we will not be discussing in more detail.)

- 1: **procedure** DPLL(S)
- 2: $S := UP(S)$;
- 3: **if** $\perp \in S$ **then return** false;
- 4: $x :=$ any atomic proposition such that $\{x, \neg x\} \cap S = \emptyset$;
- 5: **if** no such x exists **then return** true;
- 6: **if** DPLL($S \cup \{x\}$) **then return** true;
- 7: **return** DPLL($S \cup \{\neg x\}$);

Figure 3.2: The DPLL procedure

Theorem 3.6 (Resolution) In

$$\frac{l \vee l'_1 \vee \dots \vee l'_m \quad \bar{l} \vee l_1 \vee \dots \vee l_n}{l'_1 \vee \dots \vee l'_m \vee l_1 \vee l_2 \vee \dots \vee l_n}$$

the formula below the line is a logical consequence of the formulas above the line.

3.2.2 Subsumption

If a clause set contains two clauses with literals $\{l_1, \dots, l_n\}$ and $\{l_1, \dots, l_n, l_{n+1}, \dots, l_m\}$, respectively, then the latter clause can be removed, as it is a logical consequence of the former. The two clause sets, with and without the second clause, are *true* in exactly the same valuations.

Example 3.7 Applying the Subsumption rule to $\{A \vee B \vee C, A \vee C\}$ yields the equivalent set $\{A \vee C\}$. ■

If the shorter clause is a unit clause, then this is called Unit Subsumption.

3.2.3 Unit Propagation

Performing all unit resolutions exhaustively leads to the Unit Propagation procedure, also known as Boolean Constraint Propagation (BCP).

In practice, when a clause $l_1 \vee \dots \vee l_n$ with $n > 2$ can be resolved with a unit clause, the shorter clause of length $n - 1 > 1$ would not be useful, so, instead of producing these $n - 1$ clauses, of lengths $1, 2, \dots, n - 1$, unit propagation attempts to do the same useful inferences of unit clauses only, without producing the longer intermediat clauses: only produce a new clause when the complements of all but 1 of the n literals of the clause have been inferred. This can be formalized as the following rule, with $n \geq 2$.

$$\frac{\bar{l}_1 \quad \bar{l}_2 \quad \bar{l}_3 \quad \dots \quad \bar{l}_{n-1} \quad l_1 \vee \dots \vee l_n}{l_n}$$

We denote by $UP(S)$ the clause set obtained by performing all possible unit propagations to a clause set S , followed by applying the subsumption rule.

3.2.4 The Davis-Putnam Procedure

The Davis-Putnam-Logemann-Loveland procedure (DPLL) [DLL62], often known simply as the Davis-Putnam procedure, is given in Figure 3.2.

We have sketched the algorithm so that the first branch assigns the chosen atomic proposition x true in the first subtree (line 6) and false in the second (line 7). However, these two branches can be traversed in either order (exchanging x and $\neg x$ on lines 6 and 7), and efficient implementations use this freedom by using powerful heuristics for choosing first the atomic proposition (line 4) and then choosing its truth-value.

Theorem 3.8 Let S be a set of clauses. Then DPLL(S) returns true if and only if S is satisfiable.

References

Automation of reasoning in the classical propositional logic started in the works of Davis and Putnam and their co-workers, resulting in the Davis-Putnam-Logemann-Loveland procedure [DLL62] (earlier known as the Davis-Putnam procedure) and other procedures [DP60].

Efforts to implement the Davis-Putnam procedure efficiently lead to fast progress from mid-1990ies on.

The current generation of solvers for solving the SAT problem are mostly based on the Conflict-Driven Clause Learning procedure which emerged from the work of Marques-Silva and Sakallah [MSS96]. Many of the current leading implementation techniques and heuristics for selecting the decision variables come from the work on the Chaff solver [MMZ⁺01]. Several other recent developments have strongly contributed to current solvers [PD07, AS09].

Unit propagation can be implemented to run in linear time in the size of the clause set, which in the simplest variants involves updating a counter every time a literal in a clause gets a truth-value [DG84]. Modern SAT solvers attempt to minimize memory accesses (cache misses), and even counter-based linear time procedures are considered too expensive, and unit propagation schemes that do not use counters are currently used [MMZ⁺01].

Also stochastic local search algorithms have been used for solving the satisfiability problem [SLM92], but they are not used in many applications because of their inability to detect unsatisfiability.

The algorithms in this chapter assumed the formulas to be in CNF. As discussed in Section 4, the translation into CNF may in some cases exponentially increase the size of the formula. However, for the purposes of satisfiability testing, there are transformations to CNF that are of linear size and preserve satisfiability (but not logical equivalence.) These transformations are used for formulas that are not already in CNF [Tse68, CMV09].

Chapter 4

Normal Forms and Logic Data Structures

Arbitrary propositional formulas can be translated into syntactically restricted forms which are still capable of expressing all Boolean functions. Use of such normal forms can serve two main purposes. First, it may be more straightforward to define algorithms and inference methods for formulas of a simple form. The *resolution rule* (Section 3.2.1) is an example of this. Second, the process of translating a formula into certain normal form does much of the work in solving important computational problems related to propositional formulas. For example, an answer to the question of whether a formula is valid is obtained as a by-product of translating the formula normal forms such as DNF or OBDD. A number of other operations on propositional formulas are in practice much more efficient when the formulas are in certain normal forms rather than unrestricted propositional formulas.

In this chapter we present some of the best known normal forms, including the disjunctive and conjunctive normal forms, the negation normal form, as well as binary decision diagrams.

Definition 4.1 (Literals) *If x is an atomic proposition, then x and $\neg x$ are literals.*

Definition 4.2 (Clauses) *If l_1, \dots, l_n are literals, then $l_1 \vee \dots \vee l_n$ is a clause.*

Definition 4.3 (Terms) *If l_1, \dots, l_n are literals, then $l_1 \wedge \dots \wedge l_n$ is a term.*

Definition 4.4 (Conjunctive Normal Form) *If ϕ_1, \dots, ϕ_m are clauses, then the formula $\phi_1 \wedge \dots \wedge \phi_m$ is in conjunctive normal form.*

Example 4.5 The following formulas are in conjunctive normal form.

$$\begin{aligned} &\neg x \\ &\neg x_1 \vee \neg x_2 \\ &x_3 \wedge x_4 \\ &(\neg x_1 \vee \neg x_2) \wedge \neg x_3 \wedge (x_4 \vee \neg x_5) \end{aligned}$$

■

Table 4.1 lists rules that transform any formula into CNF. These rules are instances of the equivalence listed in Table 2.1

Algorithm 4.6 *Any formula can be translated into conjunctive normal form as follows.*

1. Eliminate connective \leftrightarrow (Rule 4.1).
2. Eliminate connective \rightarrow (Rule 4.2).
3. Move \neg inside \vee and \wedge (Rules 4.3, 4.4 and 4.5), also eliminating double negations.
4. Move \wedge outside \vee (Rules 4.6 and 4.7).

Notice that the last step of the transformation multiplies the number of copies of subformulas α and γ . For some classes of formulas this transformation therefore leads to exponentially big normal forms.

$$\alpha \leftrightarrow \beta \rightsquigarrow (\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha) \quad (4.1)$$

$$\alpha \rightarrow \beta \rightsquigarrow \neg\alpha \vee \beta \quad (4.2)$$

$$\neg(\alpha \vee \beta) \rightsquigarrow \neg\alpha \wedge \neg\beta \quad (4.3)$$

$$\neg(\alpha \wedge \beta) \rightsquigarrow \neg\alpha \vee \neg\beta \quad (4.4)$$

$$\neg\neg\alpha \rightsquigarrow \alpha \quad (4.5)$$

$$\alpha \vee (\beta \wedge \gamma) \rightsquigarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \quad (4.6)$$

$$(\alpha \wedge \beta) \vee \gamma \rightsquigarrow (\alpha \vee \gamma) \wedge (\beta \vee \gamma) \quad (4.7)$$

Table 4.1: Rewriting Rules for Translating Formulas into Conjunctive Normal Form

Formulas in CNF are often represented as sets of clauses (rather than a conjunction of clauses), with clauses represented as sets of literals. The placement of connectives \vee and \wedge can be left implicit because of the simple structure of CNF. The CNF $(A \vee B \vee C) \wedge (\neg A \vee D) \wedge E$ can be represented as $\{A \vee B \vee C, \neg A \vee D, E\}$ or, equivalently, as $\{\{A, B, C\}, \{\neg A, D\}, \{E\}\}$. Any clause set $\{\emptyset, \dots\}$ that contains the empty clause \emptyset is logically equivalent to \perp , and the empty clause set \emptyset is equivalent to \top .

Example 4.7 Translate $A \vee B \rightarrow (B \leftrightarrow C)$ into CNF.

$$\begin{aligned} &\rightsquigarrow A \vee B \rightarrow (B \rightarrow C) \wedge (C \rightarrow B) \\ &\rightsquigarrow \neg(A \vee B) \vee ((\neg B \vee C) \wedge (\neg C \vee B)) \\ &\rightsquigarrow (\neg A \wedge \neg B) \vee ((\neg B \vee C) \wedge (\neg C \vee B)) \\ &\rightsquigarrow (\neg A \vee ((\neg B \vee C) \wedge (\neg C \vee B))) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B))) \\ &\rightsquigarrow (\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B))) \\ &\rightsquigarrow (\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B) \wedge (\neg B \vee \neg B \vee C) \wedge (\neg B \vee \neg C \vee B) \end{aligned}$$

■

Another often used normal form is the Disjunctive Normal Form (DNF).

Definition 4.8 (Disjunctive Normal Form) If ϕ_1, \dots, ϕ_m are terms, then the formula $\phi_1 \vee \dots \vee \phi_m$ is in disjunctive normal form.

Disjunctive normal forms are found with a procedure similar to that for CNF. The only difference is that a different set of distributivity rules is applied, for moving \wedge inside \vee instead of the other way round.

A less restrictive normal form that includes both CNF and DNF is the Negation Normal Form (NNF). It is useful in many types of automated processing of formulas because of its simplicity and the fact that translation into NNF only minimally affects the size of a formula, unlike the translations into DNF and CNF which may increase the size exponentially.

Definition 4.9 (Negation Normal Form) Let X be a set of atomic propositions.

1. \perp and \top are in negation normal form.
2. x and $\neg x$ for any $x \in X$ are in negation normal form.
3. $\phi_1 \wedge \phi_2$ is in negation normal form if ϕ_1 and ϕ_2 both are.
4. $\phi_1 \vee \phi_2$ is in negation normal form if ϕ_1 and ϕ_2 both are.

No other formula is in negation normal form.

In other words, ϕ is in NNF if and only if it only contains the connectives \neg , \vee and \wedge , and \neg only occurs right in front of atomic propositions.

After performing the first three steps in the translation into CNF, a formula is in NNF. Notice that only the application of the distributivity equivalences increases the number of atomic propositions in the CNF translation. Hence the translation into NNF does not affect the number of occurrences of atomic propositions or the total number of connectives \vee and \wedge . Therefore the size of the resulting formula can be bigger only because of the

higher number of negation symbols \neg , and there cannot be more of \neg symbols than there are occurrences of atomic propositions.

4.1 Complexity of Validity and Satisfiability for DNF and CNF

Lemma 4.10 *A formula in CNF is valid if and only if for every clause in the formula, there is some atomic proposition x such that both x and $\neg x$ are in the clause.*

Lemma 4.11 *A formula in DNF is satisfiable if and only if at least one of its terms does not contain both x and $\neg x$ for any atomic proposition x .*

So, for formulas in CNF, testing for validity is a simple syntactic operation that can be done in polynomial time, and for formulas in DNF, satisfiability testing is similarly a polynomial time operation. These operations for arbitrary propositional formulas are far harder, co-NP-complete and NP-complete, respectively. For NNF these operations are similarly hard.

While validity for CNF is easy, testing satisfiability is still NP-complete. Similarly, validity for DNF is co-NP-complete.

Constructing a CNF or a DNF for an unrestricted propositional formula may take exponential time (due to the doubling of size of subformulas when applying the distributivity rules), but as a result, some of the computational problems are exponentially easier.

Validity and satisfiability for NNF are respectively co-NP-complete and NP-complete, because DNF and CNF are special cases of NNF.

4.2 Formulas as a Data Structure

Traditional application of logic is as a language of representing data and knowledge and for deductively making inferences from it.

A different use of logic emerged in the 1980ies [Bry86, CBM90, Bry92], when logical formulas were used as a data structure for representing sets (of valuations) and relations (on valuations). As will be discussed in Section 5.1, many logical operations have set-theoretic counterparts, for example union can be identified with disjunction. It turned out that logic-based data structures could scale up to much bigger sets and relations than conventional (enumerative) data structures (trees, lists, arrays, and so on.)

To use formulas as a data structure, we need operations for constructing formulas, for manipulating formulas, and for asking questions about formulas.

Constructing formulas:	
Negation:	Given ϕ , form $\neg\phi$
Conjunction:	Given ϕ_1 and ϕ_2 , form $\phi_1 \wedge \phi_2$
Disjunction:	Given ϕ_1 and ϕ_2 , form $\phi_1 \vee \phi_2$

Manipulating formulas:	
Substitution:	Given ϕ , construct $\phi[\psi/x]$ by replacing occurrences of x by ψ
Existential Abstraction:	Given ϕ , construct $\phi[\top/x] \vee \phi[\perp/x]$
Universal Abstraction:	Given ϕ , construct $\phi[\top/x] \wedge \phi[\perp/x]$

A commonly occurring special case of Substitution is Renaming, replacing occurrences of atomic propositions by other atomic propositions.

The abstraction operations are central in implementing relational projection operations (Section 5.2) and multiplication of Boolean matrices.

Analyzing formulas:	
Satisfiability:	Given ϕ , is ϕ satisfiable?
Validity:	Given ϕ , is ϕ valid?
Logical Consequence:	Given ϕ_1 and ϕ_2 , does $\phi_1 \models \phi_2$ hold?
Model Counting:	Given ϕ over X , for how many valuations $v : X \rightarrow \{0, 1\}$ does $v \models \phi$ hold?

We have already seen that with some normal forms, some of these operations are easier than with arbitrary propositional formulas. For unlimited formulas, the construction operations are efficient, but many of the other operations in turn are not. Similarly, for limited normal forms of propositional formulas, some of the construction operations can be expensive, but as soon as the normal form has been constructed, some of the other operations are far easier than with general propositional formulas.

Interestingly, for some normal forms, although their construction can be expensive (exponential in the worst case), the cheaper other operations have made them far preferable to general propositional logic.

A case in point is the abstraction operations, which are central in relational projections (Section 5.2), and for which no good algorithms exist for general propositional formulas, but which can often be very effectively computed for example with Binary Decision Diagrams (see next section).

4.3 Binary Decision Diagrams

Any propositional formula can be turned to a case analysis on a single atomic proposition by using the following equivalence-preserving transformation (also known as *Shannon expansion*.)

Theorem 4.12 (Boole's Expansion) *For any propositional formula ϕ over X and any atomic proposition $x \in X$, the formula $(x \wedge \phi[\top/x]) \vee (\neg x \wedge \phi[\perp/x])$ is logically equivalent to ϕ .*

One can also express this in terms of the ternary Boolean connective ITE (or IF-THEN-ELSE)

$$\text{ITE}(x, \phi_1, \phi_2) = (x \wedge \phi_1) \vee (\neg x \wedge \phi_2)$$

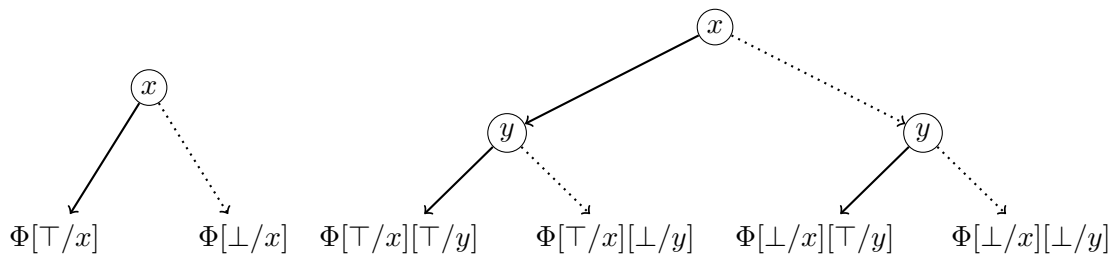
as

$$\phi \equiv \text{ITE}(x, \phi[\top/x], \phi[\perp/x]).$$

The special form of the formulas resulting from the expansion operation gives rise to a graphical representation of any propositional formula: at the topmost level, the formula represents a case analysis on the value of one atomic proposition – either x is true, or x is false – and depending on the value of x , either $\phi[\top/x]$ or $\phi[\perp/x]$ equals the value of the original formula. And, neither of these latter formulas contain occurrences of x .

The binary decision diagram construction proceeds by eliminating x first, and then eliminating further variables from the subformulas $\phi[\top/x]$ and $\phi[\perp/x]$.

The idea is that after translating ϕ to $(x \wedge \phi[\top/x]) \vee (\neg x \wedge \phi[\perp/x])$, we continue applying Boole's expansion to the subformulas $\phi[\top/x]$ and $\phi[\perp/x]$ for other atomic propositions.

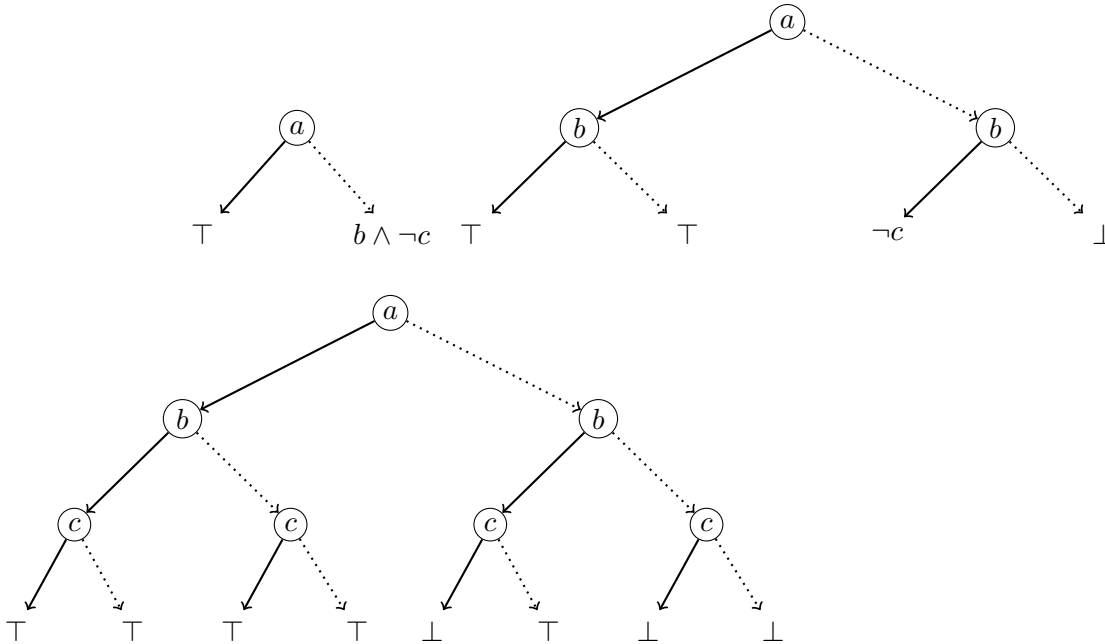


The dashed line is followed when the atomic proposition in the node is *false*, and the non-dashed one when it is *true*.

Doing this systematically for any formula with n atomic propositions yields a binary tree with 2^n leaf nodes labelled with formulas that consist of the constants \top and \perp and connectives only, without any atomic propositions.

Our first observation is that the formulas in the leaves can always be simplified to \top or \perp . At this point, we essentially have a graphical representation of the truth table constructed for the n variables: for any valuation of the atomic propositions we can determine the corresponding truth-value of the formula by following the path from root that correspond to the valuation, and reading the truth-value from the resulting leaf node.

Example 4.13 If we do this for the formula $a \vee (b \wedge \neg c)$, we get the following binary trees.



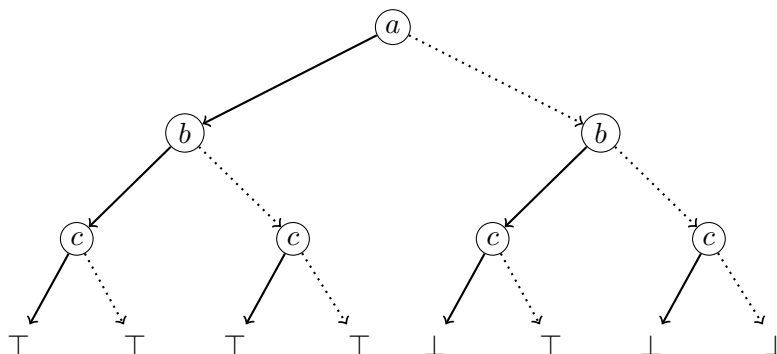
Our second observation is that this tree can often be represented far more compactly as a directed acyclic graph, in which two isomorphic subgraphs are identified. First, there will be only two leaf nodes \top and \perp (in the special cases for valid and unsatisfiable formulas there is only one leaf node.) After this, higher up in the graph, if two nodes have the same child nodes, the nodes can be merged to one.

A third observation is that if both outgoing arcs of a node point to the same subgraph, then this node can be eliminated by directing the arc from its parents to the node's child node.

After these operations, we have a *reduced ordered binary decision diagram* (ROBDD, OBDD). The *reduced* refers to the merging of nodes that are logically equivalent. The *ordered* refers to the diagram being constructed with a fixed total ordering of the atomic propositions. Sometimes more relaxed binary decision diagrams are used, especially without the ordering condition.

Observe that for a given variable ordering, the OBDD is unique up to an isomorphism, that is, all OBDDs generated for a given formula and a given variable ordering are isomorphic to each other, represented by essentially the same graph.

Example 4.14 Consider the binary tree we constructed for the formula $a \vee (b \wedge \neg c)$.

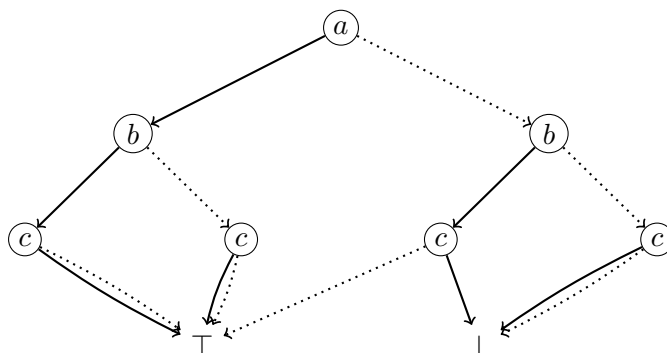


The correspondence with the truth table is exact. (Imagine rotating the truth table 90 degrees clockwise.)

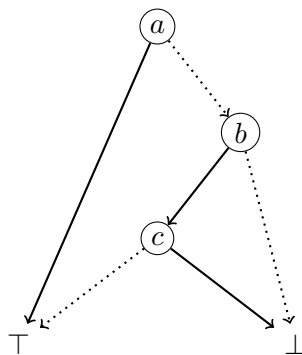
a	b	c	$a \vee (b \wedge \neg c)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



Example 4.15 Merge all \top nodes and all \perp nodes.

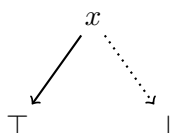


Then eliminate all nodes for which both arcs point to the same node. Above this means first eliminating three of the c nodes, after which the leftmost b node also needs to be eliminated. The resulting diagram is an OBDD.



The above construction progressed top-down starting from an arbitrary formula, generating an exponential size tree as an intermediate stage, before arriving at an OBDD. This OBDD could be constructed bottom up from the original formula, by first generating the OBDD for atomic propositions, and then constructing the OBDDs for $\phi \wedge \phi'$ and $\phi \vee \phi'$ directly from the OBDDs for ϕ and ϕ' .

The OBDD for an atomic proposition x is the following.



An OBDD is negated by swapping the leaf nodes \top and \perp .

By Boole's expansion theorem, $\phi_1 \circ \phi_2$ with any connective \circ is logically equivalent to $(x \wedge \phi_1[\top/x] \circ \phi_2[\top/x]) \vee (\neg x \wedge \phi_1[\perp/x] \circ \phi_2[\perp/x])$. This yields as a way of constructing an OBDD for any formula $\phi_1 \circ \phi_2$, given the OBDDs for ϕ_1 and ϕ_2 .

This is what is known as the *apply* procedure.

- If the root nodes of both constituent OBDDs have the same atomic propositions, then we do as follows.

$$\text{apply}(\circ, \begin{array}{c} (x) \\ \swarrow \quad \searrow \\ B \quad B' \end{array}, \begin{array}{c} (x) \\ \swarrow \quad \searrow \\ C \quad C' \end{array}) = \begin{array}{c} (x) \\ \swarrow \quad \searrow \\ \text{apply}(\circ, B, C) \quad \text{apply}(\circ, B', C') \end{array}$$

- If the atomic proposition in the root node of the OBDD C comes later in the variable ordering than x , then we do the following.

$$\text{apply}(\circ, \begin{array}{c} (x) \\ \swarrow \quad \searrow \\ B \quad B' \end{array}, C) = \begin{array}{c} (x) \\ \swarrow \quad \searrow \\ \text{apply}(\circ, B, C) \quad \text{apply}(\circ, B', C) \end{array}$$

- Analogously to the previous case, if the atomic proposition in the root node of the OBDD B comes later in the variable ordering than x , then we do the following.

$$\text{apply}(\circ, B, \begin{array}{c} (x) \\ \swarrow \quad \searrow \\ C \quad C' \end{array}) = \begin{array}{c} (x) \\ \swarrow \quad \searrow \\ \text{apply}(\circ, B, C) \quad \text{apply}(\circ, B, C') \end{array}$$

- The remaining case is when both OBDDs are leaf nodes \top or \perp . Then we apply the truth table for the connective \circ to the truth values.

$$\text{apply}(\circ, b, b') = b \circ b'$$

By using *apply* and the negation operation, an OBDD can be constructed from any propositional formula with negation and binary Boolean connectives $\wedge, \vee, \rightarrow, \leftrightarrow$, and so on.

The above description of *apply* does not include the reduction part of OBDD construction. Including this is critical for efficiency. On the first three cases, the two recursive calls are first made, and if the result from both calls is the same OBDD, the node for x is not constructed, and the OBDD from the two recursive calls is returned instead. Further, memoization is used: before constructing an OBDD node for x and two constituent OBDDs B_1 and B_2 , it is checked whether a node (x, B_1, B_2) has already been constructed. If so, that already-existing OBDD node is returned instead.

Although the runtime of one call to *apply* is polynomial in the size of the two OBDDs, the construction of an OBDD can be exponential time in the size of the original formula, as we call *apply* repeatedly, and OBDDs produced by *apply* can be bigger than the inputs to *apply*. This is as it should be, since constructing an OBDD solves the NP-complete satisfiability problem: a formula ϕ is satisfiable if and only if its OBDD is not \perp .

Example 4.16 We construct the OBDD for $a \vee (b \wedge \neg c)$ with variable ordering a, b, c .

This is by $\text{apply}(\vee, A, \text{apply}(\wedge, B, \text{negate}(C)))$, where A, B and C are the OBDDs for the atomic propositions a, b and c . We do this computation bottom up starting from $\text{negate}(C)$ and $\text{apply}(\wedge, B, \text{negate}(C))$.

TODO ■

OBDD for ϕ trivially answers the validity and satisfiability questions by equality with \top and inequality with \perp , respectively. Logical consequence of $\Phi \models \phi$ is by the validity of $\Phi \rightarrow \phi$ by satisfiability of $\Phi \wedge \neg\phi$. The OBDD for one of these two formulas (which are negations of each other) needs to be constructed. For logical consequence tests $\Phi \models c$ where c is a single literal or a disjunction of literals, simpler graph-theoretic tests, which do not require constructing a new OBDD, exist. See Section 4.3.4.

4.3.1 Properties of OBDDs

Lemma 4.17 *The only valid OBDD is the one consisting of the single node \top .*

The only unsatisfiable OBDD is the one consisting of the single node \perp .

An OBDD can be exponentially larger than an equivalent propositional formula.

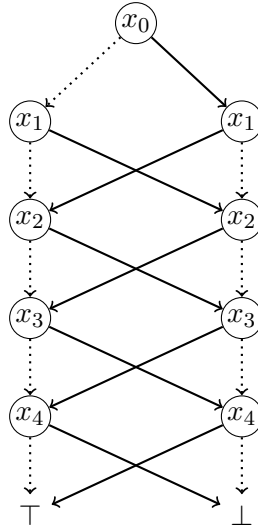


Figure 4.1: OBDD for determining the parity of an assignment

Example 4.18 Consider the formula $(x_1 \leftrightarrow x'_1) \wedge \dots \wedge (x_n \leftrightarrow x'_n)$. We build an OBDD with the variable ordering $x_1, \dots, x_n, x'_1, \dots, x'_n$.

TODO

The number of nodes in the middle is 2^n . Hence the OBDD size is exponential in n and the size of the formula we started is linear in n . ■

An OBDD can be exponentially smaller than an equivalent propositional formula.

Example 4.19 The OBDD in Figure 4.1 is true if and only if the number of true atomic propositions is even.

This OBDD is true iff the number of true atomic propositions is even. An equivalent propositional formula *over the atomic propositions* x_0, \dots, x_n has an exponential size. Representing the OBDD compactly is possible, but requires auxiliary variables as in the Tseitin transformation.

Any path after an even number of true atomic propositions ends up on the right, and after an odd number it ends up on the left.

An equivalent propositional formula *over the atomic propositions* x_0, \dots, x_n has an exponential size. The OBDD can, of course, be represented compactly as a propositional formula by using auxiliary variables as in the Tseitin transformation [Tse68]. ■

4.3.2 Model Counting

For any node in the OBDD (representing some formula Φ), the two subtrees rooted at that node represent formulas that contradict each other: $(x \wedge \Phi[\top/x])$ and $(\neg x \wedge \Phi[\perp/x])$, because of x and $\neg x$. Hence the number of models of Φ is the sum of the numbers of models of the formulas for the child nodes.

This property makes it possible to count the number of satisfying assignments of an OBDD in polynomial time. This is by traversing the OBDD once. Since the number of paths to a given node can be exponential (see Example 4.19), it is critical that the traversal counts the number of partial assignments for an internal node only once. For this it is enough that the model count for a node is recorded the first time it is needed, and subsequent uses of the count use the recorded number rather than does the same computation once more.

- 1: **procedure** MC($B : \text{OBDD}, i : \text{int}$)
- 2: **begin**
- 3: **if** $B = \top$ **then return** 2^{n+1-i} ;
- 4: **if** $B = \perp$ **then return** 0;
- 5: **if** $\text{memo}[B] \geq 0$ **then return** $\text{memo}[B]$;
- 6: $i_B := B.\text{nodeVar}$;

```

7:    $M := 2^{i_B - i}$ ;
8:   memo[B] :=  $M \times (\text{MC}(\text{B.posChild}, i_B + 1) + \text{MC}(\text{B.negChild}, i_B + 1))$ ;
9:   return memo[B];
10: end

```

Let the variable ordering in the OBDD b be x_1, \dots, x_n . Then the call $\text{MC}(B, 1)$ will return the number of satisfying assignments for the OBDD, assuming that $\text{memo}[b]$ has been initialized to -1 for all nodes b of the OBDD.

The second parameter i to MC represents the *expected* next variable in the variable ordering. It is needed for calculating the correction term M on line 7: if in a given path the variable x_i is not followed by x_{i+1} but some later variable x_k with $k > i + 1$, then the variables $x_{i+1}, x_{i+2}, \dots, x_{k-1}$ are “don’t cares”, and all assignments to these variables are possible when looking at the assignments for x_{i+1}, \dots, x_n . Since there are M of such “don’t care” variables, we get the model count for $x_{i+1}, x_{i+2}, \dots, x_{k-1}$ by multiplying the number of assignments for x_b, \dots, x_n by M .

The 2^{n+1-i} on line 3 is similarly due to the missing atomic propositions x_i, \dots, x_n on a path with last two nodes x_{i-1} and \top .

4.3.3 Restriction

The restriction operation with x and an OBDD for ϕ constructs an OBDD for $\phi[\top/x]$. This OBDD is obtained by eliminating all x -nodes, and directing the parents of x -nodes to the positive child of the x -node in question. Similarly restriction with $\neg x$ constructs OBDD for $\phi[\perp/x]$.

Restriction is used in implementing the abstraction operations.

```

1: procedure restrict(B : OBDD, x : variable, positive : bool)
2: begin
3:   if positive
4:   then
5:     for each node  $n$  in the OBDD  $B$  do
6:       if  $n.\text{posChild}.\text{nodeVar} = x$  then  $n.\text{posChild} := n.\text{posChild}.\text{posChild}$ ;
7:       if  $n.\text{negChild}.\text{nodeVar} = x$  then  $n.\text{negChild} := n.\text{negChild}.\text{posChild}$ ;
8:     end do
9:   else
10:    for each node  $n$  in the OBDD  $B$  do
11:      if  $n.\text{posChild}.\text{nodeVar} = x$  then  $n.\text{posChild} := n.\text{posChild}.\text{negChild}$ ;
12:      if  $n.\text{negChild}.\text{nodeVar} = x$  then  $n.\text{negChild} := n.\text{negChild}.\text{negChild}$ ;
13:    end do
14: end

```

4.3.4 Clausal consequences

Testing whether a clause $l_1 \vee \dots \vee l_n$ is a logical consequence of an OBDD can be implemented as a graph-theoretic test: Do all paths from the root node of the OBDD to the leaf node \top satisfy at least one of the literals in the clause, that is, for every path, is there at least literal l_i such that the atomic proposition x_i in l_i does not appear on the path, or the arc on the path from x_i matches the polarity of x in l_i (the outgoing arc from the node is dashed if $l_i = x_i$ and solid if $l_i = \neg x_i$.)

The test is simplest done as follows: remove all such outgoing arcs for nodes with x_1, \dots, x_n , as well as all arcs between nodes y and y' such that one of the x_i is between y and y' in the variable ordering. Then the clause is logical consequence of the OBDD if there is still at least one path from the root node to the leaf node \top : this means that the OBDD is true in at least one valuation in which all of l_1, \dots, l_n are false.

4.3.5 Abstraction

A key operation, needed in relational operations such as the join, is existential abstraction. This is also the operation that seems to be the reason why OBDDs are in many applications preferable to unrestricted propositional

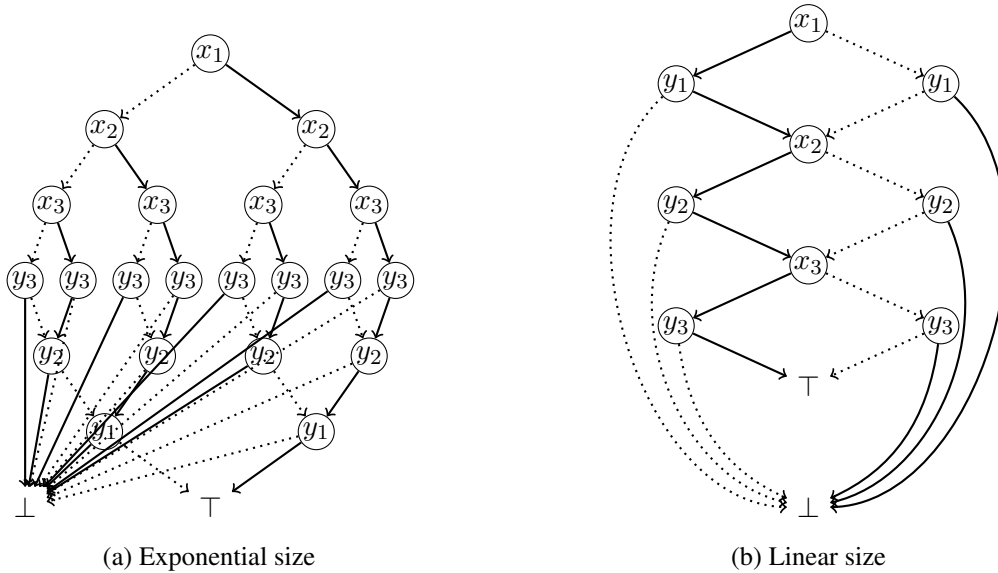


Figure 4.2: OBDDs for $(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2) \wedge (x_3 \leftrightarrow y_3)$

formulas: repeated existential abstraction unavoidably explodes the size of propositional formulas, whereas the size of OBDDs can remain in reasonable bounds.

For unrestricted propositional formulas there are no very effective methods for simplifying $\phi[\top/x] \vee \phi[\perp/x]$ as resulting from abstraction $\exists x.\phi$. Other operations, such as conjunction, renaming, and emptiness/satisfiability testing, would often not be an obstacle for using unrestricted propositional formulas in the applications where OBDDs have been very successful. NP-completeness of SAT is often mentioned as a reason why OBDDs are preferable, but the numbers of variables in many applications are relatively low (hundreds or at most thousands), and modern SAT solvers perform these tests very efficiently for smallish formulas.

Abstraction operations for an OBDD can be computed with the help of the restriction operation: $\text{restrict}(B, x, \text{true}) \vee \text{restrict}(B, x, \text{false})$.

4.3.6 Variable Ordering

For a given Boolean function, OBDDs with different variable ordering may look completely different, and the differences in their sizes may be dramatic. Finding a good variable ordering is one of the critical issues in the efficient use of OBDDs. A bad variable ordering may lead to an astronomically large OBDD, and a good variable ordering may lead to a small one.

Example 4.20 Figures 4.2a and 4.2b contain two OBDDs for the formula $(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2) \wedge (x_3 \leftrightarrow y_3)$.

The first one has an exponential size, and it has the variable ordering $x_1, y_1, x_2, y_2, \dots, x_n, y_n$. Intuitively, each of the y_3 nodes in the middle of the OBDD encodes one of the valuations of x_1, \dots, x_n , corresponding to the path from the root node. Since there are 2^n valuations, there are 2^n nodes for y_3 . The lower part of the OBDD reads the valuation of y_1, y_2, y_3 , and it does not match the valuation of x_1, x_2, x_3 , an arc to the \perp -node is followed.

The second OBDD for the same formula has only $3n$ non-leaf nodes, and hence it is linear in n . The OBDD reads the x_i variables one by one, and immediately makes a comparison to the corresponding y_i variable. Hence the OBDD does not need to “remember” previous values, only the last x_i value.

Clearly the ordering $x_1, y_1, x_2, y_2, \dots, x_n, y_n$ (or any ordering in which the x_i and y_i for the same i are next to each other) is better than $x_1, \dots, x_n, y_1, \dots, y_n$ (and other similar ones.) ■

Variable ordering can be designed manually, or they can be found by the automated variable ordering methods provided by OBDD packages. The automated methods perform a form of local search, trying out small modifications in the ordering, and keeping a change if the size of the OBDD got smaller. These methods have not guarantee of finding a good ordering even if an obvious one existed.

4.4 Normal Forms DNNF and d-DNNF

Darwiche et al. [Dar01, DM02] have carried out a thorough study on features of propositional formulas, to identify normal forms of practical importance.

The first new normal form identified by Darwiche was Decomposable NNF [Dar01]. This is NNF with the additional property of *decomposability*.

Definition 4.21 (Decomposability) *A formula ϕ in NNF is decomposable if for every subformula $\psi_1 \wedge \psi_2$ of ϕ , the subformulas ψ_1 and ψ_2 don't share atomic propositions.*

The normal form DNNF (Decomposable Negation Normal Form) is defined as formulas in NNF that additionally satisfy the decomposability condition.

Notice that OBDDs are subclass of DNNF: In OBDDs conjunctions are of the form $x \wedge \phi[\top/x]$ and $\neg x \wedge \phi[\perp/x]$, where x is an atomic proposition and $\phi[\top/x]$ and $\phi[\perp/x]$ trivially have no occurrences of x .

Unlike for general NNF formulas, formulas in DNNF have the interesting property that satisfiability testing can be done in polynomial time. The algorithm for doing this is as follows.

$$\begin{aligned} SAT(x) &= \text{true} \\ SAT(\neg x) &= \text{true} \\ SAT(\top) &= \text{true} \\ SAT(\perp) &= \text{false} \\ SAT(\phi_1 \wedge \phi_2) &= SAT(\phi_1) \text{ and } SAT(\phi_2) \\ SAT(\phi_1 \vee \phi_2) &= SAT(\phi_1) \text{ or } SAT(\phi_2) \end{aligned}$$

This algorithm does not work correctly for general NNF. For example, $SAT(x \wedge \neg x)$ will incorrectly return *true*. But the decomposability property guarantees that the satisfiability of ϕ_1 and the satisfiability of ϕ_2 entail the satisfiability of $\phi_1 \wedge \phi_2$. Since the two formulas share no atomic propositions, if they are both satisfiable, we can always easily construct a satisfying valuation from the satisfying valuations of the two formulas.

Translation from general propositional formulas into NNF is polynomial time, but translation from NNF into DNNF takes exponential time in the worst case. This is to be expected, as SAT is NP-hard but for DNNF satisfiability testing is polynomial time. The most effective known methods for translating formulas into DNNF always translate the formulas into the normal form d-DNNF too [Dar02, HD05]. We define d-DNNF next.

A further condition on NNF is *determinism*.

Definition 4.22 (Determinism) *A formula ϕ in NNF is deterministic if for every subformula $\psi_1 \vee \psi_2$ of ϕ , the subformulas ψ_1 and ψ_2 are mutually contradictory.*

The normal form d-DNNF (Deterministic DNNF) is defined as formulas in DNNF that additionally satisfy the determinism condition.

Notice that OBDDs are subclass of d-DNNF: In OBDDs disjunctions are of the form $(x \wedge \phi[\top/x]) \vee (\neg x \wedge \phi[\perp/x])$, and the two disjuncts contradict because of x and $\neg x$.

In addition to polynomial-time satisfiability testing, d-DNNF additionally have polynomial-time model-counting. This is because the model counts, that is the number of satisfying assignments, for a subformula $\psi_1 \vee \psi_2$ can be obtained as the sum of the model-counts of ψ_1 and ψ_2 .

There are examples of Boolean functions that can be expressed more compactly in DNNF than in d-DNNF, and more compactly in d-DNNF than in OBDD. A property separating OBDD from d-DNNF and DNNF is *canonicity*: the DNNF and d-DNNF for a given Boolean function are not unique. This seems to be the main reason why OBDDs, and not some other normal form, has been extensively used in practice. Canonicity of OBDDs relies on the ordering condition, whereas for DNNF and d-DNNF no obvious ordering condition exists. A different type of normal form, called Sentential Decision Diagram (SDD) [Dar11], located between OBDD and DNNF, has an ordering condition and canonicity.

For n atomic propositions, the truth-table has 2^n rows, and the last row (indicating the value of a Boolean function for one valuation of the atomic propositions) can be assigned a value in 2^{2^n} different ways. Hence there are 2^{2^n}

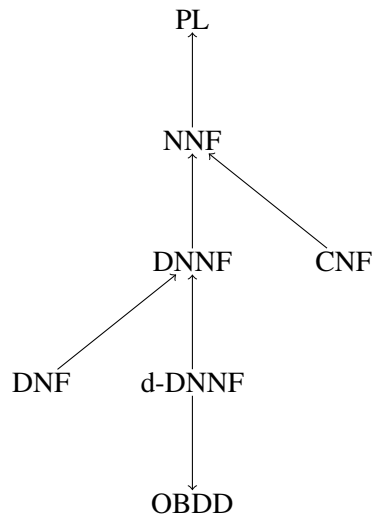


Figure 4.3: Inclusion Relations of Normal Forms

operation	PL	NNF	CNF	DNF	DNNF	d-DNNF	OBDD
$\phi \in \text{SAT}$	NP	NP	NP	P	P	P	P
$\phi \in \text{TAUT}$	co-NP	co-NP	P	co-NP	co-NP	P	P
$\phi \models c$	co-NP	co-NP	co-NP	P	P	P	P
model counting	NP	NP	NP	NP	NP	P	P
$n \times \wedge$	P	P	P	exp	exp	exp	exp
$n \times \vee$	P	P	exp	P	P	exp	exp
\neg	P	P	exp	exp	exp	?	P

Table 4.2: Complexity of Logical Operations for Different Normal Forms

different Boolean functions of n variables (atomic propositions). One can show by a simple cardinality argument that “most” Boolean functions do not have “compact” representation (e.g. size that is polynomial in the number of propositional variables.)

4.5 Normal Forms Summary

Figure 4.3 depicts the inclusion relations between some of the best known normal forms, and Table 4.2 summarizes the complexities of some important operations on logical formulas for these normal forms. In the table, NP denotes NP-hard, co-NP denotes co-NP-hard, and P denotes “polynomial time”. The complexity of negating a d-DNNF formula is not known.

For the construction operations \vee , \wedge and \neg the result has to be in the same normal form. For one \wedge or \vee , construction is polynomial size and time (at most $n_1 \times n_2$ when the constituent formulas have sizes n_1 and n_2 , respectively), but repeating the operation n times is exponential in n , as the size can increase exponentially $((n_1 \times n_2) \times n_3) \times n_4 \times \dots \times n_k$, as the product $n_1 n_2 \dots n_k$ is exponential in k .

The *clausal logical consequence* $\phi \models c$ tests whether a clause c is a logical consequence of the formula ϕ . Only the formula ϕ here is expected to be in the normal form in question. (A clause, a disjunction of literals, is not a DNNF, d-DNNF, or OBDD.)

Chapter 5

Sets and Relations in the Propositional Logic

Formulas can be viewed as a data structure for representing sets and relations. The advantage of logic in this application is *succinctness*: a formula can represent a set that has a size that is exponential in the size of the formula. Logic representation is therefore succinct. This is in strong contrast to conventional data structures which would explicitly enumerate all elements of a set, and therefore have a size that is linear in the size of the set.

5.1 Representation of Sets

Formulas can be considered as a representation of those sets of valuations that make the formula true. We can identify a valuation $v : X \rightarrow \{0, 1\}$ with a vector of length $|X|$, with each element of the vector corresponding to one of the atomic propositions in X .

Example 5.1 Let $X = \{A, B, C, D\}$. Now a valuation that assigns 1 to A and C and 0 to B and D corresponds to the bit-vector $\overset{ABCD}{1010}$, and the valuation assigning 1 only to B corresponds to the bit-vector $\overset{ABCD}{0100}$. ■

Any propositional formula ϕ can be understood as a representation of those valuations v such that $v(\phi) = 1$. Since we have identified valuations and bit-vectors, a formula naturally represents a set of bit-vectors.

Example 5.2 Formula B represents all bit-vectors of the form $?1??$, and the formula A represents all bit-vectors $1???$. Formula B therefore represents the *set*

$$\{0100, 0101, 0110, 0111, 1100, 1101, 1110, 1111\}$$

and formula A represents the set

$$\{1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111\}.$$

Similarly, $\neg B$ represents all bit-vectors of the form $?0??$, which is the set

$$\{0000, 0001, 0010, 0011, 1000, 1001, 1010, 1011\}.$$

This is the *complement* of the set represented by B .

5.1.1 Set Operations as Logical Operations

There is a close connection between the Boolean connectives \vee , \wedge , \neg and the set-theoretical operations of *union*, *intersection* and *complementation*, which is also historically the origin of Boole's work on Boolean functions.

If ϕ_1 and ϕ_2 represent bit-vector sets S_1 and S_2 , then

1. $\phi_1 \wedge \phi_2$ represents set $S_1 \cap S_2$,

sets	formulas over X
those $\frac{2^{ X }}{2}$ bit-vectors where x is true	$x \in X$
\overline{E} (complement)	$\neg E$
$E \cup F$	$E \vee F$
$E \cap F$	$E \wedge F$
$E \setminus F$ (set difference)	$E \wedge \neg F$
the empty set \emptyset	\perp (constant <i>false</i>)
the universal set	\top (constant <i>true</i>)
question about sets	question about formulas
$E \subseteq F?$	$E \models F?$
$E \subset F?$	$E \models F$ and $F \not\models E?$
$E = F?$	$E \models F$ and $F \models E?$

Table 5.1: Connections between Set-Theory and Propositional Logic

- $\phi_1 \vee \phi_2$ represents set $S_1 \cup S_2$, and
- $\neg\phi_1$ represents set $\overline{S_1}$.

Example 5.3 $A \wedge B$ represents the set $\{1100, 1101, 1110, 1111\}$ and $A \vee B$ represents the set $\{0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111\}$. ■

Questions about the relations between sets represented as formulas can be reduced to the basic logical concepts we already know, namely logical consequence, satisfiability, and validity.

- “Is ϕ satisfiable?” corresponds to “Is the set represented by ϕ *non-empty*?”
- $\phi \models \alpha$ corresponds to “Is the set represented by ϕ a *subset* of the set represented by α ?”
- “Is ϕ valid?” corresponds to “Is the set represented by ϕ *the universal set*?”

These connections allow using propositional formulas as a *data structure* in some applications in which conventional enumerative data structures for sets are not suitable because of the astronomic number of elements in the sets. For example, if there are 100 atomic propositions, then any formula consisting of just one atomic proposition represents a set of $2^{99} = 633825300114114700748351602688$ bit-vectors, which would require 7493989779944505344 TB in an explicit enumerative representation if each of the 100-bit vectors was represented with 13 bytes (wasting only 4 bits in the 13th byte.)

5.2 Relations as Formulas

Similarly to finite sets of atomic objects, *relations* on any finite set of objects can be represented as propositional formulas.

A binary relation $R \subseteq X \times X$ is a set of pairs $(a, b) \in X \times X$, and the representation of this relation is similar to representing sets before, except that the elements are pairs.

As before, we assume that the atomic objects are bit-vectors. A pair of bit-vectors of lengths n and m can of course be represented as a bit-vector of length $n + m$, simply by attaching the two bit-vectors together.

Example 5.4 To represent the *pair* (0001, 1100) of bit-vectors, both expressed as valuations of atomic propositions $X = \{A, B, C, D\}$, instead use the atomic propositions $X_{01} = \{A_0, B_0, C_0, D_0, A_1, B_1, C_1, D_1\}$ as the index variables.

The pair (0001, 1100) is hence represented as 00011100, a valuation of X_{01} . ■

Pair $\binom{A_0 B_0 C_0 D_0}{0 0 0 1}, \binom{A_1 B_1 C_1 D_1}{1 1 0 0}$ therefore corresponds to the valuation that assigns 1 to D_0, A_1 and B_1 and 0 to all other variables.

Example 5.5 $(A_0 \leftrightarrow A_1) \wedge (B_0 \leftrightarrow B_1) \wedge (C_0 \leftrightarrow C_1) \wedge (D_0 \leftrightarrow D_1)$ represents the identity relation of 4-bit bit-vectors. ■

Example 5.6 The formula

$$\begin{aligned} inc_{01} = & (\neg c_0 \wedge c_1 \wedge (b_0 \leftrightarrow b_1) \wedge (a_0 \leftrightarrow a_1)) \\ & \vee (\neg b_0 \wedge c_0 \wedge b_1 \wedge \neg c_1 \wedge (a_0 \leftrightarrow a_1)) \\ & \vee (\neg a_0 \wedge b_0 \wedge c_0 \wedge a_1 \wedge \neg b_1 \wedge \neg c_1) \\ & \vee (a_0 \wedge b_0 \wedge c_0 \wedge \neg a_1 \wedge \neg b_1 \wedge \neg c_1) \end{aligned}$$

represents the successor relation of 3-bit integers

$$\{(000, 001), (001, 010), (010, 011), (011, 100), (100, 101), (101, 110), (110, 111), (111, 000)\},$$

which can also be depicted in a tabular form as follows, in order to make the variables for the columns explicit.

$a_0 b_0 c_0$	$a_1 b_1 c_1$
000	001
001	010
010	011
011	100
100	101
101	110
110	111
111	000

Notice in this example that the tabular representation only has 8 rows, whereas the corresponding truth-table for the formula inc_{01} has $2^6 = 64$ rows. The tabular representation of the relation only enumerates those rows for which the formula evaluates to *true*.

Example 5.7 Consider the relation $\{(001, 111), (010, 110), (011, 010), (111, 110)\}$. The relation can be represented as the following truth-table (listing only those of the $2^6 = 64$ lines that have 1 in the column for ϕ),

$a_0 b_0 c_0 a_1 b_1 c_1$	ϕ
⋮	
0 0 1 1 1 1	1
⋮	
0 1 0 1 1 0	1
⋮	
0 1 1 0 1 0	1
⋮	
1 1 1 1 1 0	1
⋮	

which is equivalent to the following formula.

$$\begin{aligned} & (\neg a_0 \wedge \neg b_0 \wedge c_0 \wedge a_1 \wedge b_1 \wedge c_1) \vee \\ & (\neg a_0 \wedge b_0 \wedge \neg c_0 \wedge a_1 \wedge b_1 \wedge \neg c_1) \vee \\ & (\neg a_0 \wedge b_0 \wedge c_0 \wedge \neg a_1 \wedge b_1 \wedge \neg c_1) \vee \\ & (a_0 \wedge b_0 \wedge c_0 \wedge a_1 \wedge b_1 \wedge \neg c_1) \end{aligned}$$

Any binary relation over a finite set can be represented as a propositional formula in this way. These formulas can be used as components of formulas that represent *paths* in the graphs corresponding to the binary relation. ■

5.2.1 Relational Operations in Logic

As relations as sets of pairs (or, for n -ary relations, sets of n -tuples), the set operations that we already defined (\cup , \cap , and so on) immediately apply to relations as well. Of interest in many of our applications are also well-known relational algebra operations such as join operations (natural join \bowtie), projections, and so on. In this section, we will show how the most important relational operations can be implemented by formula manipulation when the relations in question are represented as formulas. We focus on the core operations of natural join, selection, and projection, as most of the other operations, for example many types of join operations, can be reduced to these basic operations.

The join and projection operations are the basis operations when computing the successors of a set of states with respect to a binary transition relation, when both the set and the relation are represented as formulas.

We first consider the natural join operation. The natural join operation takes two relations, and constructs a new table with columns from the two constituent tables, and each row in the new table consists of rows from the constituent tables that match on the shared columns. There may be zero, one, or more shared columns.

Example 5.8 We form the natural join $R_1 \bowtie R_2$ of two relations R_1 and R_2 .

$$\begin{array}{c|c} 0 & 1 \\ \hline 01 & 10 \\ 10 & 01 \\ 11 & 11 \end{array} \bowtie \begin{array}{c|c} 1 & 2 \\ \hline 00 & 01 \\ 01 & 10 \\ 10 & 11 \end{array} = \begin{array}{c|c|c} 0 & 1 & 2 \\ \hline 01 & 10 & 11 \\ 10 & 01 & 10 \end{array}$$

The relation R_1 could be called *non-zero swap*, and R_2 could be called *increment*. Formulas to represent R_1 and R_2 are respectively

$$\phi_1 = (a_0 \vee b_0) \wedge (a_1 \leftrightarrow b_0) \wedge (b_1 \leftrightarrow a_0)$$

and

$$\phi_2 = (b_2 \leftrightarrow \neg a_1) \wedge (a_2 \leftrightarrow (a_1 \wedge \neg b_1)) \wedge \neg(a_1 \wedge b_1).$$

(The same Boolean functions can of course be represented by many other logically equivalent formulas.)

Although ϕ_1 only contains occurrences of a_0, b_0, a_1, b_1 , one should think about its truth-table that also includes columns for a_2 and b_2 . Essentially, a_2 and b_2 for ϕ_1 are *don't cares*: they are not limited by ϕ_1 , and they can obtain any truth-values.

Similarly, ϕ_2 does not explicitly refer to a_0 and b_0 .

Now, the join $R_1 \bowtie R_2$ represents all those valuations of $a_0, b_0, a_1, b_1, a_2, b_2$ that satisfy both ϕ_1 and ϕ_2 .

The process of matching column 1 values, that is central in computing the join of relations represented as tables, corresponds to the requirement that valuations that satisfy the formula for $R_1 \bowtie R_2$ have to satisfy both ϕ_1 and ϕ_2 .

Hence the natural join operation simply corresponds to the logical conjunction \wedge , and $R_1 \bowtie R_2$ is represented by $\phi_1 \wedge \phi_2$, as can be easily verified. ■

The *selection* operation $\sigma_\beta(R)$ has a straightforward representation when the relation R is represented as a formula ϕ . This is again simply a conjunction $\phi \wedge \beta$, when β has been expressed in terms of propositional variables occurring in ϕ .

Finally, we consider the important *projection* operation $\pi_c(R)$. This operation select some subset c of the columns of a relation.

Example 5.9 Consider the relation R

$$\begin{array}{c|c} 0 & 1 \\ \hline 00 & 00 \\ 01 & 00 \\ 10 & 11 \\ 11 & 11 \end{array}$$

represented by the formula $\phi_1 = (a_1 \leftrightarrow a_0) \wedge (b_1 \leftrightarrow a_0)$ (over variables a_0, b_0, a_1, b_1), which we would like to project to the column 1 by $\pi_1(R)$ to obtain

$$\pi_1 \left(\begin{array}{c|c} 0 & 1 \\ \hline 00 & 00 \\ 01 & 00 \\ 10 & 11 \\ 11 & 11 \end{array} \right) = \frac{1}{\begin{array}{c} 00 \\ 11 \end{array}}$$

The result of the projection is the atomic formula $a_1 \leftrightarrow b_1$.

Consider the truth-tables of the original relation and its projection to 1:

a_0	b_0	a_1	b_1	$(a_1 \leftrightarrow a_0) \wedge (b_1 \leftrightarrow a_0)$	
0	0	0	0	1	
0	0	0	1	0	
0	0	1	0	0	
0	0	1	1	0	
0	1	0	0	1	
0	1	0	1	0	
0	1	1	0	0	a_1
0	1	1	1	0	b_1
1	0	0	0	0	$a_1 \leftrightarrow b_1$
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	1	1
1	1	0	0	0	
1	1	0	1	0	
1	1	1	0	0	
1	1	1	1	1	

In the second table exactly those rows (valuations of a_1, b_1) are mapped to 1, for which there is at least one row in the first table with matching values for a_1, b_1 and 1 in the last column. We have highlighted the rows in the first table that correspond to $a_1 = 0, b_1 = 0$. This valuation is mapped to 1 in the second table. Similarly, for valuation $a_1 = 0, b_1 = 1$ we get 0 in the second table, because the first table always maps this valuation to 0. ■

What the above example illustrates turns out to exactly match what Existential Abstraction does: eliminating variables a_0 and b_0 from the truth table for ϕ corresponds to existentially abstracting a_0 and b_0 in ϕ , that is, $\exists a_0 \exists b_0. \phi$.

Definition 5.10 (Existential abstraction) *Existential abstraction of ϕ with respect to x is defined by*

$$\exists x. \phi = \phi[\top/x] \vee \phi[\perp/x].$$

This operation allows eliminating atomic proposition x from any formula. Make two copies of the formula, with x replaced by the constant true \top in one copy and with constant false \perp in the other, and form their disjunction.

Analogously we can define *universal abstraction*, with conjunction instead of disjunction.

Definition 5.11 (Universal abstraction) *Universal abstraction of ϕ with respect to x is defined by*

$$\forall x. \phi = \phi[\top/x] \wedge \phi[\perp/x].$$

Example 5.12

$$\begin{aligned}
& \exists B.((A \rightarrow B) \wedge (B \rightarrow C)) \\
& = ((A \rightarrow \top) \wedge (\top \rightarrow C)) \vee ((A \rightarrow \perp) \wedge (\perp \rightarrow C)) \\
& \equiv C \vee \neg A \\
& \equiv A \rightarrow C
\end{aligned}$$

$$\begin{aligned}
& \exists AB.(A \vee B) = \exists B.(\top \vee B) \vee (\perp \vee B) \\
& = ((\top \vee \top) \vee (\perp \vee \top)) \vee ((\top \vee \perp) \vee (\perp \vee \perp)) \\
& \equiv (\top \vee \top) \vee (\top \vee \perp) \equiv \top
\end{aligned}$$

■

Both universal $\forall c$ and existential $\exists c$ abstraction can be viewed as eliminating a column for c in a truth-table of a formula ϕ by combining lines with the same valuation for variables other than c . There are always two lines that agree on the valuation of variables other than c , one in which $c = 0$ and the other with $c = 1$.

For universal abstraction these lines are combined by the Boolean function *and*, and for existential abstraction these lines are combined by the Boolean function *or*. We will illustrate this in the next example.

Example 5.13 We will abstract $a \vee (b \wedge c)$ both existentially and universally, obtaining $\exists c.(a \vee (b \wedge c)) \equiv a \vee b$ and $\forall c.(a \vee (b \wedge c)) \equiv a$. The corresponding truth tables for the original formula and the results of the two abstractions are as follows.

a	b	c	$a \vee (b \wedge c)$	a	b	$\exists c.(a \vee (b \wedge c))$	a	b	$\forall c.(a \vee (b \wedge c))$
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	1	0
0	1	0	0	0	1	1	0	1	0
0	1	1	1	0	1	1	0	1	0
1	0	0	1	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0	1
1	1	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Here we have colored the pairs of rows in the first truth-table which will be merged to the corresponding rows in the truth-tables for the abstracted formulas. The difference between the abstractions is that the existential abstraction yields 0 if and only if both of the two rows in the original table are 0, and for universal abstraction we get 0 if and only if at least one of the rows is 0. ■

Example

From $\neg A_0 \wedge \neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1))$ produce $(\neg A_1 \wedge B_1 \wedge C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge C_1)$.

$$\Phi = \neg A_0 \wedge \neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1))$$

$$\exists A_0 B_0 C_0. \Phi$$

$$= \exists B_0 C_0. (\Phi[0/A_0] \vee \Phi[1/A_0])$$

$$= \exists B_0 C_0. (\neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1)))$$

$$= \exists C_0. ((\neg A_1 \wedge (\neg C_0 \wedge B_1 \wedge C_1)) \vee (\neg A_1 \wedge (\neg C_0 \wedge \neg B_1 \wedge C_1)))$$

$$= (\neg A_1 \wedge B_1 \wedge C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge C_1)$$

Chapter 6

Transition Systems

6.1 Basic Models

Definition 6.1 (Transition System) A transition system $\langle S, E, R \rangle$ consists of

- a set S of states,
- a set E of events, and
- an assignment $R : E \rightarrow 2^{S \times S}$ of transition relations $R(e) \subseteq S \times S$ to all events $e \in E$.

Transition systems model the dynamics of a system: the state of the system is changed by events. Zero or more events may be possible in a state, as indicated by the transition relation associated with the event: if s is related by $R(e)$ to some state, then the event e is possible in s . If s is related to s' by $R(e)$ then it is possible that the event e changes the state from s to s' . The transition system does not determine which events take place: if there are more than one event possible in a state, there may be one or more possible next states for a state.

Definition 6.2 (Execution) In a transition system $\langle S, E, R \rangle$, an execution $s_0, e_0, s_1, e_1, s_2, e_2, \dots, e_{n-1}, s_n$ with $\{s_0, \dots, s_n\} \subseteq S$ and $e_0, \dots, e_{n-1} \subseteq E$ is possible if $(s_i, s_{i+1} \in R(e_i))$ for all i such that $0 \leq i < n$.

A sequence s consisting of a single state $s \in S$ is a trivial execution.

Definition 6.3 (Transition System with State Variables) A transition system with state variables $\langle S, E, R, X, v \rangle$ consists of a transition system $\langle S, E, R \rangle$ and

- a set X of state variables,
- a valuation $v : S \times X \rightarrow \{0, 1\}$ of state variables in states.

For convenience, we will write $v_s(x)$ for $v(s, x)$.

In a transition system with state variables we can talk about the truth of a propositional formula in a state of the transition system. It is trivial to adapt Definition 2.3 to transition systems with state variables to define $v_s(\phi)$ for any state $s \in S$ and any propositional formula ϕ over X .

Definition 6.4 (Succinct Transition System) A succinct transition system variables $\langle X, E, R, I \rangle$ consists of

- a set X of state variables,
- a set E of events
- a succinctly represented transition relation $R(e) \in \mathcal{L}(X \cup X')$ for every $e \in E$, and
- a succinctly represented set of initial states $I \in \mathcal{L}(X)$.

If the formula I is the constant \top , then any valuation of X is a possible initial state of the transition system. Typically I expresses at least dependencies between state variables, if not specifying a unique initial state (by a conjunction of literals for every $x \in X$.)

The relations are represented as formulas $R(e)$ with occurrences of atomic propositions for values of state variables $X = \{a, b, c, \dots\}$ and values of state variables in the next state $X' = \{a', b', c', \dots\}$.

Representation of transition relations in Section 5.2.

6.2 Procedural Representations

While we have defined succinct transition systems in terms of transitions relations represented as logical formulas, this is not how large transition systems are modelled in practise.

Practical modeling languages represent states as assignments of values to state variables, exactly as with the logic representation, but transitions are typically defined *procedurally*, in terms of *changes* they induce on the state variables. This, in turn, is similar to programming languages: changes are assignments of values to state variables, unconditionally or conditionally on some facts holding.

In this section we describe how such procedural representations can be mapped to the logic representation. The logic representation is important for many of the state space search methods that are scalable to very large state spaces. If it was not for the logic based representations, we could have defined succinct transition systems (Definition 6.4) in terms of functions $f : S \rightarrow 2^S$, where S is the set of all valuations of the state variables, that map individual states to sets of states (zero or one states for a deterministic transitions, and 2 or more states for nondeterministic transitions.) Clearly, both types of definitions are equivalent, because a function $f : S \rightarrow 2^S$ can be understood as a many-to-many binary relation on S , exactly as we have defined transitions in Definition 6.4.

The plan next is to give a simplest possible procedural transition representation and show how it is mapped to the propositional logic (Section 6.2.1), and then give a more expressive definition that covers a complex modeling language (Section 6.2.2)

6.2.1 Deterministic Unconditional Transitions

We first consider a transition representation in which a transition is characterized by a pair (p, e) where p is an arbitrary propositional formula (the precondition), and e is a set of assignments $x := 0$ or $x := 1$. That is, every change to the state variables is fixed and independent of the state the transition takes place. The set e of effects can be identified with literals x and $\neg x$, corresponding to assignments to 1 and 0.

This language cannot compactly represent many types of changes. For example, it cannot compactly represent a transition that complements the value of n state variables x_1, \dots, x_n . One would need 2^n separate transitions for representing this type of complementation.

Nevertheless, many types of complex systems can still be represented, with conditional behavior split to multiple transitions. Incrementing an n -bit binary number represented by x_{n-1}, \dots, x_0 can be done with n unconditional transitions.

Example 6.5 Incrementing a 5-bit binary number is achieved by the following transitions, exactly one of which can be applicable in any given state.

$$\begin{aligned} &(\neg x_0, \{x_0\}) \\ &(\neg x_1 \wedge x_0, \{x_1, \neg x_0\}) \\ &(\neg x_2 \wedge x_1 \wedge x_0, \{x_2, \neg x_1, \neg x_0\}) \\ &(\neg x_3 \wedge x_2 \wedge x_1 \wedge x_0, \{x_3, \neg x_2, \neg x_1, \neg x_0\}) \\ &(\neg x_4 \wedge x_3 \wedge x_2 \wedge x_1 \wedge x_0, \{x_4, \neg x_3, \neg x_2, \neg x_1, \neg x_0\}) \end{aligned}$$

■

Definition 6.6 (Unconditional deterministic transition) *Given an unconditional transition (p, e) , its translation into the propositional logic is*

$$p \wedge \bigwedge \{x' | x \in e, x \in X\} \wedge \{\neg x' | \neg x \in e, x \in X\} \wedge \{x \leftrightarrow x' | x \in X, x \notin e, \neg x \notin e\}$$

In addition to the precondition, the formula essentially gives, for every state variable $x \in X$, an equivalence $x' \leftrightarrow \phi_x$, where $\phi_x = \top$ if $x \in p$, $\phi_x = \perp$ if $\neg x \in p$, and $\phi_x = x$ if $\{x, \neg x\} \cap p = \emptyset$. These equivalences describe how the new value of every state variable depends on the values of the old values of state variables. In the simple case covered here, the dependencies are simple, as either a state variable retains its old value, or becomes unconditional true or false. In the next section we cover far more complex dependencies.

Exercise 6.1 Given a state in which a, b and c are all true, and the transition $(a \wedge b, \{\neg b, \neg c\})$,

1. express both the state and the transition relation as propositional formulas,
2. form their conjunction,
3. existentially abstract away all atomic propositions $x \in X$, leaving only atomic propositions $x \in X'$ left,
4. simplify the formula by eliminating all occurrences of \top and \perp .

6.2.2 General Definition

The simple procedural representation for unconditional transitions covered in the previous section can be restrictive, as the changes taking place in a state cannot be dependent on the state, which may require the use of a large number of unconditional transitions. For representing more complex systems it is customary to use a transition model with more complex features, which allow representing many problem far more compactly.

Definition 6.7 (Succinct transition procedure) *Given a set X of state variables, a transition is described by a pair (p, e) where p is a propositional formula over X , and e is an effect. The set of all possible effects over X is recursively defined by*

1. ϵ (the NO-OP effect) is an effect,
2. x and $\neg x$ for any $x \in X$ are (atomic) effects,
3. $eITE(\phi, e_1, e_2)$ is an effect if ϕ is a formula over X and e_1 and e_2 are effects over X , and
4. $e_1; e_2$ is an effect if e_1 and e_2 are effects over X .

The empty effect ϵ does not do anything. The conditional effect $eITE\phi e_1 e_2$ is the procedural IF-THEN-ELSE statement: if the condition ϕ holds, then e_1 is executed, and otherwise e_2 is executed. We could define IF-THEN $eIT(\phi, e)$ as a special case $eITE(\phi, e, \epsilon)$. The sequential composition $e_1; e_2$ of two effects means that e_1 is executed first, and then e_2 after that.

Next we present a systematic derivation of a translation from our action definition to the propositional logic. We could give the definition directly (and you can skip to the table given on the next page), but we can alternatively use the weakest precondition predicate familiar from programming logics [Dij75] to do the derivation systematically not only for our limited language for actions, but also for a substantially more general one.

The weakest precondition predicate $wp_\pi(\phi)$ gives for a program π (effect of an action) and a formula ϕ the weakest (most general) formula $\chi = wp_\pi(\phi)$ so that if a state s satisfies χ , that is $s \models \chi$, then the state obtained from s by executing π will satisfy ϕ .

Example 6.8 If a condition is not affected by the program, then the weakest precondition is the condition itself.

$$wp_{x:=1}(y = 1) = y = 1$$

If the program achieves the condition unconditionally, then the weakest precondition is the constant \top .

$$wp_{x:=1}(x = 1) = \top$$

■

Definition 6.9 (Expressions) *Only the constant symbols 0 and 1 are expressions.*

Definition 6.10 (Programs) 1. *The empty program ϵ is a program.*

2. *If x is a state variable and f is an expression, then $x := f$ is a program*
3. *If ϕ is a formula over equalities $x = f$ where x is a state variable and f is an expression, and π_1 and π_2 are programs, then $eITE(\phi, \pi_1, \pi_2)$ is a program.*
4. *If π_1 and π_2 are programs, then their sequential composition $\pi_1; \pi_2$ is a program. This program first executes π_2 , and then π_1 .*
5. *If π_1 and π_2 are programs, then their parallel composition $\pi_1 | \pi_2$ is a program. This program first executes π_1 , and then π_2 .*

Nothing else is a program.

Given a program π and formula ϕ , we can compute the *weakest precondition* $wp_\pi(\phi)$, that has the property, that if it is satisfied, then executing π will end in a state that satisfies ϕ , and for any other formula χ that satisfies the same condition, $\chi \models wp_\pi(\phi)$, that is, it is the *weakest* such formula. It was first investigated by Dijkstra [Dij75].

Definition 6.11 (The Weakest Precondition)

$$wp_\epsilon(\phi) = \phi \tag{6.1}$$

$$wp_{x:=b}(\phi) = \phi \text{ with } x \text{ replaced by } b \tag{6.2}$$

$$wp_{eITE(\theta, \pi_1, \pi_2)}(\phi) = (\theta \wedge wp_{\pi_1}(\phi)) \vee (\neg\theta \wedge wp_{\pi_2}(\phi)) \tag{6.3}$$

$$wp_{\pi_1; \pi_2}(\phi) = wp_{\pi_1}(wp_{\pi_2}(\phi)) \tag{6.4}$$

$$\tag{6.5}$$

6.3 Higher Order Representations

Practical modeling languages have concepts that are at a higher level than the propositional logic. The set of state variables is not a collection x_1, \dots, x_n of unstructured variables. Instead, these atomic (Boolean) state variables are obtained from a higher level specification language.

We give an example of such a language.

There are different types of objects, each type t characterized by its domain D_t , the set of objects belonging to the type. The objects themselves are represented as their names, an alphanumeric string.

The state variables are formed by combining a *predicate* symbol P with some combination $(o_1, \dots, o_k) \in D_{t_1} \times \dots \times D_{t_n}$ of objects associated with the type (t_1, \dots, t_n) of the predicate.

Example 6.12 Let there be only one type *location*, with the domain $D_{location} = \{ \text{Helsinki, Tampere, Jyväskylä, Oulu} \}$.

Let there be only one predicate *arc*, of type (location,location).

All combinations of locations, obtained as the Cartesian product $D_{location} \times D_{location}$, yield the following state variables.

arc(Helsinki,Helsinki)	arc(Helsinki,Tampere)	arc(Helsinki,Jyväskylä)	arc(Helsinki,Oulu)
arc(Tampere,Helsinki)	arc(Tampere,Tampere)	arc(Tampere,Jyväskylä)	arc(Tampere,Oulu)
arc(Jyväskylä,Helsinki)	arc(Jyväskylä,Tampere)	arc(Jyväskylä,Jyväskylä)	arc(Jyväskylä,Oulu)
arc(Oulu,Helsinki)	arc(Oulu,Tampere)	arc(Oulu,Jyväskylä)	arc(Oulu,Oulu)

■

Similarly to creating state variables from a parametric definition (a predicate), also the transitions can be created from a parametric definition.

Our definition of a parametric transition consists of three components (i, p, e) , where

- i is a list $(v_1, t_1), \dots, (v_n, t_n)$ of pairs (v, t) where v is a variable name and t is a type,
- p is a propositional formula with schematic variables as the atomic propositions, and
- e is an effect with schematic variables as state variables.

A parametric transition is instantiated by creating all possible bindings $(v_1, \dots, v_n) \in D_{t_1} \times \dots \times D_{t_n}$, and for each binding producing a succinct transition.

Example 6.13 We have the *arc* predicate of type (location,location) as discussed before, as well as a unary predicate *isAt* of type *location*.

A schematic transition that moves (a single unspecified object) between locations is defined as (i, p, e) where

- $i = (o_1, \text{location}), (o_2, \text{location})$
- $p = \text{isAt}(o_1) \wedge \text{arc}(o_1, o_2)$
- $e = \{ \text{isAt}(o_2), \neg \text{isAt}(o_1) \}$

This schematic transtion can be instantiated in 4×4 different ways, as there are two parameters each with 4 objects in its type domain. One of the instances of the transition is (p, e) where

- $p = \text{isAt}(\text{Helsinki}) \wedge \text{arc}(\text{Helsinki}, \text{Tampere})$
- $e = \{ \text{isAt}(\text{Tampere}), \neg \text{isAt}(\text{Helsinki}) \}$



Chapter 7

Symbolic Reachability Testing

State-space search is the problem of testing whether a state in a *transition system* is reachable from one or more *initial states*. Transition systems in the most basic cases can be identified with *graphs*, and the state-space search problem in this case is the s-t-reachability problem in graphs.

For small graphs the problem can be solved with standard graph search algorithms such as Dijkstra's algorithm. For graphs with an astronomically high number of states, 10^{10} or more, standard graph algorithms are impractical. Many transition systems can be compactly represented, yet their size as graphs is very high. This is because N Boolean (0-1) state variables together with a state-variable based representation of the possible actions or events can induce a state-space with the order of 2^n reachable states. This is often known as the *combinatorial explosion*. It turns out that the s-t-reachability problem for natural compactly represented graphs is PSPACE-complete [GW83, Loz88, LB90, Byl94].

Classical propositional logic has been proposed as one solution to state-space search problems for very large graphs, due to the possibility of representing and reasoning about large numbers of states with (relatively small) formulas.

7.1 OBDD-Based Symbolic Reachability

The logic-based methods are best understood in terms of relational operations, with formulas representing sets and relations. A basic state space traversal algorithm expressed with relation join and projection is as follows.

1. $i := 0$
2. $S_0 :=$ set of initial states
3. $i := i + 1$
4. $S_i := S_{i-1} \cup \pi_2(S_{i-1} \bowtie R)$
5. if $S_i \not\subseteq S_{i-1}$, go to 3.

On step 4, set S_i of those states that are reachable from the initial states by i step or less is computed. This consists of the states in S_{i-1} as well as those states that can be reached from a state in S_{i-1} by one step. These latter states are obtained by $\pi_2(S_{i-1} \bowtie R)$, where the natural join $S_{i-1} \bowtie R$ computes those pairs (s, s') of states where $s \in S_{i-1}$ has distance $i - 1$ and where s' can be reached from s by one step, as determined by the 1-step reachability relation R . From $S_{i-1} \bowtie R$ we obtain those states that are reached from S_{i-1} by projection to the second elements of the pairs, dropping the first element.

In Section 5 we have already shown how set and relation operations can be implemented as formula manipulation, when the sets S_i and the relations R are represented as formulas. Next we show how state space reachability problems can be solved with those techniques.

The first idea is that states are valuations of some set of state variables $X = \{x_1, \dots, x_n\}$. Any set of states can be represented as a formula on X , and any formula on X represents a set of states. Next, the arcs in any transition system (with states represented by valuations of X), representing transitions from states to states is a binary relation on valuations of X . As was shown before, such a relation can be represented as a formula on propositional variables $X \cup X'$, where $X' = \{x'_1, \dots, x'_n\}$. Here we have two propositional variables for every

state variable: one indicating its value before a transition takes place, and the other indicating its value after.

We already saw an example of this kind of formula in Example 5.6, here adapted to the x, x' convention on names of propositional variables.

$$\begin{aligned} inc &= (\neg c \wedge c' \wedge (b \leftrightarrow b') \wedge (a \leftrightarrow a')) \\ &\quad \vee (\neg b \wedge c \wedge b' \wedge \neg c' \wedge (a \leftrightarrow a')) \\ &\quad \vee (\neg a \wedge b \wedge c \wedge a' \wedge \neg b' \wedge \neg c') \\ &\quad \vee (a \wedge b \wedge c \wedge \neg a' \wedge \neg b' \wedge \neg c') \end{aligned}$$

Given a set of states, expressed by a formula ϕ on X , and a transition relation, expressed by a formula ρ on $X \cup X'$, we can compute their natural join by simply forming their conjunction $\phi \wedge \rho$. After this, we can project the resulting relation to the successor states of all possible transitions (s, s') from s to s' by existentially abstracting away all variables in X , by $\exists X.(\phi \wedge \rho)$. This formula has occurrences of variables from X' only. After renaming every $x' \in X'$ to the corresponding $x \in X$, the result is a formula over X .

Corresponding to the relation image operation

$$img_S(R) = \pi_2(S \bowtie R)$$

we now have the logical image operation

$$img_\phi(\rho) = \text{subst}_{X/X'}(\exists X.(\phi \wedge \rho)),$$

as well as a *pre-image* operation

$$img_\phi(\rho) = \exists X'.((\text{subst}_{X/X'}(\phi)) \wedge \rho)$$

for computing all the possible predecessor states of a set of states with respect to a relation.

Now we can express the relation breadth-first search algorithm in terms of formulas.

1. $i := 0$
2. $\phi_0 :=$ formula that represents all initial states
3. $i := i + 1$
4. $\phi_i := \phi_{i-1} \vee \text{preimg}_{\phi_{i-1}}(\rho)$;
5. if $\phi_i \not\equiv \phi_{i-1}$, go to 4.

If we are not interested in computing all reachable states, but only testing whether some states in a set G are reachable then we can terminate the computation earlier.

1. $i := 0$
2. $\phi_0 :=$ formula that represents all initial states
3. $i := i + 1$
4. $\phi_i := \phi_{i-1} \vee \text{preimg}_{\phi_{i-1}}(\rho)$;
5. if $\phi_i \not\equiv \phi_{i-1}$ and $\phi_i \wedge G$ is not satisfiable, go to 4.

Here line 5 has a slightly stronger condition: computation is only continued if no state in G has been reached.

Finally, when interested in the reachability of a specific state or sets of states, it is often useful to explicitly construct a transition sequence that leads from an initial state to those state. If we have the formulas ϕ_i from the previous computation, this can be achieved by the following procedure. Here we assume that the transition relation formula ρ is a disjunction $\rho_1 \vee \rho_2 \vee \dots \vee \rho_m$, and we are interested in knowing which of the constituent transitions, which we could call either *events* or *actions*, were taken to reach G . The initial value of i below is the final value of i in the previous algorithm.

1. $B_i := \phi_i \wedge G$
2. if $i = 0$, terminate.
3. $i := i - 1$
4. for some $t \in \{1, \dots, m\}$ such that $B_{i+1} \wedge \rho_t$ is satisfiable, do
 - (a) $B_i := \text{preimg}_{B_{i+1}}(\rho_t)$
 - (b) $t_i := t$
5. go to 2.

Here the sets B_1, \dots, B_i are states that can be reached when firing a sequence of transitions t_1, \dots, t_i so that a state in $\phi_i \wedge G$ is reached in the end.

Notice that only if the constituent transitions ρ_i are *deterministic* (it is a (partial) function, rather than a more general relation) will the firing of t_1, \dots, t_i guarantee that a state in G is reached in the end and that sets B_1, \dots, B_{i-1} are visited on the way.

7.2 SAT-Based Symbolic Reachability

OBDD-based symbolic state-space traversal find longer and longer transition sequences by interleaved natural joins and projections, when 1-step transitions are represented by the formula ρ over propositional variables X and X' , and the initial states are represented by the formula S_0 over X .

$$\begin{aligned} S_1 &= \text{subst}_{X/X'}(\exists X.(S_0 \wedge \rho)) \\ S_2 &= \text{subst}_{X/X'}(\exists X.(\text{subst}_{X/X'}(\exists X.(S_0 \wedge \rho)) \wedge \rho)) \\ S_3 &= \text{subst}_{X/X'}(\exists X.(\text{subst}_{X/X'}(\exists X.(\text{subst}_{X/X'}(\exists X.(S_0 \wedge \rho)) \wedge \rho)) \wedge \rho)) \\ &\vdots \end{aligned}$$

Another way of using the same transition relation formulas is to perform only the natural joins, ignoring the projections that are needed for “forgetting” about the past transition sequence. This will lead to formulas of the following form, when testing the reachability of states expressed by a formula G .

$$S_0 \wedge \text{subst}_{X@1/X'}(\rho) \wedge \text{subst}_{X@1/X, X@2/X'}(\rho) \wedge \text{subst}_{X@2/X, X@3/X'}(\rho) \wedge \dots \wedge \text{subst}_{X@n/X}(G).$$

The sets $X@i$ for integers i consist of all propositional variables $x@i$ where $x \in X$ is a state variable.

This formula is satisfiable if and only if there is a transition sequence of length n from one of the initial states to one of the goal states.

Example 7.1 The successor relation of 3-bit integers (000, 001), (001, 010), (010, 011), (011, 100), (100, 101), (101, 110), (110, 111) is represented by the formula

$$\begin{aligned} inc &= (\neg c \wedge c' \wedge (b \leftrightarrow b') \wedge (a \leftrightarrow a')) \\ &\vee (\neg b \wedge c \wedge b' \wedge \neg c' \wedge (a \leftrightarrow a')) \\ &\vee (\neg a \wedge b \wedge c \wedge a' \wedge \neg b' \wedge \neg c') \\ &\vee (a \wedge b \wedge c \wedge \neg a' \wedge \neg b' \wedge \neg c'), \end{aligned}$$

and the multiplication by 2 for 3-bit integers (left shift, losing the most significant bit) is represented by the formula

$$ml2 = (a' \leftrightarrow b) \wedge (b' \leftrightarrow c) \wedge \neg c'.$$

To test whether the bit-vector 101 is reachable from bit-vector 000 by four steps by using actions inc and $ml2$ test the satisfiability of the formula

$$\begin{aligned} &\neg a@0 \wedge \neg b@0 \wedge \neg c@0 \\ &\wedge (inc[X@0/X, X@1/X'] \vee ml2[X@0/X, X@1/X']) \\ &\wedge (inc[X@1/X, X@2/X'] \vee ml2[X@1/X, X@2/X']) \\ &\wedge (inc[X@2/X, X@3/X'] \vee ml2[X@2/X, X@3/X']) \\ &\wedge (inc[X@3/X, X@4/X'] \vee ml2[X@3/X, X@4/X']) \\ &\wedge a@4 \wedge \neg b@4 \wedge c@4 \end{aligned}$$

There is exactly one satisfying assignment.

i	$a@i$	$b@i$	$c@i$	action
0	0	0	0	inc
1	0	0	1	inc or $ml2$
2	0	1	0	$ml2$
3	1	0	0	inc
4	1	0	1	



The general scheme followed here involves generating formulas Φ_i for i -step reachability, and testing their satisfiability one by one, starting from Φ_0, Φ_1, \dots , until a satisfiable formula Φ_j is found. From the satisfying assignment a transition sequence can be extracted: for every two consecutive time points $i, i + 1$, the valuation of the propositional variables in $X@i \cup X@(i + 1)$ corresponds to (at least) one of the transitions.

7.2.1 Parallel Actions

The scheme we presented involves one transition at a time. Transition systems are typically described by transition rules, which correspond to the different possible actions or events, as discussed in Section 6.2.

To reduce the length of the state sequences to be considered, it is useful to try to pack more than one action/event transition in one step. Hence the transition relation formula does not represent the firing of one transition rule, but several. In this section we demonstrate how this type of partially ordered action sequences can be represented in the propositional logic. Here the partial ordering refers to the fact that at a given step, the ordering of the actions is not total: actions a_1, \dots, a_n are taken, but no strict ordering between them is imposed (explicitly).

We consider partial ordering schemes in which the simultaneous actions a_1, \dots, a_n , represented by rules $(c_1, e_1), \dots, (c_n, e_n)$ taken in state s_t of the state sequence s_0, \dots, s_N satisfy the following two properties.

- The conditions c_i for $i \in \{1, \dots, n\}$ all satisfy $s_t \models c_i$.
- Each effect e_i of action a_i causes the same changes as it would cause alone, irrespectively of the other actions taken in parallel.

Action a_1, \dots, a_n occurring “in parallel” means that they occur in some total ordering $a_{i_1}, a_{i_2}, \dots, a_{i_n}$ with $\{i_1, \dots, i_n\} = \{1, \dots, n\}$.

The effects e of actions.

$x := F(x_1, \dots, x_n)$	assignment of a state variable $x \in X = \{x_1, \dots, x_n\}$
$eITE(\phi, e_1, e_2)$	IF-THEN-ELSE conditional
$e_1; e_2$	sequential composition: execute e_1 and then execute e_2
$e_1 : e_2$	parallel composition: execute e_1 and e_2 simultaneously

In contrast to the weakest precondition predicate wp (Definition 6.11), what we need to determine the conditions under which a state variable changes. This is to distinguish between the cases in which x will be true because it was true already and did not become false, and in which x is changed to true.

The condition under which x becomes true is denoted by $cc_a(x)$, and the condition under which x becomes false is denoted by $cc_a(\neg x)$.

Definition 7.2

$$cc_e(x) = \perp \tag{7.1}$$

$$cc_{x:=\phi}(x) = \phi \tag{7.2}$$

$$cc_{y:=\phi}(x) = \perp \quad \text{when } x \neq y \tag{7.3}$$

$$cc_{x:=\phi}(\neg x) = \neg\phi \tag{7.4}$$

$$cc_{y:=\phi}(\neg x) = \perp \quad \text{when } x \neq y \tag{7.5}$$

$$cc_{eITE(\phi, e_1, e_2)}(l) = (\phi \wedge cc_{e_1}(l)) \vee (\neg\phi \wedge cc_{e_2}(l)) \tag{7.6}$$

$$cc_{e_1; e_2}(l) = (cc_{e_1}(l) \wedge wp_{e_1}(\neg cc_{e_2}(\bar{l}))) \vee wp_{e_1}(cc_{e_2}(l)) \tag{7.7}$$

$$cc_{e_1 : e_2}(l) = cc_{e_1}(l) \vee cc_{e_2}(l) \tag{7.8}$$

The propositional variables used to express the transition relation formulas with parallel action are as follows.

x	value of state variable x
x'	value of state variable x at the next time point
a	action a is executed?

Unlike in the formula in Section 6.2, here we have propositional variables a that indicate whether a given action is taken at the current time point.

To express the possibility of taking an action we need to make sure that the condition part c of the action (c, e) is true: $a \rightarrow c$. For given values for the action variables a_1, \dots, a_n and the propositional variables $x \in X$ that represent the current state, the following formula determines the new value for each state variable $x \in X$ uniquely.

$$x' \leftrightarrow \left(\bigvee_{a \in A} (a \wedge cc_a(x)) \right) \vee \left(x \wedge \neg \bigvee_{a \in A} (a \wedge cc_a(\neg x)) \right) \quad (7.9)$$

This formula is often represented in a different, but logically equivalent form.

First, we view the equivalence as $x' \leftrightarrow (\Phi_1 \vee (x \wedge \neg\Phi_2))$, where Φ_1 is the condition for x becoming true and Φ_2 is the condition for x becoming false. The formula can be equivalently written as follows.

$$\Phi_1 \rightarrow x' \quad (7.10)$$

$$x \wedge \neg\Phi_2 \rightarrow x' \quad (7.11)$$

$$\neg\Phi_1 \wedge \neg(x \wedge \neg\Phi_2) \rightarrow \neg x' \quad (7.12)$$

The formulas 7.10 simply consists of the following for every action.

$$a \wedge cc_a(x) \rightarrow x' \quad (7.13)$$

These formulas are called (positive) *effect axioms*, as they indicate for each action when they will make x true at the next time point.

The formulas 7.11 are known as (positive) *frame axioms* for historical reasons. They indicate the conditions under which x remains true when no action turns it false. They are often written in the form $\Phi_2 \wedge x \rightarrow x'$ or $x' \vee \neg x \vee \neg\Phi_2$.

Formula 7.12 can be rewritten to any of the following implications.

$$\neg\Phi_1 \wedge (\neg x \vee \Phi_2) \rightarrow \neg x' \quad (7.14)$$

$$(\neg\Phi_1 \wedge \neg x) \vee (\neg\Phi_1 \vee \Phi_2) \rightarrow \neg x' \quad (7.15)$$

The last version can be split to two implications.

$$\neg\Phi_1 \wedge \neg x \rightarrow \neg x' \quad (7.16)$$

$$\neg\Phi_1 \wedge \Phi_2 \rightarrow \neg x' \quad (7.17)$$

The implication 7.16 is the (negative) frame axiom, analogous to the (positive) frame axiom 7.11, and indicating the conditions under which x is false and remains false.

The implication 7.17 is the negative counterpart of the effect axiom 7.10. It contains the seemingly superfluous conjunct $\neg\Phi_1$. We have assumed that a state variable cannot simultaneously become true and false. Hence the seemingly superfluous conjunct could be removed. However, our equivalence 7.9 entails that x will be true whenever the conditions for it becoming true hold (by formula 7.10), and x will only become false if conditions for becoming false hold and conditions for becoming true *do not* hold, as is made explicit in formula 7.17.

7.2.2 Alternative Meanings of Parallelism

However, we need something more than the above formulas to guarantee that any set of simultaneous actions can be meaningfully executed.

Example 7.3 Consider actions

- $a_1 = (x, y := 0)$
- $a_2 = (y, x := 0)$

As a formula this is

$$\begin{aligned} x' &\leftrightarrow (x \wedge \neg a_2) & a_1 &\rightarrow x \\ y' &\leftrightarrow (y \wedge \neg a_1) & a_2 &\rightarrow y \end{aligned}$$

If both a_1 and a_2 are true, both x' and y' will be false. This does not correspond to any sequential execution of the actions, as the actions disable each other. ■

Only such sets of actions are acceptable that can be executed in some total ordering. To guarantee that such a total ordering exists we need some auxiliary concepts.

Definition 7.4 Action (p_1, e_1) (possibly) disables (p_2, e_2) if some $x \in X$ occurs in e_1 in an assignment $x := \dots$ and in p_2 .

Definition 7.5 Action (p_1, e_1) affects (p_2, e_2) if some $x \in X$ occurs in e_1 in an assignment $x := \dots$ and in ϕ for some $eITE(\phi, \dots, \dots)$ in p_2 .

Note that these definitions are approximate. E.g. $(\top, (a := 1))$ “disables” $(a, (x := 1))$, although it really doesn't. These definitions could be strengthened e.g. by considering whether the state variables occur positively or negatively.

Example 7.6 $(\top, (x := 0))$ disables $(x \vee y, (a := 1))$ ■

Example 7.7 $(\top, (x := 0))$ affects $(\top, (x := 1))$ ■

Example 7.8 $(\top, (x := 0))$ affects $(\top, eITE(x, (a := 1), (b := 1)))$ ■

The simplest condition for simultaneous actions is that they can be executed in any total ordering. This is achieved by the concept of interference that expresses a symmetric dependency relation.

Definition 7.9 (Interference) a_1 and a_2 interfere if

1. a_1 disables or affects a_2 , or
2. a_2 disables or affects a_1

To guarantee that execution in any order is possible and leads to the same state the following suffices. If a_1 and a_2 interfere, then include the following formula.

$$\neg a_1 \vee \neg a_2$$

A weaker, directed definition of dependence, sometimes allowing many more actions in parallel, is the following.

Definition 7.10 (Directed Interference) a_1 interferes with a_2 if a_1 disables or affects a_2 .

Now, even a set of actions that have interfere can be taken in parallel, as long as the directed interference relation is acyclic. A particularly simple way of achieving this is based on an arbitrarily total ordering a_1, \dots, a_n on the set of all actions. If the action a_i interferes with a_j and $j > i$, then we need include the following formula.

$$\neg a_i \vee \neg a_j$$

Now, given a state s in which a set $P \subseteq \{a_1, \dots, a_n\}$ of actions is applicable (their conditions are true, and their effects are non-conflicting) and the above constraints are satisfied, these actions can be executed in an ordering indicated by the ordering of the set of all actions. This is because each action is applicable in s , and if an action disables (or can disable) another action, then the latter action is earlier in the ordering.

Example 7.11 Consider the nesting of Russian matryoshka dolls, expressed as the following actions.

$$\begin{aligned} a_1 &= (\text{out1} \wedge \text{out2} \wedge \text{empty2}, (1\text{in2} := 1; \text{out1} := 0; \text{empty2} := 0)) \\ a_2 &= (\text{out2} \wedge \text{out3} \wedge \text{empty3}, (2\text{in3} := 1; \text{out2} := 0; \text{empty3} := 0)) \\ a_3 &= (\text{out3} \wedge \text{out4} \wedge \text{empty4}, (3\text{in4} := 1; \text{out3} := 0; \text{empty4} := 0)) \end{aligned}$$

A doll can be put inside another if neither is inside another doll, and the bigger one is empty. As a result, the bigger will not be empty any more, and the smaller is inside the bigger one. Notice that action a_3 disables a_2 , and a_2 disables a_1 .

Consider the initial state in which the four dolls are initially un-nested. With the stricter definition of parallelism, they can be nested one by one, by taking the actions a_1, a_2, a_3 in this order. But with the more relax definition of parallelism, the nesting can be performed in one step, as all actions are initially applicable, none of the effects conflict, and the directed interference relation is acyclic. The ordering of the actions is still implicitly the same a_1, a_2, a_3 , but in terms of the definition of parallelism they can be taken at the same time. ■

References

For the solution of reachability problems in discrete transition systems the use of propositional logic as a representation of sets and relations was first proposed in the works by Coudert et al. [CBM90, CM90] and Burch et al. [BCL⁺94].

The use of these logic-based techniques was possible with (ordered, reduced) Binary Decision Diagrams [Bry92], generally known as BDDs or OBDDs. BDDs are a restrictive normal form that guarantees a *canonicity* property: any two logically equivalent Boolean formulas are represented by a unique BDD. The canonicity property guarantees the polynomial-time solvability of many central problems about Boolean formulas, including *equivalence*, *satisfiability*, and *model-counting*. Though in some cases necessarily (even exponentially) larger than unlimited Boolean formulas, BDDs also in practice often lead to compact enough representations when unlimited formulas would in practice be too large.

Other similar normal forms, with useful polynomial-time operations, exist, including Decomposable Negation Normal form (DNNF) [Dar01, Dar02]. There are interesting connections between this type of normal forms and the Davis-Putnam procedure [HD05].

Problems with scalability of BDD-based methods was well-understood already in mid-1990ies. Another way of using the propositional logic for solving state-space search problems was proposed by Kautz and Selman [KS92] in 1992, and in 1996 they demonstrated it to be often far better scalable than alternative search methods for solving state-space search problems in AI [KS96]. This method was based on mapping one reachability query, whether a state satisfying a given goal formula is reachable in a given number of steps from a specified initial state. The method suffered from the large size of the formulas far less than BDDs, and was soon shown to be in many cases far better scalable than BDD-based methods for example in Computer Aided Verification, for solving the model-checking problem [BCCZ99].

In Sections 5.2 and 6.2 we only presented the most basic translation of action sequences into propositional formulas, with the exactly one action at each step. This is what is known as a *sequential* encoding. This representation suffers from the possibility of sequences of actions a_1, \dots, a_n that are independent of each other, and which therefore can be permuted to every one of $n!$ different orders. These representations are what is typically used in OBDD-based reachability methods. Having several actions can simultaneous decreases the horizon length. This substantially improves the scalability of SAT-based methods in the main applications including the planning, diagnosis, and model-checking problems. The parallel representations, with multiple simultaneous events or actions, come from Kautz and Selman's works [KS92, KS96]. The more relaxed notion of parallelism, with weaker constraints and potential for more parallelism, by requiring that at least one ordering, rather than all total orderings, are executable, were investigated later [DNK97, RHN06]. The total orderings can be often chosen so that no constraints limiting the parallelism are needed at all [RHN06].

That the SAT problems in testing the existence of transition sequences of increasing lengths are related in a way that can be used for speeding up the SAT solving process, was recognized and utilized by Eén and Sörensson in *incremental SAT solving* [ES03], in which the clauses learned by the CDCL algorithm for the formula for horizon length i could be used also for horizon length $i + 1$, with potentially high speed-ups. Another way of speeding up the search for the satisfiable formulas is to solve the sequence of formulas in parallel, rather than one by one by considering the horizon length i only after the formula for horizon length $i - 1$ has been determined to be unsatisfiable [SS07, Rin04, RHN06].

Chapter 8

Modal Logics

Many concepts are not definable truth-functionally. Consider the following.

- It is possible that ϕ holds.
- At the next time point ϕ holds.
- After executing the program π , the formula ϕ will hold.

In all of these, the truth of ϕ does not in all cases determine the truth of the sentence in question. Modal logics are logics that, in addition to the usual Boolean connectives, also include *modal operators* for defining concepts such as the ones mentioned above. A major difference between modal operators and Boolean connectives is that the former are not truth-functional: whereas the truth of a formula like $\phi_1 \vee \phi_2$ is a function of the truth-values of ϕ_1 and ϕ_2 , the truth of a modal formula $\Box\phi$ is *not* a function of the truth of ϕ . If a is true, the formula $\Box a$ could be true or false, depending on the circumstances. This difference becomes obvious from the way the model-theory (semantics) of modal logics is defined.

The interesting and intriguing thing is that the generalization of the propositional logic for the above, and many other, concepts can be achieved in the same kind of formal framework. This framework involves defining the truth of generalized formulas in a *graph*, in which the nodes are propositional valuations, and the nodes are connected to each other by arcs. This framework is related to the notion of transition systems with state variables (Definition 6.3), as we will see later.

In the next sections we first consider basic uni-modal logics with one modality only (the dual modal operators \Box and \Diamond). Then we consider multi-modal logics with multiple modalities, temporal logics which talk about time, as well as dynamic logics which allow talking about programs and paths in graphs.

8.1 Modal Logics with One Modality

Modal logics with a single modality were the earliest ones, first used in philosophical investigations in concepts like *necessity* and *possibility*, and later with other concepts. These logics have a modal operator \Box , with $\Box\phi$ read as “it is necessary that ϕ ”, and its dual operator \Diamond with $\Diamond\phi$ read as “it is possible that ϕ ”. Depending on the application, these operators have multiple readings.

- $\Box\phi$: It is necessary that ϕ (Alethic logic)
- $\Box\phi$: It is obligatory that ϕ (Deontic logic)
- $\Box\phi$: It is known that ϕ (Epistemic Logic)
- $\Box\phi$: It is believed that ϕ (Epistemic Logic)
- $\Box\phi$: Always in the future ϕ (Temporal Logic)
- $[\pi]\phi$: After executing π necessarily ϕ (Dynamic Logic)

As pointed out, the dual concept of *necessity* is *possibility*, with the operators \Box and \Diamond related by $\Diamond\phi \equiv \neg\Box\neg\phi$: something is possible if its negation is not necessary. Similarly, “obligatory” is dual to “permissible”, and “always” is dual to “sometimes”. There is no unique dual for “is believed” or “is known” in English, but the dual concepts could be expressed as “it is considered possible (as far as knowledge/belief is concerned)”. Notice that these two modal operators are related similarly to the universal and existential quantification operators $\forall x.\phi \equiv \neg\exists x.\neg\phi$. It

would be sufficient to use only one of these pairs of operators, but for convenience both are used.

Beliefs, for example, do not in general characterize the actual world exactly, and as beliefs may be false, they might not concern the actual world at all. And, it turns out, beliefs can be characterized by the set of worlds that are considered possible by the person/agent in question:

1. If I consider only such worlds possible in which ϕ is true, then I believe ϕ .
2. If I don't believe ϕ (this is different from believing $\neg\phi$!), then I consider at least one world possible in which A is false.

Clearly, I could not normally believe both A and $\neg A$ (the formula $(\Box A) \wedge (\Box \neg A)$ should in general be false) and none of the possible worlds satisfy both A and $\neg A$, but I might consider both A and $\neg A$ as possible as far as my beliefs are concerned ($(\Diamond A) \wedge (\Diamond \neg A)$ could well be true), and there could well be two possible worlds, one of which satisfying A and the other $\neg A$.

In deontic logics, in which the operator \Box is interpreted as “it is obligatory that”, the set of worlds that are referred to when interpreting sentences like “it is obligatory that” are those that satisfy all the laws and norms that we are considering. ϕ is obligatory if and only if ϕ is true in all of those worlds. If ϕ is permitted, then there is at least one world in which ϕ is true. Further, in temporal logics (which are considered in more detail later in Section 8.3 and 8.4), the worlds that are referred to by operators like “always in the future” are those representing the state of the world in later time points.

A simple semantics for a fragment of modal logics could be given by having a valuation w for the actual world and a set of valuations W for all the possible worlds. Then we could say that $\Box\phi$ is true if ϕ is true in all worlds/valuations in W . However, this definition works only when ϕ does not contain modal operators. For the more general case, with the possibility of nesting modal operators, a more sophisticated semantics is needed.

Starting in 1959, Saul Kripke (a 19-year old Harvard undergraduate) developed a semantics for a wide range of modal logics based on possible worlds [Kri63a, Kri63b, Kri65]. The semantics is very intuitive and powerful, and it is the main reason for the wide use of different modal logics in different kinds of applications, varying from philosophical logic to computer science. The idea is to view the set of all possible worlds as a graph. Formally, a *frame* is a pair $F = \langle W, R \rangle$, where W is a set of possible worlds and $R \subseteq W \times W$ (the accessibility relation) is a binary relation on W . A frame just describes how the possible worlds are related to each other, it does not yet say anything about the truth and falsity of formulae.

Let X be a finite set of propositional variables. A *model* M based on the frame F is a 4-tuple $M = \langle W, R, X, v \rangle$ where W and R are like in F , and $v : W \times X \rightarrow \{0, 1\}$ is a valuation that assigns the truth value *true* or *false* (respectively represented by 1 and 0) to each propositional variable in every possible world.

Definition 8.1 (Uni-modal Kripke model) A Kripke model $M = \langle W, R, X, v \rangle$ consists of

- a set W of worlds,
- a accessibility relation $R \subseteq W \times W$,
- a set X of atomic formulas,
- a valuation $v : W \times X \rightarrow \{0, 1\}$ of propositional variables in worlds.

For convenience, we will write $v_w(x)$ for $v(w, x)$.

The truth of a formula is defined with respect to one of the nodes of a Kripke model.

We write $M \models_w \phi$ for the truth of a modal formula ϕ in world w in a Kripke model M .

Definition 8.2 (Truth of uni-modal formulas) Let $M = \langle W, R, X, v \rangle$ be a Kripke model.

1. $M \models_w x$ iff $v_w(x) = 1$
2. $M \models_w \phi \wedge \psi$ iff $M \models_w \phi$ and $M \models_w \psi$
3. $M \models_w \phi \vee \psi$ iff $M \models_w \phi$ or $M \models_w \psi$
4. $M \models_w \neg\phi$ iff not $M \models_w \phi$
5. $M \models_w \Box\phi$ iff $M \models_{w'} \phi$ for all $w' \in W$ such that wRw'
6. $M \models_w \Diamond\phi$ iff $M \models_{w'} \phi$ for at least one $w' \in W$ such that wRw'

All but the last two lines in the above definition are essentially the same as the truth-definition for the propositional

logic. The second last line defines $\Box\phi$ to be true in the world w if and only if ϕ is true in all worlds w' that are accessible from the current world w according to the relation R .

The necessity operator requires that something holds in *all* accessible states, whereas the possibility requires only that something holds in *at least one* accessible state. So it is not surprising that the relation between \Box and \Diamond is analogous to the *universal* \forall and *existential* \exists quantifications which are related by $\forall x.\phi \equiv \neg\exists x.\neg\phi$ and $\exists x.\phi \equiv \neg\forall x.\neg\phi$.

8.1.1 Validity in classes of frames

The following formulae all valid in *all* frames, meaning that they automatically follow if we are to have Kripke models as the semantics of the logic.

- $\Box\top$
- $\Box(a \rightarrow b) \rightarrow (\Box a \rightarrow \Box b)$ (the formula schema **K**)
- $\Box(a \wedge b) \leftrightarrow (\Box a \wedge \Box b)$
- $\Diamond(a \vee b) \leftrightarrow (\Diamond a \vee \Diamond b)$
- $\Box(a \rightarrow b) \rightarrow (\Diamond a \rightarrow \Diamond b)$

Also, as we will see later when we axiomatically formalize different modal logics, the set of formulae that are valid in a class of frames have the following properties.

- If ϕ is valid in a class C of frames, then also $\Box\phi$ is valid in C .
- If ϕ and $\phi \rightarrow \phi'$ are valid in a class C of frames, then also ϕ' is valid in C .
- If ϕ is valid in a class C of frames, then any substitution instance of ϕ is valid in C .

In the first two cases this is simply because the set of formulae true in one model have the analogous properties.

8.1.2 Properties of the accessibility relation

What makes Kripke models interesting is that there is a close connection between several important axioms schemata and the formal properties of the accessibility relations.

Let us consider the following axiom schemata.

1. **T**: $\Box\varphi \rightarrow \varphi$ (alternatively $\varphi \rightarrow \Diamond\varphi$)
2. **4**: $\Box\varphi \rightarrow \Box\Box\varphi$ (alternatively $\Diamond\Diamond\varphi \rightarrow \Diamond\varphi$)
3. **5**: $\Diamond\varphi \rightarrow \Box\Diamond\varphi$
4. **B**: $\varphi \rightarrow \Box\Diamond\varphi$
5. **D**: $\Box\varphi \rightarrow \Diamond\varphi$ (alternatively $\Box\varphi \rightarrow \neg\Box\neg\varphi$)
6. **L**: $\Box(\varphi \wedge \Box\varphi \rightarrow \chi) \vee (\Box(\chi \wedge \Box\chi \rightarrow \varphi)$

Now, these schemata exactly correspond to the following properties of the accessibility relation. We say that a frame is reflexive, for example, if its accessibility relation is reflexive.

1. reflexivity (wRw for every world w),
2. transitivity ($wRu \wedge uRv$ implies wRv),
3. euclidicity ($wRu \wedge wRv$ implies uRv),
4. symmetry (wRu implies uRw),
5. seriality (for every w there exists v with wRv)
6. weak connectedness ($sRt \wedge sRu$ implies that either tRu , $t = u$ or uRt)

For example, if the frame F is reflexive, then the axiom schema **T** is valid in F . Also the opposite holds. If the axiom schema **T** is valid in a frame F , then F is reflexive. We prove these things and leave the analogous proofs for the other properties as an exercise.

Theorem 8.3 *Axiom schema **T** is valid in the class of reflexive frames.*

Proof: Let F be a reflexive frame. Let M be a model based on F and let w be any world in M . If $\Box\varphi$ is not true in w , then axiom **T** is true in w . If $\Box\varphi$ is true in w , then φ is true in all worlds that are accessible from w . Since the accessibility relation is *reflexive*, w is among the accessible worlds, i.e., φ is true in w . This means that also in this case **T** is true in w . This means, **T** is true in all worlds in all models based on reflexive frames, which we wanted to show. \square

Theorem 8.4 *If **T** is valid in a frame F , then F is reflexive.*

Proof: We prove the contraposition of the claim, that is, start from the negation of the conclusion and derive the negation of the antecedent.

Assume that F is not reflexive. We will construct a model based on F that falsifies **T**. Because F is not reflexive, there is a world w such that not wRw . Construct a model M such that $M \not\models_w p$ and $M \models_v p$ for all v such that wRv . Now $M \models_w \Box p$ and $M \not\models_w p$, and hence $M \not\models_w \Box p \rightarrow p$. \square

The next table summarizes the correspondences between relational properties and modal axioms.

Name	Property	Axiom schema
K	–	$\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$
T	reflexivity	$\Box\varphi \rightarrow \varphi$
4	transitivity	$\Box\varphi \rightarrow \Box\Box\varphi$
5	euclidicity	$\Diamond\varphi \rightarrow \Box\Diamond\varphi$
B	symmetry	$\varphi \rightarrow \Box\Diamond\varphi$
D	seriality	$\Box\varphi \rightarrow \Diamond\varphi$
L	weak connectedness	$\Box(\varphi \wedge \Box\varphi \rightarrow \chi) \vee (\Box(\chi \wedge \Box\chi \rightarrow \varphi))$

8.1.3 Different uni-modal logics

Because the different properties considered above are to some extent mutually independent, there is a possibility to construct many different modal logics corresponding to different classes of frames. Interestingly, the early modal logics $S4$ and $S5$, originally developed without reference to Kripke-style semantics by C. I. Lewis in 1910, are some of these logics.

One axiom schema is common to most logics that correspond to a class of frames, namely the axiom schema

$$\mathbf{K} = \Box(a \rightarrow b) \rightarrow (\Box a \rightarrow \Box b).$$

We call the corresponding logic **K**. In general, we simply give the list of names of axioms schemata to name a logic, but many of the logics of historical importance have their own name that is usually used.

$$\begin{aligned} & K \\ S4 &= KT4 \\ S5 &= KT5 \\ K4.3 &= K4L \\ S4.3 &= KT4L \\ & \vdots \end{aligned}$$

These logics have been applied for different purposes. The choice of axiom schemata depends on how one exactly wants to interpret the modal operators. Some properties often used for formalizations of natural language notions like knowledge, belief, and obligation, among others, are summarized in the following table.

logics	\Box	$\Diamond = \neg\Box\neg$	K	T	4	5	B	D
alethic	necessary	possible	Y	Y	?	?	?	Y
epistemic	known	possible	Y	Y	Y	Y	N	Y
doxastic	believed	possible	Y	N	Y	Y	N	Y
deontic	obligatory	permitted	Y	N	N	?	?	Y
temporal	always (in future)	sometimes	Y	Y	Y	N	N	Y

8.1.4 Undefinable properties

Many properties of binary relations exactly correspond to modal logic formulae. However, there are simple properties of binary relations that do not have a corresponding axiom schema. One such property is irreflexivity. For showing that there is no modal axiom schema corresponding to irreflexivity we use p-morphisms.

Definition 8.5 Let $M_1 = \langle W_1, R_1, X, v_1 \rangle$ and $M_2 = \langle W_2, R_2, X, v_2 \rangle$ be models over the same propositional variables X . Then a p-morphism from M_1 to M_2 is a function $f : W_1 \rightarrow W_2$ such that

- sR_1t implies $f(s)R_2f(t)$,
- $f(s)R_2u$ implies that there is $t \in W_1$ such that sR_1t and $f(t) = u$, and
- $M_1 \models_s p$ iff $M_2 \models_{f(s)} p$ for all $p \in X$ and $s \in W_1$.

So, every world in M_1 is mapped to a world in M_2 , and the arrows between world in M_1 are similarly mapped to arrows in M_2 . Notice that M_2 may have worlds that do not correspond to any world in M_1 , and that there may be arrows in M_2 that do not correspond to any arrow in M_1 as long as the starting world does not correspond to any world in M_1 .

A p-morphism between the frames $F_1 = \langle W_1, R_1 \rangle$ and $F_2 = \langle W_2, R_2 \rangle$ is a function $f : W_1 \rightarrow W_2$ that satisfies the first two conditions of the definition of p-morphisms between models.

Lemma 8.6 Let f be a p-morphism from $M_1 = \langle W_1, R_1, X, v_1 \rangle$ to $M_2 = \langle W_2, R_2, X, v_2 \rangle$. Then for all $s \in W_1$ and all formulae ϕ , $M_1 \models_s \phi$ if and only if $M_2 \models_{f(s)} \phi$.

Proof: By induction on the structure of the formula.

Base case: ϕ is an atomic formula. For any $s \in W_1$ by definition of p-morphisms, $M_1 \models_s \phi$ if and only if $M_2 \models_{f(s)} \phi$.

Inductive cases:

1. Let $\phi = \psi \rightarrow \psi'$. By induction hypothesis $M_1 \models_s \psi$ iff $M_2 \models_s \psi$, and $M_1 \models_s \psi'$ iff $M_2 \models_s \psi'$. From this the result directly follows.
2. Let $\phi = \neg\psi$. By induction hypothesis $M_1 \models_s \psi$ iff $M_2 \models_s \psi$. From this the result directly follows.
3. Let $\phi = \Box\psi$. By induction hypothesis $M_1 \models_{s'} \psi$ iff $M_2 \models_{f(s')} \psi$ for all worlds $s' \in W_1$, including those for which sR_1s' . From this the result directly follows.

Therefore, $M_1 \models_s \phi$ if and only if $M_2 \models_{f(s)} \phi$, for all $s \in W_1$ and all formulae ϕ . \square

Lemma 8.7 Let there be a surjective p-morphism f from frame F_1 to F_2 . Then for any formula ϕ , $F_1 \models \phi$ implies $F_2 \models \phi$.

Proof: Assume there is a model $M_2 = \langle W_2, R_2, X, v_2 \rangle$ based on F_2 such that $M_2 \not\models_t \phi$ for some $t \in W_2$. Define the model M_1 based on F_1 by

$$M_1 \models_s p \text{ iff } M_2 \models_{f(s)} p \text{ for all } s \in W_1$$

Take any s such that $f(s) = t$ (such an s exists because f is a surjection.) Now by Lemma 8.6 $M_1 \not\models_s \phi$. \square

Theorem 8.8 The class of irreflexive frames is not defined by any axiom schema.

Proof: We show that there is an irreflexive frame F_1 that has a p-morphic image that is not irreflexive. Consider the frame $\langle N, < \rangle$ of natural numbers ordered by the less than relation. The frame $\langle \{1\}, \{(1, 1)\} \rangle$ is its p-morphic image but not irreflexive.

Hence the class of irreflexive frames is not closed under p-morphic images. But the class of frames defined by any axiom schema is closed under p-morphic images by Lemma 8.7. This shows that the class of irreflexive frames is not definable by any axiom schema. \square

8.1.5 Logical consequence in modal logics

Logical consequence $\Sigma \models \phi$ in the classical propositional logic is defined as truth of the consequent ϕ in all models in which the antecedents Σ are true.

In modal logic there are several alternative definitions of logical consequence, because the truth of the antecedent can be considered in several different ways: do we require that the antecedents are true in all worlds of the model and then check whether ϕ is true in all worlds of a model, or do we just check that ϕ is true in those worlds of a model in which Σ are true?

These two possibilities lead to two different notions of logical consequence. The first is known as the *global* definition of logical consequence, and the second as the *local* definition of logical consequence. In this lecture we use the local definition.

Definition 8.9 (Logical consequence) *The formula ϕ is a logical consequence of a set Σ of formulae in a class C of frames (or models) (written $\Sigma \models_C \phi$) iff for all worlds s in all models M based on $F \in C$ (or all models M in C) $M \models_s \Sigma$ implies $M \models_s \phi$*

Clearly, logical consequence from the empty set $\Sigma = \emptyset$ in a class of frames coincides with validity in that class of frames.

8.2 Multi-Modal Logics

We can easily have multiple modalities by having multiple modal-operators each corresponding to a different accessibility relation.

Definition 8.10 (Multi-modal Kripke model) *A Kripke model $M = \langle W, D, R, X, v \rangle$ consists of*

- a set W of worlds,
- a set D of modalities,
- an assignment $R : D \rightarrow 2^{W \times W}$ of accessibility relations to each modality,
- a set X of atomic formulas,
- a valuation $v : W \times X \rightarrow \{0, 1\}$ of propositional variables in worlds.

Now we have multiple modal operators \Box_d , one for each $d \in D$.

Definition 8.11 (Truth of multi-modal formulas) *1. $M \models_w x$ iff $v_w(x) = 1$*

2. $M \models_w \phi \wedge \psi$ iff $M \models_w \phi$ and $M \models_w \psi$
3. $M \models_w \phi \vee \psi$ iff $M \models_w \phi$ or $M \models_w \psi$
4. $M \models_w \neg \phi$ iff not $M \models_w \phi$
5. $M \models_w \Box_d \phi$ iff $M \models_{w'} \phi$ for all $w' \in W$ such that $wR(d)w'$

We consider multi-modal logics of belief and knowledge of several agents. Each modality $d \in D$ correspond to one of the agents, and the operator \Box_d refers to the beliefs of the agent: $\Box_d \phi$ means that the agent d belief that ϕ , that is, ϕ is true in all worlds that the agent considers possible. What the agent considers possible is represented by the accessibility relation $R(d)$.

The beliefs of an agent depend on the agent's accessibility relation.

Commonly used axioms in epistemic logics.

1. $\Box_i(p \rightarrow q) \rightarrow (\Box_i p \rightarrow \Box_i q)$ (Distribution Axiom for Knowledge)
2. $\Box_i p \rightarrow p$ (Knowledge/Truth Axiom)
3. $\Box_i p \rightarrow \Box_i \Box_i p$ (Positive Introspection)
4. $\neg \Box_i p \rightarrow \Box_i \neg \Box_i p$ (Negative Introspection)

We are often interested in the multi-agent epistemic logic $S5_n$ that satisfies these principles. The accessibility relation of each agent is then of course reflexive, transitive and euclidian.

Knowledge generalization rule:

If $M \models \phi$, then $M \models_s \Box_i \phi$ for every M , ϕ and s .

For many applications it is useful to introduce further modal operators based on the operator K .

Define the operator “everybody knows that ϕ ” by

$$M \models_s E\phi \text{ iff } M \models_s \Box_i \phi \text{ for every } i \in \{1, \dots, n\}$$

This of course corresponds to the axioms schema

$$E\phi \leftrightarrow (\Box_1 \phi \wedge \dots \wedge \Box_n \phi).$$

Define the operator “it is common knowledge that ϕ ” by

$$M \models_s C\phi \text{ iff } M \models_s E^i \phi \text{ for every } i \geq 1$$

Here E^i means a sequence $EEEE \dots E$ of i Es.

Common knowledge has the following property

$$C\phi \leftrightarrow E(\phi \wedge C\phi).$$

There is a simple and very useful graph-theoretic view of these two operators in terms of paths in the Kripke model.

Definition 8.12 A world t is reachable from a world in s in k steps if there is a sequence of worlds s_0, s_1, \dots, s_k so that

1. $s_0 = s$ and $s_k = t$, and
2. $s_i R_j s_{i+1}$ for every $i \in \{0, \dots, k-1\}$ and any $j \in \{1, \dots, n\}$.

Lemma 8.13 $M \models_s E^k \phi$ iff $M \models_t \phi$ in every world t that is reachable from s in k steps.

$M \models_s C\phi$ iff $M \models_t \phi$ in every world t that is reachable from s in any number of steps.

8.3 Linear Temporal Logic LTL

Linear temporal logic is one of the best-known and most used modal logics in computer science. It can be viewed as a sub-logic of the branching time temporal logic CTL* (Section 8.4), omitting the path quantifiers A and E that refer to the branching of time.

Since LTL cannot refer to alternative futures, the truth of its formulas are interpreted over individual paths or state sequences, and in the CTL* terminology, all LTL formulas are *path formulas*, meaning that they talk about truth of facts along a single timeline, often corresponding to a path in a tree or graph of computations.

The semantics of LTL, similarly to the temporal logics CTL and CTL*, is not given in the standard framework of Kripke models as for the uni-modal and multi-modal logics.

Definition 8.14 A linear temporal model $M = (X, \sigma)$ consists of propositional variables X and an infinite sequence of propositional valuations $\sigma = (v_0, v_1, v_2, \dots)$

- $M \models_i x$ if and only if $v_i(x) = 1$
- $M \models_i \neg\phi$ if and only if $M \not\models_i \phi$
- $M \models_i \phi_1 \vee \phi_2$ if and only if $M \models_i \phi_1$ or $M \models_i \phi_2$
- $M \models_i \phi_1 \wedge \phi_2$ if and only if $M \models_i \phi_1$ and $M \models_i \phi_2$
- $M \models_i G\phi$ if and only if $M \models_j \phi$ for all $j \geq i$
- $M \models_i F\phi$ if and only if $M \models_j \phi$ for some $j \geq i$
- $M \models_i X\phi$ if and only if $M \models_{i+1} \phi$
- $M \models_i \phi U \psi$ if and only if for some $j \geq i$, $M \models_j \psi$ and $M \models_k \phi$ for all $k \in \{i, \dots, j-1\}$

8.4 Branching Time Temporal Logics CTL and CTL*

In many applications of temporal logics in computer science the temporal logic models are based on transition systems, best understood as arbitrary Kripke structures, instead of linearly ordered structures like in the linear temporal logic.

An arbitrary Kripke model naturally represents not just one linear ordering of worlds, but a tree, representing a high number of linear orderings corresponding to the paths in the tree. In this setting it is natural to extend the linear logic language further to be able to express properties of such trees. This leads to branching time temporal logics, of which we discuss the computation tree logic CTL*, and its sublogics LTL and CTL [Eme90].

8.4.1 The language of CTL*

1. Every propositional variable $x \in X$ is a formula.
2. If φ and ψ are formulae, then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$, $F\varphi$, $G\varphi$, $X\varphi$, $\varphi U \psi$, $A\varphi$, $E\varphi$.

The semantics of the operators F , G , X and U is known from the linear time temporal logics. The *path quantifiers* quantify over all of the possibly infinite number of computation paths in a transition system. Formulae $E\varphi$ with existential path quantification say that φ must be true on at least one computation path, and formulae $A\varphi$ say that φ must be true on all computation paths through the current state.

Example 8.15 CTL* formulae with path quantification are useful in expressing many correctness properties programs, controllers or other systems might have to fulfill.

$AG\neg\text{failure}$	A failure state is never reached.
$AG(EF\text{restart})$	From every state of the system it is possible to reach the <i>restart</i> state. That is, the system cannot get stuck anywhere.
$AG(\text{request} \rightarrow AF\text{ack})$	Any request will be eventually acknowledged.

■

8.4.2 The semantics of CTL*

A model in the computation tree logic is a tuple $M = \langle W, R, X, v \rangle$ where W is a set of states (possible worlds), R is an accessibility relation so that for every $w \in W$ there is $w' \in W$ such that wRw' , X is the set of propositional variables, and $v : W \times X \rightarrow \{0, 1\}$ is a valuation that assigns truth values to propositional variables in every state.

The truth of CTL* formulae are in general defined over a computation path, an infinite sequence of states in a model. However, for *state formulae* that only have E and A as their outermost modal operators the truth can also be defined with respect to a state.

Definition 8.16 A computation path is an infinite sequence of states $\lambda = w_0w_1w_2\dots$ such that w_iRw_{i+1} for each $i \geq 0$. For a path $\lambda = w_0w_1w_2\dots$, $\lambda[i]$ denotes the state w_i . For a path λ , λ^i denotes the *i*th suffix of a path λ , that is, λ^i is defined by $\lambda^i[j] := \lambda[i+j]$ for all $j \geq 0$.

We often write briefly “path” instead of “computation path”.

The truth-definition of formulae in states $w \in W$ or over computation paths λ is as follows.

1. $M \models_w x$ iff $v(w, x) = 1$
2. $M \models_w \neg\varphi$ iff $M \not\models_w \varphi$
3. $M \models_w \varphi \vee \psi$ iff $M \models_w \varphi$ or $M \models_w \psi$
4. $M \models_w A\varphi$ iff $M \models_\lambda \varphi$ for every path λ such that $\lambda[0] = w$
5. $M \models_w E\varphi$ iff $M \models_\lambda \varphi$ for some path λ such that $\lambda[0] = w$
6. $M \models_\lambda \varphi$ iff $M \models_{\lambda[0]} \varphi$ for state formulae φ
7. $M \models_\lambda \neg\varphi$ iff $M \not\models_\lambda \varphi$
8. $M \models_\lambda \varphi \vee \psi$ iff $M \models_\lambda \varphi$ or $M \models_\lambda \psi$
9. $M \models_\lambda G\varphi$ iff $M \models_{\lambda^i} \varphi$ for every $i \geq 0$
10. $M \models_\lambda F\varphi$ iff $M \models_{\lambda^i} \varphi$ for some $i \geq 0$
11. $M \models_\lambda X\varphi$ iff $M \models_{\lambda^1} \varphi$
12. $M \models_\lambda \varphi U \psi$ iff there is $i \geq 0$ such that $M \models_{\lambda^i} \psi$ and $M \models_{\lambda^j} \varphi$ for every $j \in \{0, \dots, i-1\}$.

8.4.3 Computation tree logic CTL

CTL is a simpler variant of CTL* that does not allow arbitrary combination of temporal operators. Specifically, every occurrence of operators X , F , G and U is *preceded by the path quantifier E or A* . Essentially, we have 8 modal operators $AX AF AG AU EX EF EG EU$. Hence every formula is a state formula and can be evaluated with respect to a state without referring to a designated computation path. This restriction on modal operators reduces the class of properties expressible in CTL, but it also makes CTL much easier to handle computationally.

8.4.4 Relations between CTL, LTL and CTL*

Every LTL formula and every CTL formulae has an equivalent CTL* formula, but LTL and CTL are not comparable in expressivity, that is, there are formulae in both of these logics that express properties of computation trees not expressible in the other logic.

For the LTL formula $F(p \wedge Xp)$ (interpreted universally with implicit prefix A) there is no equivalent CTL formula.

For the CTL formula $AG(EFp)$ there is no equivalent LTL formula.

The conjunction $AF(p \wedge Xp) \wedge AG(EFp)$ of the above two formulae is a CTL* formula that has no counterpart in LTL nor in CTL. Hence CTL* is strictly more expressive than either of the these two logics.

8.5 Dynamic Logic

Both Floyd and Hoare [Flo67, Hoa69] have used correctness assertions

$$\{\varphi\}\alpha\{\psi\}$$

for stating that executing the program α in a state satisfying φ will always end in a state satisfying ψ . There are proof systems that allow the stepwise derivation of such assertions about programs α from similar assertions on components of α .

The possible steps a program can take can be understood as a Kripke structure, and this idea leads to the definition of dynamic logics [Pra76, FL77, FL79], in which we have an infinite number of modal operators $[\alpha]$ for all possible programs α , and the above correctness assertion corresponds to the dynamic logic formula

$$\varphi \rightarrow [\alpha]\psi.$$

8.5.1 Propositional dynamic logic PDL

The language of the propositional dynamic logic is based on a set X of propositional variables and a set A of atomic programs.

Then formulas and programs based on A and X are defined recursively as follows.

1. The constant true \top is a formula.
2. Every propositional variable $x \in X$ is a formula.
3. If φ and ψ are formulae, then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \leftrightarrow \psi$ and $\varphi \rightarrow \psi$.
4. Every atomic program $a \in A$ is a program.
5. If α and β are programs and φ is a formula, then $\alpha \cup \beta$, $\alpha; \beta$, α^* and $\varphi?$ are programs.
6. If α is a program and φ is a formula, then $[\alpha]\varphi$ and $\langle\alpha\rangle\varphi$ are formulae.

The intuitive meaning of $\alpha \cup \beta$ is nondeterministic choice: execute either α or β . The construct $\alpha; \beta$ is sequential composition: first execute α and then execute β . The α^* programs correspond to looping: execute α zero or any finite number of times. Conditional execution is expressed by $\varphi?$: the execution continues only if φ is true.

When considering ordinary programming languages, the atomic programs would most naturally be assignment statements that take us from one possible world to another in which one or more propositional variables have a different truth-value. Of course, the definition of PDL allows more general types of atomic programs, for example ones that are nondeterministic and therefore do not define exactly one unique successor for each possible world.

Familiar constructs from procedural programming languages can be defined in terms of the PDL program constructs as follows.

$$\begin{aligned} \text{if } \varphi \text{ then } \alpha \text{ else } \beta &\equiv (\varphi?; \alpha) \cup (\neg\varphi?; \beta) \\ \text{while } \varphi \text{ do } \alpha &\equiv (\varphi?; \alpha)^*; \neg\varphi? \\ \text{repeat } \alpha \text{ until } \varphi &\equiv \alpha; (\neg\varphi?; \alpha)^*; \varphi? \end{aligned}$$

Definition 8.17 A model in the propositional dynamic logic is a tuple $M = \langle W, R_{a_1}, \dots, R_{a_n}, A, X, v \rangle$ where

- W is a set of possible worlds,
- A is a set of atomic programs,
- $R_a \subseteq W \times W$ for every $a \in A$ is an accessibility relation,
- X is the set of propositional variables, and
- $v : W \times X \rightarrow \{0, 1\}$ is a valuation that assigns truth values to propositional variables in every possible world.

The truth-definition of modal formulae in possible worlds $w \in W$ is as follows (the truth-definition of non-modal connectives is just like earlier.)

$$M \models_w [\alpha]\varphi \text{ iff } M \models_{w'} \varphi \text{ for every possible world } w' \text{ such that } wR_\alpha w'.$$

Here the difference to earlier modal logics is that there is an infinite number of different accessibility relations for all of the non-atomic programs α . The accessibility relations R_α for these programs are defined as follows.

1. The accessibility relations of atomic programs $a \in A$ are given explicitly by the model M .
2. If α and β have respectively accessibility relations R_α and R_β , then
 - (a) $R_{\alpha \cup \beta} = R_\alpha \cup R_\beta$ (union of relations),
 - (b) $R_{\alpha; \beta} = \{\langle w, w' \rangle \mid u \in W, wR_\alpha u, uR_\beta w'\}$ (composition of relations),
 - (c) $R_{\alpha^*} = (R_\alpha)^*$ (the reflexive transitive closure of a relation), and
 - (d) $R_{\varphi?} = \{\langle w, w \rangle \mid w \in W, M \models_w \varphi\}$

8.5.2 The semantics of PDL

For a given set X of propositional variables and set $A = \{a_1, \dots, a_n\}$ of atomic programs, a model in the propositional dynamic logic is a tuple $M = \langle W, R_{a_1}, \dots, R_{a_n}, X, v \rangle$ where W is a set of possible worlds and R_a for every $a \in A$ is the accessibility relation for atomic program a , and $v : W \times X \rightarrow \{0, 1\}$ is a valuation that assigns truth values to propositional variables in every possible world.

Analogously to modal logics defined earlier we can talk about the frame $F = \langle W, R_{a_1}, \dots, R_{a_n} \rangle$.

The truth-definition of modal formulae in possible worlds $s \in W$ is as follows (the truth-definition of non-modal connectives is just like earlier.)

$M \models_w [\alpha]\varphi$ iff $M \models'_w \varphi$ for every possible world w' such that $wR_\alpha w'$.

Here the difference to earlier modal logics is that there is an infinite number of different accessibility relations for all of the non-atomic programs α . The accessibility relations for these programs are defined as follows.

1. The accessibility relations of atomic programs $a \in A$ are given explicitly by the model M .
2. If α and β have respectively accessibility relations R_α and R_β , then
 - (a) $R_{\alpha\cup\beta} = R_\alpha \cup R_\beta$ (union of relations),
 - (b) $R_{\alpha;\beta} = \{\langle w, w' \rangle \mid u \in W, wR_\alpha u, uR_\beta w'\}$ (composition of relations),
 - (c) $R_{\alpha^*} = (R_\alpha)^*$ (the reflexive transitive closure of a relation), and
 - (d) $R_{\varphi?} = \{\langle w, w \rangle \mid w \in W, M \models_w \varphi\}$

8.5.3 Axiom system for PDL

The propositional dynamic logic includes the propositional logic, and hence its set of theorems includes all the theorems of the classical propositional logic and its (modal) substitution instances, and it is closed under substitution and modus ponens.

To formalize PDL axiomatically, one additionally needs axioms for describing the properties of the modal operators.

Definition 8.18 (Propositional dynamic logic) *PDL is the smallest logic (according to Definition ??) that is closed under the necessitation/generalization rule*

If $\vdash_{PDL} \varphi$, then $\vdash_{PDL} [\alpha]\varphi$ for any program α .

and contains the following axiom schemata for programs α and β and formulae/propositions p and q .

1. $[\alpha](p \rightarrow q) \rightarrow ([\alpha]p \rightarrow [\alpha]q)$
2. $[\alpha; \beta]p \leftrightarrow [\alpha][\beta]p$
3. $[\alpha \cup \beta]p \leftrightarrow [\alpha]p \wedge [\beta]p$
4. $[\alpha^*]p \leftrightarrow p \wedge [\alpha][\alpha^*]p$
5. $[\alpha^*](p \rightarrow [\alpha]p) \rightarrow (p \rightarrow [\alpha^*]p)$
6. $[q?]p \leftrightarrow (q \rightarrow p)$

8.5.4 Relations to other modal logics and Hoare logic

Relation to LTL

Variants of the temporal operators F , X , G and U can be translated into the propositional dynamic logic. Essentially, temporal logics can be understood as dynamic logics that do not make the different programs explicit, or alternatively, have only one atomic program a . The following is an embedding of LTL operators in PDL.

operator	translation into PDL
$F\varphi$	$\langle a^* \rangle \varphi$
$X\varphi$	$\langle a \rangle \varphi$
$G\varphi$	$[\alpha^*]\varphi$
$\varphi U \psi$	$\langle (\varphi?; a)^* \rangle \psi$

Relation to unimodal logics

There are simple embeddings of some the unimodal logics with operators \Box and \Diamond in the dynamic logic. The interesting thing here is that what is the properties of the relations in these uni-modal logics, are represented in the PDL modal operator.

logic	translation of $\Box\varphi$ into PDL
K	$[a]\varphi$
S4	$[a*]\varphi$
S5	$[(a \cup a^{-1})*]\varphi$

Here we need the converse connective $^{-1}$ that is given a semantics as follows.

$$R_{\alpha^{-1}} = \{\langle t, s \rangle \mid \langle s, t \rangle \in \alpha\}.$$

The converse program α^{-1} of α computes the computation of α in reverse from the end states back to the starting states.

With the above translation the satisfiability problems in the logics in question can be directly translated into PDL. Notice that the relations in the PDL frames may be arbitrary (K frames), and the reflexivity and transitivity for S4 and reflexivity, transitivity and symmetry for S5 are properties of the PDL modal operator, and need not be properties of the frames in question.

Relation to Hoare logic

The Hoare logic is a programming logic that may be used for proving the correctness of programs in procedural programming languages [Hoa69]. There are different variants of the logic, depending on the definition of programs and the language in which the correctness assertions are expressed. Here we consider a propositional variant of the logic, in which the assertions are propositional formulae.

We can consider atomic programs a that are associated with assertions $\{\varphi\}a\{\psi\}$ explaining their behavior. Assertions about more complex programs can be proved by using these simplest assertions and inference rules concerning the program constructs.

The Hoare logic has the following inference rules.¹

$$\frac{\{\varphi\}\alpha\{\sigma\}, \{\sigma\}\beta\{\psi\}}{\{\varphi\}\alpha; \beta\{\psi\}} \quad (8.1)$$

$$\frac{\{\varphi \wedge \sigma\}\alpha\{\psi\}, \{\neg\varphi \wedge \sigma\}\beta\{\psi\}}{\{\sigma\} \text{ if } \varphi \text{ then } \alpha \text{ else } \beta\{\psi\}} \quad (8.2)$$

$$\frac{\{\varphi \wedge \psi\}\alpha\{\psi\}}{\{\psi\} \text{ while } \varphi \text{ do } \alpha\{\neg\varphi \wedge \psi\}} \quad (8.3)$$

$$\frac{\varphi' \rightarrow \varphi, \{\varphi\}\alpha\{\psi\}, \psi \rightarrow \psi'}{\{\varphi'\}\alpha\{\psi'\}} \quad (8.4)$$

These inference rules are derivable in PDL. We need some auxiliary results for the proofs.

Lemma 8.19 *The following inference rules are valid in PDL.*

$$\frac{\varphi \rightarrow \psi}{[\alpha]\varphi \rightarrow [\alpha]\psi} \quad (8.5)$$

$$\frac{\psi \rightarrow [\alpha]\psi}{\psi \rightarrow [\alpha*]\psi} \quad (8.6)$$

Proof: Inference rule 8.5: See the proof of Lemma ??.

Inference rule 8.6: So assume $\vdash_{PDL'} \psi \rightarrow [\alpha]\psi$.

By the necessitation rule $\vdash_{PDL'} [\alpha*](\psi \rightarrow [\alpha]\psi)$.

By the tautology $\chi \rightarrow (\psi \rightarrow (\psi \wedge \chi))$ and MP we have $\vdash_{PDL'} \psi \rightarrow (\psi \wedge [\alpha*](\psi \rightarrow [\alpha]\psi))$.

¹Additionally, a proof system for reasoning about the formulae in the assertions is needed.

Axiom 8.18.5 is logically equivalent to $(\psi \wedge [\alpha^*](\psi \rightarrow [\alpha]\psi)) \rightarrow [\alpha^*]\psi$.

By the tautology $((\chi_1 \rightarrow \chi_2) \wedge (\chi_2 \rightarrow \chi_3)) \rightarrow (\chi_1 \rightarrow \chi_3)$ and the previous two formulae we finally have $\vdash_{PDL'} \psi \rightarrow [\alpha^*]\psi$. \square

Theorem 8.20 *The inference rule 8.3 is derivable in PDL.*

Proof: So we assume that $\{\varphi \wedge \psi\} \alpha \{\psi\}$ and have to derive $\{\psi\}$ while φ do $\alpha\{\neg\varphi \wedge \psi\}$. In PDL these assertions are

$$(\varphi \wedge \psi) \rightarrow [\alpha]\psi$$

and

$$\psi \rightarrow [(\varphi?; \alpha)^*; \neg\varphi?](\neg\varphi \wedge \psi).$$

So assume that $\vdash_{PDL'} (\varphi \wedge \psi) \rightarrow [\alpha]\psi$.

By tautological equivalence $\vdash_{PDL'} \psi \rightarrow (\varphi \rightarrow [\alpha]\psi)$.

With Axiom 8.18.6 we have $\vdash_{PDL'} \psi \rightarrow [\varphi?][\alpha]\psi$.

With Axiom 8.18.2 we have $\vdash_{PDL'} \psi \rightarrow [\varphi?; \alpha]\psi$.

With inference rule 8.6 we have $\vdash_{PDL'} \psi \rightarrow [(\varphi?; \alpha)^*]\psi$.

With inference rule 8.5, MP, and the propositional tautology $\psi \rightarrow (\neg\varphi \rightarrow (\neg\varphi \wedge \psi))$ we have $\vdash_{PDL'} \psi \rightarrow [(\varphi?; \alpha)^*](\neg\varphi \rightarrow (\neg\varphi \wedge \psi))$.

With Axiom 8.18.6 we have $\vdash_{PDL'} \psi \rightarrow [(\varphi?; \alpha)^*][\neg\varphi?](\neg\varphi \wedge \psi)$.

With Axiom 8.18.2 we have $\vdash_{PDL'} \psi \rightarrow [(\varphi?; \alpha)^*; \neg\varphi?](\neg\varphi \wedge \psi)$. \square

Derivation of the remaining three rules is left as an exercise.

Index

- abstraction, 14
- accessibility relation, 43
- alethic logic, 45
- apply, 18
- atomic formulas, 43, 47
- BDD, 16, 41
- binary decision diagrams, 41
- Boole's expansion, 15
- Boolean Constraint Propagation, 10
- branching time temporal logic, 49
- clause, 12
- CNF, 12
- combinatorial explosion, 35
- complement (of a literal), 9
- conjunctive normal form, 12
- consistency, 6
- CTL, 50
- CTL*, 49
- d-DNNF, 22
- Davis-Putnam procedure, 11
- Davis-Putnam-Logemann-Loveland procedure, 11
- decomposability (formula), 22
- determinism (formula), 22
- disjunctive normal form, 13
- DNF, 13
- DNNF, 22
- doxastic logic, 45
- dynamic logic, 45, 50
- empty clause, 9
- epistemic logic, 45
- event, 30
- existential abstraction, 14, 20, 28
- formula, 3
- frame, 43
- frame axiom, 39
- Hoare logic, 53
- immediate subformula, 3
- inconsistency, 6
- join (relational), 27
- Kripke model, 43, 47
- linear temporal logic, 48
- literal, 12
- logical consequence, 5, 47
- LTL, 48
- modal logic S4, 45, 53
- modal logic S5, 45, 53
- model counting, 15, 19, 22
- model, based on a frame, 43
- natural join, 27
- negation normal form, 13
- NNF, 13
- OBDD, 16, 41
- p-morphism, 46
- path formula, 48
- path quantifier, 49
- PDL, 50
- pre-image operation, 36
- precondition, 31
- projection (relational), 27
- propositional dynamic logic, 50
- resolution, 10
- ROBDD, 16
- S4, 45, 53
- S5, 45, 53
- SAT, 6
- satisfiability, 6
- SDD, 22
- selection (relational), 27
- Sentential decision diagram (SDD), 22
- Shannon expansion, 15
- state, 30
- state formula, 49
- state variable, 30
- subformula, 3
- succinct transition system, 30
- temporal logic, 45
- term, 12
- transition relation, 30
- transition system, 30
- transition system with state variables, 30
- truth table, 5
- unit clause, 9
- unit propagation, 10
- unit resolution, 9
- unit subsumption, 10
- universal abstraction, 14, 20, 28
- validity, 5
- valuation, 3
- weakest precondition, 33

Bibliography

- [AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 399–404. Morgan Kaufmann Publishers, 2009.
- [BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.
- [BCL⁺94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. MacMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.
- [Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.
- [Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
- [Byl94] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.
- [CBM90] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, 1990.
- [CM90] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 126–129. IEEE Computer Society Press, 1990.
- [CMV09] Benjamin Chambers, Panagiotis Manolios, and Daron Vroon. Faster SAT solving with better CNF generation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1590–1595, 2009.
- [Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.
- [Dar01] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.
- [Dar02] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002) and the 14th Conference on Innovative Applications of Artificial Intelligence (IAAI-2002)*, pages 627–634, 2002.
- [Dar11] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 819–826. AAAI Press, 2011.

- [DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.
- [Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
- [DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.
- [DNK97] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In *Recent Advances in AI Planning. Fourth European Conference on Planning (ECP'97)*, number 1348 in Lecture Notes in Computer Science, pages 169–181. Springer-Verlag, 1997.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
- [Eme90] E. Allen Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.
- [ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.
- [FL77] M. J. Fischer and R. E. Ladner. Propositional modal logic of programs. In *Proceedings of the 9th Symposium on the Theory of Computing*, pages 286–294, 1977.
- [FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal for Computer and System Sciences*, 18(2):194–211, 1979.
- [Flo67] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, 1967.
- [GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, San Francisco, 1979.
- [GW83] Hana Galperin and Avi Wigderson. Succinct representations of graphs. *Information and Control*, 56:183–198, 1983. See [Loz88] for a correction.
- [HD05] Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In Leslie Pack Kaelbling, editor, *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 156–162. Professional Book Center, 2005.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [Kri63a] Saul Kripke. Semantical analysis of modal logic i, normal propositional calculi. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.
- [Kri63b] Saul Kripke. Semantical considerations on modal logics. *Acta Philosophica Fennica*, pages 83–94, 1963.
- [Kri65] Saul Kripke. Semantical analysis of modal logic ii. non-normal modal propositional calculi. In J. W. Addison, L. Henkin, and A. Tarski, editors, *The Theory of Models*, pages 206–220. North-Holland, Amsterdam, 1965.
- [KS92] Henry Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley & Sons, 1992.

- [KS96] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press, 1996.
- [LB90] Antonio Lozano and José L. Balcázar. The complexity of graph problems for succinctly represented graphs. In Manfred Nagl, editor, *Graph-Theoretic Concepts in Computer Science, 15th International Workshop, WG'89*, number 411 in Lecture Notes in Computer Science, pages 277–286. Springer-Verlag, 1990.
- [Loz88] Antonio Lozano. NP-hardness of succinct representations of graphs. *Bulletin of the European Association for Theoretical Computer Science*, 35:158–163, June 1988.
- [MMZ⁺01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01)*, pages 530–535. ACM Press, 2001.
- [MSS96] João P. Marques-Silva and Karem A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 220–227, 1996.
- [PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In Joao Marques-Silva and Karem A. Sakallah, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2007)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.
- [Pra76] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.
- [RHN06] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.
- [Rin04] Jussi Rintanen. Evaluation strategies for planning as satisfiability. In *ECAI 2004. Proceedings of the 16th European Conference on Artificial Intelligence*, pages 682–687. IOS Press, 2004.
- [SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 46–51, 1992.
- [SS07] Matthew Streeter and Stephen F. Smith. Using decision procedures efficiently for optimization. In *ICAPS 2007. Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, pages 312–319. AAAI Press, 2007.
- [Tse68] G. S. Tseitin. On the complexity of derivations in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, 1968.