2006

Modal logic in computer science

Leigh Lambert

Modal Logic in Computer Science

Leigh Lambert
Edith Hemaspaandra, Chair
Chris Homan, Reader
Michael Van Wie, Observer

June 25, 2004

1 Abstract

Modal logic is a widely applicable method of reasoning for many areas of computer science. These areas include artificial intelligence, database theory, distributed systems, program verification, and cryptography theory. Modal logic operators contain propositional logic operators, such as conjunction and negation, and operators that can have the following meanings: "it is necessary that," "after a program has terminated," "an agent knows or believes that," "it is always the case that," etc. Computer scientists have examined problems in modal logic, such as satisfiability. Satisfiability determines whether a formula in a given logic is satisfiable. The complexity of satisfiability in modal logic has a wide range. Depending on how a modal logic is restricted, the complexity can be anywhere from NP-complete to highly undecidable. This project gives an introduction to common variations of modal logic in computer science and their complexity results.

2 Introduction

Modal logic is an extension of traditional propositional logic with the addition of extra operators, such as "it is possible that," "it is necessary that," "it will always be the case that," "an agent believes that," etc. In the last fifty years, modal logic has become extremely useful in different areas of computer science, such as artificial intelligence, database theory, distributed systems, program verification, and cryptography theory. For example, propositional dynamic logic is well suited to verifying the correctness of complex and integrated systems. Epistemic logic reasons about knowledge and belief and is useful when discussing the knowledge of individuals in a group. Epistemic logic is useful for distributed systems and artificial intelligence theory. Particular forms of epistemic logic are used for reasoning about zero-knowledge proofs, which is a part of cryptography theory.

What makes modal logic so easily adaptable to many areas of computer science stems from the flexibility of the meanings of modal operators. The basic modal operator, referred to as the "necessary" operator, and its dual, the "possibility" operator, can take on different meanings depending on the application. For example, the following are different meanings for the "necessary" operator: "after all possible computations of a program have terminated," "agent i knows that," "it is always true that," "it ought to be true that," etc. Modal logic semantics is based on possible-worlds semantics. *Possible-worlds* semantics observes different worlds, or states, that are possibly true after a change occurs at one world. For instance, there are many possibilities of what tomorrow might be, but only one of those possibilities is realized. Modal operators need to follow certain rules. For instance, if "always φ" is true, then "always always φ" is true. For processes in a distributed system, we want to make sure that if a process "knows that φ" is true, then φ is actually true. These rules will lead to different variations of modal logic. We refer to each variation as itself a modal logic.

Computer scientists have explored problems, such as satisfiability and validity, in different variations of modal logic. In this paper, we examine complexity results for satisfiability. Satisfiability in a modal logic S, or S-SAT, determines if a formula is satisfiable in S.

This paper is a survey of previous research of modal logic in computer science. It gives an introduction of the following: (1) modal logic, (2) its uses in computer science, and (3) the complexity of satisfiability in variations of modal logic. This paper gives the necessary background in modal logic, its applications in computer science, and complexity theory. It will put the complexity of different modal logics into a general framework.

The rest of the paper is organized as follows. Section 3 discusses the basics of uni-modal logics and other general concepts for each modal logic. Section 4 gives an introduction of the uses of modal logic in computer science. In this section, we present variations of modal logic, including multi-modal logics. Section 5 will discuss the problem of satisfiability of formulas in propositional logic and uni-modal logics. We will examine some of the basics of computational complexity as well. Our last section lists complexity results for the modal logics introduced in Section 4.

3 Uni-Modal Logics

A logic is described by either syntax and semantics or a deductive system. We begin by presenting the syntax and semantics for basic uni-modal logics, along with general information useful to the discussion of other modal logics. We then discuss deductive systems common to uni-modal logics. We will present more complex modal logics in section 4.

3.1 Syntax

Modal logic consists of syntactical and semantical information that describes modal formulas. We begin by presenting the syntax for uni-modal logics. The simplest, most basic modal formulas are propositions. Propositions describe facts about a particular world or state, such as "The sky is cloudy today" or "Max's chair is broken." Let $\Phi = \{p_1, p_2, p_3, \ldots\}$ be a nonempty set of propositions. A modal formula is either an element of Φ or has the form $\varphi \wedge \psi$, $\neg\varphi$, or $\Box\varphi$ where φ and ψ are modal formulas. Intuitively, the formula $\Box\varphi$ means "φ is necessarily true." Let $L(\Phi)$ be the smallest set of formulas containing Φ that is closed under conjunction, negation, and the modal operator $\Box$. Remaining operators can be defined in terms of other operators. We define the following:

- $\varphi \vee \psi \overset{def}{=} \neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \rightarrow \psi \overset{def}{=} \neg(\varphi \wedge \neg\psi)$
- $\mathit{true} \overset{def}{=}$ a propositional tautology (such as $p_1 \vee \neg p_1$)
- $\mathit{false} \overset{def}{=} \neg\mathit{true}$
- $\Diamond\psi \overset{def}{=} \neg\Box\neg\psi$.

The formula $\Diamond\varphi$ means that "φ is possibly true." The order of operations for modal formulas is left-associative by default. Parentheses are used to modify order of operations, as normally seen in algebra. Unary operators have more precedence than binary operators. The operators $\Diamond$, $\vee$, $\rightarrow$, and $\leftrightarrow$ and the assignments *true* and *false* do not normally appear in modal formulas in this report, due to the ease of working with as few operators as possible in later sections of this paper.

Other important definitions about formulas are as follows. The size of a modal formula φ, abbreviated as size(φ), is the number of symbols in φ. Each symbol is a member of the set $\Phi \cup \{\wedge, \neg, \Box, (,)\}$. The encoding of a formula φ is φ with each proposition replaced with its binary encoding. For example, the elements $p_1, p_2, p_3, \ldots$ would be replaced with $p1, p10, p11, \ldots$, respectively. The length of φ, written as $|\varphi|$, is the length of the encoding of φ. Thus, $|\varphi| \geq \text{size}(\varphi)$. Also, $|\varphi|$ can be approximately $\text{size}(\varphi) \times \log_2(\text{size}(\varphi))$, depending on the propositions used in φ.

Let formula φ be of the form $\neg\varphi'$ (respectively, $\Box\varphi', \varphi' \wedge \varphi''$). We say that formula ψ is a *subformula* of φ if ψ is φ or ψ is a subformula of φ' (respectively, φ', φ' or φ''). Subformulas are defined inductively. Thus, each proposition contained in φ is a subformula of φ. Define $\text{sub}(\varphi)$ to be the set of all subformulas of φ. We denote the size of a set A, or the number of elements in A, to be written as $||A||$. An important property of $\text{sub}(\varphi)$ is $||\text{sub}(\varphi)|| \leq \text{size}(\varphi)$. This property can be proven by induction on the size of φ. In addition, the *modal depth* of a formula φ is the maximum number of nested modal operators in φ. For example, the modal depth of the formula $\Box(\Box\varphi \wedge \Box\neg\Box\varphi')$ is at least 3 because φ' is nested in three modal operators.

3.2 Semantics

Now that we have described the syntax of uni-modal logics, we introduce the semantics, which determines whether a given formula is true or false. Modal semantics is formally defined using Kripke structures, which is similar to possible-worlds semantics. A Kripke structure is a tuple $M = (W, R, V)$ where W is a set, R is a binary relation on W, and V maps $\Phi \times W$ to $\{\textit{true, false}\}$ [22]. The set W is a set of 'possible worlds,' or states, and R determines which states are accessible from any given state in W. We say that state $b \in W$ is *accessible* from state $a \in W$ if and only if $(a, b) \in R$. R is known as the *accessibility* relation. The function V determines which facts, or propositions, are true at each of the worlds.

Each modal structure is depicted using a directed graph. Each node is named after its corresponding state in W. An edge from node w to node w' means that $(w, w') \in R$. At each node, there are labels indicating which propositions are true at the particular state. Since Φ is usually infinite, we only use labels for propositions that are needed (i.e., propositions in formulas we are working with).

Example For example, suppose $M = (W, R, V)$ where $W = \{u, v, w\}$. Let $V(p_2, u) = V(p_1, v) = V(p_1, w) = \textit{true}$. Also, let $R = \{(u, v), (u, w), (v, v), (v, w)\}$. The following graph captures this situation.

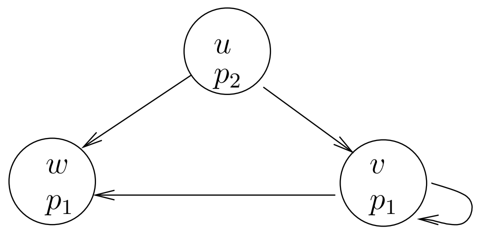


Figure 1.1

We now describe the process of how to determine whether a formula is true at a given state in a structure. Truth depends on both the state and the structure. Formulas can be true in one state and false in another, even within the same structure. We will define the notation $(M, w) \models \varphi$ as meaning "φ is true at state w in structure M." We use induction on the structure of φ to determine the $\models$ relation. V gives the information needed for the base case, in which φ is a proposition:

$$(M, w) \models p \text{ (for a proposition } p \in \Phi) \text{ iff } V(p, w) = \textit{true}.$$

Induction on conjunction and negation is similar to how truth is determined in propositional logic. The formula $\varphi \wedge \psi$ is true if and only if φ is true and ψ is true, and $\neg\varphi$ is true if and only if φ is false:

$$(M, w) \models \varphi \wedge \psi \text{ iff } (M, w) \models \varphi \text{ and } (M, w) \models \psi.$$

$$(M, w) \models \neg\varphi \text{ iff } (M, w) \not\models \varphi.$$

The next rule separates modal logic from propositional logic. The formula φ is necessarily true at state w of structure M if and only if φ is true at all states accessible from w. Written formally, we have

$$(M, w) \models \Box\varphi \text{ iff } (M, w') \models \varphi \text{ for all } w' \text{ such that } (w, w') \in R.$$

So, the semantics for $\Diamond$ is a combination of semantics for $\neg$ and $\Box$, due to the definition of $\Diamond$.

$$(M, w) \models \Diamond\varphi \text{ iff } (M, w') \models \varphi \text{ for some } w' \in W \text{ such that } (w, w') \in R.$$

Example We refer the reader back to Figure 1.1 for the following examples. Note that we have $(M, w) \models p_1$ and $(M, v) \models p_1$. Thus, $(M, u) \models \Box p_1$ and $(M, v) \models \Box p_1$, since p_1 holds in all states accessible from u and v. So, $(M, u) \models \Diamond\Box\ p_1$ due to $(u, v) \in R$. Furthermore, $(M, w) \models \neg p_2$ and $(M, v) \models \neg p_2$. It follows that $(M, v) \models \Box\neg p_2$. Also, $(M, w) \models \Box\neg p_2$ because there is no state accessible from w. Therefore, $(M, u) \models \Box\Box\neg p_2$ since $\Box\neg p_2$ holds in all states accessible from u. Therefore, we can say that $(M, u) \models \Diamond\Box p_1 \wedge \Box\Box\neg p_2 \wedge \Box p_1$. These are just a few of the formulas that can be developed from this structure.

Formally, we say that φ is valid in a modal structure M, written $M \models \varphi$, if for every $w \in W$, $(M, w) \models \varphi$. Formula φ is satisfiable in M if and only if $(M, w) \models \varphi$ at some state $w \in W$. We say that φ is valid in a set of structures $\mathcal{M}$, written $\mathcal{M} \models \varphi$ iff φ is valid in all structures in $\mathcal{M}$, and φ is satisfiable in a set of structures $\mathcal{M}$ iff φ is satisfiable in some structure in $\mathcal{M}$. Formula φ is valid in a structure M, a set of structures $\mathcal{M}$, iff $\neg\varphi$ is not satisfiable in M, $\mathcal{M}$, respectively. The following theorem gives us basic properties about necessity.

Theorem 3.1 *[22] For all formulas $\varphi, \psi \in L(\Phi)$, structures $M \in \mathcal{M}$,*

1. *if φ is a propositional tautology, then $M \models \varphi$,*
2. *if $M \models \varphi$ and $M \models \varphi \rightarrow \psi$, then $M \models \psi$,*
3. *$M \models (\Box\varphi \wedge \Box(\varphi \rightarrow \psi)) \rightarrow \Box\psi$,*
4. *if $M \models \varphi$, then $M \models \Box\varphi$.*

An accessible proof of Theorem 3.1 can be found in [15].

3.3 Axiomatizations

This section introduces deductive systems for uni-modal logics. We define a logic as a collection of axioms and rules of inference. For the basic uni-modal logic K, the axioms and rules of inference are based on Theorem 3.1. All other uni-modal logics we will look at are extensions of K.

A1 All instances of tautologies of propositional calculus are permitted.

A2 $(\Box\varphi \wedge \Box(\varphi \rightarrow \psi)) \rightarrow \Box\psi$.

R1 If $\vdash \varphi$ and $\vdash \varphi \rightarrow \psi$, then $\vdash \psi$ (Modus ponens).

R2 If $\vdash \varphi$, then $\vdash \Box\varphi$ (Generalization).

Formula φ is said to be *K-provable*, denoted $K \vdash \varphi$, if φ can be inferred from instances of A1 and A2, using R1 and R2. Inference is formally defined by the rules of inference that are true in K. Some well-known axioms are listed below.

A3 $\Box\varphi \rightarrow \varphi$ ("if φ is necessarily true, then φ is true").

A4 $\Box\varphi \to \Box\Box\varphi$ ("if φ is necessarily true, then it is necessary that φ is necessarily true").

A5 $\neg\Box\varphi \to \Box\neg\Box\varphi$ ("if φ is not necessarily true, then it is necessary that φ is not necessarily true").

A6 $\neg\Box(\mathit{false})$ ("inconsistent facts are not necessarily true").

Listed below are a few common modal logics along with axioms that are always true in each. Each modal logic below contains the axioms and rules of reference that make up K. In addition,

- T only contains A3,
- $S4$ only contains A3 and A4,
- $S5$ only contains A3, A4, and A5,
- $KD45$ only contains A3, A4, A5, and A6.

A modal logic S is *sound* with respect to a set of modal structures $\mathcal{M}$ if every formula that is provable in S is valid in $\mathcal{M}$. S is *complete* with respect to $\mathcal{M}$ if every formula that is valid in $\mathcal{M}$ is provable in S. A modal logic characterizes a set of structures if and only if it is sound and complete with respect to that set. Thus, S characterizes $\mathcal{M}$ if, for all modal formulas φ, $S \vdash \varphi \iff \mathcal{M} \models \varphi$.

Let A be a binary relation on a set S. We say that A is *Euclidean* if, for all $s, t, u \in S, (s,t) \in A$, then $(t,u) \in A$.

Definition If M is a modal structure, then it is a K-model. If M is a modal structure and R is reflexive (respectively; reflexive and transitive; reflexive, transitive, and symmetric; Euclidean and transitive) then M is a T-model (respectively, $S4$-model, $S5$-model, $KD45$-model).

Theorem 3.2 *[21] For $X \in \{K, T, S4, S5, KD45\}$, X is sound and complete with respect to the set of all X-models. A modal formula φ is:*

- *satisfiable in an X-model if and only if $\neg\varphi$ is not X-provable.*
- *valid in an X-model if and only if φ is X-provable.*

To determine if φ is X-provable, φ is inferred from instances of the axioms contained in X, using R1 and R2.

The following proposition addresses properties of $S5$ and $KD45$. It is useful for later sections of this paper.

Proposition 3.3 *[6]*

- *If φ is satisfiable in $S5$, then φ is satisfiable in an $S5$-model $M = (W, R, V)$ where R is universal (i.e., $(w, w') \in R$ for all $w, w' \in W$).*
- *If φ is satisfiable in $KD45$, then φ is satisfiable in a $KD45$-model $M = (\{w_0\} \cup W, R, V)$ where W is nonempty and $(w, w') \in R$ for all $w \in \{w_0\} \cup W, w' \in W$.*

4 Uses of Modal Logics in Computer Science

In this section, we will discuss variations of modal logic, how these logics compare to uni-modal logics, and applications of these logics in computer science. The logics we present in this section may have more than one modal operator, as opposed to the uni-modal logics presented in Section 3. They are useful in such areas as artificial intelligence (AI), distributed systems theory, database theory, program verification, and cryptography theory.

4.1 Epistemic Logic

Epistemic logic is the modal logic that reasons about knowledge and/or belief. Some areas of study that this logic is a particular concern to is philosophy, artificial intelligence (AI), and distributed systems. Other areas where epistemic logic has some use in is cryptography and database theory [8, 33]. We will mainly focus on this logic's impact in AI and distributed systems. However, we give descriptions of an application in cryptography in Section 4.2.1. Epistemic logic is useful when designing complicated AI or distributed systems, allowing designers to formalize details of systems. We will first present the syntax, semantics, and deductive systems for epistemic logic.

4.1.1 Syntax

Let n be a finite number of agents. Recall that $\Phi = \{p_1, p_2, p_3, \ldots\}$ is a nonempty set of propositions. A modal formula is built inductively from Φ; it is either an element of Φ or has the form $\varphi \wedge \psi$, $\neg\varphi$, or $[i]\varphi$, where $1 \leq i \leq n$ and φ and ψ are modal formulas. The formula $[i]\varphi$ intuitively means that "agent i knows or believes φ." Let $L_n(\Phi)$ be defined as the smallest set of formulas containing Φ that is closed under conjunction, negation, and the modal operator $[i], 1 \leq i \leq n$. We define $<i>\varphi$ to be $\neg[i]\neg\varphi$. The dual formula $<i>\varphi$ means that "agent i knows or believes that φ is true at some state."

In addition to reasoning about what each agent in a group knows, it may be helpful, depending on the application, to reason about the knowledge common to all agents. Common knowledge describes those facts that everyone in the group knows that everyone knows that ... everyone knows to be true. This allows us to formalize facts about a group's "culture." The language that agents use to communicate is an example of common knowledge. We define the following:

- $E\varphi$, which intuitively means "everyone knows φ," as $[1]\varphi \wedge [2]\varphi \wedge \ldots \wedge [n]\varphi$;
- $C\varphi$, which intuitively means "φ is common knowledge," as $E\varphi \wedge EE\varphi \wedge \ldots$.

Let $L_n^C(\Phi)$ be the extension of $L_n(\Phi)$ that is closed under conjunction, negation, the n modal operators, E, and C.

Reasoning about distributed knowledge is also useful in some applications. Distributed knowledge describes the facts than can be gathered from combining the knowledge of all agents in a group. For example, if agent i knows φ and agent j knows $\varphi \rightarrow \psi$, then ψ is considered distributed knowledge[15]. Let D be the distributed knowledge operator. If φ is a formula, then so is $D\varphi$. Let $L_n^D(\Phi)$ be an extension of $L_n(\Phi)$ that is closed under conjunction, negation, the n modal operators, and D. Distributed knowledge is categorized as knowledge that a 'wise agent' would know, if there was such an agent in the group.

4.1.2 Semantics

The semantics for epistemic logic is based on the semantics introduced in Section 3. We redefine a modal structure as $M = (W, R_1, R_2, \ldots, R_n, V)$. W is a set of states, $R_i, 1 \leq i \leq n$, is a relation that determines which states agent i believes to be accessible from any state in W, and V maps $\Phi \times W$ to $\{\mathit{true}, \mathit{false}\}$.

Our definition for $\models$ comes from Section 3, with slight changes in the modal case.

$$(M, w) \models p \text{ (for a proposition } p \in \Phi) \text{ iff } V(p, w) = \mathit{true}.$$

$$(M, w) \models \varphi \wedge \psi \text{ iff } (M, w) \models \varphi \text{ and } (M, w) \models \psi.$$

$$(M, w) \models \neg\varphi \text{ iff } (M, w) \not\models \varphi.$$

$$(M, w) \models [i]\varphi \text{ iff } (M, w') \models \varphi \text{ for all } w' \text{ such that } (w, w') \in R_i, 1 \leq i \leq n.$$

Next, we describe the semantics for common and distributed knowledge. The following notation is helpful for understanding the definition of C. Let $E^1\varphi = E\varphi$, and $E^{k+1} = E(E^k\varphi)$ for $k \geq 1$. We add to the definition of $\models$.

$$(M, w) \models E\varphi \text{ iff } (M, w) \models [i]\varphi \text{ for } 1 \leq i \leq n.$$

$$(M, w) \models C\varphi \text{ iff } (M, w) \models \bigwedge_{k=1}^{\infty} E^k\varphi.$$

The semantics for E basically says that $E\varphi$ is true at a particular state $w \in W$ if and only if all agents believe φ to be true at w. The semantics for C guarantees that all agents know that φ is true at every state w_1 accessible from w, and every state w_2 accessible from w_1, and so on.

For the distributed knowledge operator, we can add to the definition of $\models$ the following statement:

$$(M, w) \models D\varphi \text{ iff } (M, w') \models \varphi \text{ for all } w' \text{ such that } (w, w') \in R_1 \cap R_2 \cap \ldots \cap R_n.$$

This definition states that the relation containing the intersection of all the accessibility relations is the relation that results when combining the knowledge of all agents. This new relation describes a new, 'wise agent' that knows everything that each of the agents know.

4.2 Axiomatizations

The following uni-modal logics, $K, T, S4, S5, KD45$, correspond to the following modal logics $K_n, T_n, S4_n, S5_n, KD45_n$, respectively. The axioms and rules of inference are the same, except for changes due to modal operators.

A1 All instances of tautologies of propositional calculus are permitted.

A2 $([i]\varphi \wedge [i](\varphi \rightarrow \psi)) \rightarrow [i]\psi$, where $1 \leq i \leq n$.

R1 If $\vdash \varphi$ and $\vdash \varphi \rightarrow \psi$, then $\vdash \psi$ (Modus ponens).

R2 If $\vdash \varphi$, then $\vdash [i]\varphi$, where $1 \leq i \leq n$ (Generalization).

A3 $[i]\varphi \rightarrow \varphi$, where $1 \leq i \leq n$;

A4 $[i]\varphi \rightarrow [i][i]\varphi$, where $1 \leq i \leq n$;

A5 $\neg[i]\varphi \rightarrow [i]\neg[i]\varphi$, where $1 \leq i \leq n$;

A6 $\neg[i](\mathit{false})$, where $1 \leq i \leq n$.

Sato proved that K_n is sound and complete with respect to the set of all modal structures [35]. Furthermore, T_n (respectively, $S4_n, S5_n, KD45_n$) is sound and complete with respect to the set of modal structures containing accessibility relations that are reflexive (respectively; reflexive and transitive; reflexive, symmetric, and transitive; transitive and Euclidean) [35].

The following are axioms and a rule of reference that characterize axiomatizations for common knowledge.

A7 $E\varphi \equiv [1]\varphi \wedge \ldots [n]\varphi$ ("everyone knows φ if and only if each agent knows φ").

A8 $C\varphi \rightarrow E(\varphi \wedge C\varphi)$ intuitively means that ("if everyone knows that φ is true and that φ is common knowledge, then φ is common knowledge").

R3 From $\vdash \varphi \rightarrow E(\psi \wedge \varphi)$ infer $\vdash \varphi \rightarrow C\psi$.

Let K_n^C (respectively, $T_n^C, S4_n^C, S5_n^C$) be the modal logic that results from adding A7, A8, and R1 to the axioms for K_n (respectively, $T_n, S4_n, S5_n$). Milgrom proved that the logic K_n^C is sound and complete with respect to the set of all modal structures [27]. In addition, T_n^C (respectively, $S4_n^C, S5_n^C$) is sound and complete with respect to the set of modal structures containing accessibility relations that are reflexive (respectively; reflexive and transitive; reflexive, symetric, and transitive) [27].

The axiom that characterizes distributed knowledge is the following:

A9 $[i]\varphi \rightarrow D\varphi, 1 \leq i \leq n$ ("if an agent knows φ is true, then φ is distributed knowledge").

Let K_n^D (respectively, $T_n^D, S4_n^D, S5_n^D$) be the modal logic that results from adding A9 to the axioms for K_n (respectively, $T_n, S4_n, S5_n$). Since the operator D is in essence a knowledge operator, we expect axioms such as A2, A3, A4, and A5 and rule of reference R2 to hold true for D as well as the operator $[i], 1 \leq i \leq n$. For example, the statement $(D\varphi \wedge D(\varphi \rightarrow \psi)) \rightarrow D\psi$ must be true for A2 to be true in the modal logics for distributed knowledge. For $n \geq 2$ agents, K_n^D is sound and complete with respect to the set of all modal structures [10]. Furthermore, for $n \geq 2$ agents, T_n^D (respectively, $S4_n^D, S5_n^D$) is sound and complete to the set of modal structures containing accessibility relations that are reflexive (respectively; reflexive and transitive; reflexive, symmetric, and transitive) [10].

4.2.1 Applications of Epistemic Logic

We now present some applications of epistemic logic in areas of computer science. One of the most widely-used application of epistemic logic is to capture knowledge in a multi-agent system. Some examples of multi-agent systems are processes over a computer network, a simulation of persons playing a game, like scrabble, or object sensors on vehicles. We will give a desciption of this system and how it relates to epistemic logic. This system was first introduced by [13].

In a multi-agent system, there are n agents, and each agent i has a *local environment.* An agent's local environment consists of information of what i's local state is in the system. For example, in a scrabble game, agent i's local state consists of the following:

- the letters i contained in its hand,
- the letters that have been currently played,
- which words were played by which players,
- the current score.

In a distributed system, the local environment of process i might contain messages i has sent or received, the values of local variables, the clock time, etc.

In addition, there is a *global environment* that includes information that agents might not necessarily know but is still important for the system to run. This information is usually categorized as seen from a "bird's eye" view of the system. In a scrabble game, the global environment might include the letters that have not been chosen by any player, i.e., those contained in the bag. A *global state* is viewed as a tuple $(s_e, s_1, \ldots, s_n)$ of environments, where s_e is the global environment and $s_1, \ldots, s_n$ are local states for agents $1 \ldots n$.

A *run* is defined as a function from time to global states; it is a description of what happens to global states over time in one possible run of the system. A *point* is a pair (r, m), where r is a run at some time m. We take time to be the natural numbers; this makes time infinite and discrete. At point (r, m), the system is in some global state $r(m)$. Let $r_i(m)$ be the local environment for agent i.

A *system* is defined as a set of runs. Thus, our description of a system entails a collection of interacting agents. For example, a system of a scrabble game consists of all possible word combinations and sequences of plays. A system can be viewed in terms of a modal structure with the exception of V. The set of states, W, would be associated with a set of points. The accessibility relation, R_i, where $1 \leq i \leq n$, corresponds to the relation for agent i. This is determined by $((r, m), (r', m')) \in R_i$ if $r_i(m) = r'_i(m')$. This means that agent i considers (r', m') possible at point (r, m) if i has the same local environment at both points.

Let Φ be a set of basic propositions. These propositions describe facts about the system. For example, in a distributed system, some facts might be "the system is deadlocked," "the value of variable x is 3," "process 2 receives message m in round 6 of this run," etc. An *interpreted system* is a tuple (S, V), where S is a system and V is a function that maps propositions in Φ, depending on the global state, to truth values. In other words, $V(p, s) \in \{\textit{true, false}\}$, where $p \in \Phi$ and s is a global state.

We associate $I = (S, V)$ with a modal structure $M = (W, R_1, \ldots, R_n, V)$, using the definitions we already presented for $W, R_1, \ldots, R_n$ and V. Thus, agents' knowledge is determined by their local environments. Now, we define what it means for a formula φ to be true at a point (r, m) in an interpreted system I, written $(I, r, m) \models \varphi$, by applying our earlier definitions:

$$(I, r, m) \models \varphi \text{ iff } (M, (r, m)) \models \varphi.$$

Some properties of interpreted systems are the following:

- A system I that displays unbounded message delays is one where an agent j in I does not know if or when other agents will receive a message j sends.
- It can never be common knowledge that a message was delivered in systems that display unbounded message delays [12].
- So, coordination can never be acheived in these systems [45].

These have been very important insights and fallbacks to the design of distributed protocols.

Although multi-agent systems have been a common application, epistemic logic has been used with other areas of computer science as well. For example, there have been connections with epistemic logic and zero-knowledge proofs, which are important to cryptography theory [8]. Complex forms of epistemic logic, containing some probability theory in axioms, has been useful in reasoning about zero-knowledge proofs. Zero-knowledge proofs fall under the category of cryptography theory. Knowledge-based programming is a type of programming language where, based on what an agent knows, explicit tests are performed that can decide what the agent's actions are. Halpern and Fagin provide a formal semantics for knowledge-based protocols, but a complete programming language has not yet been produced [13].

These are only a few areas of computer science where epistemic logic is used. There are still areas of study where the use of epistemic logic is unrealized. Next, we will look at dynamic logic and applications where it is useful.

4.3 Propositional Dynamic Logic

Program verification ensures that a program is correct, meaning that any possible input/output combination is expected based on the purpose and specifications of the program. Program verification is a major concern when developing complex and integrated programs [30, 11]. The more complicated a system becomes, the harder it is to verify its correctness. A logic system, called dynamic logic, was developed to verify programs using formal mathematics. We will be looking at propositional dynamic logic (PDL), which is a simplified version of dynamic logic and heavily based on modal logic.

PDL introduces programs to propositional logic as modal operators. Let $[\alpha]$ be a modal operator where α is a program and φ be a fact about the program's state. The formula $[\alpha]\varphi$ means that "if α terminates, then φ holds." The fundamental meaning behind $[\alpha]$ is similar to $\Box$. The main difference between the two operators is the program name in the dynamic operator. This allows us to verify more than one program at once, which is useful for concurrency and interrelated programs, making dynamic logic multi-modal. It comes as no surprise, then, that the dual formula $<\alpha>\varphi$ relates to $\Diamond$ and means that "there is an execution of α that terminates with φ as true." Thus, $<\alpha>\varphi \overset{def}{=} \neg[\alpha]\neg\varphi$.

4.3.1 Syntax

We will extend $L(\Phi)$ introduced in Section 3. Let $\Phi = \{p_1, p_2, p_3 \ldots\}$ be a nonempty set of propositions. An 'atomic' program is a smallest basic program, meaning it does not consist of other programs. Let $\Pi = \{a_1, a_2, a_3, \ldots\}$ be a nonempty set of atomic programs. Formulas are built inductively from Φ and Π using the following operators:

- if $p \in \Phi$, then p is a formula,
- if $a \in \Pi$, then a is a program,

- if φ and ψ are formulas, then $\varphi \wedge \psi$ and $\neg\varphi$ are formulas,
- if φ is a formula and α is a program, then $[\alpha]\varphi$ is a formula,
- if α and β are programs, then $\alpha;\beta$ (sequential composition), $\alpha \cup \beta$ (nondeterministic choice), and α^* (iteration) are programs,
- if φ is a formula, then $\varphi?$ (test) is a program.

The program $\alpha;\beta$ means "do α and then β." The sequential composition operator addresses the need for order in programs. For example, ';' is a common operator in many programming languages used to separate statements. The program $\alpha \cup \beta$ means "do either α or β (nondeterministically)." The program α^* means to "repeat α some finite number of times." The iteration operator is related to loops. The program $\varphi?$ reflects if-then conditions, as it means to "test φ: continue if φ is true, otherwise 'fail'."

Let $L_{PDL}(\Phi)$ be defined as the smallest set of formulas containing Φ that is closed under conjunction, negation, and program necessity. Let $L_{PDL}(\Pi)$ be defined as the smallest set of programs containing Π that is closed under sequential composition, nondeterministic choice, iteration, and test. Note that the definitions of programs and formulas rely on each other. Formulas depend on programs because of $[\alpha]\varphi$, and programs depend on formulas because of $\varphi?$.

Parentheses can be dropped by assigning precedence to operators in the following order: unary operators, the operator ';', and the operator '$\cup$.' Thus, the expression

$$[\varphi?;\alpha^*;\beta]\varphi \wedge \psi$$

should be read as

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

The use of parentheses is utilized for parsing an expression in a particular way or for readability.

We can write some classical programming statements, such as loop constructs, using PDL program operators.

- 'if φ then α else β' $\overset{def}{=} (\varphi?;\alpha) \cup (\neg\varphi?;\beta)$
- 'while φ do α' $\overset{def}{=} (\varphi?;\alpha)^*;\neg\varphi?$
- 'repeat α until φ' $\overset{def}{=} \alpha;(\neg\varphi?;\alpha)^*;\varphi?$

4.3.2 Semantics

The semantics for PDL is based on the semantics presented in Section 3. We redefine a modal structure as $M = (W, \{R_a | a \in \Pi\}, V)$. W is a set of program states, R_a is one or more binary relation(s) that determines which states are accessible from any state in W. V maps $\Phi \times W$ into $\{\textit{true, false}\}$.

Intuitively, we consider $(w, w') \in R_a$ as the case where w is the initial state of program a and w' is an ending state of a. We develop more accessibility relations when combining programs. These newer relations allow us to discuss the input and output states of compound programs easily. The following are the meanings for each of the program operators:

1. $R_{\alpha;\beta} \overset{def}{=} \{(w, w') \mid \exists w'' \text{ such that } wR_\alpha w'' \wedge w''R_\beta w'\}$,
2. $R_{\alpha\cup\beta} \overset{def}{=} R_\alpha \cup R_\beta$,
3. $R_{\alpha^*} \overset{def}{=} \{(u, v) \mid \exists\, u_0, \ldots, u_n \text{ where } n \geq 0, u = u_0 \text{ and } v = u_n \text{ such that } (u_i, u_{i+1}) \in R_\alpha \text{ for } 0 \leq i \leq n\}$.

We give the meaning of $R_{\varphi?}$ after presenting the definiton of $\models$. Our definition for $\models$ is similar to the $\models$ presented in Section 3. Recall that V gives the information for the base case, in which φ is a proposition. The following definitions come straight from Section 3.

$$(M, w) \models p \text{ (for a proposition } p \in \Phi) \text{ iff } V(p, w) = \textit{true}.$$

$$(M, w) \models \varphi \wedge \psi \text{ iff } (M, w) \models \varphi \text{ and } (M, w) \models \psi.$$

$$(M, w) \models \neg\varphi \text{ iff } (M, w) \not\models \varphi.$$

$$(M, w) \models [\alpha]\varphi \text{ iff } (M, w') \models \varphi \text{ for all } w' \text{ such that } (w, w') \in R_\alpha.$$

Note that the last definition is changed to accommodate $[\alpha]$.

Now that we have defined $\models$, we can present the meaning for the test program operator.

$$R_{\varphi?} \overset{def}{=} \{(u, u) \mid (M, u) \models \varphi\}.$$

The loop constructs described earlier inherit their semantics from the above semantics. For example, for the while-do program, we have the following meaning [17]:

$$\{(u, v) \mid \exists\; u_0, \ldots, u_n \text{ where } n \geq 0 \text{ such that } u = u_0, v = u_n, (M, u_0) \models \varphi,$$
$$\text{and } (u_i, u_{i+1}) \in R_\alpha \text{ for } 0 \leq i \leq n, \text{ and } (M, u_n) \not\models \varphi\}.$$

4.3.3 Axiomatizations

The following is a list of rules of inference and axioms for the deductive system of PDL. Let φ and ψ be formulas and α and β be programs.

R1 If $\vdash \varphi$ and $\vdash \varphi \to \psi$, then $\vdash \psi$. (Modus Ponens)

R2 If $\vdash \varphi$, then $\vdash [\alpha]\varphi$. (Generalization)

A1 All instances of tautologies of propositional calculus are permitted.

A2 $([\alpha]\varphi \land [\alpha](\varphi \to \psi)) \to [\alpha]\psi$.

A3 $[\alpha](\varphi \land \psi) \leftrightarrow [\alpha]\varphi \land [\alpha]\psi$.

A4 $[\alpha \cup \beta]\varphi \leftrightarrow [\alpha]\varphi \land [\beta]\varphi$.

A5 $[\alpha; \beta]\varphi \leftrightarrow [\alpha][\beta]\varphi$.

A6 $[\psi?]\varphi \leftrightarrow (\psi \to \varphi)$.

A7 $\varphi \land [\alpha][\alpha^*]\varphi \leftrightarrow [\alpha^*]\varphi$.

A8 $\varphi \land [\alpha^*](\varphi \to [\alpha]\varphi) \to [\alpha^*]\varphi$. (Induction Axiom)

This is a sound and complete axiom system for PDL [37]. The axioms A1, A2, and the rules of inference are taken from the axioms and rules of inference presented in Section 3.2. A8, called the Induction Axiom for PDL, intuitively means "Suppose φ is true. After a number of iterations of α, if φ is true, then after one more iteration of α, φ is still true. Then, φ will be true after any number of iterations of α." For a more in-depth discussion of PDL, look to Unit 2 of [17].

4.3.4 Applications of Propositional Dynamic Logic

One of the major applications of PDL, and dynamic logic, is program verification. Dynamic logic and other programming logics are meant to be useful in producing correct programs. Readers can see from the previous descriptions of PDL how it applies to program verification.

What does it mean for a program to be correct? A program is like a recipe of how to solve a problem. For verification tools to work, the program must have a formal specification that clearly delineates what it means for it to be correct. A *correctness specification* is a formal description of how a program is to behave. A program is correct if its output meets the correctness specification. Correctness specification is very important, especially with large programs. Programmers tend to redefine problems so that they will know exactly what they are supposed to build. In formulating specifications, many unforeseen cases may arise, which is very useful for error handling.

PDL, and hence dynamic logic, is not well-suited to reasoning about program behavior at intermediary states. Other logics that do so are process logic and temporal logic. PDL is better suited to reasoning about program behavior with respect to only input and output states. For example, the accessibility relation for a program α only contains information about an input and an output state, i.e., $(w, w') \in R_\alpha$ means that w' is an output state when program α is run with initial state w. A reasonable restriction for dynamic logic is to only consider programs that halt. There are some programs that are not meant to halt, like operating systems, and dynamic logic should not be used to reason about them.

For programs meant to halt, correctness specifications are usually in the form of input and output specifications. For instance, formal documentation of programs usually consists of detailed descriptions of input and output specifications.

Dynamic logic is then used to reason about a program or a collection of programs. PDL is used to reason about regular programs. Regular programs are defined as follows:

- any atomic program is a regular program,
- if φ is a test, then $\varphi?$ is a regular program,
- if α and β are regular programs, then $\alpha;\beta$ is a regular program,
- if α and β are regular programs, then $\alpha \cup \beta$ is a regular program,
- if α is a regular program, then α^* is a regular program.

It is no surprise that the set of regular programs is equivalent to $L_{PDL}(\Pi)$. Dynamic logic follows the modal logic concepts of PDL with additional predicate logic concepts. This allows dynamic logic to reason about a wider range of programs than $L_{PDL}(\Pi)$.

After reasoning about a program, or collection of programs, the correctness specification of each program α is matched with each of the states in R_α. For each $(w, w') \in R_\alpha$, w and w' must follow from the correctness specification. These are the general concepts behind the use of dynamic logic towards program verification.

Next, we will describe another logic that is useful for reasoning about intermediary states of programs, temporal logic.

4.4 Temporal Logic

Temporal logic mixes time with mathematical reasoning. Although people have reasoned about time for centuries, Arthur Prior introduced a method in the 1950's that has since been a cornerstone of temporal logic [32]. There are many variants of temporal logic. We will discuss most of the variants that use modal logic concepts, such as propositional tense logic (PTL).

4.4.1 Syntax and Semantics

We add to traditional propositional logic the modal operators G ("always in the future") and H ("always in the past"). A temporal formula is defined as:

- if $p \in \Phi$, then p is a formula;
- if φ and ψ are formulas, then so are $\neg\varphi$ and $\varphi \wedge \psi$;
- if φ is a formula, then so are $G\varphi$ and $H\varphi$.

The dual operators of G and H are F('at least once in the future') and P('at least once in the past'), respectively. They are defined as $F\varphi \equiv \neg G \neg \varphi$ and $P\varphi \equiv \neg H \neg \varphi$. Let $L_t(\Phi)$ contains the least set of formulas closed under conjunction, negation, and the modal operators G and H.

The semantics for PTL uses Kripke structures, as seen in previous logics. Let $M = (W, R, V)$ be a modal structure as is defined in Section 3. We define the relation $\models$ as usual.

$$(M, w) \models p \text{ where } p \in \Phi \text{ iff } V(p, w) = \textit{true}$$

$$(M, w) \models \neg\varphi \text{ iff } (M, w) \not\models \varphi$$

$$(M, w) \models \varphi \wedge \psi \text{ iff } (M, w) \models \varphi \text{ and } (M, w) \models \psi$$

$$(M, w) \models G\varphi \text{ iff } (M, w') \models \varphi \text{ for all } w' \in W \text{ such that } (w, w') \in R$$

$$(M, w) \models H\varphi \text{ iff } (M, w') \models \varphi \text{ for all } w' \in W \text{ such that } (w', w) \in R$$

4.4.2 Axiomatizations

The following modal logics for PTL are similar to those introduced in Section 3. They are called $K_t, T_t, K4_t, S4_t$. Each of these logics contain the following axioms and rules of references.

A1: All tautologies are permitted.

A2: $(G\varphi \wedge G(\varphi \rightarrow \psi)) \rightarrow G\psi$.

A3: $(H\varphi \wedge H(\varphi \rightarrow \psi)) \rightarrow H\psi$.

A4: $PG\varphi \rightarrow \varphi$.

A5: $FH\varphi \rightarrow \varphi$.

R1: If $\vdash \varphi$ and $\vdash \varphi \rightarrow \psi$, then, $\vdash \psi$ (Modus Ponens).

R2: If $\vdash \varphi$, then $\vdash G\varphi$. (G-generalization)

R3: If $\vdash \varphi$, then $\vdash H\varphi$. (H-generalization)

The axioms $A1, A2, A3$ and the rules $R1, R2, R3$ hold for each of the modal logics $K_t, T_t, K4_t, S4_t$. In addition,

- $A4$ holds for T_t,
- $A5$ holds for $K4_t$,
- $A4$ and $A5$ hold for $S4_t$.

If M is a modal structure, then it is a K_t-model. If M is a modal structure and R is reflexive (respectively; transitive; reflexive and transitive) then M is a T_t-model (respectively, $K4_t$-model, $S4_t$-model). For $X \in \{K, T, K4, S4\}, X_t$ is sound and complete with respect to all X_t-models. [3] shows K_t and $K4_t$ to be modally complete.

4.4.3 Applications of Temporal Logic

There are many different applications for temporal logic. Some of these applications include databases, multi-agent systems, like the one presented in Section 4.3.4, robotics, planning, and natural language understanding. We will present how temporal logic can be useful for databases.

A *historical database* is a database where changes to data occur over time due to changes in the world or changes in our knowledge of the world. For example, a database for a bank subtracts an amount b of an owner's bank account if the owner has made a withdrawal of amount b. To reason about such a system, we need to know what facts are true at which points in time. This implies using temporal logic.

4.5 Deontic Logic

Deontic logic reasons about norms and normative behavior, introducing operators for obligatory, permission, and forbidden. It is a logic that reasons about what is considered normal behavior for a particular entity. Deontic logic first began as a philosophical logic but has since become helpful in politics, artificial intelligence, and computer science.

4.5.1 Syntax and Semantics

A modal formula is either $p \in \Phi, \varphi \wedge \psi, \neg\varphi, O\varphi$ where φ and ψ are deontic formulas. The formula $O\varphi$ intuitively means "φ is obligatory" or "φ ought to be true." Let $L_D(\Phi)$ be the least set of modal formulas that is closed under negation, conjunction, and the obligatory operator. Let $P\varphi \overset{def}{=} \neg O \neg\varphi$. The intuitive meaning of $P\varphi$ is "φ is permitted." Also, $F\varphi \overset{def}{=} \neg P\varphi$ means that "φ is forbidden."

The semantics for deontic logic is similar to the semantics presented in Section 3. Let $M = (W, R, V)$ be a modal structure. The definition for $\models$ changes to the following.

$$(M, w) \models p,\ \text{where } p \in \Phi,\ \text{iff } V(p, w) = \mathit{true}$$

$$(M, w) \models \neg\varphi \text{ iff } (M, w) \not\models \varphi$$

$$(M, w) \models \varphi \wedge \psi \text{ iff } (M, w) \models \varphi \text{ and } (M, w) \models \psi$$

$$(M, w) \models O\varphi \text{ iff } (M, w') \models \varphi \text{ for all } w' \in W \text{ such that } (w, w') \in R$$

4.5.2 Axiomatizations

The following is sound and complete for the logic KD with respect to the class of modal structures where R is serial [1]. Let φ and ψ be deontic formulas.

A1 All instances of tautologies of propositional calculus are permitted.

A2 $O(\varphi \rightarrow \psi) \rightarrow (O\varphi \rightarrow O\psi)$.

A3 $O\varphi \rightarrow P\varphi$.

R1 If $\vdash \varphi$ and $\vdash \varphi \rightarrow \psi$, then $\vdash \psi$ (Modus Ponens).

R2 If $\vdash \varphi$, then $\vdash O\varphi$ (Generalization).

The logic that contains only A1, A2, A3, R1, and R2 is called the standard system of deontic logic, or KD.

4.5.3 Applications of Deontic Logic

Deontic logic is useful for reasoning about normative law. It comes as no surprise that applications of deontic logic was first seen in legal applications. In recent years, deontic logic has been useful for applications in artificial intelligence as well. Applications with deontic concepts can be classified by the following list:

- fault-tolerant computer systems,
- normative use behavior,
- normative behavior in or of the organization,
- and normative behavior of the object system.

We will discuss applications of deontic logic in fault-tolerant computer systems and normative use behavior. Wieringa and Meyer explain each of the categories in detail [44].

No computer is fail-safe. However, *fault-tolerant computer systems* are designed to handle violations of normal computer behavior, such as when a computer's hard drive crashes. They can engage in behavior that, although not ideal, may be meaningful depending on the violation and current state of the system. This application of deontic logic is regarded as the specifications of exception-handling in fault-tolerant systems.

Now, we look at how deontic logic is useful for normative use behavior. End users do not always behave as expected. For example, a user might input text when asked for a number or input a six character string when asked for an eight to sixteen character string. He or she might press the wrong keys or provide inconsistent data [44]. Thus, there should be a clear distinction between a user's desired behavior and the behavior the user may actually engage in. The specifications for error-handling determine how an application will respond to the user's unexpected behavior. Deontic logic is useful in formulating these specifications. This is just a sample of the applications of deontic logic. More applications appear in [44].

5 Introduction to Complexity Theory

Complexity theory investigates the difficulty of problems in computer science. This is a fairly new field in computer science and has greatly expanded in the last thiry years. Resources such as time and space are used as tools to measure difficulty. Problems that are considered computationally practical are those that can be solved in a polynomial of the length of the input, where the resource used is time. To understand this section, readers should have basic foundations in algorithms and graph theory. The information presented in this section will provide a foundation to build on in Section 6, where we discuss the complexity of some problems in the logics introduced in the previous section. In addition to reading this section, other sources that may be of aid when reading Section 6 are [28, 38].

5.1 Decidability

A decision problem is answered by either a 'yes' or a 'no' for each input. A solution for a decision problem is an algorithm that gives the right answer for every input. An algorithm is a detailed series of steps, like a recipe, that solves a problem. Problems that are solved by algorithms are decidable. In complexity theory, we will consider only decidable problems.

Example Let φ be a modal formula and $M = (W, R, V)$ be a structure where W is a finite set of states; R is a subset of $W \times W$; and V is a function such that $W \times \Phi \rightarrow \{\mathit{true, false}\}$. Recall that Φ is a non-empty set of propositions. A common decidable problem asks if φ is satisfiable in M. This problem is known as the model-checking problem.

Notice how the problem was stated. The only solutions that would answer the problem are 'yes' or 'no.' The problem could also have been stated as: Given a structure M and a modal formula φ, at what state $w \in W$ is φ satisfiable? The form of the question changes the problem type since answers would be states.

One approach to solving the model-checking problem is the following algorithm, which is based on the algorithm found in [15]. Recall that size(φ) is defined to be the number of symbols in φ, where φ is a string over $\Phi \cup \{(,), \Box, \wedge, \neg\}$.

```
name: model-check
input: a modal formula $\varphi$,
1. let $S$ be the set sub($\varphi$)
2. sort $S$ such that each element $\psi \in S$ is listed in ascending order
with respect to size($\psi$)
3. while ($S \neq \emptyset$)
4.     remove a formula $\psi$ from the front of $S$
5.     for each $w \in W$
6.         if $\psi \in \Phi$
7.             if $V(w, \psi) = true$
8.                 label $w$ with $\psi$
9.             else
10.                label $w$ with $\neg\psi$
11.        if $\psi = \neg\psi'$
12.            if $w$ is labeled with $\neg\psi'$
13.                label $w$ with $\psi$
14.            else
15.                label $w$ with $\neg\psi$
16.        if $\psi = \psi' \wedge \psi''$
17.            if $w$ is labeled with $\psi'$ and $\psi''$
18.                label $w$ with $\psi$
19.            else
20.                label $w$ with $\neg\psi$
21.        if $\psi = \Box\psi'$
22.            if $w'$ is labeled with $\psi'$ for each $(w, w') \in R$
23.                label $w$ with $\psi$
24.            else
25.                label $w$ with $\neg\psi$
26. for each $w \in W$
27.     if $w$ is labeled with $\varphi$
28.         return 'yes'
29. return 'no'
```

The verification of model-check is fairly straightforward. It is strongly based on the definition of $\models$; however, instead of observing whether a formula holds at a single state in M, we look at all states in M. Recall that φ is satisfiable in M iff φ is satisfiable at some state in M.

Some details are left out of model-check. For example, one important detail is the representation of M. We will look in detail at the representation of M since the data structure for M is important to obtaining complexity results for the model-checking problem. Let M be a directed graph, where W is a list of "nodes" and R is a set of "edges." There are two main representations used for graphs: adjacency-list and adjacency-matrix representations. We will use a modified version of an adjacency-matrix representation. The modification is an additional bit-matrix that contains the information for V. In row i, column j, a '1' means that $V(i, j) = \textit{true}$, and a '0' means that $V(j, i) = \textit{false}$. The propositions used in the matrix are only those that occur in φ. Figure 5.1 is an example of a typical modal structure M, where $W = \{u, v, w\}, R = \{(u, w), (v, u), (v, v)\}$, and $V(p_1, u) = V(p_2, u) = V(p_2, v) = V(p_1, w) = V(p_2, w) = \textit{true}$. Figure 5.2 gives us the corresponding adjacency-matrix representation for M.

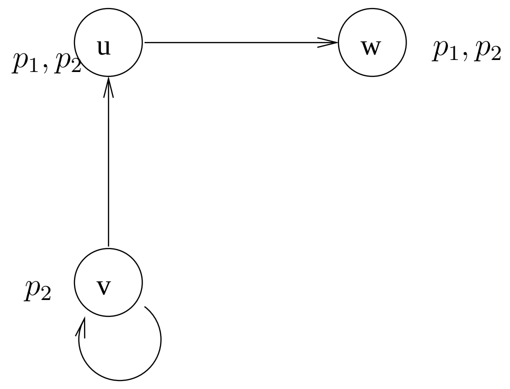


Figure 5.1

Array A

$$\begin{array}{c|ccc|} & u & v & w \\ \hline u & 0 & 0 & 1 \\ v & 1 & 1 & 0 \\ w & 0 & 0 & 0 \\ \hline \end{array}$$

Array B

$$\begin{array}{c|ccc|} & u & v & w \\ \hline p_1 & 1 & 0 & 1 \\ p_2 & 1 & 1 & 1 \\ \hline \end{array}$$

Figure 5.2

In Figure 5.2, Array A is a typical adjacency-matrix representation for a graph. For M, Array A contains the information for W and R. Array B contains the information for V. Let m be the number of rows in Array B. We can now define the size of M, abbreviated as $||M||$, to be $||W||^2 + ||W|| \times m$. Recall the length of φ, abbreviated as $|\varphi|$, is the length of the encoding of φ.

To analyze algorithms, we use an important and useful tool for complexity theory, the $\mathcal{O}$ notation. Let $\mathcal{N}$ be the set of positive integers.

Definition Assume f and g are functions from $\mathcal{N}$ to $\mathcal{N}$. We say $f(n) = \mathcal{O}(g(n))$ if there are integers $c, n_0 \in \mathcal{N}$ such that, for all $n \geq n_0, f(n) \leq c \times g(n)$. This notation is normally read as "$f(n)$ has a big-$\mathcal{O}$ of $g(n)$" or "$f(n)$ is on the order of $g(n)$."

Intuitively, this means that the function $f(n)$ grows slower or at the same rate as $g(n)$. To continue our example, we will analyze the algorithm for the model-checking problem. Remember that S contains the subformulas of φ. As a reminder, the size of sub(φ) is no greater than size(φ). Thus, $||S|| \leq$ size(φ) $\leq |\varphi|$. For each subformula of φ, we visit each state in M once. The body of the loop between lines 5 and 25 is executed at most $\mathcal{O}(|\varphi| \times ||W||)$. However, not all the steps inside the loop take a constant time, that is $\mathcal{O}(1)$. For example, in line 22, to find each $w' \in W$ such that $(w, w') \in R$, the algorithm looks at w's row in Array A. If there is a '1' in row w, column w', then the algorithm checks if w' has been labeled with ψ'. If a state's labels are actually pointers to subformulas in φ, then this step could take $\mathcal{O}(|\varphi|)$ steps. Thus, one execution of line 22 would take $\mathcal{O}(||W|| \times |\varphi|)$. Since lines 3 through 21 are the most "complex" section of model-check, we can say model-check has a running time of $\mathcal{O}(|\varphi|^2 \times ||W||^2)$.

5.1.1 Turing Machines

Turing machines are a formalized method of representing algorithms. Complexity theory uses a Turing machine representation for algorithms due to its simplicity and unifying approach. In this section, we present an introduction to Turing machines. Turing machines model how a human being would solve a problem in an algorithmic way.

Church-Turing Thesis A language can be solved by an algorithm if and only if it can be accepted by a Turing machine that halts on every input.

The following definition is a simplified Turing machine introduced by Sipser. For extra readings on Turing machines, look to [28, 38].

Definition [41]A *Turing machine* is a quintuple $M = (Q, \Sigma, \Gamma, q_0, \delta)$, where

- Q is a finite set of states,
- Σ contains the input alphabet,
- Γ contains the tape alphabet,

- $q_0 \in Q$ is the initial state,
- δ is the transition function from $Q \times \Gamma \to Q \times \Gamma \times \{\text{right, left, stay}\}$.

The transition function δ is basically the "program" of the machine. The set {right, left, stay} refers to which direction the tape head moves. Note that for each input to δ, there is only one output.

5.1.2 Nondeterministic Turing machines

Nondeterministic Turing machines are similar to typical Turing machines, with the exception that the transition function allows for one or more outputs for each input. The output is in the form of subsets. For example, let M be a nondeterministic Turing machine where $q \in Q, a \in \Gamma$, and $\delta(q, a)$ is a set A where $||A|| > 1$. At this point in the "program," M chooses from A which output to take. We say that M always chooses a path that will allow M to accept the input.

Theorem 5.1 *Let M_1 be a nondeterministic Turing machine. Then, there exists a Turing machine M_2 such that $L(M_1) = L(M_2)$.*

The proof idea behind this theorem is to simulate each computation possibility of a nondeterministic Turing machine until an accepting state is found. Where the path of computations of a Turing machine makes a straight line, the path of computations for a nondeterministic Turing machine is a N-ary branching tree, where N is the number of choices that a Turing machine can guess from during a particular computation. This means that each nondeterminstic Turing machine can be simulated with a deterministic Turing machine.

5.2 Time Complexity

We introduced complexity theory as an investigation of why some problems are difficult to solve by computers. We will first introduce complexity using time as a measurement tool, or time complexity.

Definition Let M be a Turing machine that halts on all inputs. We say the *running time*, or *time complexity*, of M is the function $f : \mathcal{N} \to \mathcal{N}$, the maximum number of steps it takes for M to run on any input n. If $f(n)$ is the running time of M, M is an $f(n)$ time Turing machine and M runs in time $f(n)$.

Worst-case analysis of M is used to find $f(n)$ since this will give us an accurate description of the running time for *all* inputs. Worst-case analysis is the largest possible running time for a Turing machine.

In comparing running times of different Turing machines, it is easier and more convenient to compare the growth rates instead of accurate running time functions. Thus, we generally use $\mathcal{O}$ notation to describe running times.

Definition Suppose that some language L is decided by a time $f(n)$ Turing machine. Then, we say that $L \in \text{DTIME}(f(n))$. DTIME($f(n)$) is a set of languages where the Turing machine for each element has a running time of at most $f(n)$. We call DTIME($f(n)$) a complexity class. Furthermore, NTIME($f(n)$) is a class of languages that can be decided by an $\mathcal{O}(f(n))$ nondeterministic Turing machine.

Some of the most well-known time complexity classes are the following: polynomial time (P), nondeterministic polynomial time (NP), and exponential time (EXPTIME). We will begin by presenting the complexity classes P and NP.

5.2.1 P and NP

Definition P is the class of languages A that are decidable in polynomial time (i.e., $\text{P} = \bigcup_k \text{DTIME}(n^k)$, where n is the length of an input to A, and k is a constant).

P is robust, meaning it is the same class for all reasonable computation models. We can represent algorithms for languages in P with high-level descriptions, or pseudocode, instead of using complicated Turing machines. For example, model-check presented in Section 5.1 is easily shown to have a polynomial running time but does not give the formal description of a Turing machine. Problems in P can be realistically solved by a computer. We know that P is at least a subset of the complexity class NP.

Definition NP is defined as the class of languages A that are decided in nondeterministic polynomial time. $\text{NP} = \bigcup_k \text{NTIME}(n^k)$, where n is the length of an input to A, and k is a constant.

Verifiers are an important concept used to understand the class NP. We present the definition of a verifier to be used as a second definition for NP.

Definition A *verifier* is an algorithm V that decides a language L such that $L = \{w \mid V$ accepts on inputs w and c for some $c\}$. A *polynomial-time verifier* is a verifier that runs in polynomial time in the length of w. L is *polynomially verifiable* if it has a polynomial-time verfier.

The difference between V and traditional algorithms for L is the additional information c, which is sometimes called the certificate. Basically, the concept behind polynomial-time verifiers is to remove the nondeterminism aspect from traditional algorithms.

Definition *NP* is the class of languages that are polynomially verifiable.

Any reasonable nondeterministic model of computation will suffice to show that a language A is in NP if A is in NP. We can see why this is true because polynomial-time verifiers can be easily related to polynomial time algorithms.

Example One example of a problem in NP is the satisfiability problem for propositional formulas, or SAT for short. SAT is used to test whether a propositional formula is satisfiable. For example, let p_1, p_2, p_3 be propositions and ψ be the propositional formula $\neg p_1 \wedge (p_2 \wedge p_3)$ If ψ is satisfiable, then there is at least one assignment for each of p_1, p_2, p_3 such that ψ is *true*. The assignments $p_1 = \textit{true}, p_2 = \textit{true}$, and $p_3 = \textit{false}$ will result in ψ being satisfiable.

To show that SAT is in NP, we give the following proof.

Proof The following is a nondeterministic polynomial time algorithm for SAT.

```
name: sat
On input $\psi$, where $\psi$ is a propositional formula,
1.  Nondeterministically replace each of the propositions in $\psi$ with
    either *true* or *false*.
2.  Simplify $\psi$ to a single truth value.
3.  If $\psi$ is *true*, return 'yes'; otherwise, return 'no'.
```

Proof As an example, we also give an alternate proof that uses a polynomial-time verifier. For this proof, let a truth assignment for ψ be the certificate c.

```
name: sat-v
Input is a propositional formula $\psi$ and a truth assignment $c$,
1.  Test whether $c$ is a truth assignment of $\psi$ and
    replace each proposition in $\psi$ with its corresponding truth value.
2.  Simplify $\psi$ to a single truth value.
3.  If step 1 passes and $\psi$ is *true*, return 'yes'; otherwise, return 'no'.
```

It has not yet been proven whether P = NP. This question is one of the most important unsolved problems in complexity theory.

5.2.2 NP-completeness

Complexity theory uses polynomial-time reductions as a simple way to show that a certain problem X is at least as hard as a problem Y, without explicitly giving an algorithm for X. This is done by finding a polynomial-time function that reduces Y to X.

Definition A function f is a *polynomial-time computable function* if there exists a Turing machine that runs in polynomial time and outputs $f(w)$ on any input w. A language A is *polynomial-time reducible*, or *polynomial-time many-one reducible*, to B, written as $A \leq_m^p B$, if a polynomial-time computable function exists such that $w \in A$ iff $f(w) \in B$. The function f is called the *polynomial-time reduction* from A to B. The reduction $A \leq_m^p B$ is also read as "A reduces to B."

To show that a language A reduces to B, there are three parts that must be expressed in the proof:

- a function f must be given,
- it must be proven that $w \in A$ iff $f(w) \in B$,
- and it must be shown that f is computable in polynomial time.

We can now present what it means for a language to be hard compared to a complexity class.

Definition A language A is said to be *hard* with respect to a complexity class $\mathcal{C}$ if every language in $\mathcal{C}$ can be reduced to A.

Some examples of hardness are NP-hard, PSPACE-hard, and EXPTIME-hard. We will speak more of the latter two in a later section. An important idea to grasp is that a language A that is NP-hard intuitively means that A is at least as hard as all languages in NP. This does not tell us which complexity class A resides in.

Definition A language A is *complete* with respect to a complexity class $\mathcal{C}$ if A is in $\mathcal{C}$ and A is also $\mathcal{C}$-hard.

Intuitively, A is complete for $\mathcal{C}$ iff A is a hardest language in $\mathcal{C}$. Like for $\mathcal{C}$-hard, some examples of $\mathcal{C}$-complete classes are NP-complete, PSPACE-complete, and EXPTIME-complete. For now, we will explore the properties of languages that are NP-complete. It is not trivial that NP-complete languages exist. To prove that a language X is NP-complete, we would have to show that that a polynomial-time reduction exists from every language in NP to X, and that X is in NP.

Theorem 5.2 *[7, 26] SAT is NP-complete.*

In the proof of Theorem 5.2, it is shown that any language A that is in NP polynomial-time reduces to SAT. Now that we know of one NP-hard problem, we can use SAT (or in general, NP-hard problems) to show that new problems are NP-hard.

Theorem 5.3 *If A is $\mathcal{C}$-hard and $A \leq_m^p B$, then B is $\mathcal{C}$-hard.*

Proof Let A be a language that is $\mathcal{C}$-hard and $A \leq_m^p B$. Let D be an arbitrary language in $\mathcal{C}$. Then, $D \leq_m^p A$. There exists a polynomial-time computability function f such that $w \in D \iff f(w) \in A$. Also, there exists a polynomial-time computability function g such that $w \in A \iff g(w) \in B$. Let $h = g \circ f$. Then, $w \in D \iff h(w) \in B$. Since a polynomial of a polynomial is a polynomial, h is a polynomial-time computability function. Without loss of generality, B is $\mathcal{C}$-hard.

It is much easier to show that a problem X is NP-complete by using Theorem 5.3 rather than finding a polynomial-time reduction for each problem in NP to X. We will begin to prove that $S5$-satisfiability ($S5$-SAT) is NP-complete. $S5$-SAT is the set of all formulas that are satisfiable in $S5$. Trivially, SAT reduces to $S5$-SAT, because a propositional formula φ is in SAT iff it is in $S5$-SAT. Since SAT is NP-complete, it is NP-hard. So, by Theorem 5.3, $S5$-SAT is NP-hard. We will continue to show that $S5$-SAT is NP-complete by proving that $S5$-SAT is in NP. All we need to do is present a polynomial-time verifier or a nondeterministic polynomial time algorithm for finding if a formula φ is $S5$-SAT. We begin by presenting a proposition that will help us find such an algorithm.

Proposition 5.4 *[25] A modal formula φ is satisfiable in $S5$ iff it is satisfiable in an $S5$-model with at most $|\varphi|$ states.*

Proof Suppose φ is satisfiable in $S5$. By Proposition 3.3, φ is satisfiable in an $S5$-model $M = (W, R, V)$ where, for all $v, v' \in W, (v, v') \in R$, i.e., R is universal. Suppose $(M, u) \models \varphi$. Let F be the set of subformulas of φ of the form $\Box\psi$ for which $(M, u) \models \neg\Box\psi$. For each $\Box\psi \in F$, there must be some state $u_{\neg\psi} \in W$ such that $(M, u_{\neg\psi}) \models \neg\psi$. Let $M' = (W', R', V')$, where $W' = \{u\} \cup \{u_{\neg\psi} \mid \Box\psi \in F\}$, V' is the restriction of V to $W' \times \Phi$, and $R' = \{(v, v') \mid v, v' \in W'\}$. $||F|| < ||\text{sub}(\varphi)||$ since F is a subset of $\text{sub}(\varphi) - \Phi$ and $\text{sub}(\varphi) \cap \Phi$ is not empty since φ contains at least one proposition. We already know that $||\text{sub}(\varphi)|| \leq \text{size}(\varphi) \leq |\varphi|$, so $||W'|| \leq |\varphi|$. We now show that for all states $u' \in W'$ and for all subformulas ψ of φ, $(M, u') \models \psi \iff (M', u') \models \psi$. We perform induction on the structure of ψ. For the following cases, let $u' \in W'$.

Case 1: ψ is of the form $p \in \Phi$. This is the base case. If $(M, u') \models p$, i.e., $V(u', p) = \mathit{true}$, then $V'(u', p) = \mathit{true}$. So, $(M', u') \models p$. Similarly, if $(M', u') \models p$, then $(M, u') \models p$.

Case 2: ψ is of the form $\psi_1 \wedge \psi_2$. If $(M, u') \models \psi_1 \wedge \psi_2$, then $(M, u') \models \psi_1$ and $(M, u') \models \psi_2$. By the induction hypothesis, $(M', u') \models \psi_1$ and $(M', u') \models \psi_2$. So, $(M', u') \models \psi_1 \wedge \psi_2$. Similarly, if $(M', u') \models \psi_1 \wedge \psi_2$, then $(M', u') \models \psi_1$ and $(M', u') \models \psi_2$. By the induction hypothesis, $(M, u') \models \psi_1$ and $(M, u') \models \psi_2$. So, $(M, u') \models \psi_1 \wedge \psi_2$.

Case 3: ψ is of the form $\neg\psi'$ If $(M, u') \models \neg\psi'$, then $(M, u') \not\models \psi'$. By the induction hypothesis, $(M', u') \not\models \psi'$. So, $(M', u') \models \neg\psi$. Likewise, if $(M', u') \models \neg\psi'$, then $(M', u') \not\models \psi'$. By the induction hypothesis, $(M, u') \not\models \psi'$. So, $(M, u') \models \neg\psi'$.

Case 4: ψ is of the form $\Box\psi'$. If $(M, u') \models \Box\psi'$, then $(M, v) \models \psi'$ for all $v \in W$. Thus, $(M, v') \models \psi'$ for all $v' \in W'$, since $W' \subseteq W$. By the induction hypothesis, $(M', v') \models \psi'$ for all $v' \in W'$. So $(M', u') \models \Box\psi'$ because $(u', v') \in R'$ for all $v' \in W'$. If $(M, u') \not\models \Box\psi'$, then $(M, u') \models \neg\Box\psi'$. There is some state $u_1 \in W$ such that $(M, u_1) \models \neg\psi'$. Since $(v, u_1) \in R$ for all $v \in W, (M, v) \models \neg\Box\psi'$. In particular, $(M, u) \models \neg\Box\psi'$. We know that $\Box\psi' \in F$ due to our previous construction. So, there is a state $u_{\neg\psi'} \in W'$ such that $(M, u_{\neg\psi'}) \models \neg\psi'$. Since $(v', u_{\neg\psi'}) \in R'$ for all $v' \in W'$, $(M', v') \models \neg\Box\psi'$. In particular, $(M', u') \models \neg\Box\psi'$, or $(M', u') \not\models \Box\psi'$.

Since $u \in W'$ and $(M, u) \models \varphi$, we also have $(M', u) \models \varphi$. Thus, we have proved that φ is $S5$-satisfiable iff it is satisfiable in an $S5$-model with at most $|\varphi|$ states.

Theorem 5.5 *[25] The satisfiability problem for $S5$ is in NP.*

Proof We will give a nondeterministic polynomial-time algorithm for deciding $S5$-satisfiability.

```
name: $S5$-sat
input: a modal formula $\varphi$,
1.  Nondeterministically choose an $S5$-model $M = (W, R, V)$
    such that such that $||W|| \leq |\varphi|$. Assume propositions in $\Phi$
    but not used in $\varphi$ are false at each state in $W$.
2.  for each $w \in W$,
3.      check that $(M, w) \models \varphi$ using model-check.
4.      if model-check returns ‘yes,’ return ‘yes.’
5.  return ‘no.’
```

To guess a modal structure M, we would have to guess each of the truth values for the propositions in φ for each state in W. Since there are at most $|\varphi|$ propositions in φ, we have a nondeterministic running time of $\mathcal{O}(|\varphi|^2)$ for step 1. In Section 5.1, we showed that model-check has a running time of $\mathcal{O}(|\varphi^2| \times ||W||^2)$. In the above algorithm, we perform model-check for each state in W. This shows that $S5$-sat runs in polynomial time in the length of the input. By Proposition 5.4 if φ is satisfiable in $S5$, then at least one of our guesses is bound to be right. Thus, we have a nondeterministic polynomial time algorithm for deciding if φ is satisfiable in $S5$.

We have proved that $S5$-SAT is NP-complete. We will now present a proof to show that $KD45$-satisfiability ($KD45$-SAT) is also NP-complete. $KD45$-SAT is the set containing all modal formulas that are satisfiable in $KD45$. As with $S5$-SAT, a propositional formula φ is SAT iff it is $KD45$-SAT. By Theorem 4.3, $KD45$-SAT is NP-hard. To show $KD45$-SAT is in NP, we will prove the following proposition.

Proposition 5.6 *[15] A modal formula φ is satisfiable in $KD45$ iff it is satisfiable in a $KD45$-model with at most $|\varphi|$ states.*

Proof Suppose φ is satisfiable in $KD45$. By Propostion 3, there exists a $KD45$-model $M = (\{w_0\} \cup W, R, V)$ where W is nonempty and $R = \{(u, v) \mid u \in \{w_0\} \cup W, v \in W\}$. Suppose $(M, u) \models \varphi$. Let F be the set of subformulas of φ of the form $\Box\psi$ for which $(M, u) \models \neg\Box\psi$. For each $\Box\psi \in F$, there is some state $u_{\neg\psi} \in W$ such that $(M, u_{\neg\psi}) \models \neg\psi$. Let $M' = (W', R', V')$, where $W' = \{u\} \cup \{u_{\neg\psi} \mid \Box\psi \in F\}, V'$ is the restriction of V with respect to $W' \times \Phi$, and $R' = \{(u,v) | u \in W', v \in W' - \{u\}\}$. $||F|| < ||\text{sub}(\varphi)||$ since F is a subset of $\text{sub}(\varphi) - \Phi$. We already know that $||\text{sub}(\varphi)|| \leq \text{size}(\varphi) \leq |\varphi|$, so $||W'|| \leq |\varphi|$. We now show that for all states $u' \in W'$ and for all subformulas ψ of $\varphi, (M, u') \models \psi \Leftrightarrow (M', u') \models \psi$. We perform induction on the structure of ψ. For those cases where ψ is not of the form $\Box\psi'$, we refer the reader to cases 1 through 3 of Proposition 5.4, since R and R' have no effect on these forms. Let ψ be of the form $\Box\psi'$ and $u' \in W'$.

Case 1: $u = u_0$ In this case, u_0 is both in $\{w_0\} \cup W$ and W'. If $(M, u') \models \Box\psi'$, then $(M, v) \models \psi'$ for all $v \in W$. Thus, $(M, v') \models \psi'$ for all $v' \in W' - \{u_0\}$, since $W' \subseteq (\{w_0\} \cup W)$. By the induction hypothesis, $(M', v') \models \psi'$ for all $v' \in W' - \{u_0\}$. So $(M', u') \models \Box\psi'$ because $(u', v') \in R'$ for all $v' \in W' - \{u_0\}$. If $(M, u') \not\models \Box\psi'$, then $(M, u') \models \neg\Box\psi'$. There is some state $u_1 \in W$ such that $(M, u_1) \models \neg\psi'$. Since $(v, u_1) \in R$ for all $v \in \{w_0\} \cup W, (M, v) \models \neg\Box\psi'$. In particular, $(M, u) \models \neg\Box\psi'$. We know that $\Box\psi' \in F$ due to our previous construction. So, there is a state $u_{\neg\psi'} \in W' - \{u\}$ such that $(M, u_{\neg\psi'}) \models \neg\psi'$. Since $(t', u_{\neg\psi'}) \in R'$ for all $v' \in W'$, $(M', v') \models \neg\Box\psi'$. In particular, $(M', u') \models \neg\Box\psi'$, or $(M', u') \not\models \Box\psi'$.

Case 2: $u \neq u_0$ If $(M, u') \models \Box\psi'$, then $(M, v) \models \psi'$ for all $v \in W$. Thus, $(M, v') \models \psi'$ for all $v' \in W'$, since $W' \subseteq \{w_0\} \cup W$. By the induction hypothesis, $(M', v') \models \psi'$ for all $v' \in W'$. So $(M', u') \models \Box\psi'$ because $(u', v') \in R'$ for all $v' \in W'$. If $(M, u') \not\models \Box\psi'$, then $(M, u') \models \neg\Box\psi'$. There is some state $u_1 \in W$ such that $(M, u_1) \models \neg\psi'$. Since $(v, u_1) \in R$ for all $v \in \{w_0\} \cup W, (M, v) \models \neg\Box\psi'$. In particular, $(M, u) \models \neg\Box\psi'$. We know that $\Box\psi' \in F$ due to our previous construction. So, there is a state $u_{\neg\psi'} \in W' - \{u\}$ such that $(M, u_{\neg\psi'}) \models \neg\psi'$. Since $(v', u_{\neg\psi'}) \in R'$ for all $v' \in W'$, $(M', v') \models \neg\Box\psi'$. In particular, $(M', u') \models \neg\Box\psi'$, or $(M', u') \not\models \Box\psi'$.

Since $u \in W'$ and $(M, u) \models \varphi$, we also have $(M', u) \models \varphi$. Thus, we have proved that φ is $KD45$-satisfiable iff it is satisfiable in a $KD45$-model with at most $|\varphi|$ states.

Theorem 5.7 *[15] The satisfiability problem for $KD45$ is in NP.*

The proof for this problem follows that of Theorem 5.5, except we replace the reference to Proposition 5.4 with Proposition 5.6.

5.3 Space Complexity

At the beginning of the previous subsection, we mentioned that determining the difficulty of a problem is based on resources such as time and space. In this section, we present attributes of space complexity.

Definition Let M be a Turing machine that halts on all inputs. We say the *space complexity* of M is the function $f : \mathcal{N} \rightarrow \mathcal{N}$, the maximum number of tape cells that M uses to run on any input n. If $f(n)$ is the space complexity of M, M is an $f(n)$ space Turing machine and M runs in space $f(n)$.

Space complexity, like time complexity, uses $\mathcal{O}$-notation to describe space upper bounds for a problem in terms of the length of its input.

Definition Suppose that some language L is decided by a space $f(n)$ Turing machine. Then, we say that $L \in$ DSPACE($f(n)$). DSPACE($f(n)$) is a set of languages where each element has a space complexity of at most $f(n)$. We call DSPACE($f(n)$) a complexity class.

One major space complexity class is called PSPACE.

Definition PSPACE is the class of languages A that are decided in polynomial space, i.e. PSPACE $= \bigcup_k$ DSPACE(n^k), where n is the length of an input to A and k is a constant.

PSPACE has similarities with P. For example, PSPACE is robust. This means that for any problem A in PSPACE, any reasonable model that decides A will show A to be in PSPACE. It is generally believed that P $\subseteq$ NP $\subseteq$ PSPACE. However, as with the case of P and NP, it has not been determined whether P = PSPACE. It is believed that both inclusions are proper. NPSPACE is the class of languages that are decided in nondeterministic polynomial space. By Savitch's theorem, NPSPACE = PSPACE [36].

Example Quantified boolean formulas, or QBF, introduces two new symbols to traditional propositional formulas. They are $\forall$, the universal operator, and $\exists$, the existential operator. A QBF is usually in the form of $Q_1x_1 \ldots Q_mx_m\, \varphi(x_1, \ldots, x_m)$, where $Q_i \in \{\forall, \exists\}, 1 \leq i \leq m$, and $\varphi(x_1, \ldots, x_m)$ is a propositional formula that contains only propositions from $x_1, \ldots, x_m$. An example of a QBF is $\sigma = \forall x_1 \exists x_2 (x_1 \vee x_2)$. The following steps show how to determine if a QBF is true. For each $\forall x_i$, we replace the formula

$\forall x_i\ \psi(x_i)$ with $\psi(x_i = \mathit{true}) \wedge \psi(x_i = \mathit{false})$. For each $\exists x_i$, we replace the formula $\exists x_i\ \psi(x_i)$ with $\psi(x_i = \mathit{true}) \vee \psi(x_i = \mathit{false})$. Let us return to the example $\sigma = \forall x_1 \exists x_2 (x_1 \vee x_2)$.

$$\begin{aligned}
\sigma &= \forall x_1 \exists x_2 (x_1 \vee x_2) \\
&\equiv \exists x_2 (\mathit{true} \vee x_2) \wedge \exists x_2 (\mathit{false} \vee x_2) \\
&\equiv ((\mathit{true} \vee \mathit{true}) \vee (\mathit{true} \vee \mathit{false})) \wedge ((\mathit{false} \vee \mathit{true}) \vee (\mathit{false} \vee \mathit{false})).
\end{aligned}$$

We then use propositional logic to simplify the formula to a single value. In our example, $\sigma = \mathit{true}$.

We now give an algorithm, called qbf, that shows QBF is in PSPACE [38].

```
name: qbf
input: a QBF formula $\sigma$,
1.  If $\sigma$ has no quantifiers, simplify $\sigma$ to a single value. If $\sigma$ is true,
    return 'yes;' otherwise, return 'no.'
2.  If $\sigma$ is in the form of $\forall x(\psi)$, recursively call qbf with input $\psi$
    twice, once with $x$ replaced by false and once with $x$
    replaced by true. If 'yes' is returned on both calls, return 'yes;'
    otherwise return 'no.'
3.  If $\sigma$ is in the form of $\exists x(\psi)$, recursively call qbf with input $\psi$
    twice, once with $x$ replaced by false and once with $x$
    replaced by true. If 'yes' is returned on either call, return 'yes;'
    otherwise return 'no.'
```

To analyze qbf, we first look at how σ changes at each recursive call. Let m be the number of quantifiers in σ. The depth of the recursion is at most m. For each recursive call, the only change in σ is the substitution of a truth value for a proposition. Step 1 takes a space of $\mathcal{O}(|\varphi|)$ where φ is the propositional formula in σ. Thus, algorithm qbf has a space complexity of $\mathcal{O}(m \times |\varphi|)$. So, QBF is in PSPACE.

Like NP-complete, PSPACE-complete is a subclass of PSPACE that contains the hardest languages in PSPACE. To show that a language A is PSPACE-complete, we must prove that A is in PSPACE and A is PSPACE-hard, that is, for all languages B in PSPACE, $B \leq^p_m A$.

Theorem 5.8 *[40] QBF is PSPACE-complete.*

We already showed QBF to be in PSPACE. This theorem is similar to Theorem 5.2 in that QBF was the first language that was shown to be PSPACE-complete. Because of Theorem 5.3, we can show a problem to be PSPACE-hard by reducing it from QBF instead of reducing it from all problems in PSPACE.

Let K-Satisfiability (K-SAT) be the set containing all formulas that are satisfiable in K. Ladner showed K-SAT is PSPACE-complete. We will present a proof to show that a version of K-SAT is also PSPACE-complete. Let a K_2-model be a K-model M where each state in M has exactly 2 successors. K_2-satisfiability (K_2-SAT) contains those formulas that are satisfiable in the set containing all K_2-models. We will show that K_2-SAT is PSPACE-complete.

Theorem 5.9 *K_2-SAT is in PSPACE.*

Proof Our algorithm is similar to Ladner's K-WORLD. We introduce two algorithms, a simple one called K_2-start and another one that does most of the work called K_2-sat. K_2-start takes in as input a modal formula φ, and K_2-sat has as input 4 sets, $T, F, \widehat{T}, \widehat{F}$. Each of these sets contains modal formulas. K_2-sat$(T, F, \widehat{T}, \widehat{F})$ returns 'yes' if and only if the following formula is satisfiable in a K_2-model.

$$\bigwedge_{\psi \in T} \psi \wedge \bigwedge_{\psi \in F} \neg\psi \wedge \bigwedge_{\psi \in \widehat{T}} \Box\psi \wedge \bigwedge_{\psi \in \widehat{F}} \neg\Box\psi$$

Otherwise, K_2-sat returns 'no.' K_2-start calls K_2-sat with T containing φ and $F, \widehat{T}, \widehat{F}$ as empty. The purpose of K_2-sat is to break down formulas in these sets, based on their operators, to obtain underlying propositions. If there are inconsistencies with propositions in T and F (i.e., $\neg p$ and p are contained in either T or F), then the algorithm returns 'no,' meaning φ is not satisfiable.

```
Algorithm $K_2$-start
input: a modal formula $\varphi$,
1.  return $K_2$-sat$(\{\varphi\}, \emptyset, \emptyset, \emptyset)$
```

```
Algorithm $K_2$-sat
input: $(T, F, \widehat{T}, \widehat{F})$, where $T, F, \widehat{T}, \widehat{F}$ are sets of modal formulas
1. if $T \cup F \not\subseteq \Phi$
2.     choose $\psi \in (T \cup F) - \Phi$
3.     if $\psi = \neg\psi'$ and $\psi \in T$
4.         return $K_2$-sat$(T - \{\psi\}, F \cup \{\psi'\}, \widehat{T}, \widehat{F})$
5.     if $\psi = \neg\psi'$ and $\psi \in F$
6.         return $K_2$-sat$(T \cup \{\psi'\}, F - \{\psi\}, \widehat{T}, \widehat{F})$
7.     if $\psi = \psi' \wedge \psi''$ and $\psi \in T$
8.         return $K_2$-sat$((T \cup \{\psi', \psi''\}) - \{\psi\}, F, \widehat{T}, \widehat{F})$
9.     if $\psi = \psi' \wedge \psi''$ and $\psi \in F$
10.        return $(K_2$-sat$(T, (F \cup \{\psi'\}) - \{\psi\}, \widehat{T}, \widehat{F}))\vee$
           $(K_2$-sat$(T, (F \cup \{\psi''\}) - \psi, \widehat{T}, \widehat{F}))$
11.    if $\psi = \Box\psi'$ and $\psi \in T$
12.        return $K_2$-sat$(T - \{\psi\}, F, \widehat{T} \cup \{\psi'\}, \widehat{F})$
13.    if $\psi = \Box\psi'$ and $\psi \in F$
14.        return $K_2$-sat$(T, F - \{\psi\}, \widehat{T}, \widehat{F} \cup \{\psi'\})$
15. if $T \cup F \subseteq \Phi$
16.    if $T \cap F \neq \emptyset$
17.        return ‘no’
18.    else
19.        for each possible subset $B$ of $\widehat{F}$,
20.            if $(K_2$-sat$(\widehat{T}, B, \emptyset, \emptyset) \wedge K_2$-sat$(\widehat{T}, \widehat{F} - B, \emptyset, \emptyset))$
21.                return ‘yes’
22.        return ‘no’
```

The above algorithm is based on Ladner's K-WORLD algorithm. The main difference is that lines 18-22 of K_2-sat restrict the number of successor states to only 2. Line 20 ensures that if φ is K_2-satisfiable, it is K_2-satisfiable in a structure where each state has only 2 successor states. In lines 1-18, we break down φ into its underlying propositions and modal subformulas (i.e., those subformulas of the form $\Box\psi$). This process involves observing if each subformula ψ of φ that is not a proposition is of the form $\neg\psi', \psi' \wedge \psi''$, or $\Box\psi'$, where ψ', ψ'' are subformulas of φ. Whether the set containing ψ', ψ'' is appended to $T, F, \widehat{T}$, or $\widehat{F}$ depends on both which set ψ belongs to and the semantics for the operators $\neg, \wedge, \Box$. One way to understand this algorithm is to imagine that while in the process of breaking down φ, there is some given state w at some given $K2$-model $M = (W, R, V)$ that K_2-sat is observing. When $T \cup F \subseteq \Phi$ and $T \cap F = \emptyset$, the recursive calls in line 20 observe the two states $w_1, w_2 \in W$ such that $(w, w_1), (w, w_2) \in R$. So, the algorithm now observes w_1 or w_2. One can see how this is possible since $T = \widehat{T}$ and F becomes some subset of $\widehat{F}$ after the recursive calls in line 20. For each formula $\psi \in \widehat{F}, \neg\Box\psi$ is true at w. So, ψ must not be true at some successor state of w. We try different combinations of subsets of $\widehat{F}$ until the condition in line 20 is true. We need only one combination for φ to be satisfiable.

To analyze K_2-SAT, we look at how much storage is used per recursion call, and then observe how many recursive calls are made. The sets $T, F, \widehat{T}, \widehat{F}$ are pairwise disjoint sets of subformulas of φ. The size of the space for $T, F, \widehat{T}, \widehat{F}$ is no more than $||\mathrm{sub}(\varphi)||$. Since $||\mathrm{sub}(\varphi)|| \leq |\varphi|$, we have a space complexity of $\mathcal{O}(|\varphi|)$ for each recursive call. Also the recursion depth is $\mathcal{O}(|\varphi|)$ since we analyze each symbol in φ per recursive call. So, our algorithm has a space complexity of $\mathcal{O}(|\varphi|^2)$. Ladner gives a more in-depth analysis. Thus, K_2-SAT is in PSPACE.

To continue showing that K_2-SAT is PSPACE-complete, we will show that K_2-SAT is PSPACE-hard. As a helpful sidenote, we explain what it means for a modal logic to be in between two other modal logics. Let S, S_1, S_2 be sound and complete modal logics with respect to the set of all S-models, S_1-models, S_2-models, respectively. Also, let A, B, C be the set containing all formulas that are valid in the set of all, respectiveley, S-models, S_1-models, S_2-models. We say S is in between S_1 and S_2 if $B \subseteq A \subseteq C$.

Theorem 5.10 *K_2-SAT is PSPACE-hard.*

Proof Ladner uses QBF to show that X-SAT, where X is "in between" K and $S4$, is PSPACE-hard. We will reduce QBF to K_2-SAT. By Theorem 5.3, this will show that K_2-SAT is PSPACE-hard. To clarify matters, K_2 is not in between K and $S4$. The set of all formulas valid in $\mathcal{M}_2$ is a subset of the set of all formulas valid in $\mathcal{M}$, meaning K_2 is a restriction of K. The following proof is similar to Ladner's proof.

Let $\sigma = Q_1 p_1 \ldots Q_m p_m \; \varphi(p_1, \ldots, p_m)$ be a QBF. We construct a formula ψ that is satisfiable in K_2 iff σ is true. We construct the formula ψ so that the existence of a binary tree is forced. Each leaf node represents a unique truth assignment for the propositions $p_1, \ldots, p_m$, which are in ψ and φ. Then, there would be 2^m different leaves for the binary tree. We need an equation that captures the following diagram.

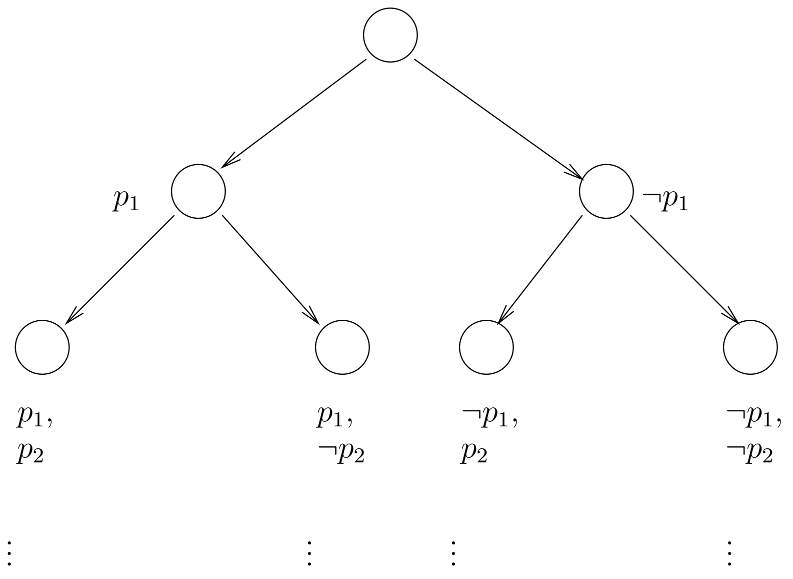


Figure 5.3

The following equation, abbreviated as *tree* gives us this effect.

$$\bigwedge_{i=1}^{m}\bigwedge_{j=0}^{m-i}(\Box^{(i-1)}\Diamond\Box^{(j)}p_i \wedge \Box^{(i-1)}\Diamond\Box^{(j)}\neg p_i)$$

Let $\Box^{(k)}\psi'$ be defined inductively as: $\Box^{(0)}\psi' = \psi', \Box^{(1)}\psi' = \Box\psi'$, and $\Box^{(k)}\psi' = \Box\Box^{(k-1)}\psi'$. Semantically, $(M, w) \models \Box^{(k)}\psi'$ means that there exists a structure M such that ψ' is true at all states reachable from w in k steps. If a propositional formula containing propositions $p_1, p_2, \ldots, p_m$ is satisfiable, then it is true at some state in a modal structure where *tree* is satisfiable. Next, we need to develop an expression based off of σ. We introduce the following notation to be a sequence of $\Diamond$'s and $\Box$'s to precede $\varphi(p_1, \ldots, p_m)$ based on the order of the quantifier $Q_i, 1 \leq i \leq m$.

$$\left\{ \begin{array}{c|c} \Diamond & Q_i = \exists \\ \Box & Q_i = \forall \end{array} \right\}$$

For example, if $\sigma = \forall p_1 \exists p_2 \varphi(p_1, p_2)$, then

$$\left\{ \begin{array}{c|c} \Diamond & Q_i = \exists \\ \Box & Q_i = \forall \end{array} \right\} \varphi(p_1, p_2) = \Box\Diamond\varphi(p_1, p_2).$$

Let ψ' be an abbreviation for the following equation:

$$\left\{ \begin{array}{c|c} \Diamond & Q_i = \exists \\ \Box & Q_i = \forall \end{array} \right\} \varphi(p_1, \ldots, p_m), 1 \leq i \leq m$$

We define ψ to be the following:

$$\mathit{tree} \wedge \psi'$$

We now show that σ is true iff ψ is K_2-SAT. Suppose σ is true. For ψ to be K_2-SAT, ψ must be satisfiable in a K_2-model. Let M be a K_2-model such that its diagram makes an infinite binary tree. Let $W = \{w_i \mid 0 \leq \lfloor \log_2(i+1) \rfloor \leq m\}$. Also, let $R = \{(w_i, w_{2i+1}) \mid 0 \leq \lfloor \log_2(i+1) \rfloor \leq m\} \cup \{w_i, w_{2i+2}) \mid 0 \leq \lfloor \log_2(i+1) \rfloor \leq m\}$. Define V such that for every state w in M of depth $0 \leq d < m$, p_{d+1} is true at all states in the left subtree of w, and $\neg p_{d+1}$ is true at all states in the right subtree of w. More formally, let $w_i \in W$ such that $d = \lfloor \log_2(i+1) \rfloor < m$. Then, $V(w_{2i+1}, p_{d+1}) = \mathit{true}$ and $V(w_{2i+2}, p_{d+1}) = \mathit{false}$. In addition, for all states u (respectively, v) reachable from w_{2i+1}, $V(u, p_{d+1}) = \mathit{true}$ (respectively, $w_{2i+2}, V(v, p_{d+1}) = \mathit{false}$). It is easy to see that ψ is true at w_0.

Now suppose that ψ is K_2-SAT. Then, there is a structure $M = (W, R, V) \in \mathcal{M}_2$ and some $w_0 \in W$ such that $(M, w_0) \models \psi$. For states $w_j \in W$, where $0 \leq j \leq m-1$, let $(w_j, w_{j+1}) \in R$. Let w_k be some given state, where $1 \leq k \leq m$ and $\sigma_{w_k} = Q_{k+1}p_{k+1} \ldots Q_m p_m \varphi(p_{k+1}, \ldots, p_m)$, where each proposition p_j, where $j \leq k$, be replaced with *true* if $V(w_k, p_j) = \mathit{true}$ and *false* otherwise. Note that $\sigma_{w_0} = \sigma$ and $\sigma_{w_m} = \varphi()$ with all propositions $p_1, \ldots, p_m$ replaced by *true* or *false* appropriately. We will use induction on k to show that given some $w_k \in W$, the QBF σ_{w_k} is true.

Base Case: $k = m$ In this case, $\sigma_{w_m} = \varphi()$, where each proposition $p_j, 1 \leq j \leq m$, is *true* if $V(w_m, p_j) = \mathit{true}$ and *false* otherwise. Since $(w_j, w_{j+1}) \in R$ for $0 \leq j \leq m-1$ and $(M, w_0) \models \psi, \varphi(p_1, \ldots, p_m)$ is true at some w_m. Thus, σ_{w_m} is true.

Inductive Hypothesis For all $k < j \leq m, \sigma_{w_j}$ is true given some state $w_j \in W$.

Inductive Step We will show that σ_{w_k} is true.

Case 1: $Q_{k+1} = \forall$ So,

$$\begin{aligned} \sigma_{w_k} = {} & Q_{k+2}p_{k+2} \ldots Q_m p_m \varphi(p_{k+1} = \mathit{true}, p_{k+2}, \ldots, p_m) \wedge \\ & Q_{k+2}p_{k+2} \ldots Q_m p_m \varphi(p_{k+1} = \mathit{false}, p_{k+2}, \ldots, p_m) \end{aligned}$$

by the definition of $\forall$. We also know that

$$(M, w_k) \models \Box \left\{ \begin{array}{c|c} \Diamond & Q_i = \exists \\ \Box & Q_i = \forall \end{array} \right\} \varphi(p_{k+1}, \ldots, p_m) \text{for } k+2 \leq i \leq m$$

for some w_k. Since $\sigma_{w_{k+1}}$ is true at both successors of w_k, and $p_{k+1} = \mathit{true}$ for one successor and *false* for the other successor, σ_{w_k} is true.

Case 2: $Q_{k+1} = \exists$ In this case,

$$\begin{aligned} \sigma_{w_k} = {} & Q_{k+2}p_{k+2} \ldots Q_m p_m \varphi(p_{k+1} = \mathit{true}, p_{k+2}, \ldots, p_m) \vee \\ & Q_{k+2}p_{k+2} \ldots Q_m p_m \varphi(p_{k+1} = \mathit{false}, p_{k+2}, \ldots, p_m) \end{aligned}$$

by the definition of $\exists$. We also know that

$$(M, w_k) \models \Diamond \left\{ \begin{array}{c|c} \Diamond & Q_i = \exists \\ \Box & Q_i = \forall \end{array} \right\} \varphi(p_{k+1}, \ldots, p_m) \text{for } k+2 \leq i \leq m$$

for some $w_k \in W$. Since $\sigma_{w_{k+1}}$ is true at a successor of w_k, σ_{w_k} is true.

Thus, σ_{w_k} is true.

We can now say that $\sigma_{w_0} = \sigma$ is true.

The last part of our proof is to show that our construction for σ is in polynomial-time. To develop ψ from σ, it would take $\mathcal{O}(m^2)$ due to the double $\bigwedge$ in *tree*. Thus, K_2-SAT is PSPACE-hard.

We have shown K_2-SAT to be PSPACE-complete. We will now introduce a complexity class that encompasses problems that are more difficult than PSPACE.

5.4 EXPTIME

EXPTIME is the class of languages that have algorithms that are solved in an exponential of the input. The algorithms of these languages are sometimes referred to as brute-force algorithms.

Definition *EXPTIME* is the class of languages A that are decidable in exponential time (i.e., EXPTIME = $\bigcup_k$ DTIME(k^n), where n is the length of an input to A and k is a constant).

EXPTIME is a superset of the previously mentioned complexity classes. So, it is generally believed that P $\subseteq$ NP $\subseteq$ PSPACE = NPSPACE $\subseteq$ EXPTIME. We will present examples of EXPTIME in the next section.

6 Classification of Complexity Results for Modal Logics

Using Sections 4 and 5 as a basis, we present complexity results of those logics presented in Section 4 with regards to satisfiabilty. We begin with a logic that is in EXPTIME-complete, to continue the explanation of EXPTIME that was started at the end of Section 5.

6.1 PDL

Recall from Section 4.2 that propositional dynamic logic (PDL) is a logic used for the verification of programs. As with uni-modal logics, PDL uses propositional logic as a foundation. The main modal operater is $[\alpha]$, where α is a program. Also, recall that $\Pi = \{a_1, a_2, a_3, \ldots\}$ is defined as a non-empty set of atomic programs. Programs are built inductively. Let α and β be programs and φ be a modal formula. Then, $\alpha;\beta, \alpha \cup \beta, \varphi?, \alpha^*$ are programs. Refer back to Section 4.2 for more of a review.

Based on the definitions of EXPTIME and completeness (not to be confused with modal completeness), EXPTIME-complete describes the hardest problems in EXPTIME. PDL-satisfiability, or PDL-SAT, is the problem that determines whether a formula is satisfiable in some given state in some given modal structure. Fischer, Ladner, and Pratt show that PDL-SAT is EXPTIME-complete [11, 30].

Theorem 6.1 *[11, 30] PDL-SAT is EXPTIME-complete.*

We will sketch the ideas behind the proof of Theorem 6.1. We explain the concepts behind the proof without going into in-depth details. The complexity of the deterministic Turing machine that decides PDL-SAT shows PDL-SAT to be in EXPTIME. As with previously introduced logics, there is a finite model property for PDL.

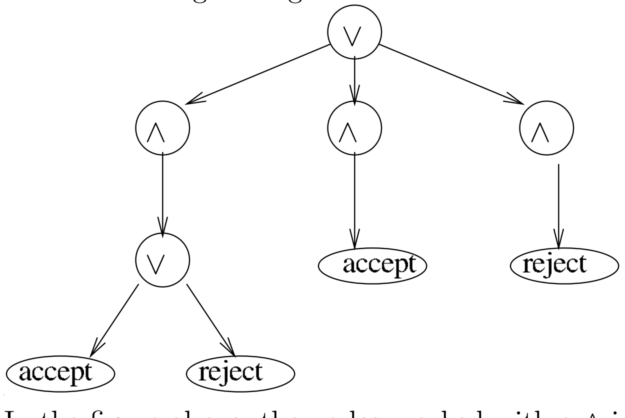
Proposition 6.2 *[11] If φ is a satisfiable PDL formula, then φ is satisfiable in a modal structure that has at most $2^{|\varphi|}$ states.*

We now give a sketch of the proof of Proposition 6.2. Fischer and Ladner presented a technique, called the filtration process, that essentially proves Proposition 6.2. Given a modal structure M such that φ is satisfiable in M, those states that have the same truth assignments for all subformulas of φ can be collapsed into a single state. We know that $||\text{sub}(\varphi)|| \leq |\varphi|$. So, the size of the power set of sub(φ), which is at most $2^{|\varphi|}$, gives the greatest number of states possible in the collapsed M.

With this property, a naïve algorithm would guess a structure $M = (W, R, V)$ such that $||W||$ is at most $2^{|\varphi|}$. The complexity of this algorithm is nondeterministic exponential time (NEXPTIME), since we are guessing a modal structure that follows the finite modal property. Since it is not certain that NEXPTIME = EXPTIME, this algorithm will not be deterministic.

However, there is a solution to this problem. Let M_u be a structure such that every formula φ that is PDL-consistent is satisfied at some state in M_u. Berman, Kozen, and Pratt each showed that such a structure does exist [4, 23, 31]. The deterministic algorithm presented by Pratt uses the filtration process on M_u with respect to a given formula φ. This produces a new structure M_φ that contains at most $2^{|\varphi|}$ states. M_φ is simply the ending product of M_u after the filtration process has finished. Pratt shows that the process does not take more than $\mathcal{O}(2^{|\varphi|})$ steps. Using the model-checking problem, we can find if φ is satisfiable in M_φ easily. Thus, the improved algorithm has a complexity of $\mathcal{O}(2^{|\varphi|})$.

We now present alternating Turing machines as an aid to considering the complexity of PDL. Recall how a nondeterministic Turing machine can be simulated by a deterministic Turing machine. At each of the steps that require a "guess," the deterministic Turing machine would run each possible computation path until an accepting state is reached. Alternating Turing machines encompass nondeterminism along with an additional feature. Within computation trees of alternating Turing machines, we label each node as universal or existential. Existential nodes are basically points of nondeterminism. For a universal node to be true, each possible computation path must accept. Below is an example of a computation tree for a possible alternating Turing machine.



In the figure above, the nodes marked with a $\wedge$ indicate universal nodes and nodes marked with a $\vee$ are existential nodes.

There are also complexity classes based off of alternating Turing machines. For example, P, PSPACE, and EXPTIME are based off of deterministic Turing machines and NP is based off of nondeterministic Turing machines. ATIME($f(n)$) is the set of alternating Turing machines that have a time complexity of $f(n)$,and ASPACE($f(n)$) is the set of alternating Turing machines that have a space complexity of $f(n)$. How do these "newer" complexity classes compare with established complexity classes?

Theorem 6.3 *[5] For $f(n) \geq n$, ASPACE$(f(n))$ = DTIME$(2^{\mathcal{O}(f(n))})$.*

That is, each alternating Turing machines that has a linear space complexity can be converted into a deterministic Turing machines that has a time complexity within a single exponent of the input, and vice versa. This theorem becomes helpful when explaining the upper bound of PDL-SAT.

Fischer and Ladner showed PDL-SAT to be EXPTIME-hard by constructing a formula φ of PDL such that the structure encodes the computation tree of a given linear-space bounded one-tape alternating Turing machine M on a given input of length n over M's input alphabet. Because of Theorem 5.1, we would be showing PDL-SAT is harder than any problem in EXPTIME.

6.1.1 Variations of PDL

Now we will list some extensions of PDL and the different complexities for each. We will mainly focus on PDL extensions that are decidable. It comes as no surprise that PDL variations are mainly due to differences in the definition of $L_{PDL}(\Pi)$, the least set of programs built inductively from Π. Some variations include enforcing deterministic programming features, like while loops or entire programs; allowing concurrency, communication, and/or well-foundedness, that is, where each possible computation path must accept; and allowing only automatas or context-free programs. We will present a few explanations of how these possibilities would be represented formally and the complexity of each.

We will first explore variations of determinism. Deterministic PDL (DPDL) is syntactically identical to PDL but interpreted over deterministic structures only. A structure $M = (W, R, V)$ is deterministic if for all $(s,t), (s,t') \in R$, $t = t'$. This essentially means that if $<a>\ \varphi$ for some $a \in \Pi$ and $\varphi \in L(\Phi)$, then $[a]\varphi$ is also true.

Strict PDL (SPDL) is the name given to the PDL variation in which only deterministic while programs are allowed. Deterministic while programs is a name for a set of programs described by the following:

- operators $\cup$, ?, and * may appear only in the context of conditional test, while loop, skip, or fail,
- tests in conditional test and while loop have no occurence of $<>$ or $[\,]$ operators.

SPDL, unlike DPDL, restricts only the syntax of PDL.

Satisfiability in DPDL and SPDL, defined as it is in PDL, are both EXPTIME-complete [2, 42]. The upper bound for DPDL is shown in Ben-Ari et. al. The upper bound for SPDL is the same as the one for PDL, since the semantics does not change. The lower bound for each is shown by a simple reduction from PDL to DPDL (SPDL) [17].

A more restrictive variation of PDL is to combine the elements of SPDL and DPDL to obtain SDPDL. Unlike the other two extensions, satisfiability of SDPDL is PSPACE-complete [14].

Peleg defined concurrent-PDL to be an extension of PDL that includes a new program operator allowing two programs to be executed in parallel [29]. If α and β are programs, then $\alpha \wedge \beta$ is a program. Peleg experiments with different degrees of communication among programs executed in parallel as well. He showed concurrent-PDL to be EXPTIME-complete. Read [29] for more details.

Automata-PDL, or APDL, takes a different approach to expressing programs. As programs in PDL are really just regular expressions, we can use finite automata to decide programs in $L_{PDL}(\Pi)$. Some researchers have tried replacing programs in formulas with their corresponding automatas [31, 18]. The representation of programs was found to not have a significant effect on the complexity of APDL. APDL-satisfiability is EXPTIME-complete.

The following is a list of accumulated logics and complexity classes.

- context-free-PDL: undecidable [24]
- PDL with complementation operator added to program operators: undecidable [16]
- IPDL (PDL with intersection operator for programs): undecidable [16]
- IDPDL (DPDL with intersection operator for programs): undecidable [16]

- CPDL (PDL with converse operator, which allows a program to be run backwards): EXPTIME-complete [43]
- RPDL (PDL where programs can be augmented with **wf** predicate, which asserts well-foundedness): EXPTIME-complete [9, 34]
- LPDL (PDL where programs can be augmented with **halt** predicate, which asserts all computations of its argument α terminate): EXPTIME-complete [9, 34]
- CRPDL (RPDL with converse operator): EXPTIME-complete [9, 34]
- CLPDL (LPDL with converse operator): EXPTIME-complete [9, 34]

6.2 Epistemic Logic

For epistemic logic, we will discuss the complexity of satisfiability in the logics $K_n, T_n, S4_n, S5_n$, and $KD45_n$. We will also list the complexities of epistemic logic with the addition of the common knowledge and/or distributed knowledge operators. Recall that X_n, where $X \in \{K, T, S4, S5, KD45\}$, is the logic X containing n modal operators.

The lower bounds for X_n-SAT, where $n \geq 1$ and $X \in \{K, T, S4\}$, and Y_n-SAT, where $n \geq 2$ and $Y \in \{S5, KD45\}$, are PSPACE-hard. This is immediate from Ladner's proof on the lower-bound of satisfiability for logics between K and $S4$[25]. Halpern and Moses show that X_n-SAT and Y_n-SAT are in PSPACE by proving that the algorithms for deciding satisfiability of X_n and Y_n run in polynomial space [15]. These algorithms use tableaus, which are constructed from a given formula φ, to determine if φ is satisfiable [15]. Recall from Section 4.1 that $L_n^D(\Phi)$ is an extension of $L_n(\Phi)$ that incorporates distributed knowledge. If φ is a formula, then $D\varphi$ is a formula. Adding the distributed knowledge operator does not change the complexity of X_n-SAT and Y_n-SAT due to the fact that adding a distributed knowledge operator is essentially adding a new modal operator that is considered the "wise" agent.

Adding the common knowledge operator to $K_n, T_n, S4_n, S5_n$, and $KD45_n$ adds an extra level of difficulty to the satisfiability problem. The problems X_n^C-SAT, where $X \in \{K, T, S4\}$ and $n \geq 1$, and Y_n^C-SAT, where $Y \in \{S5, KD45\}$ and $n \geq 2$, are each EXPTIME-complete [15]. The lower bound is shown by a reduction from PDL-SAT to X_n^C-SAT (Y_n^C-SAT). The upper bound is given by a proof in [15].

6.3 Temporal Logic and Deontic Logic

Just a few of the simplest temporal logics are listed below.

- K_t:[39] PSPACE-complete
- T_t:[39] PSPACE-complete
- $S4_t$:[39] PSPACE-complete

The logic presented in Section refdeontic, KD, is in between K and $S4$ [19]. Ladner showed that satisfiability in logic S, where S is in between K and $S4$, is PSPACE-complete [25].

Acknowledgments: We would like to thank the Laboratory for Applied Computing for its support of this project.

References

[1] L. Åqvist, Deontic Logic. In D.M. Gabbay & F. Guenther (eds.), *Handbook of Philosophical Logic Vol. II:* 605-714, Reidel, Dordrecht / Boston, 1984.

[2] M. Ben-Ari, J.Y. Halpern, and A. Pnueli, Deterministic propositional dynamic logic: finite models, complexity, and completeness. In *J. Comput. Sci.* **25:** 402-417, 1982.

[3] J. van Benthem, *The Logic of Time: a model-theoretic investigation into the varieties of temporal ontology and temporal discourse: second edition,* Kluwer Academic Publishers: Dordrecht, 1991.

[4] A completeness technique for D-axiomatizable semantics. In *Proc. 11th Symp. Theory of Comput.:* 160-166. ACM. 1979.

[5] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer, Alternation. In *J. ACM 28* **1**: 114-133,1981.

[6] B.F. Chellas, *Modal Logic.* Cambridge University Press: Cambridge, U.K. 1980.

[7] S.A. Cook, The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing:* 151-158, 1971.

[8] S. Goldwasser, S. Micali, and C. Rackoff, The knowledge complexity of interactive proof systems. In *SIAM Journal on Computing,* **18(1):** 186-208, 1989.

[9] E.A. Emerson and C. Jutla, The complexity of tree automata and logics of programs. In *Proc. 29th Symp. Foundations of Comput. Sci.,* IEEE, 1988.

[10] R. Fagin, J.Y. Halpern, and M.Y. Vardi, What can machines know? On the properties of knowledge in distributed systems. In *Journal of the ACM* **39(2):** 328-376, 1992.

[11] M.J. Fischer and R.E. Ladner, Propositional dynamic logic of regular programs. In *J. Comput. Syst. Sci.* **18(2):** 194-211, 1979.

[12] J.Y. Halpern and Y. Moses, Knowledge and common knowledge in a distributed environment. In *Journal of the ACM* **37(3):** 549-587, 1990. A preliminary version appeared in *Proc. 3rd ACM Symposium on Principles of Distributed Computing,* 1984.

[13] J.Y. Halpern and R. Fagin, Modelling knowledge and action in distributed systems. In *Distributed Computing* **3(4):** 159-179, 1989. A preliminary version appeared in *Proc. 4th ACM Symposium on Principles of Distributed Computing,* 1985 with a title "A formal model of knowledge, action, and communication in distributed systems: Preliminary report."

[14] J.Y. Halpern and J.H. Reif, The propositional dynamic logic of deterministic, well-structured programs. In *Theor. Comput. Sci.* **27:** 127-165, 1983.

[15] Joseph Y. Halpern and Yoram Moses, A guide to completeness and complexity for modal logics of knowledge and belief. In *Artificial Intelligence* **54**: 311-379, 1992.

[16] D. Harel, A. Pnueli, and M. Vardi, Two dimensional temporal logic and PDL with intersection. Unpublished, 1982.

[17] D. Harel, D. Kozen, and J. Tiuryn, *Dynamic Logic,* The MIT Press: Cambridge, MA, 2000.

[18] D. Harel and R. Sherman, Propositional dynamic logic of flowcharts. In *Infor. and Control* **64:** 119-135, 1985.

[19] R. Hilpinen, *Deontic Logic: Introductory and Systematic Readings,* Dordrecht: D. Reidel, 1971.

[20] J. Hintikka, *Knowledge and Belief,* Cornell University Press: Ithaca, NY. 1962.

[21] G.E. Hughes and M.J. Cresswell, *An Introduction to Modal Logic.* Methuen: London. 1968.

[22] S. Kripke, A semantical analysis of modal logic I: normal modal propositional calculi. In *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik,* **9:** 67-96, 1963. Announced in *Journal of Symbolic Logic* **24**: 323, 1959.

[23] D. Kozen, Logics of programs. Lecture notes, Aarhus University: Denmark, 1981.

[24] R.E. Ladner, Unpublished, 1977.

[25] R. E. Ladner, The computational complexity of provability in systems of modal propositional logic. In *SIAM Journal on Computing* **14(1)**: 113-118, 1981.

[26] L. Levin, Universal search problems (in Russian). In *Problemy Peredachi Informatsii* **9(3):** 115-116, 1973.

[27] P. Milgrom, An axiomatic characterization of common knowledge. In *Econometrica* **49(1):** 219-222, 1981.

[28] C.H. Papadimitrou, *Computational Complexity,* Addison-Wesley Publishing Company: Reading, MA, 1994.

[29] D. Peleg, Concurrent Dynamic Logic. In *Journal of the Association for Computing Machinery* **34(2):** 450-479, 1987.

[30] V. Pratt, Semantical Considerations on Floyd-Hoare logic. In *Proc. 17th IEEE Symp. on Foundations of Computer Science:* 109-121, 1976.

[31] V.R. Pratt, Dynamic algebras and the nature of induction. In *Proc. 12th Symp. Theory of Comput.:* 22-28, ACM, 1980.

[32] A. Prior, *Past, Present, and Future,* Clarendon Press: Oxford, 1967.

[33] R. Reiter, On integrity constraints. In M.Y. Vardi, editor, *Proc. Second Conference on Theoretical Aspects of Reasoning about Knowledge:* 97-112. Morgan Kaufmann: San Francisco, CA, 1988.

[34] S. Safra, On the complexity of ω-automata. In *Proc. 29th Symp. Foundations of Comput. Sci.:* 319-327, IEEE, 1988.

[35] M. Sato, A study of Kripke-style methods for some modal logics by Gentzen's sequential method. In *Publications Research Institute for Mathematical Sciences, Kyoto University* **13(2),** 1977.

[36] W.J. Savitch, Relationships between nondeterministic and deterministic tape complexities. In *JCSS* **4(2):** 177-192, 1970.

[37] K. Segerberg, A completeness theorem in the modal logic of programs (preliminary report). In *Not. Amer. Math. Soc.* **24(6):** A-552, 1977.

[38] Michael Sipser, *Introduction to the Theory of Computation,* PWS Publishing Company: Boston, 1997.

[39] E. Spaan, The complexity of propositional tense logics. In M. de Rijke (ed) *Diamonds and Defaults, Studies in Logic, Language and Information,* Kluwer, Dordrecht, 1993.

[40] L.J. Stockmeyer and A.R. Meyer, Word problems requiring exponential time: preliminary report. In *Proc.* 5^{th} *ACM Symp. on Theory of Computing:* 1-9, 1973.

[41] A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings, London Mathematical Society:* 230-256, 1936.

[42] M.K. Valiev, Decision complexity of variants of propositional dynamic logic. In *Proc. 9th Symp. Math. Found. Comput. Sci.:* Volume 88 of *Lect. Notes in Comput. Sci.:* 656-664, Springer-Verlag, 1980.

[43] M.Y. Vardi, Reasoning about the past with two-way automata. In *Proc. 25th Int. Colloq. Automata Lang. Prog.:* Volume 1443 of *Lect. Notes in Comput. Sci.:* 628-641, Springer-Verlag, 1998.

[44] R.J. Wieringa and J.-J. Ch. Meyer, Applications of Deontic Logic in Computer Science: A Concise Overview. In *Deontic Logic in Computer Science: Normative System Specification:* 17-40, John Wiley & Sons: Chichester, 1993.

[45] Y. Yemini and D. Cohen, Some issues in distributed processes communication. In *Proc. of the 1st International Conf. on Distributed Computing Systems:* 199-203, 1979.