# Logic and Games
# WS 2015/2016

Prof. Dr. Erich Grädel
Notes and Revisions by Matthias Voit

Mathematische Grundlagen der Informatik
RWTH Aachen

# Contents

1 Reachability Games and First-Order Logic 1
1.1 Model Checking . . . 1
1.2 Model Checking Games for Modal Logic . . . 2
1.3 Reachability and Safety Games . . . 5
1.4 Games as an Algorithmic Construct: Alternating Algorithms . 10
1.5 Model Checking Games for First-Order Logic . . . 20

2 Parity Games and Fixed-Point Logics 25
2.1 Parity Games . . . 25
2.2 Algorithms for parity games . . . 30
2.3 Fixed-Point Logics . . . 35
2.4 Model Checking Games for Fixed-Point Logics . . . 37
2.5 Defining Winning Regions in Parity Games . . . 42

3 Infinite Games 45
3.1 Determinacy . . . 45
3.2 Gale-Stewart Games . . . 47
3.3 Topology . . . 53
3.4 Determined Games . . . 59
3.5 Muller Games and Game Reductions . . . 61
3.6 Complexity . . . 74

4 Basic Concepts of Mathematical Game Theory 79
4.1 Games in Strategic Form . . . 79
4.2 Nash equilibria . . . 81
4.3 Two-person zero-sum games . . . 85
4.4 Regret minimization . . . 86
4.5 Iterated Elimination of Dominated Strategies . . . 89
4.6 Beliefs and Rationalisability . . . 95

4.7 Games in Extensive Form . . . . . . . . . . . . . . . . . . . . . . 98
4.8 Subgame-perfect equilibria in infinite games . . . . . . . . . . . 102

Appendix A 111
4.9 Cardinal Numbers . . . . . . . . . . . . . . . . . . . . . . . . . . 119

# 1 Reachability Games and First-Order Logic

## 1.1 Model Checking

One of the fundamental algorithmic tasks in logic is *model checking*. For a logic $L$ and a domain $\mathcal{D}$ of (finite) structures, the *model-checking problem* asks, given a structure $\mathfrak{A} \in \mathcal{D}$ and a formula $\psi \in L$, whether $\mathfrak{A}$ is a model of $\psi$. Notice that an instance of the model-checking problem has two inputs: a structure and a formula. We can measure the complexity in terms of both inputs, and this is what is commonly refered to as the *combined complexity* of the model-checking problem (for $L$ and $\mathcal{D}$). However, in many cases, one of the two inputs is fixed, and we measure the complexity only in terms of the other. If we fix the structure $\mathfrak{A}$, then the model-checking problem for $L$ on this structure amounts to deciding $\mathrm{Th}_L(\mathfrak{A}) := \{\psi \in L : \mathfrak{A} \models \psi\}$, the *L-theory* of $\mathfrak{A}$. The complexity of this problem is called the *expression complexity* of the model-checking problem (for $L$ on $\mathfrak{A}$). For first-order logic (FO) and for monadic second-order logic (MSO) in particular, such problems have a long tradition in logic and numerous applications in many fields. Of great importance in many areas of logic, in particular for finite model theory or databases, are model-checking problems for a fixed formula $\psi$, which amounts to deciding the *model class* of $\psi$ inside $\mathcal{D}$, that is $\mathrm{Mod}_{\mathcal{D}}(\psi) := \{\mathfrak{A} \in \mathcal{D} : \mathfrak{A} \models \psi\}$. Its complexity is the *structure complexity* or *data complexity* of the model-checking problem (for $\psi$ on $\mathcal{D}$).

One of the important themes in this course is a game-based approach to model checking. The general idea is to reduce the problem whether $\mathfrak{A} \models \psi$ to a strategy problem for a *model checking game* $\mathcal{G}(\mathfrak{A}, \psi)$ played by two players called *Verifier* (or *Player* 0) and *Falsifier* (or *Player* 1). We want to have the following relation between these two problems:

$$\mathfrak{A} \models \psi \text{ iff } \text{ Verifier has a winning strategy for } \mathcal{G}(\mathfrak{A}, \psi).$$

We can then do model checking by constructing, or proving the existence of, winning strategies.

To assess the efficiency of games as a solution for model checking problems, we have to consider the complexity of the resulting model checking games based on the following criteria:

- Are all plays necessarily finite?
- If not, what are the winning conditions for infinite plays?
- Do the players always have perfect information?
- What is the structural complexity of the game graphs?
- How does the size of the graph depend on different parameters of the input structure and the formula?

For first-order logic (FO) and modal logic (ML) we have only finite plays with positional winning conditions, and, as we shall see, the winning regions are computable in linear time with respect to the size of the game graph. Model checking games for fixed-point logics however admit infinite plays, and we use so-called *parity conditions* to determine the winner of such plays. It is still an open question whether winning regions and winning strategies in parity games are computable in polynomial time.

## 1.2 Model Checking Games for Modal Logic

The first logic that we discuss is propositional modal logic (ML). Let us first briefly review its syntax and semantics:

**Definition 1.1.** Given a set $A$ of actions and a set $\{P_i : i \in I\}$ of atomic propositions, the set of formulae of ML is inductively defined:

- All atomic propositions $P_i$ are formulae of ML.
- If $\psi, \varphi$ are formulae of ML, then so are $\neg\psi$, $(\psi \wedge \varphi)$ and $(\psi \vee \varphi)$.
- If $\psi \in$ ML and $a \in A$, then $\langle a \rangle \psi \in$ ML and $[a]\psi \in$ ML.





*Remark* 1.2. If there is only one action $a \in A$, we write $\Diamond\psi$ and $\Box\psi$ instead of $\langle a \rangle \psi$ and $[a]\psi$, respectively.

**Definition 1.3.** A *transition system* or *Kripke structure* with actions from a set $A$ and atomic properties $\{P_i : i \in I\}$ is a structure

$$\mathcal{K} = (V, (E_a)_{a \in A}, (P_i)_{i \in I})$$

with a universe $V$ of states, binary relations $E_a \subseteq V \times V$ describing transitions between the states, and unary relations $P_i \subseteq V$ describing the atomic properties of states.

A transition system can be seen as a labelled graph where the nodes are the states of $\mathcal{K}$, the unary relations provide labels of the states, and the binary transition relations can be pictured as sets of labelled edges.

**Definition 1.4.** Let $\mathcal{K} = (V, (E_a)_{a \in A}, (P_i)_{i \in I})$ be a transition system, $\psi \in$ ML a formula and $v$ a state of $\mathcal{K}$. The *model relationship* $\mathcal{K}, v \models \psi$, i.e., $\psi$ holds at state $v$ of $\mathcal{K}$, is inductively defined:

- $\mathcal{K}, v \models P_i$ if and only if $v \in P_i$.
- $\mathcal{K}, v \models \neg\psi$ if and only if $\mathcal{K}, v \not\models \psi$.
- $\mathcal{K}, v \models \psi \vee \varphi$ if and only if $\mathcal{K}, v \models \psi$ or $\mathcal{K}, v \models \varphi$.
- $\mathcal{K}, v \models \psi \wedge \varphi$ if and only if $\mathcal{K}, v \models \psi$ and $\mathcal{K}, v \models \varphi$.
- $\mathcal{K}, v \models \langle a \rangle \psi$ if and only if there exists $w$ such that $(v, w) \in E_a$ and $\mathcal{K}, w \models \psi$.
- $\mathcal{K}, v \models [a]\,\psi$ if and only if $\mathcal{K}, w \models \psi$ holds for all $w$ with $(v, w) \in E_a$.

For a transition system $\mathcal{K}$ and a formula $\psi$ we define the *extension* $[\![\psi]\!]^{\mathcal{K}} := \{v : \mathcal{K}, v \models \psi\}$ as the set of states of $\mathcal{K}$ where $\psi$ holds.

For the game-based approach to model-checking, it is convenient to assume that modal formulae are written in negation normal form, i.e. negation is applied to atomic propositions only. This does not reduce the expressiveness of modal logic since every formula can be efficiently translated into negation normal form by applying De Morgan's laws and the duality of $\Box$ and $\Diamond$ (i.e. $\neg\langle a \rangle \psi \equiv [a]\neg\psi$ and $\neg[a]\psi \equiv \langle a \rangle \neg\psi$) to push negations to the atomic subformulae.

Syntactically, modal logic is an extension of propositional logic. However, since ML is evaluated over transition systems, i.e. structures, it is often useful to see it as a fragment of first-order logic.

**Theorem 1.5.** For each formula $\psi \in \mathrm{ML}$ there is a first-order formula $\psi^*(x)$ (with only two variables), such that for each transition system $\mathcal{K}$ and all its states $v$ we have that $\mathcal{K}, v \models \psi \iff \mathcal{K} \models \psi^*(v)$.

*Proof.* The transformation is defined inductively, as follows:

$$\begin{aligned} P_i &\longmapsto P_i x \\ \neg\psi &\longmapsto \neg\psi^*(x) \\ (\psi \circ \varphi) &\longmapsto (\psi^*(x) \circ \varphi^*(x)), \text{ where } \circ \in \{\wedge, \vee, \rightarrow\} \\ \langle a \rangle \psi &\longmapsto \exists y (E_a xy \wedge \psi^*(y)) \\ [a]\psi &\longmapsto \forall y (E_a xy \rightarrow \psi^*(y)) \end{aligned}$$

where $\psi^*(y)$ is obtained from $\psi^*(x)$ by interchanging $x$ and $y$ everywhere in the formula. Q.E.D.

We are now ready to describe the model checking games for ML. Given a transition system $\mathcal{K}$ and a formula $\psi \in \mathrm{ML}$, we define a game $\mathcal{G}(\mathcal{K}, \psi)$ whose positions are pairs $(\varphi, v)$ where $\varphi$ is a subformula of $\psi$ and $v \in V$ is a node of $\mathcal{K}$. From any position $(\varphi, v)$ in this game, Verifier's goal is to show that $\mathcal{K}, v \models \varphi$, whereas Falsifier tries to establish that $\mathcal{K}, v \not\models \varphi$.

In the game, Verifier moves at positions of the form $(\varphi \vee \vartheta, v)$, with the choice to move either to $(\varphi, v)$ or to $(\vartheta, v)$, and at positions $(\langle a \rangle \varphi, v)$, where she can move to any position $(\varphi, w)$ with $w \in vE_a$. Analogously, Falsifier moves from positions $(\varphi \wedge \vartheta, v)$ to either $(\varphi, v)$ or $(\vartheta, v)$, and from $([a]\varphi, v)$ to any position $(\varphi, w)$ with $w \in vE_a$. Finally, at literals, i.e. if $\varphi = P_i$ or $\varphi = \neg P_i$, the position $(\varphi, v)$ is a terminal position where Verifier has won if $\mathcal{K}, v \models \varphi$, and Falsifier has won if $\mathcal{K}, v \not\models \varphi$.

The correctness of the construction of $\mathcal{G}(\mathcal{K}, \psi)$ follows readily by induction.

**Proposition 1.6.** For any position $(\varphi, v)$ of $\mathcal{G}(\mathcal{K}, \psi)$ we have that

$$\mathcal{K}, v \models \varphi \quad \Leftrightarrow \quad \text{Verifier has a winning strategy for } \mathcal{G}(\mathcal{K}, \psi) \text{ from } (\varphi, v).$$





## 1.3 Reachability and Safety Games

The model-checking games for propositional modal logic, that we have just discussed, are an instance of reachability games played on graphs or, more precisely, two-player games with perfect information and positional winning conditions, played on a *game graph* (or *arena*)

$$\mathcal{G} = (V, V_0, V_1, E)$$

where the set $V$ of positions is partitioned into sets of positions $V_0$ and $V_1$ belonging to Player 0 and Player 1, respectively. Player 0 moves from positions $v \in V_0$, while Player 1 moves from positions $v \in V_1$. All moves are along edges, and so the interaction of the players, starting from an initial position $v_0$, produces a finite or infinite *play* which is a sequence $v_0 v_1 v_2 \ldots$ with $(v_i, v_{i+1}) \in E$ for all $i$.

The winning conditions of the players are based on a simple positional principle: Move or lose! This means that Player $\sigma$ has won at a position $v$ in the case that position $v$ belongs to his opponent and there are no moves available from that position. Thus the goal of Player $\sigma$ is to reach a position in $T_\sigma := \{v \in V_{1-\sigma} : vE = \emptyset\}$. We call this a *reachability condition*.

But note that this winning condition applies to finite plays only. If the game graph admits infinite plays (for instance cycles) then we must either consider these as draws, or introduce a winning condition for infinite plays. The dual notion of a reachability condition is a *safety condition* where Player $\sigma$ just has the objective to avoid a given set of 'bad' positions, which in this case is the set $T_{1-\sigma}$, and to remain inside the safe region $V \setminus T_{1-\sigma}$.

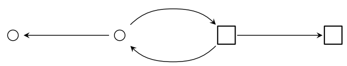
A *(positional) strategy* for Player $\sigma$ in such a game $\mathcal{G}$ is a (partial) function $f : \{v \in V_\sigma : vE \neq \emptyset\} \rightarrow V$ such that $(v, f(v)) \in E$. A finite or infinite play $v_0 v_1 v_2 \ldots$ is *consistent with* $f$ if $v_{i+1} = f(v_i)$ for every $i$ such that $v_i \in V_\sigma$. A strategy $f$ for Player $\sigma$ is winning from $v_0$ if every play that starts at initial position $v_0$ and that is consistent with $f$ is won by Player $\sigma$.

We first consider *reachability games* where both players play with the reachability objective to force the play to a position in $T_\sigma$. We define *winning regions*

$$W_\sigma := \{v \in V : \text{Player } \sigma \text{ has a winning strategy from position } v\}.$$

If $W_0 \cup W_1 = V$, i.e. for each $v \in V$ one of the players has a winning strategy, the game $\mathcal{G}$ is called *determined*. A play which is not won by any of the players is considered a draw.

*Example* 1.7. No player can win from one of the middle two nodes:



The winning regions of a reachability game $\mathcal{G} = (V, V_0, V_1, E)$ can be constructed inductively as follows:

$$W_\sigma^0 = T_\sigma \text{ and}$$
$$W_\sigma^{i+1} = W_\sigma^i \cup \{v \in V_\sigma : vE \cap W_\sigma^i \neq \emptyset\} \cup \{v \in V_{1-\sigma} : vE \subseteq W_\sigma^i\}.$$

Clearly $W_\sigma^i$ is the region of those positions from which Player $\sigma$ has a strategy to win in at most $i$ moves, and for finite game graphs, with $|V| = n$, we have that $W_\sigma = W_\sigma^n$.

Next we consider the case of a *reachability-safety game*, where Player 0, as above, plays with the reachability objective to force the play to a terminal position in $T_0$, whereas player 1 plays with the safety objective of avoiding $T_0$, i.e. to keep the play inside the safe region $S_1 := V \setminus T_0$. Notice that there are no draws in such a game.

The winning region $W_0$ of Player 0 can be defined as in the case above, but the winning region $W_1$ of Player 1 is now the maximal set $W \subseteq S_1$ such that from all $w \in W$ Player 1 has a strategy to remain inside $W$, which can be defined as the limit of the descending chain $W_1^0 \supseteq W_1^1 \supseteq W_1^2 \supseteq \dots$ with

$$W_1^0 = S_1 \text{ and}$$
$$W_1^{i+1} = W_1^i \cap \{v \in V : (v \in V_0 \text{ and } vE \subseteq W_1^i) \text{ or } (v \in V_1 \text{ and } vE \cap W_1^i \neq \emptyset)\}.$$





Again on finite game graphs, with $|V| = n$, we have that $W_1 = W_\sigma^n$.

This leads us to two fundamental concepts for the analysis of games on graphs: *attractors* and *traps*. Let $\mathcal{G} = (V, V_0, V_1, E)$ be a game graph and $X \subseteq V$.

**Definition 1.8.** The *attractor of $X$ for Player $\sigma$*, in short $\mathrm{Attr}_\sigma(X)$ is the set of those positions from which Player $\sigma$ has a strategy to reach $X$ (or to win because the opponent cannot move anymore). We can inductively define $\mathrm{Attr}_\sigma(X) := \bigcup_{n \in \mathbb{N}} X^n$, where

$$X^0 = X \text{ and}$$
$$X^{i+1} = X^i \cup \{v \in V_\sigma : vE \cap X^i \neq \emptyset\} \cup \{v \in V_{1-\sigma} : vE \subseteq X^i\}.$$

For instance, the winning region $W_\sigma$ in a reachability game is the attractor of the winning positions: $W_\sigma = \mathrm{Attr}_\sigma(T_\sigma)$.

A set $Y \subseteq V \setminus T_{1-\sigma} =: S_\sigma$ is called a *trap* for Player $1 - \sigma$ if Player $\sigma$ has a strategy to guarantee that from each $v \in Y$ the play will remain inside $Y$. Note that the complement of an attractor $\mathrm{Attr}_\sigma(X)$ is a trap for player $\sigma$. The maximal trap $Y$ of Player $1 - \sigma$ can be defined as $Y = \bigcap_{n \in \mathbb{N}} Y^n$, where

$$Y^0 = S_\sigma \text{ and}$$
$$Y^{i+1} = Y^i \cap \{v : (v \in V_\sigma \text{ and } vE \cap Y^i \neq \emptyset) \text{ or } (v \in V_{1-\sigma} \text{ and } vE \subseteq Y^i)\}.$$

The winning region of a Player $\sigma$ with the safety objective for $S_\sigma$ is the maximal trap for player $1 - \sigma$.

We consider several algorithmic problems for a given reachability game $\mathcal{G}$: The computation of winning regions $W_0$ and $W_1$, the computation of winning strategies, and the associated decision problem

$$\textsc{Game} := \{(\mathcal{G}, v) : \text{Player 0 has a winning strategy for } \mathcal{G} \text{ from } v\}.$$

**Theorem 1.9.** Game is P-complete and decidable in time $O(|V| + |E|)$.

Note that this remains true for *strictly alternating games*.

**Algorithm 1.1.** A linear time algorithm for GAME

**Input**: A game $\mathcal{G} = (V, V_0, V_1, E)$
**output**: Winning regions $W_0$ and $W_1$

```
for all v ∈ V do                    (* 1: Initialisation *)
    win[v] := ⊥
    P[v] := {u : (u, v) ∈ E}
    n[v] := |vE|
end do

for all v ∈ V_0                     (* 2: Calculate win *)
    if n[v] = 0 then Propagate(v, 1)
for all v ∈ V_1
    if n[v] = 0 then Propagate(v, 0)
return win

procedure Propagate(v, σ)
    if win[v] ≠ ⊥ then return
    win[v] := σ                     (* 3: Mark v as winning for player σ *)
    for all u ∈ P[v] do             (* 4: Propagate change to predecessors *)
        n[u] := n[u] − 1
        if u ∈ V_σ or n[u] = 0 then Propagate(u, σ)
    end do
end
```





The inductive definition of an attractor shows that winning regions for both players can be computed efficiently. Hence we can also solve GAME in polynomial time. To solve GAME in linear time, we use the slightly more involved Algorithm 1.1. Procedure Propagate will be called once for every edge in the game graph, so the running time of this algorithm is linear with respect to the number of edges in $\mathcal{G}$.

Furthermore, we can show that the decision problem GAME is equivalent to the satisfiability problem for propositional Horn formulae. We recall that propositional Horn formulae are finite conjunctions $\bigwedge_{i \in I} C_i$ of clauses $C_i$ of the form

$$\begin{aligned} X_1 \wedge \ldots \wedge X_n &\rightarrow X \quad \text{or} \\ \underbrace{X_1 \wedge \ldots \wedge X_n}_{\text{body}(C_i)} &\rightarrow \underbrace{0}_{\text{head}(C_i)} . \end{aligned}$$

A clause of the form $X$ or $1 \rightarrow X$ has an empty body.

We will show that SAT-HORN and GAME are mutually reducible via logspace and linear-time reductions.

(1) GAME $\leq_{\text{log-lin}}$ SAT-HORN
For a game $\mathcal{G} = (V, V_0, V_1, E)$, we construct a Horn formula $\psi_{\mathcal{G}}$ with clauses

$$\begin{aligned} v &\rightarrow u && \text{for all } u \in V_0 \text{ and } (u, v) \in E, \text{ and} \\ v_1 \wedge \ldots \wedge v_m &\rightarrow u && \text{for all } u \in V_1 \text{ and } uE = \{v_1, \ldots, v_m\}. \end{aligned}$$

The minimal model of $\psi_{\mathcal{G}}$ is precisely the winning region of Player 0, so

$$(\mathcal{G}, v) \in \text{GAME} \iff \psi_{\mathcal{G}} \wedge (v \rightarrow 0) \text{ is unsatisfiable.}$$

(2) SAT-HORN $\leq_{\text{log-lin}}$ GAME
For a Horn formula $\psi(X_1, \ldots, X_n) = \bigwedge_{i \in I} C_i$, we define a game $\mathcal{G}_\psi = (V, V_0, V_1, E)$ as follows:

$$V = \underbrace{\{0\} \cup \{X_1, \ldots, X_n\}}_{V_0} \cup \underbrace{\{C_i : i \in I\}}_{V_1} \text{ and}$$

$$E = \{X \to C_i : X = \text{head}(C_i)\} \cup \{C_i \to X_j : X_j \in \text{body}(C_i)\},$$

i.e., Player 0 moves from a variable to some clause containing the variable as its head, and Player 1 moves from a clause to some variable in its body. Player 0 wins a play if, and only if, the play reaches a clause $C$ with $\text{body}(C) = \emptyset$. Furthermore, Player 0 has a winning strategy from position $X$ if, and only if, $\psi \models X$, so we have

$$\text{Player 0 wins from position 0} \quad \Longleftrightarrow \quad \psi \text{ is unsatisfiable.}$$

These reductions show that Sat-Horn is also P-complete and, in particular, also decidable in linear time.

## 1.4 Games as an Algorithmic Construct: Alternating Algorithms

Alternating algorithms are algorithms whose set of configurations is divided into *accepting*, *rejecting*, *existential* and *universal* configurations. The acceptance condition of an alternating algorithm $A$ is defined by a game played by two players $\exists$ and $\forall$ on the computation graph $\mathcal{G}(A, x)$ (or equivalently, the computation tree $\mathcal{T}(A, x)$) of $A$ on input $x$. The positions in this game are the configurations of $A$, and we allow moves $C \to C'$ from a configuration $C$ to any of its successor configurations $C'$. Player $\exists$ moves at existential configurations and wins at accepting configurations, while Player $\forall$ moves at universal configurations and wins at rejecting configurations. By definition, $A$ accepts some input $x$ if and only if Player $\exists$ has a winning strategy for the game played on $\mathcal{T}_{A,x}$.

We will introduce the concept of alternating algorithms formally, using the model of a Turing machine, and we prove certain relationships between the resulting alternating complexity classes and usual deterministic complexity classes.

### 1.4.1 Turing Machines

The notion of an alternating Turing machine extends the usual model of a (deterministic) Turing machine which we introduce first. We consider





Turing machines with a separate input tape and multiple linear work tapes which are divided into basic units, called cells or fields. Informally, the Turing machine has a reading head on the input tape and a combined reading and writing head on each of its work tapes. Each of the heads is at one particular cell of the corresponding tape during each point of a computation. Moreover, the Turing machine is in a certain state. Depending on this state and the symbols the machine is currently reading on the input and work tapes, it manipulates the current fields of the work tapes, moves its heads and changes to a new state.

Formally, a *(deterministic) Turing machine with separate input tape and k linear work tapes* is given by a tuple $M = (Q, \Gamma, \Sigma, q_0, F_{\text{acc}}, F_{\text{rej}}, \delta)$, where $Q$ is a finite set of *states*, $\Sigma$ is the *work alphabet* containing a designated symbol $\square$ (*blank*), $\Gamma$ is the *input alphabet*, $q_0 \in Q$ is the *initial state*, $F := F_{\text{acc}} \cup F_{\text{rej}} \subseteq Q$ is the set of *final states* (with $F_{\text{acc}}$ the *accepting states*, $F_{\text{rej}}$ the *rejecting states* and $F_{\text{acc}} \cap F_{\text{rej}} = \emptyset$), and $\delta : (Q \setminus F) \times \Gamma \times \Sigma^k \to Q \times \{-1, 0, 1\} \times \Sigma^k \times \{-1, 0, 1\}^k$ is the *transition function*.

A *configuration* of $M$ is a complete description of all relevant facts about the machine at some point during a computation, so it is a tuple $C = (q, w_1, \dots, w_k, x, p_0, p_1, \dots, p_k) \in Q \times (\Sigma^*)^k \times \Gamma^* \times \mathbb{N}^{k+1}$ where $q$ is the current state, $w_i$ is the contents of work tape number $i$, $x$ is the contents of the input tape, $p_0$ is the position on the input tape and $p_i$ is the position on work tape number $i$. The contents of each of the tapes is represented as a finite word over the corresponding alphabet[, i.e., a finite sequence of symbols from the alphabet]. The contents of each of the fields with numbers $j > |w_i|$ on work tape number $i$ is the blank symbol (we think of the tape as being infinite). A configuration where $x$ is omitted is called a *partial configuration*. The configuration $C$ is called *final* if $q \in F$. It is called *accepting* if $q \in F_{\text{acc}}$ and *rejecting* if $q \in F_{\text{rej}}$.

The *successor configuration* of $C$ is determined by the current state and the $k + 1$ symbols on the current cells of the tapes, using the transition function: If $\delta(q, x_{p_0}, (w_1)_{p_1}, \dots, (w_k)_{p_k}) = (q', m_0, a_1, \dots, a_k, m_1, \dots, m_k, b)$, then the successor configuration of $C$ is $\Delta(C) = (q', \overline{w}', \overline{p}', x)$, where for any $i$, $w_i'$ is obtained from $w_i$ by replacing symbol number $p_i$ by $a_i$ and $p_i' = p_i + m_i$. We write $C \vdash_M C'$ if, and only if, $C' = \Delta(C)$.

The *initial configuration* $C_0(x) = C_0(M, x)$ of $M$ on input $x \in \Gamma^*$ is given by the initial state $q_0$, the blank-padded memory, i.e., $w_i = \varepsilon$ and $p_i = 0$ for any $i \geq 1$, $p_0 = 0$, and the contents $x$ on the input tape.

A *computation* of $M$ on input $x$ is a sequence $C_0, C_1, \ldots$ of configurations of $M$, such that $C_0 = C_0(x)$ and $C_i \vdash_M C_{i+1}$ for all $i \geq 0$. The computation is called *complete* if it is infinite or ends in some final configuration. A complete finite computation is called *accepting* if the last configuration is accepting, and the computation is called *rejecting* if the last configuration is rejecting. *M accepts* input $x$ if the (unique) complete computation of $M$ on $x$ is finite and accepting. *M rejects* input $x$ if the (unique) complete computation of $M$ on $x$ is finite and rejecting. The machine $M$ *decides* a language $L \subseteq \Gamma^*$ if $M$ accepts all $x \in L$ and rejects all $x \in \Gamma^* \setminus L$.

### 1.4.2 Alternating Turing Machines

Now we shall extend deterministic Turing machines to nondeterministic Turing machines from which the concept of alternating Turing machines is obtained in a very natural way, given our game theoretical framework.

A *nondeterministic Turing machine* is nondeterministic in the sense that a given configuration $C$ may have several possible successor configurations instead of at most one. Intuitively, this can be described as the ability to *guess*. This is formalised by replacing the transition function $\delta : (Q \setminus F) \times \Gamma \times \Sigma^k \to Q \times \{-1, 0, 1\} \times \Sigma^k \times \{-1, 0, 1\}^k$ by a transition relation $\Delta \subseteq ((Q \setminus F) \times \Gamma \times \Sigma^k) \times (Q \times \{-1, 0, 1\} \times \Sigma^k \times \{-1, 0, 1\}^k)$. The notion of successor configurations is defined as in the deterministic case, except that the successor configuration of a configuration $C$ may not be uniquely determined. Computations and all related notions carry over from deterministic machines in the obvious way. However, on a fixed input $x$, a nondeterministic machine now has several possible computations, which form a (possibly infinite) finitely branching computation tree $\mathcal{T}_{M,x}$. A nondeterministic Turing machine $M$ *accepts* an input $x$ if there *exists* a computation of $M$ on $x$ which is accepting, i.e., if there exists a path from the root $C_0(x)$ of $\mathcal{T}_{M,x}$ to some accepting configuration. The language of $M$ is $L(M) = \{x \in \Gamma^* \mid M \text{ accepts } x\}$. Notice that for a nondeterministic machine $M$ to decide a language $L \subseteq \Gamma^*$ it is not





necessary that all computations of $M$ are finite. (In a sense, we count infinite computations as rejecting.)

From a game-theoretical perspective, the computation of a nondeterministic machine can be viewed as a solitaire game on the computation tree in which the only player (the machine) chooses a path through the tree starting from the initial configuration. The player wins the game (and hence, the machine accepts its input) if the chosen path finally reaches an accepting configuration.

An obvious generalisation of this game is to turn it into a two-player game by assigning the nodes to the two players who are called $\exists$ and $\forall$, following the intuition that Player $\exists$ tries to show the existence of a *good* path, whereas Player $\forall$ tries to show that all selected paths are *bad*. As before, Player $\exists$ wins a play of the resulting game if, and only if, the play is finite and ends in an accepting leaf of the game tree. Hence, we call a computation tree accepting if, and only if, Player $\exists$ has a winning strategy for this game.

It is important to note that the partition of the nodes in the tree should not depend on the input $x$ but is supposed to be inherent to the machine. Actually, it is even independent of the contents of the work tapes, and thus, whether a configuration belongs to Player $\exists$ or to Player $\forall$ merely depends on the current state.

Formally, an *alternating Turing machine* is a nondeterministic Turing machine $M = (Q, \Gamma, \Sigma, q_0, F_{\text{acc}}, F_{\text{rej}}, \Delta)$ whose set of states $Q = Q_\exists \cup Q_\forall \cup F_{\text{acc}} \cup F_{\text{rej}}$ is partitioned into *existential*, *universal*, *accepting*, and *rejecting* states. The semantics of these machines is given by means of the game described above.

Now, if we let accepting configurations belong to player $\forall$ and rejecting configurations belong to player $\exists$, then we have the usual winning condition that a player loses if it is his turn but he cannot move. We can solve such games by determining the winner at leaf nodes and propagating the winner successively to parent nodes. If at some node, the winner at all of its child nodes is determined, the winner at this node can be determined as well. This method is sometimes referred to as backwards induction and it basically coincides with our method for solving GAME on trees (with possibly infinite plays). This gives the following equivalent semantics of alternating Turing machines:

The subtree $\mathcal{T}_C$ of the computation tree of $M$ on $x$ with root $C$ is called *accepting*, if

- $C$ is accepting
- $C$ is existential and there is a successor configuration $C'$ of $C$ such that $\mathcal{T}_{C'}$ is accepting or
- $C$ is universal and $\mathcal{T}_{C'}$ is accepting for all successor configurations $C'$ of $C$.

$M$ accepts an input $x$, if $\mathcal{T}_{C_0(x)} = \mathcal{T}_{M,x}$ is accepting.

For functions $T, S : \mathbb{N} \to \mathbb{N}$, an alternating Turing machine $M$ is called *$T$-time bounded* if, and only if, for any input $x$, each computation of $M$ on $x$ has length less or equal $T(|x|)$. The machine is called *$S$-space bounded* if, and only if, for any input $x$, during any computation of $M$ on $x$, at most $S(|x|)$ cells of the work tapes are used. Notice that time boundedness implies finiteness of all computations which is not the case for space boundedness. The same definitions apply for deterministic and nondeterministic Turing machines as well since these are just special cases of alternating Turing machines. These notions of resource bounds induce the complexity classes ATIME containing precisely those languages $L$ such that there is an alternating $T$-time bounded Turing machine deciding $L$ and ASPACE containing precisely those languages $L$ such that there is an alternating $S$-space bounded Turing machine deciding $L$. Similarly, these classes can be defined for nondeterministic and deterministic Turing machines.

We are especially interested in the following alternating complexity classes:

- ALOGSPACE $= \bigcup_{d \in \mathbb{N}}$ ASPACE$(d \cdot \log n)$,
- APTIME $= \bigcup_{d \in \mathbb{N}}$ ATIME$(n^d)$,
- APSPACE $= \bigcup_{d \in \mathbb{N}}$ ASPACE$(n^d)$.

Observe that GAME $\in$ ALOGSPACE. An alternating algorithm which decides GAME with logarithmic space just plays the game. The algorithm only has to store the *current* position in memory, and this can be done with logarithmic space. We shall now consider a slightly more involved example.





*Example* 1.10. QBF $\in$ ATIME$(O(n))$. W.l.o.g we assume that negation appears only at literals. We describe an alternating procedure $Eval(\varphi, \mathcal{I})$ which computes, given a quantified Boolean formula $\psi$ and a valuation $\mathcal{I} : \text{free}(\psi) \to \{0,1\}$ of the free variables of $\psi$, the value $[\![\psi]\!]^{\mathcal{I}}$.

**Algorithm 1.2.** Alternating algorithm deciding QBF.

| | |
|---|---|
| **Input**: $(\psi, \mathcal{I})$ | where $\psi \in$ QAL and $\mathcal{I}$: free$(\psi) \to \{0,1\}$ |
| **if** $\psi = Y$ | **then** |
| | **if** $\mathcal{I}(Y) = 1$ **then** accept |
| | **else** reject |
| **if** $\psi = \varphi_1 \vee \varphi_2$ | **then** „∃“ guesses $i \in \{1,2\}$ , $Eval(\varphi_i, \mathcal{I})$ |
| **if** $\psi = \varphi_1 \wedge \varphi_2$ | **then** „∀“ chooses $i \in \{1,2\}$ , $Eval(\varphi_i, \mathcal{I})$ |
| **if** $\psi = \exists X \varphi$ | **then** „∃“ guesses $j \in \{0,1\}$ , $Eval(\varphi, \mathcal{I}[X = j])$ |
| **if** $\psi = \forall X \varphi$ | **then** „∀“ chooses $j \in \{0,1\}$ , $Eval(\varphi, \mathcal{I}[X = j])$ |

### 1.4.3 Alternating versus Deterministic Complexity Classes

The main results we want to establish in this section concern the relationship between alternating complexity classes and deterministic complexity classes. We will see that alternating time corresponds to deterministic space, while by translating deterministic time into alternating space, we can reduce the complexity by one exponential. Here, we consider the special case of alternating polynomial time and polynomial space. We should mention, however, that these results can be generalised to arbitrary large complexity bounds which are well behaved in a certain sense.

**Lemma 1.11.** NPSPACE $\subseteq$ APTIME.

*Proof.* Let $L \in$ NPSPACE and let $M$ be a nondeterministic $n^l$-space bounded Turing machine which recognises $L$ for some $l \in \mathbb{N}$. The machine $M$ accepts some input $x$ if, and only if, some accepting configuration is reachable from the initial configuration $C_0(x)$ in the configuration tree of $M$ on $x$ in at most $k := 2^{cn^l}$ steps for some $c \in \mathbb{N}$. This is due to the fact that there are most $k$ different configurations of $M$ on input $x$ which use at most $n^l$ cells of the memory which can be seen using a simple combinatorial argument. So if there is some accepting configuration reachable from the initial configuration $C_0(x)$, then there is some accepting configuration reachable from $C_0(x)$ in at most $k$ steps. This is equivalent to the existence of some intermediate configuration $C'$ that is reachable from $C_0(x)$ in at most $k/2$ steps and from which some accepting configuration is reachable in at most $k/2$ steps.

So the alternating algorithm deciding $L$ proceeds as follows. The existential player guesses such a configuration $C'$ and the universal player chooses whether to check that $C'$ is reachable from $C_0(x)$ in at most $k/2$ steps or whether to check that some accepting configuration is reachable from $C'$ in at most $k/2$ steps. Then the algorithm (or equivalently, the game) proceeds with the subproblem chosen by the universal player, and continues in this binary search like fashion. Obviously, the number of steps which have to be performed by this procedure to decide whether $x$ is accepted by $M$ is logarithmic in $k$. Since $k$ is exponential in $n^l$, the time bound of $M$ is $dn^l$ for some $d \in \mathbb{N}$, so $M$ decides $L$ in polynomial time. Q.E.D.

**Lemma 1.12.** APTIME $\subseteq$ PSPACE.

*Proof.* Let $L \in$ APTIME and let $A$ be an alternating $n^l$-time bounded Turing machine that decides $L$ for some $l \in \mathbb{N}$. Then there is some $r \in \mathbb{N}$ such that any configuration of $A$ on any input $x$ has at most $r$ successor configurations and w.l.o.g. we can assume that any non-final configuration has precisely $r$ successor configurations. We can think of the successor configurations of some non-final configuration $C$ as being enumerated as $C_1, \dots, C_r$. Clearly, for given $C$ and $i$ we can compute $C_i$. The idea for a deterministic Turing machine $M$ to check whether some input $x$ is in $L$ is to perform a depth-first search on the computation tree $\mathcal{T}_{A,x}$ of $A$ on $x$. The crucial point is that we cannot construct and keep the whole configuration tree $\mathcal{T}_{A,x}$ in memory since its size is exponential in $|x|$ which exceeds our desired space bound. However, since the length of each computation is polynomially bounded, it is possible to keep a single computation path in memory and to construct the successor configurations of the configuration under consideration on the fly.

Roughly, the procedure $M$ can be described as follows. We start with the initial configuration $C_0(x)$. Given any configuration $C$ under consideration, we propagate 0 to the predecessor configuration if $C$ is rejecting and we propagate 1 to the predecessor configuration if $C$ is accepting. If $C$ is neither accepting nor rejecting, then we construct, for $i = 1, \dots, r$ the successor configuration $C_i$ of $C$ and proceed with checking $C_i$. If $C$ is existential, then as soon as we receive 1 for some $i$, we propagate 1 to the predecessor. If we encounter 0 for all $i$, then we propagate 0. Analogously, if $C$ is universal, then as soon as we receive a 0 for some $i$, we propagate 0. If we receive only 1 for all $i$, then we propagate 1. Then $x$ is in $L$ if, and only if, we finally receive 1 at $C_0(x)$. Now, at any point during such a computation we have to store at most one complete computation of $A$ on $x$. Since $A$ is $n^l$-time bounded, each such computation has length at most $n^l$ and each configuration has size at most $c \cdot n^l$ for some $c \in \mathbb{N}$. So $M$ needs at most $c \cdot n^{2l}$ memory cells which is polynomial in $n$. Q.E.D.





So we obtain the following result.

**Theorem 1.13.** (Parallel time complexity = sequential space complexity)

(1) APTIME = PSPACE.
(2) AEXPTIME = EXPSPACE.

Proposition (2) of this theorem is proved exactly the same way as we have done it for proposition (1). Now we prove that by translating sequential *time* into alternating *space*, we can reduce the complexity by one exponential.

**Lemma 1.14.** EXPTIME $\subseteq$ APSPACE

*Proof.* Let $L \in$ EXPTIME. Using a standard argument from complexity theory, there is a deterministic Turing machine $M = (Q, \Sigma, q_0, \delta)$ with time bound $m := 2^{c \cdot n^k}$ for some $c, k \in \mathbb{N}$ with only a single tape (serving as both input and work tape) which decides $L$. (The time bound of the machine with only a single tape is quadratic in that of the original machine with $k$ work tapes and a separate input tape, which, however, does not matter in the case of an exponential time bound.) Now if $\Gamma = \Sigma \uplus (Q \times \Sigma) \uplus \{\#\}$, then we can describe each configuration $C$ of $M$ by a word

$$\underline{C} = \#w_0 \dots w_{i-1}(qw_i)w_{i+1} \dots w_t\# \in \Gamma^*.$$

Since $M$ has time bound $m$ and only one single tape, it has space bound $m$. So, w.l.o.g., we can assume that $|\underline{C}| = m + 2$ for all configurations $C$ of $M$ on inputs of length $n$. (We just use a representation of the tape which has a priori the maximum length that will occur during a computation on an input of length $n$.) Now the crucial point in the argumentation is the following. If $C \vdash C'$ and $1 \leq i \leq m$, symbol number $i$ of the word $\underline{C'}$ only depends on the symbols number $i-1$, $i$ and $i+1$ of $\underline{C}$. This allows us to decide whether $x \in L(M)$ with the following alternating procedure which uses only polynomial space.

Player $\exists$ guesses some number $s \leq m$ of steps of which he claims that it is precisely the length of the computation of $M$ on input $x$. Furthermore, $\exists$ guesses some state $q \in F_{\mathrm{acc}}$, a Symbol $a \in \Sigma$ and a number $i \in \{0, \dots, s\}$, and he claims that the $i$-th symbol of the configuration $\underline{C}$ of $M$ after the computation on $x$ is $(qa)$. (So players start inspecting the computation of $M$ on $x$ from the final configuration.) If $M$ accepts input $x$, then obviously player $\exists$ has a possibility to choose all these objects such that his claims can be validated. Player $\forall$ wants to disprove the claims of $\exists$. Now, player $\exists$ guesses symbols $a_{-1}, a_0, a_1 \in \Gamma$ of which he claims that these are the symbols number $i-1$, $i$ and $i+1$ of the predecessor configuration of the final configuration $\underline{C}$. Now, $\forall$ can choose any of these symbols and demand that $\exists$ validates his claim for this particular symbol. This symbol is now the symbol under consideration, while $i$ is updated according to the movement of the (unique) head of $M$. Now, these actions of the players take place for each of the $s$ computation steps of $M$ on $x$. After $s$ such steps, we check whether the current symbol and the current position are consistent with the initial configuration $C_0(x)$. The only information that has to be stored in the memory is the position $i$ on the tape, the number $s$ which $\exists$ has initially guessed and the current number of steps. Therefore, the algorithm uses space at most $\mathrm{O}(\log(m)) = \mathrm{O}(n^k)$ which is polynomial in $n$. Moreover, if $M$ accepts input $x$ then obviously player $\exists$ has a winning strategy for the computation game. If, conversely, $M$ rejects input $x$, then the combination of all claims of player $\exists$ cannot be consistent and player





$\forall$ has a strategy to spoil any (cheating) strategy of player $\exists$ by choosing the appropriate symbol at the appropriate computation step. Q.E.D.

Finally, we make the simple observation that it is not possible to gain more than one exponential when translating from sequential time to alternating space. (Notice that Exptime is a proper subclass of 2Exptime.)

**Lemma 1.15.** APspace $\subseteq$ Exptime

*Proof.* Let $L \in$ APspace, and let $A$ be an alternating $n^k$-space bounded Turing machine which decides $L$ for some $k \in \mathbb{N}$. Moreover, for an input $x$ of $A$, let $\mathrm{Conf}(A, x)$ be the set of all configurations of $A$ on input $x$. Due to the polynomial space bound of $A$, this set is finite and its size is at most exponential in $|x|$. So we can construct the graph $G = (\mathrm{Conf}(A, x), \vdash)$ in time exponential in $|x|$. Moreover, a configuration $C$ is reachable from $C_0(x)$ in $\mathcal{T}_{A,x}$ if and only if $C$ is reachable from $C_0(x)$ in $G$. So to check whether $A$ accepts input $x$ we simply decide whether player $\exists$ has a winning strategy for the game played on $G$ from $C_0(x)$. This can be done in time linear in the size of $G$, so altogether we can decide whether $x \in L(A)$ in time exponential in $|x|$. Q.E.D.

**Theorem 1.16.** (Translating sequential time into alternating space)

(1) ALogspace = P.
(2) APspace = Exptime.

Proposition (1) of this theorem is proved using exactly the same arguments as we have used for proving proposition (2). An overview over the relationship between deterministic and alternating complexity classes is given in Figure 1.1.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Logspace | $\subseteq$ | Ptime | $\subseteq$ | Pspace | $\subseteq$ | Exptime | $\subseteq$ | Expspace |
| | | $\Vert$ | | $\Vert$ | | $\Vert$ | | $\Vert$ |
| | | ALogspace | $\subseteq$ | APtime | $\subseteq$ | APspace | $\subseteq$ | AExptime |

**Figure 1.1.** Relation between deterministic and alternating complexity classes

## 1.5 Model Checking Games for First-Order Logic

Let us first recall the syntax of FO formulae on relational structures. We have that $R_i(\bar{x})$, $\neg R_i(\bar{x})$, $x = y$ and $x \neq y$ are well-formed valid FO formulae, and inductively for FO formulae $\varphi$ and $\psi$, we have that $\varphi \vee \psi$, $\varphi \wedge \psi$, $\exists x \varphi$ and $\forall x \varphi$ are well-formed FO formulae. This way, we allow only formulae in *negation normal form* where negations occur only at atomic subformulae and all junctions except $\vee$ and $\wedge$ are eliminated. These constraints do not limit the expressiveness of the logic, but the resulting games are easier to handle.

For a structure $\mathfrak{A} = (A, R_1, \ldots, R_m)$ with $R_i \subseteq A^{r_i}$, we define the evaluation game $\mathcal{G}(\mathfrak{A}, \psi)$ as follows:

We have positions $\varphi(\bar{a})$ for every subformula $\varphi(\bar{x})$ of $\psi$ and every $\bar{a} \in A^k$.

At a position $\varphi \vee \vartheta$, Verifier can choose to move either to $\varphi$ or to $\vartheta$, while at positions $\exists x \varphi(x, \bar{b})$, he can choose an instantiation $a \in A$ of $x$ and move to $\varphi(a, \bar{b})$. Analogously, Falsifier can move from positions $\varphi \wedge \vartheta$ to either $\varphi$ or $\vartheta$ and from positions $\forall x \varphi(x, \bar{b})$ to $\varphi(a, \bar{b})$ for an $a \in A$.

The winning condition is evaluated at positions with atomic or negated atomic formulae $\varphi$, and we define that Verifier wins at $\varphi(\bar{a})$ if, and only if, $\mathfrak{A} \models \varphi(\bar{a})$, and Falsifier wins if, and only if, $\mathfrak{A} \not\models \varphi(\bar{a})$.

In order to determine the complexity of FO model checking, we have to consider the process of determining whether $\mathfrak{A} \models \psi$. To decide this question, we have to construct the game $\mathcal{G}(\mathfrak{A}, \psi)$ and check whether Verifier has a winning strategy from position $\psi$. The size of the game graph is bound by $|\mathcal{G}(\mathfrak{A}, \psi)| \leq |\psi| \cdot |A|^{\text{width}(\psi)}$, where width$(\psi)$ is the maximal number of free variables in the subformulae of $\psi$. So the game graph can be exponential, and therefore we can get only exponential time complexity for Game. In particular, we have the following complexities for the general case:

- alternating time: $O(|\psi| + \text{qd}(\psi) \log |A|)$
  where qd$(\psi)$ is the quantifier-depth of $\psi$,
- alternating space: $O(\text{width}(\psi) \cdot \log |A| + \log |\psi|)$,
- deterministic time: $O(|\psi| \cdot |A|^{\text{width}(\psi)})$ and





- deterministic space: $O(|\psi| + \text{qd}(\psi) \log |A|)$.

Efficient implementations of model checking algorithms will construct the game graph on the fly while solving the game.

We obtain that the structural complexity of FO model checking is ALogtime, and both the expression complexity and the combined complexity are PSpace.

**Fragments of FO with Efficient Model Checking**

We have seen that the size of the model checking games for first-order fomulae is exponential with respect to the width of the formulae, so we do not obtain polynomial-time model-checking algorithms in the general case. We now consider appropriate restrictions of FO, that lead to fragments with small model-checking games and thus to efficient game-based model-checking algorithms.

The *k-variable fragment of* FO is

$$\text{FO}^k := \{\psi \in \text{FO} : \text{width}(\psi) \leq k\}.$$

Clearly $|\mathcal{G}(\mathfrak{A}, \psi)| \leq |\psi| \cdot |A|^k$ for any finite structure $\mathfrak{A}$ and any $\psi \in \text{FO}^k$.

**Theorem 1.17.** ModCheck$(\text{FO}^k)$ is solvable in time $O(|\psi| \cdot |A|^k)$ and P-complete, for every $k \geq 2$.

As shown in Theorem 1.5, modal logic can be embedded (efficiently) into $\text{FO}^2$. Hence, also ML model checking has polynomial time complexity.

It is a general observation that modal logics have many convenient model-theoretic and algorithmic properties. Besides efficient model-checking the following facts are important in many applications of modal logic.

- The satisfiability problem for ML is decidable (in PSpace),
- ML has the finite model property: each satisfiable formula has a finite model,
- ML has the tree model property: each satisfiable formula has a tree-shaped model,

- algorihmic problems for ML can be solved by automata-based methods.

The embedding of ML into $FO^2$ has sometimes been proposed as an explanation for the good properties of modal logic, since $FO^2$ is a first-order fragment that shares some of these properties. However, more recently, it has been seen that this explanation has its limitations and is not really convincing. In particular, there are many extensions of ML to temporal and dynamic logics such as LTL, CTL, CTL$^*$, PDL and the $\mu$-calculus $L_\mu$ that are of great importance for applications in computer science, and that preserve many of the good algorithmic properties of ML. Especially the associated satisfiability problems remain decidable. However this is not at all true for the corresponding extension of $FO^2$.

A better and more recent explanation for the good properties of modal logic is that modal operators correspond to a restricted form of quantification, namely *guarded quantification*. Indeed, in the embedding of ML into $FO^2$, all quantifiers are *guarded* by atomic formulae. This can be vastly generalised beyond two-variable logic and involving formulae of arbitrary relational vocabularies, leading to the *guarded fragment* of FO.

**Definition 1.18.** The *guarded fragment* of first-order logic GF is the fragment of first-order logic which allows only guarded quantification

$$\exists \bar{y}(\alpha(\bar{x},\bar{y}) \land \varphi(\bar{x},\bar{y})) \text{ and } \forall \bar{y}(\alpha(\bar{x},\bar{y}) \rightarrow \varphi(\bar{x},\bar{y})),$$

where the *guards* $\alpha$ are atomic formulae containing all free variables of $\varphi$.

GF is a generalisation of modal logics: ML $\subseteq$ GF $\subseteq$ FO. Indeed, the modal operators $\Diamond$ and $\Box$ can be expressed as

$$\langle a \rangle \varphi \equiv \exists y(E_a xy \land \varphi(y)) \text{ and } [a]\varphi \equiv \forall y(E_a xy \rightarrow \varphi(y)).$$

It has turned out that the guarded fragment preserves (and explains to some extent) essentially all of the good model-theoretic and algorithmic properties of modal logics, in a far more expressive setting. In terms of model-checking games, we can observe that guarded logics have small





model checking games of size $\|\mathcal{G}(\mathfrak{A},\psi)\| = O(|\psi| \cdot \|\mathfrak{A}\|)$, and so there exist efficient game-based model-checking algorithms for them.