Strategies in Infinite Games

Strategies in Infinite Games: Structured Reactive Programs and Transducers over Infinite Alphabets

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Benedikt Brütsch
aus
Bergisch Gladbach

Berichter: Prof. Dr. Dr. h. c. Dr. h. c. Wolfgang Thomas
Directeur de Recherche Dr. Nicolas Markey
Prof. Dr. Martin Grohe

Tag der mündlichen Prüfung: 20. Dezember 2018

Diese Dissertation ist auf den Internetseiten
der Universitätsbibliothek online verfügbar.

Abstract

The subject of this thesis is the construction of winning strategies in games of the following kind: Two players alternately choose symbols from some fixed alphabet. They play forever, thus composing an infinite sequence of symbols. The winning condition is a set L of such sequences – the second player wins if the resulting sequence is in the set L, otherwise the first player wins.

Such games can be used to model the interaction between a so-called reactive system and its environment: The system continually reads input symbols provided by the environment and produces output symbols in response. A winning condition can be regarded as a specification for the system, and a winning strategy for the system constitutes an implementation of that specification.

Church's Synthesis Problem is to determine which player has a winning strategy and to present such a strategy in the form of a finite automaton, given an ω-regular winning condition $L \subseteq \Sigma^\omega$ over a finite alphabet Σ. We consider two variants of this problem.

The first variant is a refinement of the task, demanding a strategy in the form of a *structured reactive program*. It was introduced and solved in 2011 by Madhusudan. His solution involves building an alternating two-way co-Büchi tree automaton recognizing the programs that are winning strategies. We present a direct construction of a deterministic bottom-up tree automaton recognizing these programs and give a lower bound for the size of any such automaton. In both approaches, the number of (Boolean) variables available to the programs is crucial, as the time required by the synthesis algorithm is doubly exponential in that number. We show that for certain winning conditions defined in linear temporal logic (LTL), the required number of variables is exponential in the formula size, matching the known upper bound.

The second variant of the synthesis problem is a generalization where the players choose symbols from an infinite alphabet instead of a finite one. More precisely, the alphabet is of the form Σ^*, for a finite set Σ, so each symbol is a finite word. To represent winning conditions, we define automata over such infinite alphabets, namely $\mathbb{N}$-*memory automata*. We study closure properties and decidability questions for these automata, both on finite words and on ω-words. Analogously, we introduce $\mathbb{N}$-*memory transducers* to represent strategies. We show that the synthesis problem is solvable in this setting: Given a winning condition $L \subseteq (\Sigma^*)^\omega$ defined by a deterministic $\mathbb{N}$-memory parity automaton, we can determine which player has a winning strategy and construct such a strategy in the form of a deterministic $\mathbb{N}$-memory transducer.

Zusammenfassung

Das Thema dieser Arbeit ist die Konstruktion von Gewinnstrategien in unendlichen Spielen folgender Art: Zwei Spieler wählen abwechselnd jeweils ein Symbol aus einem gegebenen Alphabet, so dass sich eine unendliche Symbolfolge ergibt. Als Gewinnbedingung dient eine Menge L solcher Folgen – der zweite Spieler gewinnt, falls die produzierte Folge in L enthalten ist, andernfalls gewinnt der erste Spieler.

Solche Spiele sind ein nützliches Modell für die Interaktion zwischen einem sogenannten reaktiven System und dessen Umgebung: Das System nimmt fortwährend Eingabesymbole entgegen und antwortet jeweils mit einem Ausgabesymbol. Eine Gewinnbedingung kann als Spezifikation für solch ein System aufgefasst werden, und eine Gewinnstrategie für das System als korrekte Implementierung.

Wir untersuchen zwei Varianten von *Churchs Syntheseproblem*, das wie folgt formuliert werden kann: Gegeben ist eine ω-reguläre Gewinnbedingung $L \subseteq \Sigma^\omega$ über einem endlichen Alphabet Σ. Bestimme, welcher Spieler eine Gewinnstrategie hat, und konstruiere eine solche Strategie in Form eines endlichen Automaten.

Zuerst betrachten wir eine von Madhusudan (2011) vorgeschlagene Verfeinerung: Gesucht ist hier eine Strategie in Form eines *strukturierten reaktiven Programms*. Madhusudans Lösung basiert auf der Konstruktion eines alternierenden Zwei-Wege-Baumautomaten, der die Programme erkennt, die Gewinnstrategien darstellen. Wir präsentieren eine direkte Konstruktion eines deterministischen Baumautomaten, der ebendiese Programme erkennt, und geben eine untere Schranke für die Größe solcher Baumautomaten an. In beiden Verfahren spielt die Anzahl der (Booleschen) Programmvariablen eine wichtige Rolle, da sie sich doppelt exponentiell auf die Laufzeit des Synthesealgorithmus auswirkt. Wir zeigen, dass für bestimmte in Linear Temporal Logic (LTL) formulierte Spezifikationen eine exponentielle Variablenanzahl nötig ist.

Als zweite Variante des Syntheseproblems betrachten wir die folgende Verallgemeinerung: Die Spieler wählen nun Symbole aus einem *unendlichen* Alphabet der Form Σ^*, für eine endlichen Menge Σ. Zur Darstellung von Gewinnbedingungen definieren wir sogenannte $\mathbb{N}$-*Memory-Automaten* über solchen Alphabeten und untersuchen deren Eigenschaften sowohl auf endlichen Wörtern also auch auf ω-Wörtern. Wir definieren $\mathbb{N}$-*Memory-Transducer* als Modell für Strategien und beweisen die algorithmische Lösbarkeit des entsprechenden Syntheseproblems: Für eine Gewinnbedingung $L \subseteq (\Sigma^*)^\omega$, die durch einen deterministischen $\mathbb{N}$-Memory-Parity-Automaten gegeben ist, können wir bestimmen, welcher Spieler eine Gewinnstrategie hat, und eine solche Strategie in Form eines $\mathbb{N}$-Memory-Transducers angeben.

Contents

1

Introduction

Gale-Stewart games

Gale-Stewart games, introduced by Gale and Stewart in 1953 [GS53], are games of infinite duration between two players who alternately choose symbols from some fixed alphabet. The first symbol is chosen by Player I, the next one by Player II, and so on, and each choice can be observed by both players. This process results in an infinite sequence of symbols, which we call a *play* of the game. The winning condition is defined by a given set L of such sequences: The plays in L are won by Player II, all other plays are won by Player I.

We say that a player has a winning strategy if he can choose his symbols, based on the previous choices of his opponent, in such a way that he always wins. If a player has a winning strategy, we call him the winner of the game. For certain classes of winning conditions, it is possible to determine the winner algorithmically and even to construct a winning strategy. Most prominently, this applies to the class of ω-regular winning conditions, which are the sets of infinite sequences that can be recognized by nondeterministic Büchi automata. More specifically, algorithmic solutions to the following problem were given by Büchi and Landweber in 1969 [BL69], and – using a different approach – by Rabin in 1972 [Rab72]:

Church's Synthesis Problem

> Given an ω-regular winning condition, decide which of the two players has a winning strategy in the corresponding Gale-Stewart game, and construct a winning strategy in the form of a finite-state automaton.

This task was proposed by Church in 1957 [Chu57] (see also [Chu63]), albeit in different terminology, and is known as *Church's Synthesis Problem*. In this thesis, we consider two variations of this problem.

In the first part, we study a refined version of the problem, proposed by Madhusudan [Mad11], where the winning strategy is to be presented in a compositional and more succinct format, namely as a *structured reactive program*. Madhusudan solved this refined problem by building a two-way alternating tree automaton recognizing the set of programs, over a given finite set of variables, that implement a winning strategy. We give an alternative solution using a more elementary type of tree

automata. More specifically, we provide a direct construction of a deterministic bottom-up tree automaton recognizing the aforementioned set of programs. We also give a lower bound for the size of any deterministic or nondeterministic tree automaton recognizing this set of programs. Moreover, we address the special case of winning conditions given in linear temporal logic (LTL). For such winning conditions, we prove an exponential lower bound for the number of Boolean program variables that are required to implement a winning strategy, matching the upper bound that can be derived from previously known results.

In the second part, we generalize the result of Büchi and Landweber from finite alphabets to infinite alphabets where each symbol is in fact a finite word. Formally, we consider alphabets of the form Σ^* for some finite set Σ. In particular, the alphabet $\mathbb{N}$ can be viewed as a special case, represented by $\{1\}^*$. We define a suitable type of automata over such infinite alphabets, called $\mathbb{N}$-*memory automata*, and develop the basic theory of these automata both on finite words and on ω-words. Furthermore, we introduce a corresponding model of automata with output, namely $\mathbb{N}$-*memory transducers*. We provide an algorithmic solution to the following generalization of Church's Synthesis Problem: Given a winning condition $L \subseteq (\Sigma^*)^\omega$ defined by a deterministic $\mathbb{N}$-memory parity automaton, decide which of the two players has a winning strategy in the corresponding Gale-Stewart game, and construct a winning strategy in the form of a deterministic $\mathbb{N}$-memory transducer.

The following sections give a more detailed account of the contributions and motivations of the two parts of this thesis and provide an overview of related work.

1.1 Synthesis of Structured Reactive Programs

1.1.1 Background

Gale-Stewart games can be understood as a model for the behavior of so-called *reactive systems*, which continuously interact with their environment. Reactive systems are ubiquitous, with examples ranging from microchips to driver assistance systems to control systems in industrial plants. We may think of Player I as the environment, which in each turn provides an input symbol to the system, while Player II represents the system, which in each turn produces an output symbol. In this context, a winning condition can be regarded as a *specification*, indicating the admissible combinations of input and output sequences. A winning strategy for Player II then constitutes an implementation of

the system that satisfies the specification, by producing for every input sequence a suitable output sequence.

Most approaches to the synthesis of such winning strategies, for instance [BL69; Rab72; PR89; KV99], are designed to yield strategies in the form of transition systems such as Mealy or Moore automata. However, when designing a reactive system, we are ultimately not interested in a "static" representation of the winning strategy – instead, we want to be able to "execute" it. For this purpose, winning strategies in the form of circuits or imperative programs are arguably more suitable, since they prescribe the computational steps required to apply the strategy. In fact, in its original formulation, Church's Synthesis Problem asks for a "circuit" satisfying the given specification. A circuit or program can be much more succinct than a transition lookup table that might be used to execute a strategy defined by a transition system.

In the first part of this thesis, we consider such a succinct format of strategies, namely *structured reactive programs* over a finite set of Boolean variables, which were introduced by Madhusudan [Mad11]. We refine his results and show limitations of this approach. Structured reactive programs are nonterminating `while`-programs with input and output statements. In contrast to transition systems, they are structured in the sense that they can be decomposed into subprograms.

Madhusudan [Mad11] proposes a procedure to synthesize such programs from ω-regular specifications, using the fact that structured programs can be represented by their syntax trees. Given a finite set of Boolean variables and a nondeterministic Büchi automaton recognizing the complement of the specification, he constructs a two-way alternating ω-automaton on finite trees that recognizes the set of *all* programs over these variables that satisfy the specification. This tree automaton can be transformed into a nondeterministic one-way tree automaton (NTA) to check for emptiness and extract a minimal program (regarding the height of the corresponding tree) from that set. In contrast to the transition systems constructed by classical synthesis algorithms, the synthesized program does not depend on the specific syntactic formulation of the specification, but only on its meaning.

1.1.2 Contributions

First, to lay a foundation for our study of the synthesis of structured reactive programs, we define a formal semantics for these programs, which was only informally indicated by Madhusudan [Mad11]. To that end, we introduce the concept of *Input/Output/Internal machines*

(IOI machines), which are composable in the same way as structured programs. This allows for an inductive definition of the semantics.

We then present a direct construction of a deterministic bottom-up tree automaton (DTA) recognizing the set of programs over the given variables that satisfy the given specification, without a detour via more intricate types of tree automata. To that end, we transform the given nondeterministic Büchi automaton representing the complement of the specification into another Büchi automaton $\mathcal{B}_{\text{bad}}$ recognizing the "bad program traces", which characterize the "incorrect" programs. Based on this Büchi automaton $\mathcal{B}_{\text{bad}}$, the DTA inductively computes a summary of the behavior of a given program in the form of so-called *co-execution signatures*. A co-execution is a pair consisting of a computation of the program and a corresponding run of $\mathcal{B}_{\text{bad}}$, and the co-execution signatures capture the essential information about all possible co-executions.

Our synthesis procedure applies not only to programs that read input and write output in strict alternation, but also more generally to programs that may cause some delay, within given bounds, between the input sequence and the output sequence. Such a program corresponds to a strategy for the system in a game (against its environment) where in each move the system may either choose one or more output symbols or skip and wait for the next input (see [HKT10]), as long as the resulting delay stays within the specified bounds. This can be a suitable model for systems that interact with their environment through input and output buffers of fixed size.

For programs with strict input/output alternation, the complexity of our construction matches that of Madhusudan's algorithm. In particular, the size of the resulting DTA is exponential in the size of the given Büchi automaton (representing the complement of the specification) and doubly exponential in the number of program variables. In fact, we also establish a lower bound by showing that the set of all programs over m Boolean variables that satisfy a given specification cannot even be recognized by an NTA with less than $2^{2^{m-1}}$ states, if any such program exists. However, note that a DTA (or NTA) accepting precisely these programs enables us to extract a minimal program for the given specification and the given set of variables. Thus, while the tree automaton may be large, the synthesized program itself might be rather small.

Finally, we clarify the limits of this synthesis approach by analyzing how many Boolean variables are needed to satisfy a given specification. More specifically, we consider specifications defined by LTL formulas. An exponential upper bound (in the size of the formula) for the required

number of variables can be deduced from the doubly exponential upper bound for the size of the smallest transition system satisfying a given LTL specification (see [PR89]). We establish a corresponding lower bound by constructing a family of LTL specifications $(\varphi_n)_{n\in\mathbb{N}}$, whose size is quadratic in n, such that the least number of variables used by any structured program satisfying φ_n is $\Omega(2^n)$. For the proof, we draw on concepts from graph theory and exploit the fact that, intuitively speaking, the so-called transition graphs of structured programs over a small number of variables have small tree-width. We show that certain specifications can only be satisfied by programs whose transition graphs have large tree-width, which allows us to deduce a lower bound for the number of variables used by these programs.

The first part of this thesis is based on [Brü15].

1.1.3 Related Work

Lustig and Vardi [LV09] solve the problem of constructing for a given LTL specification an implementation that is composed of components from a given library. The composition is represented as a transition system rather than a structured program, but the synthesis algorithm involves "summaries" of the behavior of the individual components that are very similar to the co-execution signatures used in our algorithm.

Aminof, Mogavero and Murano [AMM12] provide a round-based algorithm to synthesize another type of composite systems, called *hierarchical transition systems*, which can be exponentially more succinct than corresponding "flat" transition systems. The desired system is constructed in a bottom-up manner: In each round, a specification is provided and the algorithm constructs a corresponding hierarchical transition system from a given library of available components and the hierarchical transition systems created in previous rounds. Thus, in order to obtain a small system in the last round, the specifications in the previous rounds have to be chosen in an appropriate way.

Current techniques for the synthesis of (potentially) compact implementations in the form of circuits or programs typically proceed in an indirect way, by converting a transition system into such an implementation. For example, Bloem et al. [BGJ+07] first construct a symbolic representation (a binary decision diagram) of an appropriate transition system and then extract a corresponding circuit. However, this indirect approach does not necessarily yield a succinct result.

Another succinct model for the representation of strategies in infinite games was introduced by Gelderie [Gel12; Gel14] in the form of *strategy*

machines. These are essentially Turing machines, equipped with a set of control states and a memory tape, thus providing a similar separation between control flow and memory as structured programs. Fearnley, Peled, and Schewe [FPS12; FPS15] address the question whether realizable specifications given in LTL or CTL (computation tree logic) can always be implemented by Turing machines whose transition table *and* memory tape are polynomially bounded in the length of the given formula. They show that the answer depends on the collapse of certain complexity classes.

1.2 Synthesis of Transducers over Infinite Alphabets

1.2.1 *Background*

Since the pioneering work of Büchi, Landweber [BL69] and Rabin [Rab72], many variations of Church's Synthesis Problem have been considered, including the synthesis of infinite-state strategies. For example, Walukiewicz [Wal01] showed the analogue of the Theorem of Büchi and Landweber for the case of winning conditions that are defined by pushdown parity automata – it is then possible to present a winning strategy realized by a pushdown automaton. It is remarkable that a different kind of "infinite extension" of Church's Synthesis Problem has not been addressed in the literature, namely the variant where the players of the Gale-Stewart game draw symbols from an infinite alphabet.

That is not to say that Gale-Stewart games over infinite alphabets are uncharted territory. On the contrary, games over infinite alphabets as well as over finite alphabets have been extensively studied within the framework of set theory. However, the focus of that research is the notion of *determinacy* rather than the algorithmic solvability of these games. A game is called determined if one of the players has a winning strategy, and from a set-theoretical perspective, it is a central question whether the Gale-Stewart games induced by a certain class of winning conditions are determined.

In that context, the set of all infinite sequences over a fixed alphabet is understood as a topological space, most notably the Cantor space $\{0,1\}^\omega$ or the Baire space $\mathbb{N}^\omega$. Gale and Stewart [GS53] showed (using the axiom of choice) that there exist winning conditions that do not admit winning strategies for either player, but that determinacy is guaranteed if the winning condition is an open or closed set. The latter result was subsequently extended to larger classes of winning

conditions [Wol55; Dav64; Par72], and in 1975, Martin [Mar75] showed that Gale-Stewart games are in fact determined for all Borel winning conditions. The topological classification theory of the Baire space $\mathbb{N}^\omega$ is very closely aligned with that of the Cantor space $\{0,1\}^\omega$ (see [Kec95; Mos09]), and the aforesaid determinacy results hold in either case – and in fact for arbitrary finite and infinite alphabets.

In regard to Church's Synthesis Problem, however, the step from finite to infinite alphabets is highly nontrivial. It requires automata that can process sequences over infinite alphabets. Let us consider the set of natural numbers $\mathbb{N}$ as the prototypical example of an infinite alphabet. We may represent a number $m \in \mathbb{N}$ by a finite word over a finite alphabet, for example by the word 1^m over the unary alphabet $\{1\}$. A sequence over $\mathbb{N}$ thus corresponds to a sequence of finite words.

A straightforward approach to make such a sequence of finite words amenable to be processed by standard automata is to concatenate the words, separated by some additional symbol. Thus, a sequence $m_1 m_2 m_3 \ldots$ with $m_i \in \mathbb{N}$ could be encoded as a sequence $1^{m_1}01^{m_2}01^{m_3}0\ldots$ over $\{0,1\}$, which could be processed by a finite-state automaton. A Gale-Stewart game over the alphabet $\mathbb{N}$ could then be viewed as a different kind of game over the alphabet $\{0,1\}$, namely as a so-called Banach-Mazur game, where the players choose in each move a finite sequence of symbols rather than just a single symbol (see, for example, [GL12] and [Dil89]). A major drawback of this approach is the fact that even simple properties of sequences over $\mathbb{N}$ cannot be verified by finite-state automata working on the corresponding finite-alphabet encodings. For example, a finite-state automaton cannot check whether the first two numbers m_1, m_2 of a given sequence are equal (and even a pushdown automaton cannot check whether the first three numbers are all equal). We shall therefore define more powerful tools to deal with sequences over infinite alphabets.

1.2.2 Contributions

We present $\mathbb{N}$-*memory automata* as a formalism to specify languages over infinite alphabets of the form Σ^*, for a finite set Σ. For the special case of the alphabet $\{1\}^*$, which can be viewed as a representation of $\mathbb{N}$, the $\mathbb{N}$-memory automata defined in this thesis are a simplified but expressively equivalent form of the identically named automata introduced in joint work with Landwehr and Thomas [BLT17].

An $\mathbb{N}$-memory automaton is not only equipped with a finite set of states but also with additional memory that can store a natural

number. Its transition relation is defined by a so-called pebble automaton. Given an element of the alphabet Σ^*, i.e., a finite word u over Σ, this pebble automaton performs a two-way run on u starting at the position determined by the current memory content of the $\mathbb{N}$-memory automaton. The position reached at the end of that run determines the new memory content. Equivalently, the transition relation can be defined in monadic second-order (MSO) logic over the structure of the natural numbers with the successor relation.

Let us give some examples illustrating the capabilities of our model. An $\mathbb{N}$-memory automaton can, for instance, determine whether the length of the words in a given sequence is strictly increasing: To that end, the automaton memorizes at each point of the sequence the length of the current word and checks whether the length of the next word is strictly greater. In particular, the set of strictly increasing sequences over the alphabet $\mathbb{N}$ (represented by $\{1\}^*$), can be recognized by an $\mathbb{N}$-memory automaton. In a similar way, we can construct $\mathbb{N}$-memory automata that check whether a sequence over $\mathbb{N}$ increases or decreases by 1 in every step, or whether all even numbers occurring in a given sequence are ordered increasingly.

We study the theory of $\mathbb{N}$-memory automata, including the basic closure properties and decision problems, both over finite words and over ω-words. Furthermore, we introduce corresponding automata with output, called $\mathbb{N}$-*memory transducers*. An $\mathbb{N}$-memory transducer over an alphabet Σ^* can be regarded as a strategy in a Gale-Stewart game over Σ^*.

The main result of the second part of this thesis is an algorithmic solution of Church's Synthesis Problem for a certain class of winning conditions over infinite alphabets of the form Σ^*, for a finite set Σ:

Given a winning condition $L \subseteq (\Sigma^*)^\omega$ that is defined by a deterministic $\mathbb{N}$-memory parity automaton, we can decide who wins the corresponding Gale-Stewart game and construct a winning strategy in the form of a deterministic $\mathbb{N}$-memory transducer.

A preliminary version of this result for the special case of the alphabet $\mathbb{N}$ has been presented in joint work with Thomas [BT16].

In essence, we obtain this solution of the synthesis problem by reducing the Gale-Stewart game to a parity game on the configuration graph of the given deterministic $\mathbb{N}$-memory parity automaton. The set of positions of that parity game has the form $Q \times \Gamma^*$ with finite sets Q and Γ (in fact, with $\Gamma = \{1\}$). Moreover, the game graph, the positions

of each priority, and the positions of each player are all MSO-definable in the structure $Q \times \mathcal{T}_\Gamma$, where $\mathcal{T}_\Gamma$ is the infinite Γ-branching tree. We show that in such a parity game, we can construct for each player a uniform positional winning strategy (on his winning region) that is MSO-definable in $Q \times \mathcal{T}_\Gamma$.

For the proof, we combine several constructions from the literature, in particular an algorithm by Kupferman, Piterman, and Vardi [KPV10] that yields finite-automata strategies in parity games on pushdown graphs, a technique by Cachat [Cac03] that can be adapted to extend this result from pushdown graphs to prefix-recognizable graphs, and a proof by Blumensath [Blu01] that provides an effective connection between the equivalent notions of MSO-definability and prefix-recognizability of graphs.

1.2.3 *Related Work*

Numerous models of automata over infinite alphabets, whose elements are often called data values, have been proposed in the literature. However, they have not been considered in the context of Church's Synthesis Problem. A survey of some of these automata was given by Segoufin [Seg06]. Among them are the register automata introduced by Kaminski and Francez [KF94] (under the name finite-memory automata), which are equipped with a finite number of registers and can compare the current data value for equality with those that have previously been stored in the registers. This model has been extended in [FHL16] to allow comparisons with respect to a total order on the alphabet.

Bojańczyk, Klin, and Lasota [BKL11; BKL14] propose the more general, abstract model of orbit-finite nominal G-automata as well as a concrete counterpart called Fraïssé automata. These automata work on alphabets augmented with a certain "symmetry", and for the so-called total order symmetry, they coincide with register automata with order comparisons.

Another common approach to handle sequences over infinite alphabets is to proceed in two phases: First a finite-state transducer processes the sequence, then its output is reorganized, based on a total order or the equality relation on the alphabet, and fed to a finite-state automaton. This approach, using only the equality relation, is applied by Bojańczyk et al. [BMS+06; BDM+11] to define the so-called data automata; an equivalent formalism called class memory automata is provided by Björklund and Schwentick [BS07; BS10]. Manuel [Man10] considers

sequences equipped with a total order on the occurring symbols and introduces a corresponding type of automata working in two phases, namely text automata. In a similar way, Tan [Tan12; Tan14] defines an automaton model over totally ordered alphabets that even works on trees, namely ordered-data tree automata.

Furthermore, automata with a bounded number of pebbles have been studied over infinite alphabets by Neven et al. [NSV04]. (These automata are not to be confused with the pebble automata considered in this thesis.) The pebbles are used to mark positions of the input sequence. The symbol at the current position of the automaton can be checked for equality with the symbols at the marked positions. However, the nonemptiness problem for such automata is undecidable [NSV04], unless heavy restrictions are imposed [Tan10].

The expressive power of all the models mentioned above is incomparable to that of $\mathbb{N}$-memory automata. In particular, in contrast to $\mathbb{N}$-memory automata, they cannot express properties of "incremental change" in sequences over the alphabet $\mathbb{N}$, since they do not allow comparisons of natural numbers with respect to the successor relation. For instance, they cannot check whether the difference between any two numbers at neighboring positions in a given sequence is exactly 1. In the context of temporal logic, Carapelle et al. [CFK+15] define a formalism that can express such properties, but in contrast to $\mathbb{N}$-memory automata, it only allows comparisons of numbers that occur within a bounded neighborhood.

Our model of $\mathbb{N}$-memory automata can be viewed as a generalization of the progressive grid automata and the strong automata introduced by Czyba, Spinrath, and Thomas [CST15]. They also subsume the $\mathcal{N}$-MSO-automata of Spelten, Thomas, and Winter [STW11] (see also [Bès08]).

2

Preliminaries on Languages and Automata

In this chapter, we fix some notations and define the basic concepts regarding formal languages and finite automata that are used throughout this thesis.

General notations. We let $\mathbb{N} = \{0, 1, 2, \ldots\}$ denote the set of natural numbers, and we define $\mathbb{N}_{\geq 1} = \{1, 2, 3, \ldots\}$ and $[k] = \{0, 1, 2, \ldots, k\}$. The powerset of a set M can be identified with the set of functions of the form $f : M \to \{0, 1\}$, and it is denoted by 2^M. The number of elements of a finite set M is denoted by $|M|$. We often use the notation $\overline{a}$ to refer to a tuple $(a_1, \ldots, a_k)$.

$\mathbb{N}$, $\mathbb{N}_{\geq 1}$
$[k]$
2^M
$|M|$
$\overline{a}$

Words and languages. An *alphabet* is a nonempty set, and the elements of an alphabet are called *symbols*. To refer to *finite* alphabets, we generally use the letters Σ and Γ. A finite word, also simply called a *word*, over a finite or infinite alphabet X is a finite (possibly empty) sequence $w = a_1 \ldots a_n$ with $a_i \in X$ for $i \in \{1, \ldots, n\}$, where $n \in \mathbb{N}$ is the length of w, denoted by $|w|$. The empty word ε is the word of length 0. The *concatenation* of two words $u = a_1 \ldots a_m$ and $v = b_1 \ldots b_n$ is the word $u \cdot v = a_1 \ldots a_m b_1 \ldots b_n$, also written as uv. For a word w of the form $w = u \cdot v$, we call u a *prefix* and v a *suffix* of w.

alphabet
symbol
Σ, Γ
word
$|w|$
ε
prefix, suffix

The set of all words over the alphabet X is denoted by X^*, and we let $X^+ = X^* \setminus \{\varepsilon\}$. Furthermore, $X^{\leq k}$ and $X^{<k}$ are the sets of words over X of length at most k and of length strictly less than k, respectively. A *language* over the alphabet X is a set $L \subseteq X^*$. The concatenation of languages L_1 and L_2 is defined as $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1,\ w_2 \in L_2\}$, and the iteration of a language L is $L^* = \{\varepsilon\} \cup L \cup (L \cdot L) \cup (L \cdot L \cdot L) \cup \ldots$.

X^*, X^+
$X^{\leq k}$, $X^{<k}$
language

ω-words and ω-languages. An *ω-word* over an alphabet X is an infinite sequence $\alpha = a_1 a_2 a_3 \ldots$ with $a_i \in X$ for $i \in \mathbb{N}_{\geq 1}$. A finite word $w = a_1 \ldots a_m$ can be concatenated with an ω-word $\beta = b_1 b_2 b_3 \ldots$, yielding the ω-word $w \cdot \beta = a_1 \ldots a_m b_1 b_2 b_3 \ldots$, which is also written as $w\beta$. If α is an ω-word of the form $w \cdot \beta$, then we call w a *prefix* and β a *suffix* of α.

ω-word

The set of all ω-words over the alphabet X is denoted by X^ω, and an *ω-language* over X is a set $L \subseteq X^\omega$. For a language $W \subseteq X^*$ of finite

X^ω
ω-language

words and an ω-language $L \subseteq X^\omega$, we let $W \cdot L = \{w \cdot \beta \mid w \in W,\ \beta \in L\}$. The infinite iteration of a language W of finite words is the ω-language $W^\omega = \{w_1 \cdot w_2 \cdot w_3 \cdot \ldots \mid w_i \in W \text{ and } w_i \neq \varepsilon \text{ for } i \in \mathbb{N}_{\geq 1}\}$.

FINITE AUTOMATA. Finite automata on words over finite alphabets are defined as follows.

nondeterministic finite automaton (NFA)

DEFINITION 2.0.1. A *nondeterministic finite automaton (NFA)* $\mathcal{A}$ is a tuple $(S, \Sigma, \Delta, s_0, F)$, where

- S is a finite set of states,
- Σ is a finite alphabet,
- $\Delta \subseteq S \times \Sigma \times S$ is the transition relation,
- $s_0 \in S$ is the initial state, and
- $F \subseteq S$ is the set of final states.

deterministic finite automaton (DFA)

If the transition relation Δ of an NFA $\mathcal{A}$ can in fact be written as a function $\delta\colon S \times \Sigma \to S$, then we call the automaton $\mathcal{A}$ a *deterministic finite automaton (DFA)*.

A *run* π of an NFA $\mathcal{A}$ as defined above on a word $w = a_1 \ldots a_n \in \Sigma^*$ is a finite sequence of states $s_1 \ldots s_{n+1}$ such that $(s_i, a_i, s_{i+1}) \in \Delta$ for all $i \in \{1, \ldots, n\}$. A run is *accepting* if it starts in the initial state and ends in a final state, i.e., if $s_1 = s_0$ and $s_{n+1} \in F$. The language *recognized* by the NFA $\mathcal{A}$ is defined as

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \text{there is an accepting run of } \mathcal{A} \text{ on } w\}.$$

regular language

A language that is recognized by an NFA or, equivalently, by a DFA is called a *regular language*.

BÜCHI AUTOMATA. Automata on ω-words over finite alphabets can be defined like automata on finite words, but with a different acceptance condition.

Büchi automaton

DEFINITION 2.0.2. A *(nondeterministic) Büchi automaton* $\mathcal{A}$ is a tuple $(S, \Sigma, \Delta, s_0, F)$ of the same form as a nondeterministic finite automaton. However, we call F the set of *accepting* states.

A *run* π of a Büchi automaton $\mathcal{A}$ on an ω-word $\alpha = a_1 a_2 a_3 \ldots \in \Sigma^\omega$ is an infinite sequence of states $s_1 s_2 s_3 \ldots$ such that $(s_i, a_i, s_{i+1}) \in \Delta$ for all $i \in \mathbb{N}_{\geq 1}$. (We will sometimes also consider finite runs on finite words,

which are defined as in the case of NFAs.) A run is *accepting* if it starts in the initial state and infinitely often visits an accepting state, i.e., if $s_1 = s_0$ and $s_i \in F$ for infinitely many $i \in \mathbb{N}_{\geq 1}$. The ω-language *recognized* by the Büchi automaton $\mathcal{A}$ is defined as

$$L(\mathcal{A}) = \{\alpha \in \Sigma^\omega \mid \text{there is an accepting run of } \mathcal{A} \text{ on } \alpha\}.$$

An ω-language that is recognized by a nondeterministic Büchi automaton is called a *regular ω-language* (or an *ω-regular language*).

regular ω-language

Part I

Synthesis of Structured Reactive Programs

3

Outline: Structured Reactive Programs

In Part I of this thesis, we study the synthesis of *structured reactive programs*, which continually interact with their environment by means of input and output statements.

Figure 1 shows an example of such a program, which uses the Boolean variables z_1 and z_2. To each input from the alphabet $\{0, 1\}$, this program responds with an output as follows: The output will be 0 in each step until the input is 1 for the first time, and after that, the output will be 1 until the input is 1 again, and so on.

```
z1 := false;
while true do {
    input (z2);
    output (z1);
    if z2 then {
        z1 := ¬z1;
    };
}
```

Figure 1: A simple structured reactive program.

Building on previous work by Madhusudan [Mad11], we study the following refinement of Church's Synthesis Problem:

Given a finite set of Boolean variables Z and given an ω-regular specification, construct a structured reactive program p over Z that satisfies the specification, or determine that no such program exists.

We proceed in the following way:

- First, in Chapter 4, we define the syntax of the programs that we want to consider. We also develop a formal operational semantics for these programs in terms of finite-state machines with distinguished input and output transitions, which we call IOI machines.

- In Chapter 5, we define a representation of programs in the form of finite trees and provide some basics on tree automata.
- In Chapter 6, we present a procedure to synthesize a structured reactive program from a given ω-regular specifications. The core of that procedure is the construction of a deterministic bottom-up tree automaton accepting precisely the programs that satisfy the specification.
- The size of the tree automaton constructed in Chapter 6 is doubly exponential in the number of Boolean variables available to the programs. In Chapter 7, we prove that this cannot be avoided.
- The results of Chapter 6 and Chapter 7 give rise to the question of how many program variables are in fact required to satisfy a given specification. In Chapter 8, we prove an exponential lower bound for the required number of variables for certain families of specifications defined in linear temporal logic (LTL).
- We conclude our study of the synthesis problem for structured reactive programs in Chapter 9 with a brief summary and some perspectives for future research.

4

Syntax and Semantics of Structured Programs

4.1 Syntax of Structured Programs

We consider structured programs as defined in [Mad11]. These programs read input symbols and write output symbols in the form of tuples of Boolean values (i.e., tuples of bits). More specifically, an input symbol is a tuple $\bar{a} \in \{0,1\}^{N_I}$ and an output symbol is a tuple $\bar{b} \in \{0,1\}^{N_O}$, where $N_I, N_O \in \mathbb{N}_{\geq 1}$ are two fixed numbers.

Notation 4.1.1. The input alphabet is denoted by $\Sigma_I = \{0,1\}^{N_I}$ and the output alphabet by $\Sigma_O = \{0,1\}^{N_O}$.

Σ_I, Σ_O

Definition 4.1.2. *Structured programs* (also simply called *programs*) and *Boolean expressions* (also simply called *expressions*) over a finite set Z of Boolean variables are defined by the following grammar, where z stands for a variable in Z and $\bar{z}$ for a tuple of variables:

structured programs

$$\begin{aligned} \langle prog \rangle \quad ::= \quad & z := \langle expr \rangle \mid \texttt{input } \bar{z} \mid \texttt{output } \bar{z} \mid \\ & \langle prog \rangle ; \langle prog \rangle \mid \texttt{while } \langle expr \rangle \texttt{ do } \langle prog \rangle \mid \\ & \texttt{if } \langle expr \rangle \texttt{ then } \langle prog \rangle \texttt{ else } \langle prog \rangle \end{aligned}$$

Boolean expressions

$$\begin{aligned} \langle expr \rangle \quad ::= \quad & \texttt{true} \mid \texttt{false} \mid z \mid \\ & \langle expr \rangle \wedge \langle expr \rangle \mid \langle expr \rangle \vee \langle expr \rangle \mid \neg \langle expr \rangle \end{aligned}$$

Intuitively, an assignment of the form "$z := e$" changes the value of the variable z according to the value of the expression e. A program of the form "`input` $\bar{z}$" reads an input symbol (i.e., a tuple of bits) and stores it in the variables $\bar{z}$. Hence, $\bar{z}$ must be a tuple of N_I pairwise distinct variables. Conversely, "`output` $\bar{z}$" writes the output symbol consisting of the current values of the variables $\bar{z}$. Thus, $\bar{z}$ must be a tuple of N_O variables in this case. These atomic operations can be combined by means of sequential execution (indicated by "`;`"), repetition (`while`),

and conditional execution (`if/then/else`). A formal semantics will be developed in the following sections.

4.2 Operational Semantics of Structured Programs

In order to formalize the program semantics, let us first fix some terminology and notations.

variable valuation

A *variable valuation* for a given set of Boolean variables Z is a function $\sigma\colon Z \to \{0,1\}$, which assigns to each variable a Boolean value. The set of all possible variable valuations for Z is denoted by 2^Z.

$\sigma[z/a]$

We write $\sigma[z/a]$ to denote the variable valuation that is obtained from the valuation σ by replacing the value of the variable z with $a \in \{0,1\}$. We also extend this notation to tuples $\bar{z}$ of pairwise distinct variables and tuples $\bar{a}$ of Boolean values.

The *value of an expression* e over a set of Boolean variables Z depends on the valuation of the variables and is defined in the natural way.

$[\![e]\!](\sigma)$

Formally, for an expression e, we let $[\![e]\!] : 2^Z \to \{0,1\}$ be the function that assigns to each variable valuation σ the corresponding value of e:

$$\begin{aligned} [\![\texttt{true}]\!](\sigma) &= 1 & [\![\neg e_1]\!](\sigma) &= 1 - [\![e_1]\!](\sigma) \\ [\![\texttt{false}]\!](\sigma) &= 0 & [\![e_1 \wedge e_2]\!](\sigma) &= \min\bigl\{[\![e_1]\!](\sigma), [\![e_2]\!](\sigma)\bigr\} \\ [\![z]\!](\sigma) &= \sigma(z) & [\![e_1 \vee e_2]\!](\sigma) &= \max\bigl\{[\![e_1]\!](\sigma), [\![e_2]\!](\sigma)\bigr\} \end{aligned}$$

To indicate the individual "steps" of a program, we will use symbols from the following "augmented" alphabets, based on the input alphabet Σ_I and the output alphabet Σ_O. The symbol τ will be used to signify assignment steps.

$\widehat{\Sigma}_I$, $\widehat{\Sigma}_O$, $\widehat{\Sigma}_\tau$, $\widehat{\Sigma}$

Notation 4.2.1. We let $\widehat{\Sigma}_I = \{(\text{in}, \bar{a}) \mid \bar{a} \in \Sigma_I\}$, $\widehat{\Sigma}_O = \{(\text{out}, \bar{b}) \mid \bar{b} \in \Sigma_O\}$, and $\widehat{\Sigma}_\tau = \{\tau\}$. Finally, we set $\widehat{\Sigma} = \widehat{\Sigma}_I \cup \widehat{\Sigma}_O \cup \widehat{\Sigma}_\tau$.

In the following, we define an operational semantics by mapping programs to certain transition systems, which we call *input/output/internal machines*, or IOI machines for short.

IOI machine

Definition 4.2.2. An *IOI machine* $\mathcal{M}$ over the input alphabet Σ_I, the output alphabet Σ_O, and the set of Boolean variables Z is a tuple $(Q, \Delta, \mathit{mem}, Q_{\text{entry}}, Q_{\text{exit}})$, where

- Q is a finite set of states,

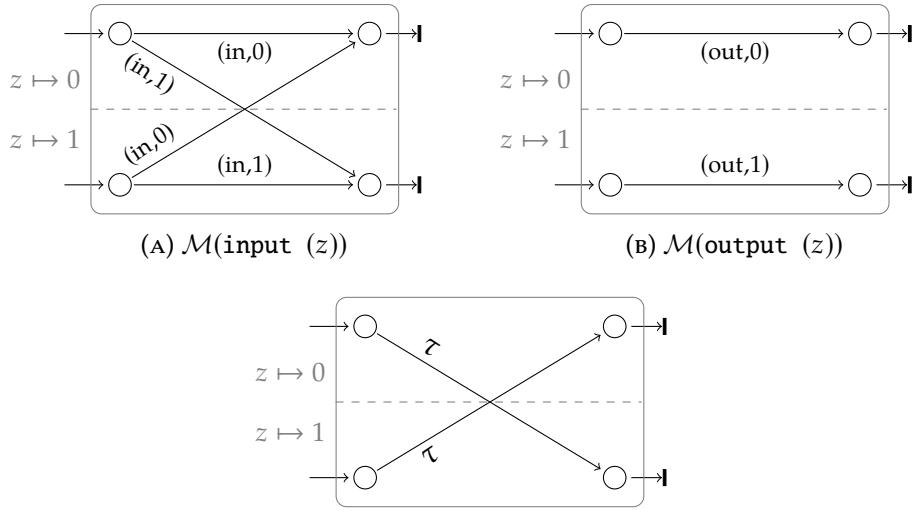
- $\Delta = \Delta_{\text{int}} \cup \Delta_{\text{in}} \cup \Delta_{\text{out}}$ is the transition relation, consisting of
  a set of input transitions $\Delta_{\text{in}} \subseteq Q \times \widehat{\Sigma}_{\text{I}} \times Q$,
  a set of output transitions $\Delta_{\text{out}} \subseteq Q \times \widehat{\Sigma}_{\text{O}} \times Q$,
  a set of internal transitions $\Delta_{\text{int}} \subseteq Q \times \widehat{\Sigma}_{\tau} \times Q$,
- $mem\colon Q \to 2^Z$ is the *memory content function*, which assigns to each state a variable valuation,
- $Q_{\text{entry}} \subseteq Q$ is a set of *entry states* such that for all distinct states $q_1, q_2 \in Q_{\text{entry}}$, we have $mem(q_1) \neq mem(q_2)$, — entry states
- $Q_{\text{exit}} \subseteq Q$ is a set of *exit states* such that for all distinct states $q_1, q_2 \in Q_{\text{exit}}$, we have $mem(q_1) \neq mem(q_2)$. — exit states

With each program p, we associate an IOI machine, denoted by $\mathcal{M}(p)$. — $\mathcal{M}(p)$
We proceed in an inductive manner, starting with the programs that consist of a single statement.

Semantics of atomic programs. An atomic program (consisting of a single assignment, input, or output statement) terminates after a single step, so each transition of the associated IOI machine leads from an entry state to an exit state.

- $\mathcal{M}(z := e) = (Q, \Delta, mem, Q_{\text{entry}}, Q_{\text{exit}})$ with
  - $Q = \{\text{entry}, \text{exit}\} \times 2^Z$,
  - $\Delta = \big\{\big((\text{entry}, \sigma_1), \tau, (\text{exit}, \sigma_2)\big) \mid \sigma_2 = \sigma_1[z/[\![e]\!](\sigma_1)]\big\}$,
  - $mem\big((x, \sigma)\big) = \sigma \quad$ (for $(x, \sigma) \in Q$),
  - $Q_{\text{entry}} = \big\{(\text{entry}, \sigma) \mid \sigma \in 2^Z\big\}$,
  - $Q_{\text{exit}} = \big\{(\text{exit}, \sigma) \mid \sigma \in 2^Z\big\}$.
- $\mathcal{M}(\texttt{input}\ \bar{z}) = (Q, \Delta, mem, Q_{\text{entry}}, Q_{\text{exit}})$ with
  - $Q, Q_{\text{entry}}, Q_{\text{exit}}$ and mem as above,
  - $\Delta = \big\{\big((\text{entry}, \sigma_1), (\text{in}, \bar{a}), (\text{exit}, \sigma_2)\big) \mid \bar{a} \in \Sigma_{\text{I}},\ \sigma_2 = \sigma_1[\bar{z}/\bar{a}]\big\}$.
- $\mathcal{M}(\texttt{output}\ \bar{z}) = (Q, \Delta, mem, Q_{\text{entry}}, Q_{\text{exit}})$ with
  - $Q, Q_{\text{entry}}, Q_{\text{exit}}$ and mem as above,
  - $\Delta = \big\{\big((\text{entry}, \sigma), (\text{out}, \bar{b}), (\text{exit}, \sigma)\big) \mid \bar{b} \in \Sigma_{\text{O}},\ \sigma(\bar{z}) = \bar{b}\big\}$.

Examples of IOI machines for atomic programs that use only a single variable are shown in Figure 2. In the examples, the input and output alphabets are simply $\{0, 1\}$.



(A) $\mathcal{M}(\texttt{input}\ (z))$ (B) $\mathcal{M}(\texttt{output}\ (z))$

(C) $\mathcal{M}(z := \neg z)$

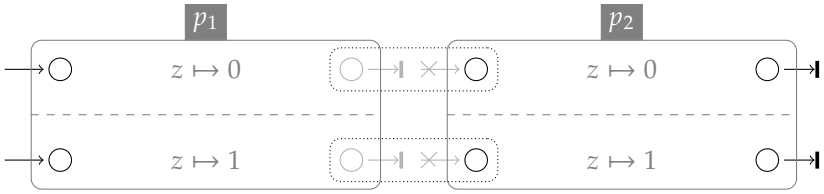
FIGURE 2: Illustration of IOI machines for atomic programs over $Z = \{z\}$. Entry states are indicated by a small incoming arrow $\longrightarrow$, exit states by an outgoing arrow $\rightarrow\!\!|$.

SEMANTICS OF COMPOSITE PROGRAMS. We will now specify the IOI machines for composite programs, which are constructed from the IOI machines of the respective subprograms. These constructions are illustrated in Figure 3. Note that for every program, the corresponding IOI machine has precisely one entry state and precisely one exit state for each variable valuation.

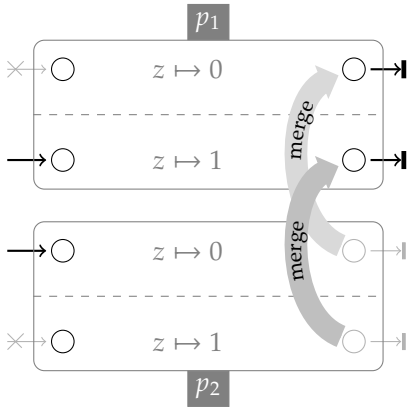
Let p_1, p_2 be two programs and let $\mathcal{M}(p_1)$ and $\mathcal{M}(p_2)$ be their associated IOI machines, with memory content functions mem_1 and mem_2, respectively. We assume that $\mathcal{M}(p_1)$ and $\mathcal{M}(p_2)$ have disjoint sets of states, which can always be achieved by renaming.

The IOI machine of the program "$p_1\,;p_2$" is the series composition ("concatenation") of $\mathcal{M}(p_1)$ and $\mathcal{M}(p_2)$, which is obtained by merging each exit state q_1 of $\mathcal{M}(p_1)$ with the corresponding entry state q_2 of $\mathcal{M}(p_2)$ with the same variable valuation, i.e., with $mem_1(q_1) = mem_2(q_2)$. Such a state q_2 always exists because by construction, $\mathcal{M}(p_2)$ has exactly one entry state for each possible variable valuation. The entry states of $\mathcal{M}(p_1\,;p_2)$ are those of $\mathcal{M}(p_1)$, and the exit states are those of $\mathcal{M}(p_2)$.

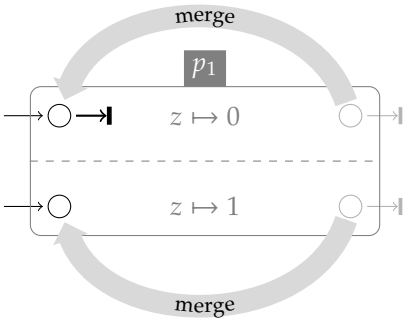
For the program "$\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2$", the corresponding IOI machine is the parallel composition of $\mathcal{M}(p_1)$ and $\mathcal{M}(p_2)$, which is constructed in the following way: The exit states of $\mathcal{M}(p_2)$ are merged with the corresponding exit states (with the same variable valuation) of $\mathcal{M}(p_1)$. Again, the existence of these corresponding states is guaranteed



(A) $\mathcal{M}(p_1 ; p_2)$. Exit states of $\mathcal{M}(p_1)$ are merged with the corresponding entry states of $\mathcal{M}(p_2)$.



(B) $\mathcal{M}(\texttt{if } z \texttt{ then } p_1 \texttt{ else } p_2)$. Note that the new entry states are determined by the branching condition.



(C) $\mathcal{M}(\texttt{while } z \texttt{ do } p_1)$. The entry state violating the loop condition becomes an exit state (with no outgoing transitions).

FIGURE 3: Illustration of IOI machines for composite programs over $Z = \{z\}$. States that get merged into others are grayed out.

by construction of $\mathcal{M}(p_2)$. Furthermore, we define a new set of entry states, which consists of those entry states q of $\mathcal{M}(p_1)$ whose variable valuation $mem_1(q)$ satisfies the condition e and those entry states q' of $\mathcal{M}(p_2)$ whose variable valuation $mem_2(q')$ violates e.

Finally, we obtain the IOI machine for "$\texttt{while } e \texttt{ do } p_1$" by merging the exit states of $\mathcal{M}(p_1)$ with the corresponding entry states with the same variable valuation. The set of entry states is the same as in $\mathcal{M}(p_1)$, but the set of exit states now consists of those entry states q whose variable valuation $mem_1(q)$ violates the loop condition e.

4.3 Computations, Traces, and I/O Sequences

In this section, we use the operational semantics introduced in Section 4.2 to define the computations of a program and also the "traces" and input/output sequences of these computations.

4.3.1 Computations and Traces

computation

Let p be a program and let $\mathcal{M}(p)$ be the corresponding IOI machine. A *computation* ϱ of the program p is a sequence

$$q_1 \xrightarrow{c_1} q_2 \xrightarrow{c_2} q_3 \xrightarrow{c_3} q_4 \xrightarrow{c_4} \cdots$$

of subsequent transitions of $\mathcal{M}(p)$ that starts with an entry state and either ends with an exit state or continues ad infinitum. We call ϱ an *initialized* computation if the variable valuation $\sigma = mem(q_1)$ at the beginning of ϱ assigns to all variables the value 0.

initialized computation

trace

The *trace* of the computation ϱ is the sequence over the alphabet $\widehat{\Sigma}$ that is induced by ϱ:

$$\text{trace}(\varrho) = c_1 c_2 c_3 c_4 \ldots$$

The set of *finite traces* and the set of *infinite traces* of the program p are defined as follows:

FinTraces(p)

$$\text{FinTraces}(p) = \{w \in \widehat{\Sigma}^* \mid w = \text{trace}(\varrho) \text{ for some initialized finite computation } \varrho \text{ of } p\},$$

InfTraces(p)

$$\text{InfTraces}(p) = \{\alpha \in \widehat{\Sigma}^\omega \mid \alpha = \text{trace}(\varrho) \text{ for some initialized infinite computation } \varrho \text{ of } p\}.$$

4.3.2 I/O Sequences and Reactive Programs

input sequence

output sequence

For an infinite trace $\alpha \in \widehat{\Sigma}^\omega$, let $(\text{in}, \bar{a}_1)(\text{in}, \bar{a}_2)(\text{in}, \bar{a}_3)\ldots$ be the sequence that is obtained from α by removing all symbols except for those in $\widehat{\Sigma}_\text{I}$. We call $\bar{a}_1\bar{a}_2\bar{a}_3\ldots$ the *input sequence* of the trace. Similarly, consider the sequence $(\text{out}, \bar{b}_1)(\text{out}, \bar{b}_2)(\text{out}, \bar{b}_3)\ldots$ that is obtained from α by removing all symbols except for those in $\widehat{\Sigma}_\text{O}$. We call $\bar{b}_1\bar{b}_2\bar{b}_3\ldots$ the *output sequence* of the trace. If both the input sequence and the output sequence are infinite, then we say that the trace α yields an infinite *input/output sequence*, or *I/O sequence* for short, defined as

I/O sequence io(α)

$$\text{io}(\alpha) = (\bar{a}_1, \bar{b}_1)(\bar{a}_2, \bar{b}_2)(\bar{a}_3, \bar{b}_3)\ldots \in (\Sigma_\text{I} \times \Sigma_\text{O})^\omega.$$

If the trace α does not yield an infinite I/O sequence, we set $\text{io}(\alpha) = \bot$. Furthermore, we define

io(p)

$$\text{io}(p) = \{\text{io}(\alpha) \mid \alpha \in \text{InfTraces}(p)\}.$$

We call a program p *reactive* if it has no finite traces and all its infinite traces yield infinite I/O sequences – that is, if $\text{FinTraces}(p) = \emptyset$ and $\bot \notin \text{io}(p)$. Intuitively, a reactive program never ceases to read input and to write output.

reactive program

4.3.3 Delay

For a finite trace $w \in \widehat{\Sigma}^*$, there might be a difference between the length of the input sequence of w and the length of the output sequence of w, written as $|w|_{\widehat{\Sigma}_\text{I}} - |w|_{\widehat{\Sigma}_\text{O}}$. We call this difference the *delay* between the input sequence and the output sequence.

Let $k_1, k_2 \in \mathbb{N}$. We say that a finite or infinite trace α has $[-k_1, k_2]$-*bounded delay* if for all finite prefixes w of α, the delay is within the bounds $[-k_1, k_2]$, that is, if $|w|_{\widehat{\Sigma}_\text{I}} - |w|_{\widehat{\Sigma}_\text{O}} \in \{-k_1, \ldots, k_2\}$. Moreover, if all finite and infinite traces of a program p have $[-k_1, k_2]$-bounded delay, then we say that p has $[-k_1, k_2]$-bounded delay. In particular, a program with $[0, 1]$-bounded delay always alternates between reading input and writing output, so we call it a *program with strict I/O alternation*.

$[-k_1, k_2]$-bounded delay

strict I/O alternation

5

Programs, Trees, and Tree Automata

Our approach to the synthesis of structured reactive programs, which we will present in Chapter 6, is based on the fact that programs can be viewed as finite trees. In this chapter, we specify the tree representation of a program in a formal way. We also define automata on finite trees, which will be the main tool in our synthesis procedure.

5.1 Structured Programs as Trees

We begin by defining trees over ranked alphabets. A *ranked alphabet* is a finite alphabet Σ that is partitioned into sets $\Sigma_0, \dots, \Sigma_m$.

ranked alphabet

Definition 5.1.1. Let $\Sigma = \Sigma_0 \cup \dots \cup \Sigma_m$ be a ranked alphabet. The set T_Σ of finite *trees* over Σ is inductively defined as follows:

- Every symbol $a \in \Sigma_0$ is a tree, that is, $a \in T_\Sigma$.
- If $t_1, \dots, t_i \in T_\Sigma$ and $a \in \Sigma_i$, then $a(t_1, \dots, t_i) \in T_\Sigma$.

T_Σ, finite tree

Alternatively, we can view a tree t over $\Sigma = \Sigma_0 \cup \dots \cup \Sigma_m$ as a labeling $t\colon \mathrm{Dom}_t \to \Sigma$ of a prefix-closed domain $\mathrm{Dom}_t \subseteq \{1, \dots, m\}^*$ such that for every $u \in \mathrm{Dom}_t$ with a label $t(u) \in \Sigma_i$, we have $uj \in \mathrm{Dom}_t$ if and only if $j \in \{1, \dots, i\}$. This is illustrated in Figure 4. The elements of Dom_t are called the *nodes* of the tree. In particular, the node ε is called the *root node* and a node u is called a *leaf node* if $t(u) \in \Sigma_0$. The *height* of t is defined as $\max\{|u| \mid u \in \mathrm{Dom}_t\}$.

height

Programs over a set of variables Z can be interpreted as trees over the ranked alphabet $\Sigma^Z_{\mathrm{prog}} = \Sigma^Z_0 \cup \Sigma^Z_1 \cup \Sigma^Z_2 \cup \Sigma^Z_3$, where

- $\Sigma^Z_0 = \{\texttt{true}, \texttt{false}\} \cup Z \cup$
  $\{\texttt{input}\ \ \bar{z} \mid \bar{z} \in Z^{N_\mathrm{I}}\} \cup \{\texttt{output}\ \ \bar{z} \mid \bar{z} \in Z^{N_\mathrm{O}}\}$,
- $\Sigma^Z_1 = \{\neg\} \cup \{\texttt{assign-}z \mid z \in Z\}$,
- $\Sigma^Z_2 = \{\vee, \wedge\} \cup \{\text{“;”}, \texttt{while}\}$,
- $\Sigma^Z_3 = \{\texttt{if}\}$.

tree(p)

The tree representation of a program p, denoted by tree(p) and also known as the syntax tree of p, is defined inductively. First, we specify the tree representation of expressions:

$$\begin{aligned}
\text{tree}(\texttt{true}) &= \texttt{true} \\
\text{tree}(\texttt{false}) &= \texttt{false} \\
\text{tree}(z) &= z \quad (\text{for } z \in Z) \\
\text{tree}(\neg e_1) &= \neg(\text{tree}(e_1)) \\
\text{tree}(e_1 \wedge e_2) &= \wedge(\text{tree}(e_1), \text{tree}(e_2)) \\
\text{tree}(e_1 \vee e_2) &= \vee(\text{tree}(e_1), \text{tree}(e_2))
\end{aligned}$$

The tree representation of programs is now defined as follows:

$$\begin{aligned}
\text{tree}(\texttt{input } \bar{z}) &= \texttt{input } \bar{z} \\
\text{tree}(\texttt{output } \bar{z}) &= \texttt{output } \bar{z} \\
\text{tree}(z := e) &= \texttt{assign-}z\,(\text{tree}(e)) \\
\text{tree}(p_1\,;p_2) &= \texttt{;}(\text{tree}(p_1), \text{tree}(p_2)) \\
\text{tree}(\texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2) &= \texttt{if}\,(\text{tree}(e), \text{tree}(p_1), \text{tree}(p_2)) \\
\text{tree}(\texttt{while } e \texttt{ do } p_1) &= \texttt{while}\,(\text{tree}(e), \text{tree}(p_1))
\end{aligned}$$

We use programs and their tree representations interchangeably, so we simply write p instead of tree(p) to refer to the tree representation of p. Figure 4 shows a simple program and its tree representation.

5.2 Automata on Finite Trees

To recognize certain sets of programs (represented as trees), we will use tree automata (see [CDG+07]).

nondeterministic tree automaton (NTA)

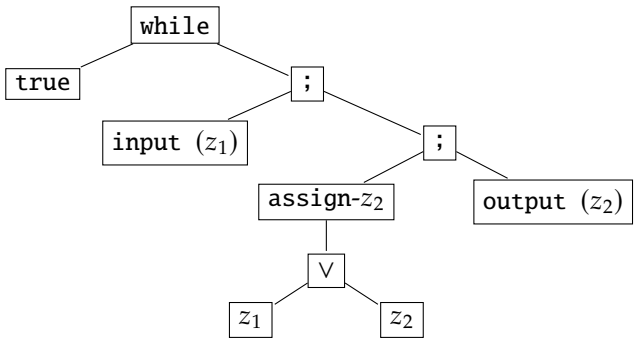
Definition 5.2.1. A *nondeterministic tree automaton (NTA)* $\mathcal{C}$ is a tuple (Q, Σ, Δ, F), where

- Q is a finite set of states,
- $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_m$ is a ranked alphabet,
- $\Delta \subseteq \bigcup_{i=0}^{m}(Q^i \times \Sigma_i \times Q)$ is the transition relation, and
- $F \subseteq Q$ is the set of final states.

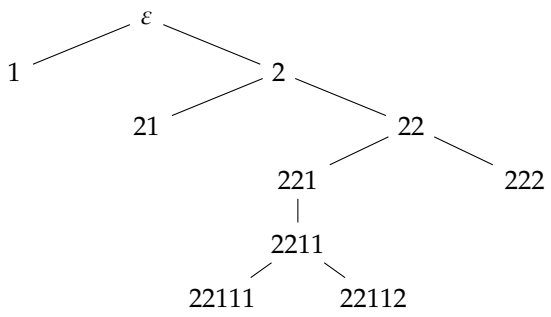
A *run* of such an NTA $\mathcal{C}$ on a given tree $t\colon \text{Dom}_t \to \Sigma$ is a labeling $\pi\colon \text{Dom}_t \to Q$ that assigns to each tree node u a state of $\mathcal{C}$ such that

```
while true do {
    input (z1);
    z2 := z1 ∨ z2;
    output (z2)
}
```

(A) A simple program.



(B) Tree representation of the program above.



(C) Domain of the tree above.

FIGURE 4: A program, its tree representation, and the tree domain.

- if $t(u) \in \Sigma_0$, then $\big(t(u), \pi(u)\big) \in \Delta$, and
- if $t(u) \in \Sigma_i$ with $i \geq 1$, then $\big(\pi(u1), \ldots, \pi(ui),\ t(u),\ \pi(u)\big) \in \Delta$.

The NTA $\mathcal{C}$ accepts a tree t if there exists a run π of $\mathcal{C}$ on t such that the state at the root node ε is a final state, i.e., $\pi(\varepsilon) \in F$. The set of trees *recognized* by the NTA $\mathcal{C}$ is defined as

$$L(\mathcal{C}) = \{t \in T_\Sigma \mid \mathcal{C} \text{ accepts } t\}.$$

We will specifically consider *deterministic* tree automata, where the transition relation is in fact a function.

deterministic tree automaton (DTA)

Definition 5.2.2. A *deterministic tree automaton (DTA)* $\mathcal{C}$, also known as a deterministic bottom-up tree automaton, is a tuple (Q, Σ, δ, F), where

- Q is a finite set of states,
- $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_m$ is a ranked alphabet,
- $\delta\colon \bigcup_{i=0}^{m}(Q^i \times \Sigma_i) \to Q$ is the transition function, and
- $F \subseteq Q$ is the set of final states.

A deterministic tree automaton $\mathcal{C}$ has exactly one run on each given tree $t \in T_\Sigma$. We view this run as an evaluation of the tree in a bottom-up manner, from the leaf nodes to the root node.

Finally, let us recapitulate the following well-known result.

Proposition 5.2.3. Given a nondeterministic (or deterministic) tree automaton $\mathcal{C} = (Q, \Sigma, \Delta, F)$ over the alphabet $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_m$, we can, in polynomial time, construct a tree $t \in L(\mathcal{C})$ or determine that no such tree exists.

Proof. We start by determining the set of states

$$Q_0 = \{q \in Q \mid (a, q) \in \Delta \text{ for some } a \in \Sigma_0\},$$

which are reachable via trees of height 0. In addition, we store for each of these states q a corresponding tree t_q witnessing the reachability of q. Formally, for $q \in Q_0$, we let $t_q = a$ for some $a \in \Sigma_0$ with $(a, q) \in \Delta$.

Then, for $\ell = 1, 2, \ldots$, we compute the set

$$Q_\ell = \{q \in Q \mid (q_1, \ldots, q_i, a, q) \in \Delta \text{ for some } i \in \{1, \ldots, m\},\ q_1, \ldots, q_i \in Q_{\ell-1},\ a \in \Sigma_i\}$$

of states that are reachable via trees of height ℓ. Again, we store a corresponding tree (of height ℓ) along with each of the newly added states $q \in Q_\ell \setminus Q_{\ell-1}$, by letting $t_q = a(t_{q_1}, \ldots, t_{q_i})$ for some $a \in \Sigma_i$ and $q_1, \ldots, q_i \in Q_{\ell-1}$ with $(q_1, \ldots, q_i, a, q) \in \Delta$. Note that t_q can be represented using pointers to the previously constructed subtrees, preventing an exponential blow-up.

We stop as soon as we have either found a final state or we have $Q_\ell = Q_{\ell-1}$. In the latter case, we have computed all reachable states without finding a final state, so the language recognized by $\mathcal{C}$ is empty. In the former case, we have found a reachable state $q \in F$ and can

present the tree t_q as a witness for nonemptiness. Since we consider trees with increasing height in each step, this tree t_q is minimal with respect to its height among all the trees accepted by $\mathcal{C}$.

Note that this algorithm will always terminate after at most $|Q|$ steps, since we add at least one state to the set of reachable states in every step. Furthermore, each step can be performed in time polynomial in $|Q|$. □

6

Synthesis Using Deterministic Tree Automata

6.1 Overview

In this chapter, we present a procedure to derive a reactive program from a given specification defining the admissible I/O sequences.

Definition 6.1.1. A *specification* is an ω-language $R \subseteq (\Sigma_I \times \Sigma_O)^\omega$, and we say that a program p *satisfies* R if p is reactive and $\mathrm{io}(p) \subseteq R$.

specification

Specifically, we consider the following variant of Church's Synthesis Problem, which we call the *synthesis problem for reactive programs*:

Given a finite set of Boolean variables Z and given an ω-regular specification $R \subseteq (\Sigma_I \times \Sigma_O)^\omega$, construct a program p over Z with strict I/O alternation that satisfies R, or determine that no such program exists.

synthesis problem for reactive programs

In game-theoretical terminology, the specification defines a winning condition for Player II in a Gale-Stewart game, and we are looking for a winning strategy for Player II in the form of a program over the variables Z.

Madhusudan [Mad11] showed that the synthesis problem for reactive programs can be solved effectively by constructing a tree automaton recognizing precisely the set of programs with the desired properties. More specifically, his solution is based on two-way alternating co-Büchi tree automata, which can then be transformed into nondeterministic tree automata. In this chapter, we present an alternative to this approach by constructing directly a deterministic tree automaton recognizing the "correct" programs.

In fact, we prove the slightly more general result stated in Theorem 6.1.2 below, which allows us to choose arbitrary bounds $[-k_1, k_2]$ for the delay of the programs. To allow only programs with strict I/O alternation, we can choose $k_1 = 0$ and $k_2 = 1$. For fixed k_1, k_2, the size of our tree automaton matches the size of the nondeterministic tree

automaton resulting from Madhusudan's algorithm. By the *size* of an automaton we mean the number of its states.

THEOREM 6.1.2. Given a finite set of Boolean variables Z, given $k_1, k_2 \in \mathbb{N}$, and given an ω-regular specification $R \subseteq (\Sigma_I \times \Sigma_O)^\omega$, represented by a nondeterministic Büchi automaton $\mathcal{A}$ recognizing the complement of R, we can construct a deterministic tree automaton that recognizes the set of programs over Z with $[-k_1, k_2]$-bounded delay that satisfy R. The size of this tree automaton is doubly exponential in $|Z|$, k_1, and k_2, and exponential in the size of $\mathcal{A}$.

The polynomial-time emptiness test described in Proposition 5.2.3 can then be employed to obtain one of the programs accepted by the tree automaton if any such program exists – in fact, a *minimal* program (with respect to the height of its syntax tree) can be extracted.

In the remainder of this chapter, we present a proof of Theorem 6.1.2. In Section 6.2, we characterize the "correct" programs, i.e., the programs to be accepted by the tree automaton, in terms of their traces. Moreover, given a nondeterministic Büchi automaton $\mathcal{A}$ recognizing the complement of the specification R, we construct a nondeterministic Büchi automaton $\mathcal{B}_{\text{bad}}$ that accepts precisely the "bad" infinite traces, which render a program "incorrect".

In Section 6.3, we outline the construction at the heart of our synthesis procedure, where we use this Büchi automaton $\mathcal{B}_{\text{bad}}$ to build the deterministic bottom-up tree automaton recognizing the "correct" programs as stated in Theorem 6.1.2. Finally, in Section 6.4, we provide the formal construction of that tree automaton.

6.2 Recognizing the Bad Traces

As a first step toward solving the synthesis problem for reactive programs, we characterize the programs that are valid solutions in terms of their traces.

The synthesis problem asks for a program over the given set of variables Z that has $[-k_1, k_2]$-bounded delay and satisfies the given specification R. In other words, we want to construct a program without finite traces and with the property that every infinite trace α of the program has $[-k_1, k_2]$-bounded delay and yields an infinite I/O sequence $\text{io}(\alpha) \in R$. The following definition captures all infinite traces that would *violate* this property.

Definition 6.2.1. Let $R \subseteq (\Sigma_I \times \Sigma_O)^\omega$ be a specification. Let $k_1, k_2 \in \mathbb{N}$. The set of *bad infinite traces* with respect to R and the delay bounds $[-k_1, k_2]$ is the following ω-language over the alphabet $\widehat{\Sigma}$ (see Notation 4.2.1): bad infinite traces

$$\mathrm{BadTraces}(R, k_1, k_2) = L_\perp \cup L_{\mathrm{delay}} \cup L_{\mathrm{spec}},$$

where

$$\begin{aligned} L_\perp &= \{\alpha \in \widehat{\Sigma}^\omega \mid \mathrm{io}(\alpha) = \perp\}, \\ L_{\mathrm{delay}} &= \{\alpha \in \widehat{\Sigma}^\omega \mid \alpha \text{ does } \textit{not} \text{ have } [-k_1, k_2]\text{-bounded delay}\}, \\ L_{\mathrm{spec}} &= \{\alpha \in \widehat{\Sigma}^\omega \mid \alpha \text{ has } [-k_1, k_2]\text{-bounded delay and} \\ &\qquad \mathrm{io}(\alpha) \in (\Sigma_I \times \Sigma_O)^\omega \setminus R\}. \end{aligned}$$

The desired programs can now be characterized as follows.

Proposition 6.2.2. Let $R \subseteq (\Sigma_I \times \Sigma_O)^\omega$ be a specification. Let $k_1, k_2 \in \mathbb{N}$. A program p has $[-k_1, k_2]$-bounded delay and satisfies R if and only if

- $\mathrm{FinTraces}(p) = \emptyset$ and
- $\mathrm{InfTraces}(p) \cap \mathrm{BadTraces}(R, k_1, k_2) = \emptyset$.

Our next goal is to construct a nondeterministic Büchi automaton $\mathcal{B}_{\mathrm{bad}}$ that recognizes the set of bad infinite traces $\mathrm{BadTraces}(R, k_1, k_2)$. $\mathcal{B}_{\mathrm{bad}}$

Proposition 6.2.3. Given a nondeterministic Büchi automaton $\mathcal{A}$ recognizing the complement of a specification $R \subseteq (\Sigma_I \times \Sigma_O)^\omega$ and given $k_1, k_2 \in \mathbb{N}$, we can construct a nondeterministic Büchi automaton $\mathcal{B}_{\mathrm{bad}}$ recognizing $\mathrm{BadTraces}(R, k_1, k_2)$. The size of $\mathcal{B}_{\mathrm{bad}}$ is polynomial in the size of $\mathcal{A}$ and exponential in k_1 and k_2.

Proof. We obtain the desired automaton by combining automata recognizing the ω-languages $L_\perp$, L_{delay}, and L_{spec} from Definition 6.2.1.

First, the ω-language $L_\perp$ consisting of the infinite traces that do not yield infinite I/O sequences can be written as $L_\perp = \widehat{\Sigma}^* \cdot (\widehat{\Sigma}_I \cup \widehat{\Sigma}_\tau)^\omega \cup \widehat{\Sigma}^* \cdot (\widehat{\Sigma}_O \cup \widehat{\Sigma}_\tau)^\omega$ and is recognized by a simple nondeterministic Büchi automaton $\mathcal{B}_\perp$ that can switch nondeterministically, at any point, to an accepting state with a $(\widehat{\Sigma}_I \cup \widehat{\Sigma}_\tau)$-self-loop or to an accepting state with a $(\widehat{\Sigma}_O \cup \widehat{\Sigma}_\tau)$-self-loop.

Secondly, the ω-language L_{delay} consisting of the infinite traces that violate the delay bounds can be recognized by a nondeterministic Büchi

automaton that keeps track of the current delay while reading a given trace and switches to an accepting state if the delay exceeds the bounds. Formally, we let $\mathcal{B}_{\text{delay}} = (S^{\text{delay}}, \widehat{\Sigma}, \Delta^{\text{delay}}, s_0^{\text{delay}}, F^{\text{delay}})$, where

- $S^{\text{delay}} = \{-k_1, \dots, k_2\} \cup \{\top\}$,
- $s_0^{\text{delay}} = 0$,
- $F^{\text{delay}} = \{\top\}$,
- $\Delta^{\text{delay}} = \big\{\big(n, (\text{in}, \bar{a}), n+1\big) \mid n \in \{-k_1, \dots, k_2 - 1\},\ \bar{a} \in \Sigma_{\text{I}}\big\}$
  $\cup \big\{\big(n, (\text{out}, \bar{b}), n-1\big) \mid n \in \{-k_1 + 1, \dots, k_2\},\ \bar{b} \in \Sigma_{\text{O}}\big\}$
  $\cup \big\{\big(n, \tau, n\big) \mid n \in \{-k_1, \dots, k_2\}\big\}$
  $\cup \big\{\big(k_2, (\text{in}, \bar{a}), \top\big) \mid \bar{a} \in \Sigma_{\text{I}}\big\}$
  $\cup \big\{\big(-k_1, (\text{out}, \bar{b}), \top\big) \mid \bar{b} \in \Sigma_{\text{O}}\big\}$
  $\cup \big\{\big(\top, c, \top\big) \mid c \in \widehat{\Sigma}\big\}$.

Thirdly, we consider the ω-language L_{spec} of the infinite traces with $[-k_1, k_2]$-bounded delay that yield infinite I/O sequences violating the specification R. We construct a nondeterministic Büchi automaton $\mathcal{B}_{\text{spec}}$ that simulates the automaton $\mathcal{A}$ recognizing the complement of R on the I/O sequence of the given trace, using a bounded "buffer" to handle the delay. Formally, given the Büchi automaton $\mathcal{A} = (S, \Sigma_{\text{I}} \times \Sigma_{\text{O}}, \Delta, s_0, F)$, we define $\mathcal{B}_{\text{spec}} = (S', \widehat{\Sigma}, \Delta', s_0', F')$ with

- $S' = S \times \big(\widehat{\Sigma}_{\text{I}}^{\leq k_2} \cup \widehat{\Sigma}_{\text{O}}^{\leq k_1}\big) \times \{0, 1\}$,
  where the last component is used to indicate whether a final state of $\mathcal{A}$ has just been reached,
- $s_0' = (s_0, \varepsilon, 0)$,
- $F' = \{(s, u, i) \in S' \mid i = 1\}$,
- $\Delta' = \big\{\big((s, u, i), (\text{in}, \bar{a}), (s, u \cdot (\text{in}, \bar{a}), 0)\big) \mid$
  $s \in S,\ u \in \widehat{\Sigma}_{\text{I}}^{<k_2},\ i \in \{0, 1\},\ a \in \Sigma_{\text{I}}\big\}$
  $\cup \big\{\big((s, u, i), (\text{out}, \bar{b}), (s, u \cdot (\text{out}, \bar{b})), 0)\big) \mid$
  $s \in S,\ u \in \widehat{\Sigma}_{\text{O}}^{<k_1},\ i \in \{0, 1\},\ \bar{b} \in \Sigma_{\text{O}}\big\}$
  $\cup \big\{\big((s, (\text{out}, \bar{b}) \cdot u, i), (\text{in}, \bar{a}), (s', u, j)\big) \mid$
  $(s, (\bar{a}, \bar{b}), s') \in \Delta,\ u \in \widehat{\Sigma}_{\text{O}}^{<k_1},\ i, j \in \{0, 1\},\ j = 1 \text{ iff } s' \in F\big\}$
  $\cup \big\{\big((s, (\text{in}, \bar{a}) \cdot u, i), (\text{out}, \bar{b}), (s', u, j)\big) \mid$
  $(s, (\bar{a}, \bar{b}), s') \in \Delta,\ u \in \widehat{\Sigma}_{\text{I}}^{<k_2},\ i, j \in \{0, 1\},\ j = 1 \text{ iff } s' \in F\big\}$
  $\cup \big\{\big((s, u, i), \tau, (s, u, 0)\big) \mid (s, u, i) \in S'\big\}$.

Combining the Büchi automata $\mathcal{B}_\perp$, $\mathcal{B}_{\text{delay}}$, and $\mathcal{B}_{\text{spec}}$, we can assemble the desired Büchi automaton $\mathcal{B}_{\text{bad}}$ recognizing $\text{BadTraces}(R, k_1, k_2) = L_\perp \cup L_{\text{delay}} \cup L_{\text{spec}}$. The size of this automaton is polynomial in the size of $\mathcal{A}$ and exponential in k_1 and k_2. □

6.3 Idea for Recognizing the Correct Programs

Our goal is to prove Theorem 6.1.2, that is, we want to construct a DTA recognizing the programs over the variables Z that have $[-k_1, k_2]$-bounded delay and satisfy the given specification R. By Proposition 6.2.2, these are precisely the programs without finite traces and without bad infinite traces in the sense of Definition 6.2.1. We can construct a nondeterministic Büchi automaton $\mathcal{B}_{\text{bad}}$ recognizing the bad infinite traces, as stated in Proposition 6.2.3, so it suffices to show the following proposition.

Proposition 6.3.1. Given a finite set of Boolean variables Z and given a nondeterministic Büchi automaton $\mathcal{B}_{\text{bad}}$ recognizing a set of traces $L(\mathcal{B}_{\text{bad}}) \subseteq \widehat{\Sigma}^\omega$, we can construct a DTA $\mathcal{C}$ recognizing the set of trees

$$L(\mathcal{C}) = \Big\{ p \in T_{\Sigma_{\text{prog}}^Z} \mid \text{FinTraces}(p) = \emptyset \text{ and } \text{InfTraces}(p) \cap L(\mathcal{B}_{\text{bad}}) = \emptyset \Big\}.$$

The size of this DTA is doubly exponential in $|Z|$ and exponential in the size of $\mathcal{B}_{\text{bad}}$.

To simplify the construction, we assume that $\mathcal{B}_{\text{bad}}$ can never "get stuck". This can be guaranteed by adding a non-accepting "sink state" (with a self-loop for all symbols) and adding, for every $c \in \widehat{\Sigma}$, to each state without outgoing c-transitions a new c-transition leading to that sink state. Then for every computation of a given program, there is at least one corresponding run of $\mathcal{B}_{\text{bad}}$ on the resulting trace.

Definition 6.3.2. A (finite or infinite) *co-execution* of a program p with a nondeterministic Büchi automaton $\mathcal{B}_{\text{bad}}$ over the alphabet $\widehat{\Sigma}$ is a pair (ϱ, π) where ϱ is a (finite or infinite) computation of p and π is a run of $\mathcal{B}_{\text{bad}}$ on the trace of that computation.

co-execution

Let S be the set of states of $\mathcal{B}_{\text{bad}}$. We say that a co-execution (ϱ, π) starts with $(\sigma, s) \in 2^Z \times S$ if σ is the variable valuation at the beginning

of the computation ϱ and s is the state at the beginning of the run π. If ϱ is finite and ends with the variable valuation σ', and if the run π ends with the state s', then we call (ϱ, π) a co-execution from (σ, s) to (σ', s'). Furthermore, we say that a co-execution is *initialized* if it starts with (σ_0, s_0), where σ_0 is the initial variable valuation, which assigns to all variables the value 0, and s_0 is the initial state of $\mathcal{B}_{\text{bad}}$.

initialized co-execution

The DTA $\mathcal{C}$ specified in Proposition 6.3.1 will evaluate a given program p in a bottom-up manner, thereby assigning one of its states to each node of the program tree. Intuitively, the DTA will inductively compute a "summary" of the co-executions of p with $\mathcal{B}_{\text{bad}}$. The program p will be accepted if the summary at the root node indicates that there are no initialized finite co-executions (i.e., $\text{FinTraces}(p) = \emptyset$) and that there are no initialized infinite co-executions (ϱ, π) such that π is an accepting run of $\mathcal{B}_{\text{bad}}$ (i.e., $\text{InfTraces}(p) \cap L(\mathcal{B}_{\text{bad}}) = \emptyset$).

Formally, the summary computed by the DTA consists of the following "signatures" of the program p.

Definition 6.3.3. Let p be a program over the set of variables Z and let $\mathcal{B}_{\text{bad}}$ be a nondeterministic Büchi automaton over $\widehat{\Sigma}$ with state set S. The *finite co-execution signature* of p with respect to $\mathcal{B}_{\text{bad}}$, denoted by $\text{FinSig}(p, \mathcal{B}_{\text{bad}})$, is defined as follows:

finite co-execution signature $\text{FinSig}(p, \mathcal{B}_{\text{bad}})$

$$\text{FinSig}(p, \mathcal{B}_{\text{bad}}) = \big\{\big((\sigma, s), f, (\sigma', s')\big) \in (2^Z \times S) \times \{0,1\} \times (2^Z \times S) \mid \text{there is a finite co-execution } (\varrho, \pi) \text{ of } p \text{ with } \mathcal{B}_{\text{bad}} \text{ from } (\sigma, s) \text{ to } (\sigma', s') \text{ such that } f = 1 \text{ iff } \pi \text{ contains an accepting state}\big\}$$

The *infinite co-execution signature* of p with respect to $\mathcal{B}_{\text{bad}}$, denoted by $\text{InfSig}(p, \mathcal{B}_{\text{bad}})$, is defined as follows:

infinite co-execution signature $\text{InfSig}(p, \mathcal{B}_{\text{bad}})$

$$\text{InfSig}(p, \mathcal{B}_{\text{bad}}) = \big\{(\sigma, s) \in 2^Z \times S \mid \text{there is an infinite co-execution } (\varrho, \pi) \text{ of } p \text{ with } \mathcal{B}_{\text{bad}} \text{ starting with } (\sigma, s) \text{ such that } \pi \text{ infinitely often visits an accepting state}\big\}$$

As we will see in the next section, these signatures can indeed be determined inductively by a DTA.

6.4 Construction of the Tree Automaton

We will now provide the formal construction of the DTA $\mathcal{C}$ accepting precisely the "correct" programs over the variables Z as stated in Propo-

sition 6.3.1. That is to say, given a nondeterministic Büchi automaton $\mathcal{B}_{\text{bad}} = (S, \widehat{\Sigma}, \Delta, s_0, F)$ recognizing the set of bad infinite traces, we construct a DTA $\mathcal{C}$ that accepts a program p if and only if $\text{FinTraces}(p) = \emptyset$ and $\text{InfTraces}(p) \cap L(\mathcal{B}_{\text{bad}}) = \emptyset$. As mentioned in Section 6.3, we assume that $\mathcal{B}_{\text{bad}}$ has at least one run on each trace.

We build $\mathcal{C} = (Q, \Sigma^Z_{\text{prog}}, \delta, F_{\mathcal{C}})$ in such a way that the state that is reached at the root of a tree $t \in \Sigma^Z_{\text{prog}}$ is

- $\big(\text{FinSig}(p, \mathcal{B}_{\text{bad}}), \text{InfSig}(p, \mathcal{B}_{\text{bad}})\big)$ if t represents a program p,
- $\big\{\sigma \in 2^Z \mid [\![e]\!](\sigma) = 1\big\}$ if t represents an expression e,
- $\bot$ otherwise.

The set of states of $\mathcal{C}$ is therefore $Q = Q_{\text{prog}} \cup Q_{\text{expr}} \cup \{\bot\}$, where

- $Q_{\text{prog}} = \big\{(X, Y) \mid X \subseteq (2^Z \times S) \times \{0,1\} \times (2^Z \times S),\ Y \subseteq 2^Z \times S\big\}$,
- $Q_{\text{expr}} = \big\{V \subseteq 2^Z\big\}$.

Note that the number of states is doubly exponential in the number of variables $|Z|$ and exponential in the size of $\mathcal{B}_{\text{bad}}$.

The final states are the pairs of finite and infinite co-execution signatures that indicate a "correct" program – one that has no initialized finite co-executions, and no initialized infinite co-executions involving an accepting run of $\mathcal{B}_{\text{bad}}$. For the formal definition, let σ_0 be the initial variable valuation, where all variables have the value 0, and recall that s_0 is the initial state of $\mathcal{B}_{\text{bad}}$. We set

$$F_{\mathcal{C}} = \big\{(X, Y) \in Q_{\text{prog}} \mid \big((\sigma_0, s_0), f, (\sigma', s')\big) \notin X \text{ for all } f, \sigma', s', \text{ and } (\sigma_0, s_0) \notin Y\big\}.$$

Let us now define the transition function δ of the DTA. In the following, we assume that $V, V_1, V_2 \in Q_{\text{expr}}$ and $(X_1, Y_1), (X_2, Y_2) \in Q_{\text{prog}}$.

(1) Transitions to evaluate expressions.

- $\delta(\texttt{true}) = 2^Z$
- $\delta(\texttt{false}) = \emptyset$
- $\delta(z) = \big\{\sigma \in 2^Z \mid \sigma(z) = 1\big\}$
- $\delta(V_1, V_2, \wedge) = V_1 \cap V_2$
- $\delta(V_1, V_2, \vee) = V_1 \cup V_2$
- $\delta(V, \neg) = 2^Z \setminus V$

(2) Transitions for assignments. A computation of a program "$z := e$" changes the variable valuation according to the value of the expression e and yields the finite trace τ, so we set

$$\delta(V, \texttt{assign-}z) = (X, \emptyset)$$

with

$$X = \Big\{\big((\sigma, s), f, (\sigma[z/c], s')\big) \mid \sigma \in 2^Z,\ c = 1 \text{ iff } \sigma \in V,\ (s, \tau, s') \in \Delta, \text{ and } f = 1 \text{ iff } \{s, s'\} \cap F \neq \emptyset\Big\}.$$

(3) Transitions for input statements. Since a computation of a program "$\texttt{input}\ \bar{z}$" changes the variable valuation according to the given input symbol $\bar{a}$ and yields the finite trace $(\text{in}, \bar{a})$, we set

$$\delta(\texttt{input}\ \bar{z}) = (X, \emptyset)$$

with

$$X = \Big\{\big((\sigma, s), f, (\sigma[\bar{z}/\bar{a}], s)\big) \mid \sigma \in 2^Z,\ \bar{a} \in \Sigma_{\text{I}},\ \big(s, (\text{in}, \bar{a}), s'\big) \in \Delta, \text{ and } f = 1 \text{ iff } \{s, s'\} \cap F \neq \emptyset\Big\}.$$

(4) Transitions for output statements. A computation of a program "$\texttt{output}\ \bar{z}$" does not change the variable valuation σ and yields the finite trace $(\text{out}, \sigma(\bar{z}))$, so we set

$$\delta(\texttt{output}\ \bar{z}) = (X, \emptyset)$$

with

$$X = \Big\{\big((\sigma, s), f, (\sigma, s')\big) \mid \sigma \in 2^Z,\ \big(s, (\text{out}, \sigma(\bar{z})), s'\big) \in \Delta, \text{ and } f = 1 \text{ iff } \{s, s'\} \cap F \neq \emptyset\Big\}.$$

(5) Transitions for concatenation. Co-executions of a program "$p_1 ; p_2$" consist of a co-execution of p_1 and a co-execution of p_2, or solely of an infinite co-execution of p_1, so we set

$$\delta\big((X_1, Y_1), (X_2, Y_2), ;\big) = (X, Y)$$

with

$$X = \Big\{\big((\sigma, s), \max\{f_1, f_2\}, (\sigma'', s'')\big) \mid \text{there exists } (\sigma', s') \text{ such that } \big((\sigma, s), f_1, (\sigma', s')\big) \in X_1 \text{ and } \big((\sigma', s'), f_2, (\sigma'', s'')\big) \in X_2\Big\},$$

and

$$Y = Y_1 \cup \big\{(\sigma, s) \mid \text{there exist } (\sigma', s') \in Y_2 \text{ and } f \text{ such that } \big((\sigma, s), f, (\sigma', s')\big) \in X_1\big\}.$$

(6) Transitions for "if". For a program "`if` e `then` p_1 `else` p_2", a co-execution is either a co-execution of p_1 starting with a variable valuation that satisfies the condition e, or a co-execution of p_2 starting with a variable valuation that violates the condition. Hence, we set

$$\delta\big(V, (X_1, Y_1), (X_2, Y_2), \texttt{if}\big) = (X, Y)$$

with

$$\begin{aligned} X &= \big\{\big((\sigma, s), f, (\sigma', s')\big) \in X_1 \mid \sigma \in V\big\} \\ &\quad \cup \big\{\big((\sigma, s), f, (\sigma', s')\big) \in X_2 \mid \sigma \notin V\big\}, \\ Y &= \big\{(\sigma, s) \in Y_1 \mid \sigma \in V\big\} \\ &\quad \cup \big\{(\sigma, s) \in Y_2 \mid \sigma \notin V\big\}. \end{aligned}$$

(7) Transitions for "while". A finite co-execution of a program "`while` e `do` p_1" can be decomposed into a finite sequence of co-executions of p_1. An infinite co-execution of such a program can either eventually stay inside a loop iteration forever or pass through infinitely many iterations. It can therefore be decomposed either into a finite sequence of co-executions of p_1 followed by an infinite co-execution of p_1, or into a finite sequence of co-executions of p_1 followed by "cycles" of co-executions of p_1 that keep returning to the same variable valuation and the same state of $\mathcal{B}_{\text{bad}}$ (since there are only finitely many valuations and states). Thus, we set

$$\delta\big(V, (X_1, Y_1), \texttt{while}\big) = (X, Y)$$

with

$$\begin{aligned} X &= \big\{\big((\sigma, s), f, (\sigma', s')\big) \in \widehat{X}_1 \mid \sigma \notin V\big\}, \\ Y &= \big\{(\sigma, s) \mid \text{there exist } (\sigma', s') \text{ and } f \text{ such that } \sigma' \in V \\ &\qquad \text{and } \big((\sigma, s), f, (\sigma', s')\big) \in \widehat{X}_1, \text{ and such that} \\ &\qquad (\sigma', s') \in Y_1 \text{ or } \big((\sigma', s'), 1, (\sigma', s')\big) \in \widehat{X}_1\big\}, \end{aligned}$$

where $\widehat{X}_1$ is, intuitively, the reflexive transitive closure of the relation $X_1 \subseteq (2^Z \times S) \times \{0, 1\} \times (2^Z \times S)$. It represents all finite

sequences of consecutive co-executions of p_1 that are compatible with the loop condition e, and it is formally defined as the smallest relation satisfying the following conditions:

- $\big((\sigma, s), 0, (\sigma, s)\big) \in \widehat{X}_1$ for all $(\sigma, s) \in 2^Z \times S$, and
- if $\big((\sigma, s), f_1, (\sigma', s')\big) \in \widehat{X}_1$, $\big((\sigma', s'), f_2, (\sigma'', s'')\big) \in X_1$, and $\sigma' \in V$, then $\big((\sigma, s), \max\{f_1, f_2\}, (\sigma'', s'')\big) \in \widehat{X}_1$.

(8) Transitions for syntactically invalid programs. In all other cases, the tree is not a syntactically correct program or expression, so we set $\delta(\ldots) = \bot$.

This completes the construction of the DTA $\mathcal{C}$ announced in Proposition 6.3.1 and thus concludes the proof of Theorem 6.1.2.

7

A Lower Bound for the Size of the Tree Automata

The size of the DTA constructed in Chapter 6 is doubly exponential in the given number of Boolean variables that the programs may use. We will now show that this cannot be avoided, even if we use nondeterministic tree automata (NTAs).

THEOREM 7.0.1. Let Z be a set of m Boolean variables, let $k_1, k_2 \in \mathbb{N}$, let $R \subseteq (\Sigma_I \times \Sigma_O)^\omega$ be a specification, and let $\mathcal{C}$ be an NTA that recognizes the set of programs over Z with $[-k_1, k_2]$-bounded delay that satisfy R. If that set of programs is nonempty, then $\mathcal{C}$ has at least $2^{2^{m-1}}$ states.

Proof. Let $z_1, \ldots, z_m$ be the Boolean variables in the set Z. There are $2^{2^{m-1}}$ functions of the form $g\colon \{0,1\}^{m-1} \to \{0,1\}$. Each such function g can be implemented by a program p_g that sets z_m to the value $g\big(\sigma(z_1), \ldots, \sigma(z_{m-1})\big)$, where $\sigma \in 2^Z$ is the valuation of the variables before p_g is executed. We will now show that the NTA $\mathcal{C}$ must be able to distinguish all of these programs and hence needs at least $2^{2^{m-1}}$ states.

For each function g as specified above, we can construct another program p_g^{check} that checks whether z_m has the right value according to the function g – if the value is correct, then the program terminates, otherwise it performs an infinite loop without any input statements. Moreover, we can build a program p_g^{verified} that executes $p_g; p_g^{\text{check}}$ for each valuation of the variables $z_1, \ldots, z_{m-1}$, using a `while` loop, and finally resets all variables to 0. Note that all checks will succeed, so p_g^{verified} will terminate.

Now suppose that there is a program p_{sat} over Z with $[-k_1, k_2]$-bounded delay that satisfies the specification R. For each g, the program $p_g^{\text{verified}}; p_{\text{sat}}$ satisfies the specification as well and is therefore accepted by the NTA $\mathcal{C}$. Consider, for each g, an arbitrary accepting run of $\mathcal{C}$ on $p_g^{\text{verified}}; p_{\text{sat}}$, and let q_g denote the state of $\mathcal{C}$ that is visited at the root of the subtree p_g (see Figure 5).

Toward a contradiction, assume that $\mathcal{C}$ has less than $2^{2^{m-1}}$ states. This implies that for at least two distinct functions g_1 and g_2, the states q_{g_1}

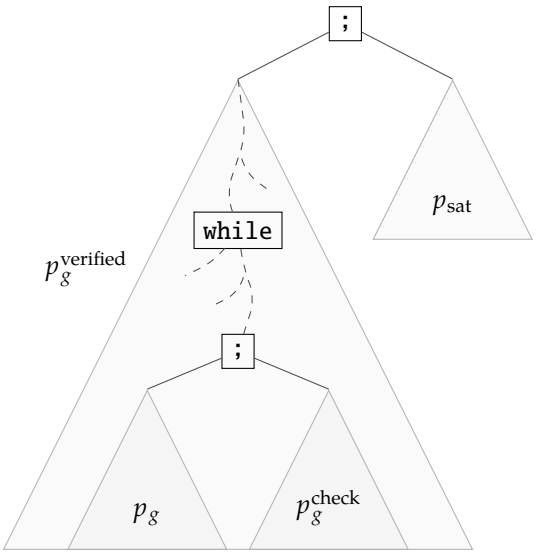


FIGURE 5: Illustration of the program $p_g^{\text{verified}}; p_{\text{sat}}$.

and q_{g_2} are equal. Therefore, if we modify the program $p_{g_1}^{\text{verified}}; p_{\text{sat}}$ by replacing the subprogram p_{g_1} with p_{g_2}, we still obtain an accepting run of $\mathcal{C}$. But the resulting program does not satisfy the specification: It will run into an infinite loop without any input statements, because at least one of the checks will fail. This is a contradiction to the premise, so $\mathcal{C}$ must indeed have at least $2^{2^{m-1}}$ states. □

8

Bounds for the Number of Program Variables

8.1 Linear Temporal Logic as a Specification Language

The programs constructed by the synthesis procedure presented in Chapter 6 are restricted to a given set of Boolean variables Z. However, there may be no program over Z satisfying the specification if Z is chosen too small. In this chapter, we investigate how many Boolean variables are needed to satisfy a given specification. We only consider specifications that are defined in *linear temporal logic (LTL)*. LTL formulas are constructed according to the following grammar:

linear temporal logic (LTL)

$$\varphi \quad ::= \quad \texttt{true} \;\mid\; \text{ap} \;\mid\; \neg\varphi \;\mid\; \varphi_1 \vee \varphi_2 \;\mid\; X\varphi \;\mid\; \varphi_1 \,U\, \varphi_2,$$

where *ap* stands for an atomic proposition. In our case, an atomic proposition is either an input symbol $\bar{a} \in \Sigma_\mathrm{I}$ or of the form "out=$\bar{b}$" for some output symbol $\bar{b} \in \Sigma_\mathrm{O}$. For convenience, we also allow the Boolean connectives $\wedge$, $\rightarrow$ and $\leftrightarrow$ (defined in the usual way), and we let $F\varphi = \texttt{true}\, U \varphi$ and $G\varphi = \neg F \neg \varphi$. We interpret LTL formulas over infinite input/output sequences $\beta = (\bar{a}_1, \bar{b}_1)(\bar{a}_2, \bar{b}_2)(\bar{a}_3, \bar{b}_3) \ldots \in (\Sigma_\mathrm{I} \times \Sigma_\mathrm{O})^\omega$ using the following satisfaction relation:

$$\begin{array}{lll}
\beta \models \texttt{true} & \text{for all } \beta, & \\
\beta \models \bar{a} & \text{if } \bar{a} = \bar{a}_1 & (\text{for } \bar{a} \in \Sigma_\mathrm{I}), \\
\beta \models \text{out=}\bar{b} & \text{if } \bar{b} = \bar{b}_1 & (\text{for } \bar{b} \in \Sigma_\mathrm{O}), \\
\beta \models \neg\varphi & \text{if } \beta \not\models \varphi, & \\
\beta \models \varphi_1 \vee \varphi_2 & \text{if } \beta \models \varphi_1 \text{ or } \beta \models \varphi_2, & \\
\beta \models X\varphi & \text{if } (\bar{a}_2, \bar{b}_2)(\bar{a}_3, \bar{b}_3)(\bar{a}_4, \bar{b}_4) \ldots \models \varphi, & \\
\beta \models \varphi_1 \,U\, \varphi_2 & \text{if for some } i \geq 1\text{: } (\bar{a}_i, \bar{b}_i)(\bar{a}_{i+1}, \bar{b}_{i+1}) \ldots \models \varphi_2 & \\
 & \text{and for all } 1 \leq j < i\text{: } (\bar{a}_j, \bar{b}_j)(\bar{a}_{j+1}, \bar{b}_{j+1}) \ldots \models \varphi_1. &
\end{array}$$

We use LTL formulas to define specifications. Formally, an LTL formula φ represents the specification $\{\beta \in (\Sigma_\mathrm{I} \times \Sigma_\mathrm{O})^\omega \mid \beta \models \varphi\}$.

8.2 Results and Proof Overview

It is well known that the size of the smallest Mealy automaton realizing a given LTL specification is at most doubly exponential in the size of the LTL formula [PR89]. A Mealy automaton is a DFA without final states whose transitions are additionally labeled with an output symbol. Given such a Mealy automaton, we can easily construct an equivalent structured program in the following way: The current state of the automaton is encoded using Boolean variables. The program performs an infinite loop that contains a case distinction (using nested `if/then/else` constructs) encoding the transition function of the automaton. Note that the number of variables required to represent the current state of the automaton is logarithmic in the number of states and hence at most exponential in the size of the LTL formula. We thus obtain the following result.

Theorem 8.2.1 (Consequence of [PR89]). For every realizable LTL specification φ, there exists a structured program p, over some set of Boolean variables Z, that satisfies φ, where $|Z|$ is at most exponential in the size of the LTL formula.

We will show a matching lower bound. Note that such a lower bound is not necessarily implied by the doubly exponential lower bound for the size of a Mealy automaton realizing an LTL specification established in [Ros92]. This is because the current "state" of a program does not only depend on the valuation of its variables but also on the current location within the program (also known as the program counter).

Before we give an outline of our approach, let us fix some terminology.

graph — An (undirected) *graph* $G = (V(G), E(G))$ consists of a set $V(G)$ of nodes and a set $E(G) \subseteq \{\{x, x'\} \mid x, x' \in V(G)\}$ of undirected edges. A *walk* π

walk — in a graph G is a sequence $x_1 \dots x_n$ of nodes such that $\{x_i, x_{i+1}\} \in E(G)$ for all $i \in \{1, \dots, n-1\}$. We are particularly interested in the transition graphs of structured programs.

transition graph G_p

Definition 8.2.2. Let p be a program with the associated IOI machine $\mathcal{M}(p) = (Q, \Delta, mem, Q_{\text{entry}}, Q_{\text{exit}})$. The *transition graph* of the program p is the graph G_p with

- $V(G_p) = Q$ and
- $E(G_p) = \big\{\{q, q'\} \mid (q, c, q') \in \Delta \text{ for some } c \in \widehat{\Sigma}\big\}$.

We call Q_{entry} and Q_{exit} the sets of entry states and exit states of G_p.

Our proof for a lower bound for the number of variables is based on the notion of tree-width (see, for example, [ST93]), which is a measure for the similarity of a graph to a tree. We will construct LTL specifications that can only be satisfied by programs whose transition graphs have "large" tree-width. (More accurately, the required tree-width grows rapidly with the size of the LTL formula.) Since the structure of the control flow of a structured program is very "tree-like", the transition graph of a structured program can only have large tree-width if the program uses a large number of variables, which yields the desired lower bound. More specifically, we proceed in three steps:

1. We construct a family of LTL specifications $(\varphi_n)_{n\in\mathbb{N}}$ whose length is quadratic in n, such that the transition graph of any program satisfying φ_n (stutter-)simulates the hypercube of dimension $d = 2^n$.

2. We show that if a (labeled) graph G_1 is (stutter-)simulated by another graph G_2, then the tree-width of G_2 is at least as large as the tree-width of G_1.

3. We show that the tree-width of the transition graph of any program with m Boolean variables is at most $3 \cdot 2^m - 1$.

Formal definitions of tree-width, the hypercube, and stutter simulation will be given later.

Since the tree-width of the d-dimensional hypercube is exponential in d (see [SK06]), we can deduce that the tree-width of the transition graph of any program satisfying φ_n is at least doubly exponential in n. Therefore, any program satisfying φ_n must use an exponential number of Boolean variables.

8.3 Preliminaries: Labeled Graphs and Stutter Simulation

First, we need to define what it means for a graph to stutter-simulate another graph. More precisely, we consider *labeled graphs* (G, ℓ), consisting of a graph G and a labeling function $\ell\colon V(G) \to L$ for some set of labels L. We extend the labeling function to sets of nodes $X \subseteq V(G)$ by defining $\ell(X) = \{\ell(x) \mid x \in X\}$ and to walks $\pi = x_1 \dots x_n$ by letting $\ell(\pi) = \{\ell(x_i) \mid i \in \{1, \dots, n\}\}$.

labeled graph

Definition 8.3.1. A *stutter simulation* between two labeled graphs (G_1, ℓ_1) and (G_2, ℓ_2) is a relation $\mathfrak{S} \subseteq V(G_1) \times V(G_2)$ such that

1. for all $x \in V(G_1)$, there is some $y \in V(G_2)$ with $(x, y) \in \mathfrak{S}$, and

stutter simulation

2. for all $(x, y) \in \mathfrak{S}$, we have
   - $\ell_1(x) = \ell_2(y)$ and
   - if $\{x, x'\} \in E(G_1)$ for some $x' \in V(G_1)$, then there exists a walk $yy_1y_2 \dots y_n y'$ in G_2 with $(x', y') \in \mathfrak{S}$ and $(x, y_i) \in \mathfrak{S}$ for all $i \in \{1, \dots, n\}$.

We say that a node x is stutter-simulated by a node y if $(x, y) \in \mathfrak{S}$.

Note that the first condition in our definition of a stutter simulation requires that each node of the first graph is stutter-simulated by some node of the second graph. If there exists a stutter simulation $\mathfrak{S} \subseteq V(G_1) \times V(G_2)$ between two labeled graphs (G_1, ℓ_1) and (G_2, ℓ_2), we say that (G_1, ℓ_1) *is stutter-simulated by* (G_2, ℓ_2).

The following proposition will be useful later.

Proposition 8.3.2. Let $\mathfrak{S} \subseteq V(G_1) \times V(G_2)$ be a stutter simulation between two labeled graphs (G_1, ℓ_1) and (G_2, ℓ_2). Let $x \in V(G_1)$ and $y \in V(G_2)$ such that $(x, y) \in \mathfrak{S}$.
For every node $x' \in V(G_1)$ that is reachable from x via some walk π_1, there exists a node $y' \in V(G_2)$ with $(x', y') \in \mathfrak{S}$ such that y' is reachable from y via some walk π_2 with $\ell_2(\pi_2) = \ell_1(\pi_1)$.

Proof. We use induction on the length of the walk π_1 from x to x'.

Induction base. If π_1 has length 1, and hence $x = x'$, then y is a correct choice for y'.

Induction step. If the length of the walk π_1 is greater than 1, then π_1 is of the form $\pi_1' x'$, where π_1' is a walk from x to some node $x'' \in V(G_1)$ with $\{x'', x'\} \in E(G_1)$. By the induction hypothesis, there exists a node $y'' \in V(G_2)$ with $(x'', y'') \in \mathfrak{S}$ such that y'' is reachable from y via some walk π_2' with $\ell_2(\pi_2') = \ell_1(\pi_1')$. Because $(x'', y'') \in \mathfrak{S}$, it follows from $\{x'', x'\} \in E(G_1)$ that there is a node $y' \in V(G_2)$ with $(x', y') \in \mathfrak{S}$ that is reachable from y'' via some walk $y'' y_1 y_2 \dots y_n y'$ with $(x'', y_i) \in \mathfrak{S}$ for all $i \in \{1, \dots, n\}$. Hence, y' is reachable from y via the walk $\pi_2 = \pi_2' y_1 y_2 \dots y_n y'$. Note that $\ell_2(y_1 y_2 \dots y_n) = \{\ell_1(x'')\}$ and $\ell_2(y') = \ell_1(x')$, so

$$\begin{aligned} \ell_2(\pi_2) &= \ell_2(\pi_2') \cup \ell_2(y_1 y_2 \dots y_n) \cup \{\ell_2(y')\} \\ &= \ell_1(\pi_1') \cup \{\ell_1(x'')\} \cup \{\ell_1(x')\} \\ &= \ell_1(\pi_1). \end{aligned}$$

□

We will be particularly interested in stutter simulations between a graph G_1 whose nodes are labeled by the identity function *id* on $V(G_1)$ and a graph G_2 whose nodes are labeled with nodes of G_1:

id

Notation 8.3.3. We write $G_1 \preccurlyeq G_2$ if there exists a labeling function $\ell\colon V(G_2) \to V(G_1)$ such that (G_1, id) is stutter-simulated by (G_2, ℓ).

$G_1 \preccurlyeq G_2$

8.4 Step 1: Forcing the Program to Simulate a Hypercube

8.4.1 A Family $(\varphi_n)_{n\in\mathbb{N}}$ of Specifications

We consider input sequences over the alphabet $\Sigma_I = \{0, 1, \boxplus, \boxminus, ?\}$. These symbols can be encoded using three bits, but we omit this encoding for the sake of readability. We interpret an input sequence over Σ_I as a series of instructions to manage a set of words of length n, that is, a set $S \subseteq \{0,1\}^n$. An input sequence infix (that is, a finite segment of an input sequence) of the form $\boxplus w$ with $w \in \{0,1\}^n$ is interpreted as an instruction to add the word w to the set. Similarly, an infix of the form $\boxminus w$ with $w \in \{0,1\}^n$ is viewed as an instruction to remove the word w from the set. Finally, an infix of the form $?w$ with $w \in \{0,1\}^n$ is a query that asks for the membership of w in the set. We call an input sequence *queryless* if it does not contain the symbol $?$.

queryless input sequence

Each finite prefix v of such an input sequence uniquely determines a set, denoted by S_v, that is obtained from the empty set by adding and removing words according to the instructions in v. More precisely, a word w is in S_v if and only if there is an occurrence of the instruction $\boxplus w$ in v such that the suffix of v after that occurrence does not contain the instruction $\boxminus w$. We will call such sets S_v *word sets*.

word set S_v

Now we construct an LTL specification over the input alphabet Σ_I as defined above and the output alphabet $\Sigma_O = \{0,1\}$ that requires that the output is 1 infinitely often if and only if both of the following conditions are met:

1. The input sequence contains exactly one query (one occurrence of the symbol $?$).
2. The queried word w is in the word set S_v that is determined by the prefix v of the input sequence up to the query.

This can be expressed by the LTL formula

$$\varphi_n = (\textsc{AtMostOneQuery} \wedge \textsc{PositiveQuery}) \leftrightarrow GF(\text{out} = 1)$$

with the following auxiliary formulas (where X^i is used as an abbreviation for $\underbrace{XX\ldots X}_{i \text{ times}}$):

$$\textsc{AtMostOneQuery} = G\,(\boxed{?} \rightarrow XG(\neg\boxed{?}))$$

$$\textsc{PositiveQuery} = F\Big(\overbrace{\boxplus \wedge \textsc{QueriedWord}}^{\text{queried word added}} \wedge \underbrace{\neg F(\boxminus \wedge \textsc{QueriedWord})}_{\text{not removed later}} \Big)$$

$$\textsc{QueriedWord} = F\,\boxed{?} \wedge \bigwedge_{1 \le i \le n} \Big(\big(X^i 0 \wedge G(\boxed{?} \rightarrow X^i 0)\big) \vee \big(X^i 1 \wedge G(\boxed{?} \rightarrow X^i 1)\big) \Big)$$

Note that the length of φ_n is only quadratic in n. Furthermore, φ_n is clearly realizable by a program that keeps track of the current word set using $\mathcal{O}(2^n)$ Boolean variables: The program outputs 0 until a query occurs and then starts outputting 1 if and only if the query was successful. If a second query is encountered, the program starts outputting 0 again.

8.4.2 The Specification φ_n and the Hypercube

If we view the possible word sets $S \subseteq \{0,1\}^n$ as the nodes of a graph and connect two sets S, S' by an edge if S' can be obtained from S by following a $\boxplus$- or $\boxminus$-instruction (that is, by either adding or removing a single word), then we obtain the hypercube of dimension 2^n:

hypercube H_d

Definition 8.4.1. The *hypercube of dimension d*, denoted by H_d, is constructed in the following way: Let D be a set containing exactly d elements. The nodes of the hypercube are the subsets of D, that is, $V(H_d) = 2^D$. Two nodes $S, S' \subseteq D$ are connected by an (undirected) edge if and only if S' can be obtained from S by either adding or removing a single element.

Intuitively, a program satisfying the specification φ_n must keep track of the current word set, determined by the input sequence that has been read so far. This means that at each state of the transition graph of the program, the current word set is uniquely determined and hence

we can label each state with such a set. The program must be able to emulate the instructions provided by the input sequence, which amount to movements along the edges of the above-mentioned hypercube of dimension 2^n. This leads us to the following lemma.

LEMMA 8.4.2. Let p be a program that satisfies the specification φ_n. Then $H_{2^n} \preccurlyeq G_p$.

To prepare for a formal proof of this lemma, we first prove the following proposition.

PROPOSITION 8.4.3. Let p be a program that satisfies φ_n. Let q be a state of the associated IOI machine $\mathcal{M}(p)$ such that q is reachable from the initial state q_0 of $\mathcal{M}(p)$ via some queryless finite input sequence v. Then for all (finite) input sequences v' that lead from q_0 to q, we have $S_{v'} = S_v$.

Proof. Assume that the proposition does not hold. Then there are two input sequences v, v' leading to q with $S_v \neq S_{v'}$.

Let w be an arbitrary word that is in S_v or in $S_{v'}$ but not in both sets. Now, if we append to v and to v' a query $?w$ for w, followed by an arbitrary queryless infinite input sequence α, then for one of the resulting infinite input sequences $v\,?\,w\alpha$ and $v'\,?\,w\alpha$, the program cannot produce a correct output sequence:

For one of the sequences, the query must yield a positive result and thus the program must output 1 infinitely often. For the other sequence, the query must yield a negative result, so the program must output 1 only finitely often. But in both cases the (IOI machine of the) program is in state q immediately before the query, so from that point on, the output will be the same for both input sequences and hence the specification will be violated for one of them. □

We can now prove Lemma 8.4.2:

Proof of Lemma 8.4.2. We have to show that there exists a labeling $\ell\colon V(G_p) \to V(H_{2^n})$ such that (H_{2^n}, id) is stutter-simulated by (G_p, ℓ). (W.l.o.g., we assume that H_{2^n} is constructed from the set $\{0,1\}^n$, so $V(H_{2^n})$ is the powerset of $\{0,1\}^n$.) Recall that the nodes of G_p are the states of the IOI machine $\mathcal{M}(p)$ and that two nodes are connected by an edge if and only if they are connected by a transition of $\mathcal{M}(p)$.

We choose the following labeling ℓ for G_p: If a state q is reachable in $\mathcal{M}(p)$ from the initial state q_0 via some queryless input sequence v,

then its label is $\ell(q) = S_v$. Due to Proposition 8.4.3, this label is uniquely determined. Otherwise, we choose an arbitrary word set $S \subseteq \{0,1\}^n$ and set $\ell(q) = S$.

We now define a stutter simulation $\mathfrak{S}$ between the labeled graphs (H_{2^n}, id) and (G_p, ℓ):

$$\mathfrak{S} = \big\{ (S, q) \in V(H_{2^n}) \times V(G_p) \mid q \text{ is reachable in } \mathcal{M}(p) \text{ from } q_0 \text{ via some queryless input sequence } v \text{ with } S_v = S \big\}$$

We will now show that $\mathfrak{S}$ is indeed a stutter simulation. First, note that for each word set $S \subseteq \{0,1\}^n$, we can construct a queryless input sequence v with $S_v = S$, and since the program p is reactive, some state q is reachable in $\mathcal{M}(p)$ via this input sequence. Thus, for each node S of H_{2^n}, there exists a node q of G_p with $(S, q) \in \mathfrak{S}$, so condition 1 in the definition of a stutter simulation (Definition 8.3.1) is satisfied.

Regarding condition 2, consider now any $(S, q) \in \mathfrak{S}$. By definition of $\mathfrak{S}$, the state q is reachable in $\mathcal{M}(p)$ from the initial state via some queryless input sequence v with $S_v = S$, so the labels $id(S)$ and $\ell(q) = S$ are equal, satisfying the first part of the condition. To verify the second part of the condition, consider a set S' with $\{S, S'\} \in E(H_{2^n})$. Note that S' is obtained from S by adding or removing a single word $w \in \{0,1\}^n$. W.l.o.g., we assume that $S' = S \cup \{w\}$. Since p is reactive, some state q' must be reachable in $\mathcal{M}(p)$ (and hence also in G_p) from q via the input sequence $\boxplus w$. This state q' is thus reachable from the initial state via the input sequence $v \boxplus w$, which yields the word set S', so we have $(S', q') \in \mathfrak{S}$. It remains to be shown that $(S, q'') \in \mathfrak{S}$ for all states q'' that are visited between q and q'. Each such state q'' is reachable in $\mathcal{M}(p)$ from the initial state via an input sequence of the form vv', where v' is a strict prefix of $\boxplus w$. Note that $S_{vv'} = S_v = S$, so we have $(S, q'') \in \mathfrak{S}$. Thus, (H_{2^n}, id) is indeed stutter-simulated by (G_p, ℓ). □

8.5 Interlude: Tree-Width and the Cops and Robber Game

To prepare for the following steps, we have to define the notion of tree-width. To that end, we recapitulate the cops and robber game introduced in [ST93], which provides a characterization of the tree-width of a graph.

cops and robber game

Definition 8.5.1. In the *cops and robber game*, k cops in helicopters try to catch a robber on a graph G. The cops can always see the robber, and vice versa. First, the cops choose a set $C_0 \in [V(G)]^{\leq k}$ of starting nodes.

(We write $[V(G)]^{\leq k}$ to denote the set of subsets of $V(G)$ that contain at most k nodes.) Then the robber chooses his starting node $x_0 \in V(G)$. The cops and the robber continue to move in alternation: At the beginning of the $(i+1)$th step of a play, we have a position (C_i, x_i), meaning that the cops are located at the nodes in the set C_i and the robber is on node x_i. Now the players make their moves: $[V(G)]^{\leq k}$

1. The cops can move around freely in their helicopters and choose a set $C_{i+1} \in [V(G)]^{\leq k}$.
2. The robber runs from x_i to some node x_{i+1} along a walk in G.

A cop that stays at his previous node, i.e., a cop on a node in $C_i \cap C_{i+1}$, lands at that node. (Nevertheless, such a cop may still fly to a different node in the next step.) We say that the nodes in $C_i \cap C_{i+1}$ are *controlled* by a cop.
A play of the game is a sequence of positions of the following form:

$$(C_0, x_0)\ (C_1, x_1)\ (C_2, x_2)\ \ldots$$

The cops win if at least one of the following conditions is met:

- At some point, the robber runs to a node $x_{i+1} \in C_{i+1}$, which allows the cops to capture him in their next move.
- On his way from a node x_i to x_{i+1}, the robber crosses a node $x \in C_i \cap C_{i+1}$ controlled by a cop.

Otherwise the robber can evade the cops forever and wins.

We can use the cops and robber game to define the tree-width of a graph. While tree-width is usually defined using so-called tree decompositions of graphs, the following definition is equivalent, as was shown in [ST93].

Definition 8.5.2. Let G be a graph and let k be the least number of cops such that the cops have a winning strategy in the cops and robber game on G. The *tree-width* of G is defined as $\mathrm{tw}(G) = k - 1$. tree-width $\mathrm{tw}(G)$

8.6 Step 2: Stutter Simulation and Tree-Width

If two graphs are related by a stutter simulation, then their tree-widths are related accordingly.

Lemma 8.6.1. Let G_1 and G_2 be graphs.
If $G_1 \preccurlyeq G_2$ then $\mathrm{tw}(G_1) \leq \mathrm{tw}(G_2)$.

Proof. Assume that $G_1 \preccurlyeq G_2$. We show that if the robber has a winning strategy against k cops on the graph G_1 (in the game presented in Definition 8.5.1), then the robber also has a winning strategy against k cops on the graph G_2. It follows that $\mathrm{tw}(G_1) \leq \mathrm{tw}(G_2)$.

Since $G_1 \preccurlyeq G_2$, there exists a labeling function $\ell \colon V(G_2) \to V(G_1)$ such that (G_1, id) is stutter-simulated by (G_2, ℓ), witnessed by some stutter-simulation $\mathfrak{S}$. W.l.o.g., we assume that all nodes $y \in V(G_2)$ simulate some node $x \in V(G_1)$, i.e., $(x, y) \in \mathfrak{S}$. (Otherwise, we can simply remove all other nodes from G_2, since this will only limit the options of the robber.) Note that this implies $(\ell(y), y) \in \mathfrak{S}$ for all $y \in V(G_2)$, since G_1 is labeled by the identity function.

Assume that the robber has a winning strategy on G_1. We will now present a winning strategy for the robber on G_2. Intuitively, the fact that G_1 is simulated by G_2 allows the robber to "simulate" his G_1-moves in G_2 and thus evade the cops. More precisely, the G_2-strategy for the robber is defined as follows.

Choosing the starting node. Let $C_0 \in [V(G_2)]^{\leq k}$ be the set of starting nodes chosen by the cops in G_2. Now let $x_0 \in V(G_1)$ be the starting node chosen by the robber in G_1 (according to his winning strategy) if the cops start on the nodes $\ell(C_0)$. In G_2, we let the robber choose an arbitrary starting node $y_0 \in V(G_2)$ such that $(x_0, y_0) \in \mathfrak{S}$, i.e., $\ell(y_0) = x_0$. Such a node always exists according to the definition of a stutter simulation. Since $\ell(y_0) = x_0 \notin \ell(C_0)$, we have $y_0 \notin C_0$, so this is a safe move for the robber in G_2. The play on G_2 now continues from the position (C_0, y_0). Note that by definition of y_0, the corresponding position $\big(\ell(C_0), \ell(y_0)\big)$ in the game on G_1 is consistent with the winning strategy for the robber on G_1.

Evading the cops forever. Now consider a play $(C_0, y_0) \dots (C_i, y_i)$ on G_2, such that $\big(\ell(C_0), \ell(y_0)\big) \dots \big(\ell(C_i), \ell(y_i)\big)$ is a play on G_1 that is consistent with the winning strategy for the robber on G_1. Furthermore, let $C_{i+1} \in [V(G_2)]^{\leq k}$ be the next move of the cops on G_2. We have to define an appropriate response by the robber.

To that end, suppose that the next move of the cops on G_1 is $\ell(C_{i+1})$. Let $x_{i+1} \in V(G_1)$ be the node that is chosen by the robber in response to that, according to his winning strategy on G_1. Furthermore, let π_1 be the walk used by the robber to get from $\ell(y_i)$ to x_{i+1}. We determine the next

node for the robber in G_2 by simulating that move: Since $(\ell(y_i), y_i) \in \mathfrak{S}$, Proposition 8.3.2 is applicable. Hence, there exists a node $y_{i+1} \in V(G_2)$ with $(x_{i+1}, y_{i+1}) \in \mathfrak{S}$, i.e., with $\ell(y_{i+1}) = x_{i+1}$, such that y_{i+1} is reachable from y_i via some walk π_2 with $\ell(\pi_2) = id(\pi_1)$. We let the robber choose an arbitrary y_{i+1} with this property and move to that node. This is illustrated in Figure 6.

Let us now show that there are no cops at the node y_{i+1} and that the robber does not have to cross a node controlled by a cop to get there. The node x_{i+1} was chosen according to the winning strategy on G_1, so $x_{i+1} \notin \ell(C_{i+1})$. Since $\ell(y_{i+1}) = x_{i+1}$, we thus have $\ell(y_{i+1}) \notin \ell(C_{i+1})$, which implies $y_{i+1} \notin C_{i+1}$.

Moreover, the walk π_1 in G_1 does not cross the set $\ell(C_i) \cap \ell(C_{i+1})$ of nodes controlled by cops. Since $\ell(\pi_2) = id(\pi_1)$, we obtain $\ell(\pi_2) \cap \ell(C_i) \cap \ell(C_{i+1}) = \emptyset$, which implies that π_2 does not contain any node in $C_i \cap C_{i+1}$.

As a result of the move of the robber to y_{i+1}, the play on G_2 is extended to $(C_0, y_0) \dots (C_{i+1}, y_{i+1})$. Note that the corresponding sequence of positions $\big(\ell(C_0), \ell(y_0)\big) \dots \big(\ell(C_{i+1}), \ell(y_{i+1})\big)$ on G_1 is again consistent with the winning strategy for the robber on G_1, so we obtain a well-defined strategy on G_2. Following this strategy, the robber will evade the cops forever. □

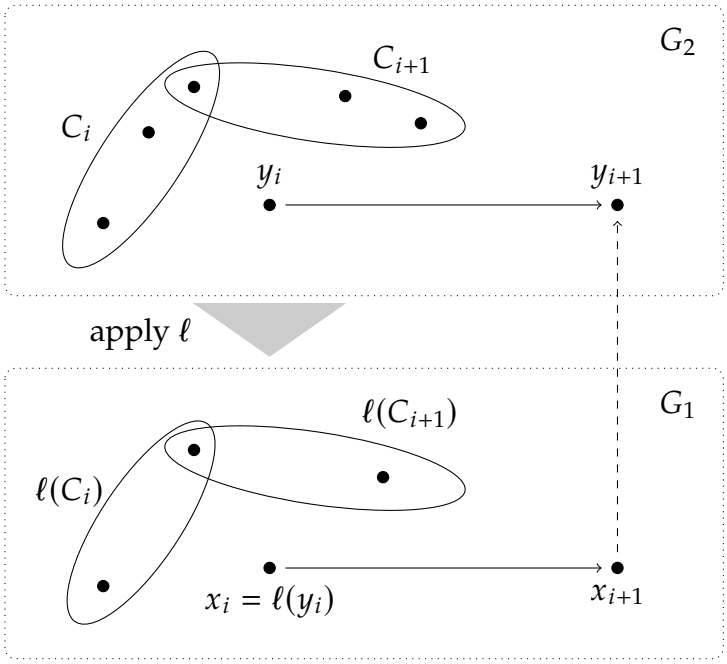


FIGURE 6: Construction of the strategy for the robber on G_2 from his winning strategy on G_1.

With Lemma 8.4.2, we obtain the following corollary.

Corollary 8.6.2. Let p be a program that satisfies the specification φ_n (as defined in Section 8.4.1). Then $\mathrm{tw}(G_p) \geq \mathrm{tw}(H_{2^n})$.

We can now apply the following lower bound for the tree-width of the hypercube.

Theorem 8.6.3 ([SK06]). Let H_d be the hypercube of dimension d. Then we have

$$\mathrm{tw}(H_d) \geq c \cdot \frac{2^d}{\sqrt{d}} \quad \text{for some constant } c > 0.$$

Thus, with Corollary 8.6.2, we have the following lower bound for the tree-width of the transition graph of any program satisfying φ_n.

Corollary 8.6.4. Let p be a program that satisfies the specification φ_n (as defined in Section 8.4.1). Then

$$\mathrm{tw}(G_p) \geq c \cdot \frac{2^{2^n}}{\sqrt{2^n}} \quad \text{for some constant } c > 0.$$

8.7 Step 3: Tree-Width of Program Transition Graphs

In the last section, we have established that the transition graph of any program that satisfies the specification φ_n (as defined in Section 8.4.1) has large tree-width (doubly exponential in n). We will now show that a structured program can only have a transition graph with large tree-width if it uses a large number of variables. This is closely related to the fact that the control-flow graphs of structured programs are so-called series-parallel graphs, which have bounded tree-width (see [Bod98; Tho98]).

Theorem 8.7.1. Let p be a program over a set of m Boolean variables. The tree-width of the transition graph of p is at most $3 \cdot 2^m - 1$, that is, $\mathrm{tw}(G_p) \leq 3 \cdot 2^m - 1$.

Proof. We have to show that $3 \cdot 2^m$ cops have a winning strategy against the robber on the transition graph G_p (in the game specified in Definition 8.5.1).

We show the following stronger statement by induction on the structure of the program p: There exists a winning strategy for $3 \cdot 2^m$ cops against the robber on G_p such that during any play that is consistent with that strategy, the robber is always *cut off* from the entry and exit states of G_p. We say that the robber is cut off from a node x if every walk from the robber's current node to x contains a node that is controlled by a cop.

Induction base. If p is an atomic program, then G_p has exactly $2 \cdot 2^m$ nodes (2^m entry states and 2^m exit states). Since there are $3 \cdot 2^m$ cops, a cop can be sent to each node at the beginning of the game. Now the robber has to go to a node with a cop, so he is captured in the next step. This also implies that he is cut off from the entry and exit states.

Induction step. We have to consider programs that are constructed by concatenation, conditional execution, or repetition.

$p =$ "$p_1\,;p_2$": In this case, the transition graph of p can be obtained from the transition graphs of p_1 and p_2 by merging the exit states of G_{p_1} with the corresponding entry states of G_{p_2}. Note that there can only be 2^m entry states in G_{p_2} and hence only 2^m merged states in G_p.

Thus, the cops win the game with the following strategy: In the beginning, they go to the entry and exit states of G_p and to the states that have resulted from merging the exit states of G_{p_1} with the entry states of G_{p_2}. (Note that this requires $3 \cdot 2^m$ cops.) If the robber runs to a node from G_{p_1}, the cops play according to their winning strategy on G_{p_1}. Otherwise, if the robber chooses a node from G_{p_2}, the cops play according to their winning strategy on G_{p_2}.

By the induction hypothesis, in both cases, the robber cannot escape from the p_1-part or p_2-part of the graph, respectively. Thus, the cops win, and the robber is always cut off from the entry and exit states of G_p.

$p =$ "`if` e `then` p_1 `else` p_2": The transition graph of p is constructed from the transition graphs of p_1 and p_2 by merging the corresponding exit states and specifying entry states according to the branching condition.

The cops therefore have the following winning strategy: In the beginning, they go to the exit states of G_p and those states

that were entry states in G_{p_1} or G_{p_2}. (This includes all entry states of G_p and requires $3 \cdot 2^m$ cops.) Then they play their winning strategy on G_{p_1} or G_{p_2}, respectively, depending on which part of G_p the robber chooses in his first move.

The induction hypothesis implies that the cops win and that the robber is cut off from the entry and exit states of G_p.

$p =$ "`while` e `do` p_1": The transition graph of p is obtained by merging the exit states of G_{p_1} with the corresponding entry states. To win the game, the cops can simply play their winning strategy on G_{p_1}: From the induction hypothesis, it follows that the robber is cut off from the entry and exit states of G_p. Hence, the robber can only make moves that are available within G_{p_1}, so the cops win the game on G_p. □

8.8 Putting It All Together

We can now prove the previously announced lower bound for the number of variables that are required in a structured program to satisfy a given LTL specification.

Theorem 8.8.1. There exists a family $(\varphi_n)_{n \in \mathbb{N}}$ of realizable LTL specifications whose size is quadratic in n, such that the least number of Boolean variables used by any program satisfying φ_n is $\Omega(2^n)$.

Proof. Let $(\varphi_n)_{n \in \mathbb{N}}$ be the family of specifications defined in Section 8.4.1. Let p be a program over m Boolean variables that satisfies φ_n. Theorem 8.7.1 and Corollary 8.6.4 yield the following inequations:

$$3 \cdot 2^m - 1 \geq \mathrm{tw}(G_p) \geq c \cdot \frac{2^{2^n}}{\sqrt{2^n}} \quad \text{for some constant } c > 0.$$

This implies $m \geq 2^n - \mathrm{ld}\left(\sqrt{2^n}\right) + \mathrm{ld}\left(\frac{c}{3}\right)$. Hence the required number of variables is $\Omega(2^n)$. □

9

Conclusion on Structured Reactive Programs

In Part I of this thesis, we provided several contributions to the study of structured reactive programs, which were introduced in [Mad11] as a succinct formalism to represent reactive systems.

We introduced a new algorithm to synthesize a structured reactive program from a given ω-regular specification using the elementary concept of deterministic bottom-up tree automata. Our construction can be used to synthesize programs with strict input/output alternation, and also programs with some delay (within given bounds) between the input and output sequences. The constructed tree automaton recognizes the set of programs over a given set of Boolean variables that satisfy the given specification and comply with the given delay bounds. The specification is represented by a nondeterministic Büchi automaton recognizing its complement, and the size of the tree automaton is exponential in the size of that Büchi automaton, doubly exponential in the number of Boolean variables that are available to the programs, and doubly exponential in the delay bounds. As in Madhusudan's algorithm, an emptiness test of the tree automaton yields a minimal program satisfying the specification.

LTL specifications, or their complement, can be translated to nondeterministic Büchi automata in exponential time (see, for example, [GPV+96]). Thus, the time required by the synthesis algorithm in that case is doubly exponential in the formula size – for a fixed number of variables (and fixed delay bounds). This matches the doubly exponential lower bound for synthesizing Mealy automata from LTL specifications (see [Ros92]). However, the size of the tree automata that are constructed to synthesize structured reactive programs also grows doubly exponentially with the number of program variables.

With that in mind, we provided two results that show the limitations of the synthesis approaches presented in this thesis and by Madhusudan: Firstly, every deterministic or nondeterministic tree automaton that recognizes the set of "specification-compliant" programs over m Boolean variables must have at least $2^{2^{m-1}}$ states. Secondly, certain LTL specifications necessitate an exponential number of program variables,

which matches the corresponding upper bound. By choosing the number of variables accordingly, we obtain a synthesis algorithm that is guaranteed to find a correct program if the specification can be satisfied at all. The time complexity of the algorithm is then triply exponential in the size of the given LTL formula. We may view this additional exponential blowup as a price that we pay for obtaining a *minimal* program. We leave it as an open question whether this price is inevitable, even with other algorithms that might avoid a representation of all correct programs by a tree automaton.

Besides the number of variables and the length of the program code, one may also study the number of computation steps performed by a program between reading an input and producing an output. Potential tradeoffs between these measures for given classes of specifications are still to be investigated.

Part II

Synthesis of Transducers over Infinite Alphabets

10

Outline: Games over Infinite Alphabets

The subject of Part II of this thesis are Gale-Stewart games over infinite alphabets of the form Σ^*, where Σ is a finite set. In each turn of such a game, the current player chooses a finite word from the set Σ^*. The resulting play of the game is an element of $(\Sigma^*)^\omega$, that is, an infinite sequence of finite words – which is not to be confused with the concatenation of these finite words.

This is illustrated in Figure 7, which shows a play over the alphabet $\{a, b\}^*$. The finite words chosen by the players are written vertically, from bottom to top, and ε signifies the empty word. For instance, the first word chosen by Player I is abb.

	b		a				
	b	a	b			a	
Chosen words:	a	b	a	b	ε	b	$\cdots$
Player:	I	II	I	II	I	II	$\cdots$

Figure 7: A play in a Gale-Stewart game over the alphabet $\{a, b\}^*$.

Our goal is to solve a generalization of Church's Synthesis Problem for infinite alphabets Σ^* as described above:

> Given a winning condition $L \subseteq (\Sigma^*)^\omega$ defined using a suitable type of automaton, determine which player has a winning strategy in the corresponding Gale-Stewart game, and construct such a strategy in the form of a transducer of a suitable type.

In order to provide an algorithmic solution, we first have to define in a precise way the automata and transducers that we shall use to represent winning conditions and strategies. We proceed in the following way:

- Chapter 11 provides some background on monadic second-order logic (MSO logic), which is used extensively in this part of the thesis. The reader familiar with MSO logic may first skip this chapter and return to it as needed.

- In Chapter 12, we define $\mathbb{N}$-memory automata on finite words over alphabets of the form Σ^* (for a finite set Σ) and investigate their expressive power, closure properties, and decision problems.
- Next, in Chapter 13, we extend our study of $\mathbb{N}$-memory automata to infinite words and establish connections to the Borel hierarchy of the Baire space.
- As a suitable formalism to represent strategies, we introduce $\mathbb{N}$-memory transducers, in Chapter 14.
- In Chapter 15, we provide a solution to Church's Synthesis Problem for winning conditions represented by deterministic $\mathbb{N}$-memory parity automata and winning strategies represented by $\mathbb{N}$-memory transducers. This solution relies on the results that are detailed in the following chapter.
- The solution to the synthesis problem given in Chapter 15 involves a translation of the Gale-Stewart game to a parity game (on an infinite graph) that has the following property: The game graph, the positions of each priority, and the positions of each player are MSO-definable in a structure of the form $Q \times \mathcal{T}_\Gamma$, where Q and Γ are finite sets and $\mathcal{T}_\Gamma$ is the infinite Γ-branching tree. In Chapter 16, we show that in such parity games, MSO-definable winning strategies can be constructed.
- Finally, in Chapter 17, we give a brief summary and indicate some directions for future research.

11

Background on MSO Logic

Many of the concepts and results in this part of the thesis are based on *monadic second-order logic (MSO logic)*. In this chapter, we recall the syntax and semantics of this logic and provide some related definitions and results that will be used in the remainder of this thesis.

11.1 MSO Logic over Relational Structures

A *relational structure* $\mathcal{S} = (U, \overline{R})$ consists of a set U, called the universe of $\mathcal{S}$, and a tuple $\overline{R} = (R_1, \ldots, R_n)$ of relations over U.

relational structure

Formulas of MSO logic over such a structure $\mathcal{S}$ are defined inductively, using first-order variables x, y, z, etc., ranging over elements of (the universe of) $\mathcal{S}$, and second-order variables X, Y, Z, etc., ranging over sets of elements. An atomic formula is of one of the following forms:

monadic second-order logic (MSO)

- $x = y$
- $Y(x)$
- $R_i(x_1, \ldots, x_k)$

If φ and ψ are MSO formulas, then so are

- $\neg\varphi$
- $\varphi \vee \psi$
- $\exists x\, \varphi$
- $\exists X\, \varphi$.

For the sake of convenience, we will also use the Boolean connectives $\wedge$, $\rightarrow$, $\leftrightarrow$, the universal quantifier $\forall$ (for first-order variables as well as for second-order variables), and the truth constants `true` and `false`.

We write $\varphi(x_1, \ldots, x_m, X_1, \ldots, X_n)$ to denote a formula φ in which the first-order variables $x_1, \ldots, x_m$ and the second-order variables $X_1, \ldots, X_n$ are not bound by a quantifier. Such variables are called free variables. We often abbreviate a tuple $(x_1, \ldots, x_m)$ as $\overline{x}$ and a tuple $(X_1, \ldots, X_n)$ as $\overline{X}$.

We use the standard semantics to evaluate MSO formulas. Consider a formula $\varphi(\overline{x}, \overline{X})$ over the structure $\mathcal{S} = (U, \overline{R})$. We write

$$\mathcal{S} \models \varphi[u_1, \ldots, u_m, V_1, \ldots, V_n]$$

to express that φ is *satisfied* by $\mathcal{S}$ if the first-order variables $x_1, \ldots, x_m$ are interpreted as $u_1, \ldots, u_m \in U$ and the second-order variables $X_1, \ldots, X_n$

are interpreted as $V_1, \ldots, V_n \subseteq U$. In particular, for the atomic formulas, we have

$$\begin{aligned} \mathcal{S} &\models x = y\,[u_1, u_2] && \text{iff} \quad u_1 = u_2, \\ \mathcal{S} &\models Y(x)\,[u, V] && \text{iff} \quad u \in V, \\ \mathcal{S} &\models R_i(x_1, \ldots, x_k)\,[u_1, \ldots, u_k] && \text{iff} \quad (u_1, \ldots, u_k) \in R_i. \end{aligned}$$

Quantifiers and Boolean operators are interpreted in the usual way. For details see [EFT94].

11.2 MSO-Definability and MSO-Family-Definability

MSO formulas can be used to define relations and functions.

MSO-definable relation or function

Definition 11.2.1. Let $\mathcal{S} = (U, \overline{P})$ be a relational structure over a universe U. A k-ary relation $R \subseteq U^k$ (with $k \in \mathbb{N}_{\geq 1}$) is called *MSO-definable in* $\mathcal{S}$ if there is an MSO formula $\varphi(x_1, \ldots, x_k)$ such that

$$(u_1, \ldots, u_k) \in R \;\Leftrightarrow\; \mathcal{S} \models \varphi[u_1, \ldots, u_k]$$

for all $u_1, \ldots, u_k \in U$.
We also apply this notion to functions by calling a function $f : U \to U$ MSO-definable in $\mathcal{S}$ if its graph, that is, the relation $\{(u, f(u)) \mid u \in U\}$, is MSO-definable in $\mathcal{S}$.

The concept of MSO-definability can be extended to structures in the natural way.

MSO-definable structure

Definition 11.2.2. Let $\mathcal{S} = (U, \overline{P})$ and $\mathcal{S}' = (U, \overline{R})$ be relational structures over U. The structure $\mathcal{S}'$ is called *MSO-definable in* $\mathcal{S}$ if all its relations $R_1, \ldots, R_n$ are MSO-definable in $\mathcal{S}$.

MSO-definability of a structure $\mathcal{S}'$ in a structure $\mathcal{S}$ is a special case of *MSO-interpretability* (for details, see for example [BCL08]). The latter notion allows the restriction of the universe by MSO formulas and allows $\mathcal{S}'$ to be isomorphic rather than strictly equal to the resulting structure. For our purposes, the simpler concept of MSO-definability is more suitable.

MSO-definability allows for the transfer of MSO formulas from one structure to another, in the following sense (analogous to the more general case of MSO-interpretable structures; see [BCL08]).

Proposition 11.2.3. Let $\mathcal{S}$ and $\mathcal{S}'$ be two relational structures over a universe U such that $\mathcal{S}'$ is MSO-definable in $\mathcal{S}$. For every MSO formula $\varphi'(x_1, \ldots, x_k)$ over $\mathcal{S}'$, we can construct an MSO formula $\varphi(x_1, \ldots, x_k)$ over $\mathcal{S}$ such that

$$\mathcal{S} \models \varphi[u_1, \ldots, u_k] \;\Leftrightarrow\; \mathcal{S}' \models \varphi'[u_1, \ldots, u_k]$$

for all $u_1, \ldots, u_k \in U$.

Proof. Let $\mathcal{S}' = (U, R_1, \ldots, R_n)$. Since $\mathcal{S}'$ is MSO-definable in $\mathcal{S}$, each relation R_i can be defined in $\mathcal{S}$ by an MSO formula ψ_{R_i}. Thus, replacing all occurrences of a relation R_i in φ' by the corresponding formula ψ_{R_i} yields the desired formula φ. □

We will often consider relations and structures whose universe is a product of a finite set Q and another (possibly infinite) set U. Such relations and structures may be defined in a structure with universe U by a *family* of formulas as follows.

MSO-family-definable relation or function

Definition 11.2.4. Let $\mathcal{S} = (U, \overline{P})$ be a relational structure over a universe U and let Q be a finite set. A k-ary relation $R \subseteq (Q \times U)^k$ (with $k \in \mathbb{N}_{\geq 1}$) is called *MSO-family-definable in* $\mathcal{S}$ if there is a family of MSO formulas $\big(\varphi_{q_1,\ldots,q_k}(x_1, \ldots, x_k)\big)_{q_1,\ldots,q_k \in Q}$ such that

$$\big((q_1, u_1), \ldots, (q_k, u_k)\big) \in R \;\Leftrightarrow\; \mathcal{S} \models \varphi_{q_1,\ldots,q_k}[u_1, \ldots, u_k]$$

for all $(q_1, u_1), \ldots, (q_k, u_k) \in Q \times U$.
We call a function $f : (Q \times U) \to (Q \times U)$ MSO-family-definable in $\mathcal{S}$ if its graph, that is, the relation $\big\{\big((q, u), f\big((q, u)\big)\big) \mid (q, u) \in Q \times U\big\}$, is MSO-family-definable in $\mathcal{S}$.

MSO-family-definable structure

Definition 11.2.5. Let $\mathcal{S} = (U, \overline{P})$ and $\mathcal{S}' = (Q \times U, \overline{R})$ be relational structures, where Q is a finite set. We say that $\mathcal{S}'$ is *MSO-family-definable in* $\mathcal{S}$ if all its relations $R_1, \ldots, R_n$ are MSO-family-definable in $\mathcal{S}$.

11.3 MSO Logic over Product Structures

In particular, we will consider structures that are obtained as a product of a structure $\mathcal{S}$ and a finite set Q. Such a product, formally defined below, consists of $|Q|$ copies of the original structure. For each copy,

there is a unary relation containing the elements of that copy. The copies are connected by a relation $\mathrm{Eq}_{\mathcal{S}}$, which relates elements that are copies of the same element of $\mathcal{S}$.

product structure $Q \times \mathcal{S}$

Definition 11.3.1. Let $\mathcal{S} = \big(U, (R_i)_{i\in\{1,\dots,n\}}\big)$ be a relational structure and let Q be a finite set. The *product structure* or simply *product* of $\mathcal{S}$ with Q is the relational structure

$$Q \times \mathcal{S} = \big(Q \times U,\ (P_q)_{q\in Q},\ (R'_i)_{i\in\{1,\dots,n\}},\ \mathrm{Eq}_{\mathcal{S}}\big),$$

where

$$\begin{aligned} P_q &= \big\{(q,u) \mid u \in U\big\},\\ R'_i &= \big\{\big((q,u_1),\dots,(q,u_k)\big) \mid q \in Q,\ (u_1,\dots,u_k) \in R_i\big\},\\ \mathrm{Eq}_{\mathcal{S}} &= \big\{\big((p,u),(q,u)\big) \mid p,q \in Q,\ u \in U\big\}. \end{aligned}$$

For example, we will consider products of the following structure over the natural numbers.

structure $\mathcal{N}$

Notation 11.3.2. We let $\mathcal{N}$ denote the structure $(\mathbb{N}, \mathrm{Succ})$ of the natural numbers with the successor relation $\mathrm{Succ} = \{(n, n+1) \mid n \in \mathbb{N}\}$.

$Q \times \mathcal{N}$

The product of the structure $\mathcal{N}$ with a finite set Q is the structure $Q \times \mathcal{N} = \big(Q \times \mathbb{N}, (P_q)_{q\in Q}, \mathrm{Succ}', \mathrm{Eq}_{\mathcal{N}}\big)$ with the following relations:

$$\begin{aligned} P_q &= \big\{(q,n) \mid n \in \mathbb{N}\big\},\\ \mathrm{Succ}' &= \big\{\big((q,n),(q,n+1)\big) \mid q \in Q,\ n \in \mathbb{N}\big\},\\ \mathrm{Eq}_{\mathcal{N}} &= \big\{\big((p,n),(q,n)\big) \mid p,q \in Q,\ n \in \mathbb{N}\big\}. \end{aligned}$$

We note that in so-called monadic second-order transductions as developed by Courcelle (see [BC10]), the operations of MSO-interpretations and of k-fold copying (that is, building a product structure) are combined into one.

MSO-family-definability translates to MSO-definability in a product structure and vice versa:

Proposition 11.3.3. Let $\mathcal{S}$ be a relational structure over a universe U, and let Q be a finite set. Let $R \subseteq (Q \times U)^k$ be a k-ary relation (with $k \in \mathbb{N}_{\geq 1}$).

R is MSO-definable in $Q \times \mathcal{S}$ $\Leftrightarrow$ R is MSO-family-definable in $\mathcal{S}$.

Suitable MSO formulas can be constructed effectively (in both directions).

Before we provide a proof, let us mention a useful consequence of this proposition.

Proposition 11.3.4. Let $\mathcal{S}$ be a relational structure over a universe U, and let Q be a finite set. Let $R \subseteq (Q \times U)^k$ be a k-ary relation (with $k \in \mathbb{N}_{\geq 1}$) that is MSO-definable in $Q \times \mathcal{S}$.
For every $Q' \subseteq Q$, the relation $R \cap (Q' \times U)^k$ is MSO-definable in $Q' \times \mathcal{S}$. Suitable MSO formulas can be constructed effectively.

Proof. Given an MSO formula $\varphi(x_1, \ldots, x_k)$ defining R in $Q \times \mathcal{S}$, an MSO formula $\varphi'(x_1, \ldots, x_k)$ defining $R \cap (Q' \times U)^k$ in $Q' \times \mathcal{S}$ can be obtained as follows: First, translate φ into a family of formulas $(\varphi_{q_1,\ldots,q_k})_{q_1,\ldots,q_k \in Q}$ defining R in $\mathcal{S}$, which is possible according to Proposition 11.3.3. Then apply the reverse direction of Proposition 11.3.3, but only translate the subfamily of formulas $(\varphi_{q_1,\ldots,q_k})_{q_1,\ldots,q_k \in Q'}$. □

We now turn to the proof of Proposition 11.3.3.

Proof for direction "⇐" of Proposition 11.3.3. Suppose R is defined in $\mathcal{S} = \big(U, (R_i)_{i \in \{1,\ldots,n\}}\big)$ by a family of formulas $\big(\varphi_{q_1,\ldots,q_k}(x_1, \ldots, x_k)\big)_{q_1,\ldots,q_k \in Q}$. We have to construct a formula $\varphi(y_1, \ldots, y_k)$ that defines R in $Q \times \mathcal{S}$. To that end, we first transform each formula $\varphi_{q_1,\ldots,q_k}$ over $\mathcal{S}$ into a corresponding formula $\widehat{\varphi}_{q_1,\ldots,q_k}$ over $Q \times \mathcal{S}$ whose first-order quantification scope is restricted to a fixed copy of $\mathcal{S}$ (we choose the one where the first component is q_1). More precisely, $\widehat{\varphi}_{q_1,\ldots,q_k}$ is obtained from $\varphi_{q_1,\ldots,q_k}$ by replacing all relation symbols R_i with their counterparts R'_i and replacing each subformula of the form $\exists x\, \psi$ with $\exists x \big(P_{q_1}(x) \wedge \psi\big)$. We can now construct the formula

$$\varphi(y_1, \ldots, y_k) = \bigvee_{q_1,\ldots,q_k \in Q} \Bigg(\bigwedge_{i=1}^{k} P_{q_i}(y_i) \wedge \exists x_1, \ldots, x_k \Bigg(\bigwedge_{i=1}^{k} \Big(\mathrm{Eq}_{\mathcal{S}}(x_i, y_i) \wedge P_{q_1}(x_i) \Big) \wedge \widehat{\varphi}_{q_1,\ldots,q_k}(x_1, \ldots, x_k) \Big) \Bigg),$$

which defines the relation R in $Q \times \mathcal{S}$. □

For the proof of the other direction ("⇒"), we will use the following notation.

Notation 11.3.5. Let Q and U be two sets, and let $M \subseteq Q \times U$. For $p \in Q$, we write M^p to denote the set $\{u \in U \mid (p, u) \in M\}$. M^p

---

First, we show a preparatory lemma (similar to Lemma 1.3.5 in [BCLo8]). Roughly speaking, it says that we can transfer MSO formulas from $Q \times \mathcal{S}$ to $\mathcal{S}$ by replacing each second-order variable X with a tuple of second-order variables $(Y^p)_{p \in Q}$, which represents a partition of X.

Lemma 11.3.6. Let $\mathcal{S} = (U, \overline{R})$ be a relational structure, and let Q be a finite set. Given an MSO formula $\varphi(x_1, \dots, x_k, X_1, \dots, X_\ell)$ over $Q \times \mathcal{S}$, we can construct a family of MSO formulas $(\varphi_{q_1,\dots,q_k})_{q_1,\dots,q_k \in Q}$ of the form $\varphi_{q_1,\dots,q_k}\big(y_1, \dots, y_k, (Y_1^p)_{p\in Q}, \dots, (Y_\ell^p)_{p\in Q}\big)$ over $\mathcal{S}$ such that

$$\begin{aligned} Q \times \mathcal{S} &\models \varphi\big[(q_1, u_1), \dots, (q_k, u_k),\ M_1, \dots, M_\ell\big] \\ &\Leftrightarrow \\ \mathcal{S} &\models \varphi_{q_1,\dots,q_k}\big[u_1, \dots, u_k,\ (M_1^p)_{p\in Q}, \dots, (M_\ell^p)_{p\in Q}\big] \end{aligned}$$

for all $(q_1, u_1), \dots, (q_k, u_k) \in Q \times U$ and all $M_1, \dots, M_\ell \subseteq Q \times U$.

Proof. We construct the desired formulas $(\varphi_{q_1,\dots,q_k})_{q_1,\dots,q_k \in Q}$ by induction on the structure of φ. To allow for a simpler inductive argument, we define these formulas for an arbitrary number of free variables, even if not all of these variables are actually used in the formula.

Base cases (atomic formulas).

- For $\varphi(x_1, \dots, x_k, \overline{X}) = x_i{=}x_j$:

$$\varphi_{q_1,\dots,q_k}(y_1, \dots, y_k, \overline{Y}) := \begin{cases} y_i{=}y_j & \text{if } q_i = q_j, \\ \texttt{false} & \text{otherwise.} \end{cases}$$

- For $\varphi(x_1, \dots, x_k, \overline{X}) = \mathrm{Eq}_{\mathcal{S}}(x_i, x_j)$:

$$\varphi_{q_1,\dots,q_k}(y_1, \dots, y_k, \overline{Y}) := y_i{=}y_j$$

- For $\varphi(x_1, \dots, x_k, X_1, \dots, X_\ell) = X_i(x_j)$:

$$\varphi_{q_1,\dots,q_k}(y_1, \dots, y_k, (Y_1^p)_{p\in Q}, \dots, (Y_\ell^p)_{p\in Q}) := Y_i^{q_j}(y_j)$$

- For $\varphi(x_1, \dots, x_k, \overline{X}) = P_p(x_i)$ (with $p \in Q$):

$$\varphi_{q_1,\dots,q_k}(y_1, \dots, y_k, \overline{Y}) := \begin{cases} \texttt{true} & \text{if } q_i = p, \\ \texttt{false} & \text{otherwise.} \end{cases}$$

- For $\varphi(x_1, \ldots, x_k, \overline{X}) = R'(x_{i_1}, \ldots, x_{i_n})$
  (for a relation R' of $Q \times \mathcal{S}$ derived from a relation R of $\mathcal{S}$):

$$\varphi_{q_1,\ldots,q_k}(y_1, \ldots, y_k, \overline{Y}) := \begin{cases} R(y_{i_1}, \ldots, y_{i_n}) & \text{if } q_{i_1} = \ldots = q_{i_n}, \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Inductive step.

- For $\varphi = \neg\psi$: $\quad \varphi_{\overline{q}} := \neg\psi_{\overline{q}}$
- For $\varphi = \psi \vee \psi'$: $\quad \varphi_{\overline{q}} := \psi_{\overline{q}} \vee \psi'_{\overline{q}}$
- For $\varphi(\overline{x}, \overline{X}) = \exists x\ \psi(\overline{x}, x, \overline{X})$:

$$\varphi_{\overline{q}}(\overline{y}, \overline{Y}) := \exists y \bigvee_{q \in Q} \psi_{\overline{q},q}(\overline{y}, y, \overline{Y})$$

- For $\varphi(\overline{x}, \overline{X}) = \exists X\ \psi(\overline{x}, \overline{X}, X)$:

$$\varphi_{\overline{q}}(\overline{y}, \overline{Y}) := \exists (Y^p)_{p \in Q}\ \psi_{\overline{q}}(\overline{y}, \overline{Y}, (Y^p)_{p \in Q})$$ □

We can now complete the proof of Proposition 11.3.3.

Proof for direction "⇒" of Proposition 11.3.3. Let R be MSO-definable in $Q \times \mathcal{S}$ by a formula $\varphi(x_1, \ldots, x_k)$. By Lemma 11.3.6, we can construct a corresponding family of MSO formulas $\left(\varphi_{q_1,\ldots,q_k}(y_1, \ldots, y_k)\right)_{q_1,\ldots,q_k \in Q}$ and we obtain, for all $(q_1, u_1), \ldots, (q_k, u_k) \in Q \times U$:

$$\begin{aligned} \big((q_1, u_1), \ldots, (q_k, u_k)\big) \in R &\Leftrightarrow Q \times \mathcal{S} \models \varphi[(q_1, u_1), \ldots, (q_k, u_k)] \\ &\Leftrightarrow \mathcal{S} \models \varphi_{q_1,\ldots,q_k}[u_1, \ldots, u_k]. \end{aligned}$$

Thus, R is MSO-family-definable in $\mathcal{S}$. □

By applying Lemma 11.3.6 to sentences (which are formulas without free variables), an MSO sentence over $Q \times \mathcal{S}$ can be transformed into a family consisting of a single MSO sentence over $\mathcal{S}$ that is true if and only if the original sentence is true. Thus, we obtain the following corollary.

Corollary 11.3.7. Let $\mathcal{S}$ be a relational structure with a decidable MSO theory. For any finite set Q, the MSO theory of $Q \times \mathcal{S}$ is decidable.

11.4 MSO Logic over Infinite Trees

We are particularly interested in MSO formulas over infinite trees.

Γ-branching tree $\mathcal{T}_\Gamma$

Definition 11.4.1. For a finite set Γ, the (infinite) Γ-*branching tree* is the structure $\mathcal{T}_\Gamma = \big(\Gamma^*, (\mathrm{Succ}_a)_{a\in\Gamma}\big)$ with $\mathrm{Succ}_a = \{(u, ua) \mid u \in \Gamma^*\}$.

Remark 11.4.2. Recall the structure $\mathcal{N} = (\mathbb{N}, \mathrm{Succ})$ of the natural numbers with the successor relation (see Notation 11.3.2). This structure is isomorphic to the Γ-branching tree $\mathcal{T}_\Gamma$ for $\Gamma = \{1\}$, by virtue of the isomorphism $f : \mathbb{N} \to \{1\}^*$ with $f(n) = 1^n$.

The following well-known fact (see, for example, [CLN+10]) will be useful in Chapter 16.

Proposition 11.4.3. Let $L \subseteq \Gamma^*$ be a language over a finite alphabet Γ.

$$L \text{ is regular} \iff \begin{array}{l} L \text{ is MSO-definable by some formula } \varphi(x) \\ \text{in the } \Gamma\text{-branching tree } \mathcal{T}_\Gamma. \end{array}$$

From a DFA recognizing L we can derive a corresponding MSO formula, and vice versa.

We will also rely on the decidability of certain MSO theories. We know from [Büc66] that the MSO theory of the structure $\mathcal{N}$ is decidable. By Corollary 11.3.7, this also holds for products of $\mathcal{N}$ with finite sets:

Proposition 11.4.4. For every finite set Q, the MSO theory of $Q \times \mathcal{N}$ is decidable.

More generally, for a finite set Γ, the MSO theory of the Γ-branching tree $\mathcal{T}_\Gamma$ is decidable (see [Rab68]), so we obtain the following proposition.

Proposition 11.4.5. For all finite sets Q and Γ, the MSO theory of $Q \times \mathcal{T}_\Gamma$ is decidable.

11.5 A Pumping Lemma for MSO-Definable Relations

Let us now establish a rather technical lemma about a specific kind of MSO-definable relations, namely binary relations of natural numbers

that are MSO-definable in $\mathcal{N}$. We show that for such a relation R the membership of a pair $(m_1, m_2) \in \mathbb{N} \times \mathbb{N}$ in R is determined by the equivalence class of (m_1, m_2) with respect to the following equivalence relation. This will be useful in Section 12.4 and, most notably, for the pumping arguments in Section 12.5.

Definition 11.5.1. Let $n \in \mathbb{N}$. Two pairs $(m_1, m_2), (m_1', m_2') \in \mathbb{N} \times \mathbb{N}$ are called *n-equivalent*, written as $(m_1, m_2) \sim_n (m_1', m_2')$, if all of the following conditions are satisfied:

n-equivalence $\sim_n$

- $m_1 < m_2 \Leftrightarrow m_1' < m_2'$
- $m_1 < n \Leftrightarrow m_1' < n$
- $m_2 < n \Leftrightarrow m_2' < n$
- $|m_1 - m_2| < n \Leftrightarrow |m_1' - m_2'| < n$
- $m_1 \equiv m_1' \pmod{n}$
- $m_2 \equiv m_2' \pmod{n}$

Lemma 11.5.2. Let $\varphi(x_1, x_2)$ be an MSO formula defining a binary relation $R_\varphi \subseteq \mathbb{N} \times \mathbb{N}$ in the structure $\mathcal{N}$. There exists a number $n_\varphi \in \mathbb{N}$, which we call the *pumping modulus of* φ, such that the following holds for all $(m_1, m_2), (m_1', m_2') \in \mathbb{N} \times \mathbb{N}$:
If $(m_1, m_2) \sim_{n_\varphi} (m_1', m_2')$, then $(m_1, m_2) \in R_\varphi \Leftrightarrow (m_1', m_2') \in R_\varphi$.

pumping modulus

Proof. The relation R_φ can be represented by the ω-language

$$L_\varphi = \{\alpha_{m_1,m_2} \mid (m_1, m_2) \in R_\varphi\},$$

where α_{m_1,m_2} is the ω-word $\binom{b_0^1}{b_0^2}\binom{b_1^1}{b_1^2}\binom{b_2^1}{b_2^2}\binom{b_3^1}{b_3^2}\ldots$ over the alphabet $\{0,1\}^2$ with

$$b_i^j = \begin{cases} 1 & \text{if } i = m_j, \\ 0 & \text{otherwise.} \end{cases}$$

This is the ω-language defined by the MSO formula φ. Since MSO formulas can define precisely the regular ω-languages (by Büchi's Theorem [Büc66]), the ω-language L_φ is regular. Thus, there is a deterministic ω-automaton $\mathcal{A}$, say with a parity acceptance condition, that recognizes L_φ (see [Tho97]). Let n be the number of states of this automaton $\mathcal{A}$. We claim that $n_\varphi := n!$ is a suitable choice for the pumping modulus of φ.

Consider two n_φ-equivalent pairs $(m_1, m_2), (m_1', m_2') \in \mathbb{N} \times \mathbb{N}$, i.e., $(m_1, m_2) \sim_{n_\varphi} (m_1', m_2')$. These pairs are represented by the ω-words α_{m_1,m_2} and $\alpha_{m_1',m_2'}$. We assume that $m_1 < m_2$, which implies $m_1' < m_2'$. (The case $m_1 > m_2$ follows by symmetry and for the case $m_1 = m_2$, we can apply a simplified version of the arguments below.) Thus we have

$$\alpha_{m_1,m_2} = (0,0)^{m_1}(1,0)(0,0)^{m_2-m_1-1}(0,1)(0,0)^\omega \quad \text{and}$$
$$\alpha_{m_1',m_2'} = (0,0)^{m_1'}(1,0)(0,0)^{m_2'-m_1'-1}(0,1)(0,0)^\omega.$$

Consider the unique runs π and π' of the automaton $\mathcal{A}$ on these ω-words. We will show that both π and π' reach the same state when they arrive at the symbol $(0,1)$, so either both are accepting or both are rejecting, and hence $(m_1, m_2) \in R_\varphi \Leftrightarrow (m_1', m_2') \in R_\varphi$.

First, we show that both runs reach the same state when they arrive at the symbol $(1,0)$. Consider the run of $\mathcal{A}$ on $(0,0)^\omega$. At some position i with $i \leq n \leq n_\varphi$, this run will become periodic with a period that is at most n and hence divides $n_\varphi = n!$. Therefore, since $m_1 \geq n_\varphi$ if and only if $m_1' \geq n_\varphi$ and since $m_1 \equiv m_1' \pmod{n_\varphi}$, the state q reached after the prefix $(0,0)^{m_1}$ is the same as the state reached after the prefix $(0,0)^{m_1'}$.

Now consider the run of $\mathcal{A}$ starting in state q on the ω-word $(1,0)(0,0)^\omega$. Again, at some position $j \leq n_\varphi$, that run will become periodic with a period dividing n_φ. Since $m_2 - m_1 \geq n_\varphi$ if and only if $m_2' - m_1' \geq n_\varphi$ and since $m_2 - m_1 \equiv m_2' - m_1' \pmod{n_\varphi}$, the state that is reached after reading $(1,0)(0,0)^{m_2-m_1-1}$ is indeed the same as after reading $(1,0)(0,0)^{m_2'-m_1'-1}$. □

12

ℕ-Memory Automata on Finite Words

12.1 Overview

In this chapter, we present an automaton model for infinite alphabets, called ℕ-*memory automata*. A formal definition will be provided in Section 12.3, but let us start with an intuitive description.

We consider automata on words of the form $w = u_1 u_2 u_3 \dots u_n$, where each symbol u_ℓ, for $\ell \in \{1, \dots, n\}$, is itself a word over a finite alphabet Σ. The input alphabet of such an automaton is hence the infinite set Σ^*. We call the elements of this infinite alphabet *micro words*. We think of a word of micro words as a "horizontal" sequence (arranged from left to right) of "vertical" sequences (arranged from bottom to top), as shown in Figure 8.

micro word

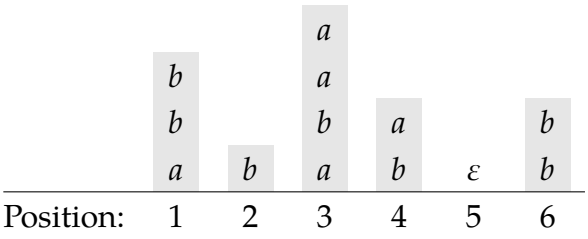


Figure 8: A finite word over the infinite alphabet $\{a, b\}^*$.

Remark 12.1.1. A word $w \in (\Sigma^*)^*$ consisting of the symbols (i.e., micro words) $u_1, \dots, u_n \in \Sigma^*$ is written as $u_1 \dots u_n$. This should not be confused with the usual concatenation of these micro words, which would yield a word over the finite alphabet Σ.

Consider the following languages over the alphabet $\{a, b\}^*$.

Example 12.1.2. Let $\Sigma = \{a, b\}$.

- $L_1 = \{u_1 \dots u_n \in (\Sigma^*)^* \mid |u_1| = |u_n|\}$
- $L_2 = \{u_1 \dots u_n \in (\Sigma^*)^* \mid |u_n| \geq n\}$
- $L_3 = \{u_1 \dots u_n \in (\Sigma^*)^* \mid u_1 = a^{k_1} b^{\ell_1}, \dots, u_n = a^{k_n} b^{\ell_n}$ with $k_i = k_{i-1} + \ell_{i-1}$ for all $i \in \{2, \dots, n\}\}$

To recognize such languages, our automata need some sort of unbounded memory. Consider an input word $w = u_1 \dots u_n \in (\Sigma^*)^*$. Note that each micro word $u_\ell = a_1 \dots a_m \in \Sigma^*$ can be viewed as a partial labeling of the structure $\mathcal{N} = (\mathbb{N}, \mathrm{Succ})$, i.e., of the natural numbers with the successor relation, such that each position $i \in \{1, \dots, m\}$ is labeled by the corresponding symbol a_i. To carry information from a micro word u_ℓ to the next micro word $u_{\ell+1}$, an ℕ-memory automaton is not only equipped with a finite set of states Q but can also memorize in every step a position $i \in \mathbb{N}$ of the structure $\mathcal{N}$ – hence the name ℕ-memory automata.

memorized position

A configuration of an ℕ-memory automaton is therefore a pair $(p, i) \in Q \times \mathbb{N}$ consisting of a state and a *memorized position*, and a transition has the form

$$\big((p, i), u, (q, j)\big) \quad \text{with} \quad p, q \in Q, u \in \Sigma^*, \text{and } i, j \in \mathbb{N}.$$

In general, the transition relation of such an automaton can be infinite. To allow for a finite representation, we have to consider a restricted class of transition relations. We will use transition relations that can be computed by so-called *pebble automata*. These relations turn out to be the ones that can be defined in MSO logic over the structure $\mathcal{N}$.

To get a sense of the expressive power of ℕ-memory automata, let us take a closer look at the languages from Example 12.1.2. To recognize the language L_1, an ℕ-memory automaton could memorize the last position of the first micro word u_1 until it reaches the end of the input word, and then switch to a final state if the memorized position matches the last position of the final micro word u_n. The language L_2 can be recognized by incrementing the memorized position by 1 in each step (starting with the position 0) and verifying that the last position of the last micro word is greater than the final memorized position. Finally, the language L_3 is recognized by an ℕ-memory automaton that memorizes in each step the last position of the current micro word and checks that in the next micro word, all positions from 1 to the memorized position are labeled by a and the remaining positions are labeled by b.

The remainder of this chapter is structured as follows: We begin by presenting the required concepts, in particular pebble automata and their expressive equivalence to MSO formulas (Section 12.2). We can then provide the formal definition of ℕ-memory automata and show that their transition relations can equivalently be defined by MSO formulas (Section 12.3). Subsequently, we take a look at the expressive power of ℕ-memory automata and compare them to related automata models (Section 12.4). We examine the closure properties of the class

of languages recognized by $\mathbb{N}$-memory automata (Section 12.5), and finally we address the decidability of some important decision problems (Section 12.6).

Remark 12.1.3. We will often regard the set $\mathbb{N}$ of the natural numbers as an alphabet Σ^* with a unary alphabet Σ, specifically $\Sigma = \{1\}$. This is justified by the fact that each number $n \in \mathbb{N}$ can be represented as a micro word $\underbrace{111\ldots1}_{n \text{ times}} \in \{1\}^*$, and vice versa.

12.2 Preliminaries: Trips and Pebble Automata

As described above, a transition $\big((p,i),u,(q,j)\big)$ of an $\mathbb{N}$-memory automaton involves an update of the memorized position from $i \in \mathbb{N}$ to $j \in \mathbb{N}$. This update can be understood as a *trip* from the old position i to the new position j, a notion that was introduced in a slightly different form by Engelfriet et al. [EHB99].

In this section, we consider the class of regular sets of trips, which coincide with those that can be defined by MSO formulas. We employ a slight adaptation of a proof from [EHB99] to show that these are also precisely the sets of trips computed by pebble automata. This serves as a preparation for the formal definition of $\mathbb{N}$-memory automata in Section 12.3.

12.2.1 Regular Sets of Trips

Definition 12.2.1. A *trip* is a triple (u,i,j) consisting of a finite word $u \in \Sigma^*$ over a finite alphabet Σ and two numbers $i,j \in \mathbb{N}$.

trip

Metaphorically speaking, a trip (u,i,j) represents a journey from position i to position j in a certain "landscape" (namely the structure $\mathcal{N}$ augmented with a labeling of the positions $1,\ldots,|u|$, defined by the word u).

Note: In contrast to the definition in [EHB99], i and j need not lie in $\{1,\ldots,|u|\}$.

Definition 12.2.2. A set of trips $T \subseteq \Sigma^* \times \mathbb{N} \times \mathbb{N}$ is called *functional* if for every $u \in \Sigma^*$ and $i \in \mathbb{N}$, there is at most one $j \in \mathbb{N}$ such that $(u,i,j) \in T$.

functional set of trips

We are particularly interested in so-called regular sets of trips. In order to define the notion of regularity for sets of trips, we encode

these sets as ω-languages. To that end, we represent a finite word u by a corresponding ω-word, and we then augment that ω-word with markers at the positions i and j to represent a trip (u, i, j).

extended alphabet $\widehat{\Sigma}$

ω-extension $\widehat{u}$

Definition 12.2.3. For a finite alphabet Σ, we let $\widehat{\Sigma} = \Sigma \cup \{\bot, \#\}$. We call $\widehat{\Sigma}$ an *extended alphabet*. The *ω-extension* of a finite word $u \in \Sigma^*$ is the ω-word $\widehat{u} = \bot u \#^\omega$ over the extended alphabet $\widehat{\Sigma}$.

marked ω-word mark$(\widehat{u}, i, j)$

Definition 12.2.4. Let $u \in \Sigma^*$ be a finite word, and let $\widehat{u} = a_0 a_1 a_2 \dots$ be its ω-extension (with $a_0 = \bot$). Let $i, j \in \mathbb{N}$.
The *marked ω-word* associated with u, i, j, denoted by mark$(\widehat{u}, i, j)$, is the ω-word $c_0 c_1 c_2 \dots$ over the alphabet $\bigcup_{a \in \widehat{\Sigma}} \{a, (a, 1, 0), (a, 0, 1), (a, 1, 1)\}$ with

$$c_n = \begin{cases} a_n & \text{if } n \neq i \text{ and } n \neq j, \\ (a_n, 1, 0) & \text{if } n = i \neq j, \\ (a_n, 0, 1) & \text{if } n = j \neq i, \\ (a_n, 1, 1) & \text{if } n = i = j. \end{cases}$$

mark$(\widehat{u}, i)$

Similarly, the marked ω-word mark$(\widehat{u}, i)$ associated with u and i is the ω-word $c_0 c_1 c_2 \dots$ over the alphabet $\bigcup_{a \in \widehat{\Sigma}} \{a, (a, 1)\}$ with

$$c_n = \begin{cases} a_n & \text{if } n \neq i, \\ (a_n, 1) & \text{if } n = i. \end{cases}$$

For example, the trip $(u, 2, 5)$ with $u = a_1 a_2 a_3$ is represented by the marked ω-word mark$(\widehat{u}, 2, 5) = \bot\ a_1\ (a_2, 1, 0)\ a_3\ \#\ (\#, 0, 1)\ \#^\omega$. A set of trips can be represented by a corresponding ω-language:

mark(T)

Notation 12.2.5. For a set of trips $T \subseteq \Sigma^* \times \mathbb{N} \times \mathbb{N}$, we let

$$\text{mark}(T) = \{\text{mark}(\widehat{u}, i, j) \mid (u, i, j) \in T\}.$$

regular set of trips

Definition 12.2.6. A set of trips $T \subseteq \Sigma^* \times \mathbb{N} \times \mathbb{N}$ is called *regular* if the ω-language mark(T) is regular.

Regular sets of trips can be represented by MSO formulas, as we will see now.

Definition 12.2.7. Let Σ be a finite alphabet, and let $\varphi(\overline{Z}, x, y)$ be an MSO formula over the structure $\mathcal{N}$, with $\overline{Z} = (Z_a)_{a\in\Sigma}$. The *set of trips induced by* φ, denoted by $\mathrm{Trips}(\varphi)$, is defined as follows:

set of trips induced by an MSO formula

$$\mathrm{Trips}(\varphi) = \left\{(u, i, j) \in \Sigma^* \times \mathbb{N} \times \mathbb{N} \mid \mathcal{N} \models \varphi[(K_a^u)_{a\in\Sigma}, i, j]\right\},$$

$\mathrm{Trips}(\varphi)$

where for $u = a_1 \dots a_m \in \Sigma^*$ and $a \in \Sigma$, we let

$$K_a^u = \left\{k \in \{1, \dots, m\} \mid a_k = a\right\}.$$

For an MSO formula $\varphi(\overline{Z}, x, y)$ inducing a set of trips T, we can regard $\mathrm{mark}(T)$ as the ω-language defined by φ. By Büchi's Theorem [Büc66], an ω-language can be defined by an MSO formula if and only if it is regular (and we can effectively transform a given MSO formula into a corresponding Büchi automaton, and vice versa). The sets of trips induced by MSO formulas are therefore the regular ones:

Proposition 12.2.8. A set of trips $T \subseteq \Sigma^* \times \mathbb{N} \times \mathbb{N}$ is regular if and only if there is an MSO formula $\varphi(\overline{Z}, x, y)$ with $\mathrm{Trips}(\varphi) = T$.

12.2.2 Pebble Automata

As described in [EHB99], we can use pebble automata to compute sets of trips.

Definition 12.2.9. A *pebble automaton* $\mathcal{B}$ is a tuple $(S, \Sigma, \Delta, S_0, S_f)$, where

pebble automaton

- S is a finite set of states,
- Σ is a finite alphabet,
- $\Delta \subseteq (S \setminus S_f) \times \Sigma \times \{0, 1\} \times S \times \mathrm{Act}$ is the transition relation, where $\mathrm{Act} = \{\uparrow, \downarrow, \diamond\}$ is the set of possible actions,
- $S_0 \subseteq S$ is the set of initial states, and
- $S_f \subseteq S$ is the set of final states.

A pebble automaton is a "two-way'" automaton equipped with one pebble. Intuitively, a transition $(s, a, b, s', d) \in \Delta$ allows the automaton to switch from state s to state s', provided that the input symbol at the current position is a, and that the pebble is on the current position if

and only if $b = 1$. Moreover, the automaton then performs the action indicated by d, where $\uparrow$ and $\downarrow$ signify an upward or downward move, respectively, and $\diamond$ means that the automaton places the pebble at the current position and stays there. Note that we think of the words processed by pebble automata as sequences of symbols arranged in a vertical way, starting at the bottom.

A pebble automaton as defined above is called *deterministic* if it only has a single initial state and for every triple $(s, a, b) \in (S \setminus S_f) \times \Sigma \times \{0, 1\}$, there is exactly one pair $(s', d) \in S \times \mathrm{Act}$ such that $(s, a, b, s', d) \in \Delta$.

Let us now define the behavior of a pebble automaton in a formal way. We consider both finite and infinite input words. In general, a *configuration* of a pebble automaton $\mathcal{B} = (S, \Sigma, \Delta, S_0, S_f)$ is a tuple (s, h, ℓ), consisting of a state $s \in S$, the position $h \in \mathbb{N}$ of the automaton in the input word, and the position $\ell \in \mathbb{N}$ of the pebble in the input word. A *run* of $\mathcal{B}$ on a finite or infinite word $a_0 a_1 a_2 \ldots$ over the alphabet Σ is a (possibly infinite) sequence of configurations

$$(s_1, h_1, \ell_1)(s_2, h_2, \ell_2)(s_3, h_3, \ell_3) \ldots$$

such that for every pair of consecutive configurations (s_k, h_k, ℓ_k) and $(s_{k+1}, h_{k+1}, \ell_{k+1})$, there exists a transition $(s_k, a_{h_k}, b_k, s_{k+1}, d_k) \in \Delta$, with $b_k = 1$ if and only if $h_k = \ell_k$, that satisfies one of the following conditions:

- $d_k = \uparrow$ and $h_{k+1} = h_k + 1,\ \ell_{k+1} = \ell_k$,
- $d_k = \downarrow$ and $h_k > 0,\ h_{k+1} = h_k - 1,\ \ell_{k+1} = \ell_k$,
- $d_k = \diamond$ and $\ell_{k+1} = h_{k+1} = h_k$.

Pebble automata can be used to compute trips: A finite run from position i to position j on an ω-word of the form $\widehat{u} = \bot u \#^\omega$ (for a finite word $u \in \Sigma^*$) corresponds to the trip (u, i, j). By convention, we only consider runs that start and end in configurations in which the position of the pebble coincides with the position of the automaton. This can be formalized as follows.

set of trips computed by a pebble automaton

Definition 12.2.10. Let $\mathcal{B} = (S, \widehat{\Sigma}, \Delta, S_0, S_f)$ be a pebble automaton over an extended alphabet $\widehat{\Sigma}$. The *set of trips computed by* $\mathcal{B}$, denoted by $\mathrm{Trips}(\mathcal{B})$, is defined as follows:

$\mathrm{Trips}(\mathcal{B})$

$$\mathrm{Trips}(\mathcal{B}) = \big\{(u, i, j) \in \Sigma^* \times \mathbb{N} \times \mathbb{N} \mid \text{for some } s_0 \in S_0, s_f \in S_f, \text{ there is a run of } \mathcal{B} \text{ on } \widehat{u} \text{ from the configuration } (s_0, i, i) \text{ to the configuration } (s_f, j, j)\big\}$$

While we will use pebble automata mainly to compute sets of trips as described above, the classical application of pebble automata is to recognize languages of finite words. In that latter context, a pebble automaton $\mathcal{B}$ *accepts* a given *finite* word $v \in \Sigma^*$ if there is a run of $\mathcal{B}$ on v that starts in a configuration $(s_0, 0, 0)$ with $s_0 \in S_0$ and ends in a configuration (s_f, h, ℓ) with $s_f \in S_f$. The *language recognized by* $\mathcal{B}$ is then defined as $L(\mathcal{B}) = \{v \in \Sigma^* \mid \mathcal{B} \text{ accepts } v\}$.

language recognized by a pebble automaton

It was shown by Blum and Hewitt [BH67] that for every pebble automaton, one can construct an NFA recognizing the same language. Conversely, every NFA can be regarded as a pebble automaton that does not use its pebble and only moves in one direction. Therefore, the languages recognized by pebble automata are precisely the regular ones.

12.2.3 Pebble Automata vs. MSO Formulas

The goal of this subsection is to show that the sets of trips computed by pebble automata are precisely those that can be induced by MSO formulas, namely the regular ones. The proof is essentially the same as in [EHB99], adapted to our generalized definition of trips (u, i, j), where $i, j \in \mathbb{N}$ are not restricted to positions of the finite word u.

First, we prove that pebble automata can only compute regular sets of trips. The following simple proposition will be useful.

Proposition 12.2.11. Let Σ be a finite alphabet with $\# \in \Sigma$. For every regular language (of finite words) $W \subseteq \Sigma^*$, the ω-language $W \cdot \{\#\}^\omega$ is regular.

Proof. An NFA recognizing W can be converted into a Büchi automaton recognizing $W \cdot \{\#\}^\omega$ by adding a single accepting state with a #-loop, which can be entered from every final state of the NFA. □

Proposition 12.2.12. Let $\mathcal{B}$ be a pebble automaton over an extended alphabet. The set of trips $\mathrm{Trips}(\mathcal{B})$ is regular. Moreover, if $\mathcal{B}$ is deterministic, then $\mathrm{Trips}(\mathcal{B})$ is functional.

Proof. To show that the ω-language $\mathrm{mark}(\mathrm{Trips}(\mathcal{B}))$ and hence the set of trips $\mathrm{Trips}(\mathcal{B})$ are regular, we will construct a pebble automaton $\mathcal{B}'$ such that $L(\mathcal{B}') \cdot \{\#\}^\omega = \mathrm{mark}(\mathrm{Trips}(\mathcal{B}))$. This pebble automaton can be converted into an equivalent NFA (see [BH67]), so $L(\mathcal{B}')$ is regular. Thus, by Proposition 12.2.11, $\mathrm{mark}(\mathrm{Trips}(\mathcal{B}))$ is a regular ω-language.

Let us now specify the pebble automaton $\mathcal{B}'$, which processes finite input words over the alphabet $\bigcup_{a \in \widehat{\Sigma}} \{a, (a,1,0), (a,0,1), (a,1,1)\}$, where $\widehat{\Sigma}$ is the extended alphabet used by the original pebble automaton $\mathcal{B}$. The automaton $\mathcal{B}'$ first verifies that the input word has the required format: The symbol $\perp$ must occur at position 0 but nowhere else; there must be exactly one position labeled with the first marker (i.e., with a symbol of the form $(a,1,*)$) and exactly one position labeled with the second marker (i.e., with a symbol of the form $(a,*,1)$); finally, if the padding symbol # occurs at some position, then it must also occur at all positions thereafter.

After this "syntax check", the automaton $\mathcal{B}'$ moves to the position i that is indicated by the first marker and places the pebble there. Then it simulates the original pebble automaton $\mathcal{B}$, starting at the position i. When a final state of $\mathcal{B}$ is reached, the automaton $\mathcal{B}'$ verifies that the pebble lies on the current position j of the automaton and that this position is labeled with the second marker. In that case, $\mathcal{B}'$ accepts, since the input word encodes a trip (u,i,j) that is computed by $\mathcal{B}$.

Note that to compute a trip (u,i,j), the automaton $\mathcal{B}$ might temporarily move beyond the end of the word u and beyond the positions i and j. Such a run of $\mathcal{B}$ can only be simulated by the automaton $\mathcal{B}'$ if the input word contains a sufficient number of trailing padding symbols #. Therefore, $\mathcal{B}'$ does not necessarily accept all words v with $v\#^\omega = \mathrm{mark}(\widehat{u}, i, j)$, but it will accept some of them, namely those with sufficient padding – which suffices to satisfy the condition $L(\mathcal{B}') \cdot \{\#\}^\omega = \mathrm{mark}(\mathrm{Trips}(\mathcal{B}))$. This concludes the proof of the first part of the proposition.

Now suppose that $\mathcal{B}$ is a deterministic pebble automaton. From a given start configuration on a given input word, such an automaton can reach at most one configuration that involves a final state. (Note that the final states of a pebble automaton cannot have outgoing transitions.) The set of trips computed by $\mathcal{B}$ is therefore functional. □

Before we address the converse of Proposition 12.2.12, we note that for a set of trips T, the ω-language $\mathrm{mark}(T)$ only contains ω-words with the suffix $\#^\omega$. We establish the following connection between such ω-languages and corresponding languages of finite words.

shortest relevant prefix relPref(α)

relPref(L)

Definition 12.2.13. Let Σ be a finite alphabet with $\# \in \Sigma$. Let α be an ω-word such that $\alpha \in \Sigma^* \cdot \{\#\}^\omega$. The *shortest relevant prefix* of α, denoted by $\mathrm{relPref}(\alpha)$, is the shortest finite word $v \in \Sigma^*$ such that $v\#^\omega = \alpha$. For an ω-language $L \subseteq \Sigma^* \cdot \{\#\}^\omega$, we let $\mathrm{relPref}(L) = \{\mathrm{relPref}(\alpha) \mid \alpha \in L\}$.

For example, consider the trip $(a_1a_2a_3, 2, 5)$ and the associated marked ω-word $\alpha = \bot\ a_1\ (a_2, 1, 0)\ a_3\ \#\ (\#, 0, 1)\ \#^\omega$. The shortest relevant prefix of α is the finite word $\mathrm{relPref}(\alpha) = \bot\ a_1\ (a_2, 1, 0)\ a_3\ \#\ (\#, 0, 1)$.

Proposition 12.2.14. Let Σ be a finite alphabet with $\# \in \Sigma$. For every regular ω-language L with $L \subseteq \Sigma^* \cdot \{\#\}^\omega$, the language $\mathrm{relPref}(L) \subseteq \Sigma^*$ is regular.

Proof. Let $\mathcal{C}$ be a nondeterministic Büchi automaton recognizing L. The language $\mathrm{relPref}(L)$ consists of all finite words $v \in \Sigma^*$ that satisfy the following conditions: The ω-word $v\#^\omega$ is accepted by $\mathcal{C}$, and v does not end with the symbol #.

We can easily convert $\mathcal{C}$ into an NFA recognizing the set of words that satisfy the first condition – it suffices to mark as final states precisely those states from which there is a sequence of #-transitions that leads to a cycle of #-transitions containing an accepting state of $\mathcal{C}$. We can also construct an NFA recognizing the set of words that do not end with #. Now we build the intersection of the two NFAs, and we obtain an NFA recognizing $\mathrm{relPref}(L)$, so this language is indeed regular. □

We can now show the converse of Proposition 12.2.12.

Proposition 12.2.15. Every regular set of trips $T \subseteq \Sigma^* \times \mathbb{N} \times \mathbb{N}$ can be computed by a pebble automaton. Moreover, if T is also functional, then it can be computed by a deterministic pebble automaton.

Proof. We prove the two parts of the proposition separately.

First part. Since T is regular, the ω-language $\mathrm{mark}(T)$ is regular (by definition). Note that all ω-words in $\mathrm{mark}(T)$ end with the suffix $\#^\omega$. Thus, we can apply Proposition 12.2.14, which says that $\mathrm{relPref}(\mathrm{mark}(T))$ is a regular language of finite words. Let $\mathcal{D}$ be a DFA recognizing $\mathrm{relPref}(\mathrm{mark}(T))$. Note that for an ω-word $\alpha = \mathrm{mark}(\widehat{u}, i, j)$ representing a trip (u, i, j), the shortest relevant prefix $\mathrm{relPref}(\alpha)$ is the prefix that ends at the position $\max\{|u|, i, j\}$.

A first approach to the construction of a pebble automaton computing the set of trips T might look like this: Starting at a position i of a given ω-word of the form $\widehat{u} = \bot u\#^\omega$ for some $u \in \Sigma^*$, the automaton guesses nondeterministically a position j and places the pebble there. Then it checks whether $(u, i, j) \in T$ by simulating the DFA $\mathcal{D}$ on the prefix up to (and including) position $\max\{|u|, i, j\}$, and finally walks back to the target position j if this test was successful.

This construction does not work because the pebble automaton would have to be able to detect both positions i and j during the simulation of $\mathcal{D}$, but by placing the pebble on position j it would "forget" the start position i. However, we can solve this problem by handling the three cases $j > i$, $j < i$, and $j = i$ separately. First, the pebble automaton places the pebble on the start position i and chooses nondeterministically one of the three cases, then it proceeds as follows.

For the case $j > i$, it simulates the DFA $\mathcal{D}$ from the beginning of $\widehat{u}$ until it arrives at the position i, treating the symbol $a \in \widehat{\Sigma}$ at that position as $(a, 1, 0)$. Then it continues the simulation and while doing so, chooses nondeterministically a position j, where it places the pebble. The symbol $a \in \widehat{\Sigma}$ at that position is treated as $(a, 0, 1)$. If at some point the simulation reaches a final state of $\mathcal{D}$ and the next symbol is the padding symbol #, then the trip (u, i, j) is in T, so the pebble automaton moves back to position j and enters a final state.

The case $j = i$ is handled similarly: Here, the symbol $a \in \widehat{\Sigma}$ at the start position $i = j$ is treated as $(a, 1, 1)$ and no nondeterministic choice for j is necessary.

For the case $j < i$, note that we can transform the DFA $\mathcal{D}$ into a DFA $\mathcal{D}^r$ recognizing the reverse of $L(\mathcal{D})$, i.e., $L(\mathcal{D}^r) = \{c_n \dots c_1 \mid c_1 \dots c_n \in L(\mathcal{D})\}$. The pebble automaton now reads the prefix of $\widehat{u}$ ending at the position $\max\{|u|, i\}$ in reverse order (downward), simulating the DFA $\mathcal{D}^r$. The symbol $a \in \widehat{\Sigma}$ at position i (indicated by the pebble) is treated as $(a, 1, 0)$. While continuing the simulation from position i downward, the pebble automaton nondeterministically guesses a position j, places the pebble there, and treats the symbol $a \in \widehat{\Sigma}$ at that position as $(a, 0, 1)$. If the simulation reaches a final state of $\mathcal{D}^r$ when it has arrived at position 0 (indicated by the symbol $\perp$), then the pebble automaton moves to position j again and enters a final state. This concludes the construction of the desired pebble automaton.

Second part (functional sets of trips). Note that in the construction of the pebble automaton above, we have used nondeterminism for two purposes: for the choice between the cases $j > i$, $j < i$, and $j = i$, and for guessing the exact position j. Now suppose that the set of trips T is not only regular but also functional. We want to show that nondeterminism is not necessary in this case, so we can construct a *deterministic* pebble automaton computing T.

First we show that for given $u \in \Sigma^*$ and $i \in \mathbb{N}$, the pebble automaton can decide deterministically whether the uniquely determined (if at all existing) j with $(u, i, j) \in T$ is larger than i, smaller than i, or equal to i.

Since $\mathrm{mark}(T) = \{\mathrm{mark}(\widehat{u}, i, j) \mid (u, i, j) \in T\}$ is a regular ω-language, so is

$$L_> := \{\mathrm{mark}(\widehat{u}, i) \mid \text{there is a } j > i \text{ such that } (u, i, j) \in T\}.$$

Thus, by Proposition 12.2.14, we can construct a DFA recognizing $\mathrm{relPref}(L_>)$. Analogously, we obtain DFAs recognizing the shortest relevant prefixes of

$$L_< := \{\mathrm{mark}(\widehat{u}, i) \mid \text{there is a } j < i \text{ such that } (u, i, j) \in T\} \text{ and}$$
$$L_= := \{\mathrm{mark}(\widehat{u}, i) \mid (u, i, i) \in T\}.$$

To determine which of the three cases for j applies, the pebble automaton simulates these three DFAs on the prefix up to position $\max\{|u|, i\}$ of $\widehat{u}$, treating the symbol $a \in \widehat{\Sigma}$ at the pebble position i as $(a, 1)$, and checks which one of them (if any) ends up in a final state.

Now we show that the exact position j can be found deterministically. We consider here the case $j > i$; the case $j < i$ is handled analogously. Instead of guessing this position nondeterministically while simulating the DFA $\mathcal{D}$, our new, deterministic pebble automaton simply tries all positions $j > i$, moving the pebble upward one by one, until the correct position is found. When placing the pebble on a position j, the pebble automaton remembers the state of $\mathcal{D}$ that was reached before that position, so that it can resume the simulation of $\mathcal{D}$ from that position if j was not yet the correct choice. □

In summary, by Proposition 12.2.12 and Proposition 12.2.15, a set of trips is regular if and only if it can be computed by a pebble automaton. Furthermore, a regular set of trips is functional if and only if it can be computed by a deterministic pebble automaton.

Since MSO formulas can induce precisely the regular sets of trips, as stated in Proposition 12.2.8, we can derive the following corollary. Note that we can effectively switch between the equivalent representations of a given set of trips because the proofs provided above are constructive.

Corollary 12.2.16. Let $T \subseteq \Sigma^* \times \mathbb{N} \times \mathbb{N}$ be a set of trips. The following are effectively equivalent:

(1) There is a pebble automaton $\mathcal{B}$ with $\mathrm{Trips}(\mathcal{B}) = T$.
(2) There is an MSO formula $\varphi(\overline{Z}, x, y)$ with $\mathrm{Trips}(\varphi) = T$.

If T is functional, then the following is effectively equivalent to (1), (2):

(3) There is a deterministic pebble automaton $\mathcal{B}$ with $\mathrm{Trips}(\mathcal{B}) = T$.

12.2.4 Direct Translation from Pebble Automata to MSO Formulas

Converting a given pebble automaton into an equivalent MSO formula using the constructions underlying Proposition 12.2.12 and Proposition 12.2.8 involves a transformation of the original pebble automaton into another pebble automaton recognizing a language of finite words, a conversion of the latter pebble automaton into an NFA and then into a Büchi automaton, and finally a translation of that Büchi automaton into an MSO formula. As an alternative, we now provide a direct translation from pebble automata to MSO formulas, as illustrated in Figure 9.

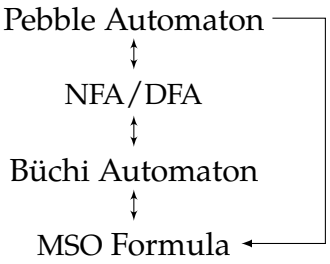


FIGURE 9: Translations from pebble automata to MSO formulas.

Let Σ be a finite alphabet and let $\mathcal{B} = (S, \widehat{\Sigma}, \Delta, S_0, S_f)$ be a pebble automaton over the extended alphabet $\widehat{\Sigma}$, computing the set of trips $\mathrm{Trips}(\mathcal{B})$. The basic idea is to express, in MSO logic, the existence of a run of $\mathcal{B}$ that computes a given trip. We begin by constructing for each pair of states (s, s') of $\mathcal{B}$ an MSO formula $\textsc{UpDownStep}_{s,s'}(x, x', z, \overline{Z})$ expressing that $\mathcal{B}$ has a transition from state s to s' that lets the automaton move (without placing the pebble) from position x to x' on the ω-word encoded by $\overline{Z} = (Z_a)_{a \in \widehat{\Sigma}}$, with the pebble lying on position z:

$$
\begin{aligned}
&\textsc{UpDown}_{s,s'}\left(x, x', z, (Z_a)_{a\in\widehat{\Sigma}}\right) = \\
&\quad \bigvee_{(s,a,0,s',\uparrow)\in\Delta} \left(Z_a(x) \wedge x \neq z \wedge x' = x + 1\right) \vee \\
&\quad \bigvee_{(s,a,1,s',\uparrow)\in\Delta} \left(Z_a(x) \wedge x = z \wedge x' = x + 1\right) \vee \\
&\quad \bigvee_{(s,a,0,s',\downarrow)\in\Delta} \left(Z_a(x) \wedge x \neq z \wedge x' = x - 1\right) \vee \\
&\quad \bigvee_{(s,a,1,s',\downarrow)\in\Delta} \left(Z_a(x) \wedge x = z \wedge x' = x - 1\right).
\end{aligned}
$$

Now we define the reflexive transitive closure of the "step up/down relation" represented by the formulas $\textsc{UpDown}_{s,s'}$. Intuitively, the re-

sulting formula expresses the existence of a run in which the pebble is fixed, lying on position z:

$$\textsc{FixedPebbleRun}_{s,s'}(x, x', z, \overline{Z}) =$$
$$\forall (X_r)_{r\in S} \left(\Big(X_s(x) \wedge \textsc{FixedClosed}\big((X_r)_{r\in S}, z, \overline{Z}\big) \Big) \rightarrow X_{s'}(x') \right)$$

with

$$\textsc{FixedClosed}\big((X_r)_{r\in S}, z, \overline{Z}\big) =$$
$$\bigwedge_{s,s'\in S} \forall x, x' \left(\Big(X_s(x) \wedge \textsc{UpDown}_{s,s'}\big(x, x', z, \overline{Z}\big) \Big) \rightarrow X_{s'}(x') \right).$$

Using $\textsc{FixedPebbleRun}_{s,s'}$, we can construct a formula that represents a sequence of transitions of $\mathcal{B}$ from one pebble placement to the next – that is, a sequence of transitions without a pebble placement, leading from position x to x', with the pebble lying on position x, followed by a placement of the pebble at position x':

$$\textsc{PebbleMove}_{s,s'}(x, x', \overline{Z}) = \bigvee_{r\in S} \big(\textsc{FixedPebbleRun}_{s,r}(x, x', x, \overline{Z}) \wedge \textsc{PutPebble}_{r,s'}(x', x, \overline{Z}) \big)$$

with

$$\textsc{PutPebble}_{r,s'}\big(x, z, (Z_a)_{a\in\widehat{\Sigma}}\big) = \bigvee_{(r,a,0,s',\diamond)\in\Delta} \big(Z_a(x) \wedge x \neq z\big) \quad \vee \bigvee_{(r,a,1,s',\diamond)\in\Delta} \big(Z_a(x) \wedge x = z\big).$$

By defining the reflexive transitive closure of the "pebble movement relation" represented by the formulas $\textsc{PebbleMove}_{s,s'}$, we obtain the following formula, which expresses the existence of a run of $\mathcal{B}$ starting with the pebble on the start position x and ending with the pebble on the final position x', with the last transition being a pebble placement transition. Such a run may involve multiple intermediate placements of the pebble along the way.

$$\textsc{PebbleMoveClosure}_{s,s'}(x, x', \overline{Z}) =$$
$$\forall (X_r)_{r\in S} \left(\Big(X_s(x) \wedge \textsc{MoveClosed}\big((X_r)_{r\in S}, \overline{Z}\big) \Big) \rightarrow X_{s'}(x') \right)$$

with

$$\textsc{MoveClosed}\big((X_r)_{r\in S}, \overline{Z}\big) =$$
$$\bigwedge_{s,s'\in S} \forall x, x' \left(\big(X_s(x) \wedge \textsc{PebbleMove}_{s,s'}(x, x', \overline{Z}) \big) \rightarrow X_{s'}(x') \right).$$

While a run that computes a trip must end with the pebble on the final position of the automaton, the last placement of the pebble does not have to occur at the very end of the run; instead, the automaton may take some additional steps after the last placement of the pebble and finally return to the pebble position. Therefore, we construct the following formula, which states, more generally, that there is a run of $\mathcal{B}$ that starts with the pebble on the start position x and ends with the pebble on the final position x':

$$\textsc{Run}_{s,s'}(x, x', \overline{Z}) = \bigvee_{r \in S} \big(\textsc{PebbleMoveClosure}_{s,r}(x, x', \overline{Z}) \wedge \textsc{FixedPebbleRun}_{r,s'}(x', x', x', \overline{Z})\big).$$

Finally, we obtain the desired MSO formula inducing the set of trips computed by $\mathcal{B}$ by expressing the existence of a run as defined by $\textsc{Run}_{s,s'}$ from an initial state to a final state. Note that in this formula, the free variables $(Z_a)_{a \in \Sigma}$ encode a finite word over Σ, but $\mathcal{B}$ operates on the corresponding ω-extension over the alphabet $\widehat{\Sigma} = \Sigma \uplus \{\bot, \#\}$, so we have to define explicitly the sets $Z_\bot$ and $Z_\#$ containing the positions of the ω-extension that are labeled with $\bot$ and #, respectively:

$$\varphi\big((Z_a)_{a \in \Sigma}, x, y\big) = \exists Z_\bot, Z_\# \Big(\forall x \, \big(Z_\bot(x) \leftrightarrow x = 0\big) \wedge \forall x \, \big(Z_\#(x) \leftrightarrow \bigwedge_{a \in \Sigma \cup \{\bot\}} \neg Z_a(x)\big) \wedge \bigvee_{\substack{s_0 \in S_0, \\ s_f \in S_f}} \textsc{Run}_{s_0,s_f}\big(x, y, (Z_a)_{a \in \widehat{\Sigma}}\big)\Big).$$

This concludes the direct translation from pebble automata to MSO formulas.

12.3 Definition of ℕ-Memory Automata

We can now give a formal definition of our automaton model for finite words over an infinite alphabet Σ^*. As mentioned in Section 12.1, we call the symbols in such an alphabet *micro words*.

Definition 12.3.1. An *ℕ-memory automaton* $\mathcal{A}$ is a tuple $(Q, \Sigma^*, \Delta, q_0, F)$, where

ℕ-memory automaton

- Q is a finite set of states,
- Σ^* is an infinite alphabet, for a finite set Σ,
- $q_0 \in Q$ is the initial state,
- $\Delta \subseteq (Q \times \mathbb{N}) \times \Sigma^* \times (Q \times \mathbb{N})$ is a pebble-automatic transition relation as defined below,
- $F \subseteq Q$ is the set of final states.

Definition 12.3.2. A transition relation $\Delta \subseteq (Q \times \mathbb{N}) \times \Sigma^* \times (Q \times \mathbb{N})$ is *pebble-automatic* if there exists a pebble automaton $\mathcal{B}$ of the form $(S, \widehat{\Sigma}, \Delta_{\mathcal{B}}, Q, \widetilde{Q})$, where $\widetilde{Q} = \{\widetilde{q} \mid q \in Q\}$ is a distinct copy of Q, that computes Δ in the following sense:

pebble-automatic transition relation

$$\Delta = \bigcup_{p,q \in Q} \left\{ \big((p, i), u, (q, j)\big) \mid (u, i, j) \in \mathrm{Trips}(\mathcal{B}_{p,\widetilde{q}}) \right\},$$

See Definition 12.2.10.

where $\mathcal{B}_{p,\widetilde{q}}$ denotes the pebble automaton $(S, \widehat{\Sigma}, \Delta_{\mathcal{B}}, \{p\}, \{\widetilde{q}\})$.
Moreover, Δ is *deterministically pebble-automatic* if it is computed by a pebble automaton $\mathcal{B}$ that is deterministic from each of its initial states.

A *configuration* of an ℕ-memory automaton $\mathcal{A} = (Q, \Sigma^*, \Delta, q_0, F)$ is a pair (p, i) consisting of a state $p \in Q$ and a memorized position $i \in \mathbb{N}$. A *run* of the ℕ-memory automaton $\mathcal{A}$ on a finite word $w = u_1 \dots u_n$ with $u_\ell \in \Sigma^*$ (for $\ell \in \{1, \dots, n\}$) is a sequence of configurations

$$(q_1, i_1) \dots (q_{n+1}, i_{n+1})$$

with $(q_1, i_1) = (q_0, 0)$ such that for all $\ell \in \{1, \dots, n\}$, there is a transition $\big((q_\ell, i_\ell), u_\ell, (q_{\ell+1}, i_{\ell+1})\big) \in \Delta$. We call such a run *accepting* if it ends with a final state, that is, if $q_{n+1} \in F$. We say that the automaton accepts a given finite word if there is an accepting run on that word.

The ℕ-memory automaton $\mathcal{A}$ and its transition relation Δ are called *deterministic* if for every configuration $(p, i) \in Q \times \mathbb{N}$ and every micro word $u \in \Sigma^*$, there is exactly one configuration $(q, j) \in Q \times \mathbb{N}$ such that $\big((p, i), u, (q, j)\big) \in \Delta$. In that case, we typically write Δ as a *transition function* $\delta \colon (Q \times \mathbb{N}) \times \Sigma^* \to (Q \times \mathbb{N})$.

Definition 12.3.3. Let $\mathcal{A}$ be an ℕ-memory automaton over an alphabet Σ^*. The *language recognized by* $\mathcal{A}$ is the set

$$L(\mathcal{A}) = \{w \in (\Sigma^*)^* \mid \text{there is an accepting run of } \mathcal{A} \text{ on } w\}.$$

A language recognized by a (deterministic) ℕ-memory automaton is called *(deterministically)* ℕ*-memory-recognizable*.

It will often be more convenient to represent the transition relation of an ℕ-memory automaton using MSO logic instead of pebble automata.

MSO-family-definable transition relation

Definition 12.3.4. A transition relation $\Delta \subseteq (Q \times \mathbb{N}) \times \Sigma^* \times (Q \times \mathbb{N})$ is *MSO-family-definable* if it is defined by a family of MSO formulas $\big(\varphi_{p,q}(\overline{Z}, x, y)\big)_{p,q \in Q}$ with $\overline{Z} = (Z_a)_{a \in \Sigma}$, meaning that

See Definition 12.2.7.

$$\Delta = \bigcup_{p,q \in Q} \big\{\big((p, i), u, (q, j)\big) \mid (u, i, j) \in \mathrm{Trips}(\varphi_{p,q})\big\}.$$

Remark 12.3.5. In the special case where the alphabet is ℕ, represented by $\{1\}^*$, a micro word $u \in \{1\}^*$ is uniquely determined by its length. We can then use formulas of the form $\varphi_{p,q}(z, x, y)$ rather than $\varphi_{p,q}(\overline{Z}, x, y)$, with z indicating the length of the micro word.

By the following theorem, we can use the two representations of transition relations, namely pebble automata and MSO formulas, interchangeably.

Theorem 12.3.6. Let $\Delta \subseteq (Q \times \mathbb{N}) \times \Sigma^* \times (Q \times \mathbb{N})$ be a transition relation. The following are effectively equivalent:

(1) The relation Δ is pebble-automatic.
(2) The relation Δ is MSO-family-definable.

If Δ is deterministic, the following is effectively equivalent to (1), (2):

(3) The relation Δ is deterministically pebble-automatic.

Proof. This is essentially a consequence of the fact that pebble automata and MSO formulas can define the same sets of trips (Corollary 12.2.16).

(1) $\Rightarrow$ (2): Let Δ be computed by a pebble automaton $\mathcal{B}$. For $p, q \in Q$, consider the pebble automaton $\mathcal{B}_{p,\widetilde{q}}$ obtained from $\mathcal{B}$ by making p the only initial state and $\widetilde{q}$ the only final state. By Corollary 12.2.16, we can construct an MSO formula $\varphi_{p,q}(\overline{Z}, x, y)$ such that $\mathrm{Trips}(\varphi_{p,q}) = \mathrm{Trips}(\mathcal{B}_{p,\widetilde{q}})$. Hence we obtain a family of MSO formulas $(\varphi_{p,q})_{p,q\in Q}$ inducing the same transition relation Δ as the pebble automaton $\mathcal{B}$.

(2) $\Rightarrow$ (1): Let Δ be defined by a family of MSO formulas $(\varphi_{p,q})_{p,q\in Q}$. By Corollary 12.2.16, we can construct for each $p, q \in Q$ a pebble automaton $\mathcal{B}^{p,q} = (S^{p,q}, \widehat{\Sigma}, \Delta^{p,q}, S_0^{p,q}, S_f^{p,q})$ such that $\mathrm{Trips}(\mathcal{B}^{p,q}) = \mathrm{Trips}(\varphi_{p,q})$. We combine all these pebble automata in a single pebble automaton $\mathcal{B} = (S, \widehat{\Sigma}, \Delta_{\mathcal{B}}, Q, \widetilde{Q})$, where $\widetilde{Q} = \{\widetilde{q} \mid q \in Q\}$: Started in a state $p \in Q$, the automaton $\mathcal{B}$ can simulate $\mathcal{B}^{p,q}$ for any $q \in Q$ (chosen nondeterministically), and upon reaching a final state of $\mathcal{B}^{p,q}$, it switches to the final state $\widetilde{q}$. Formally, we let $S = Q \cup \widetilde{Q} \cup \bigcup_{p,q\in Q} S^{p,q}$ and

$$\begin{aligned}\Delta_{\mathcal{B}} = &\bigcup_{p,q\in Q} \Delta^{p,q} \\ &\cup \bigcup_{p,q\in Q} \left\{(p, a, b, s, d) \mid (s_0, a, b, s, d) \in \Delta^{p,q} \text{ with } s_0 \in S_0^{p,q}\right\} \\ &\cup \bigcup_{p,q\in Q} \left\{(s, a, b, \widetilde{q}, d) \mid (s, a, b, s_f, d) \in \Delta^{p,q} \text{ with } s_f \in S_f^{p,q}\right\}.\end{aligned}$$

This pebble automaton $\mathcal{B}$ induces the same transition relation Δ as the family of formulas $(\varphi_{p,q})_{p,q\in Q}$.

(2) $\Rightarrow$ (3): Now suppose that Δ is a deterministic transition relation defined by a family of MSO formulas $(\varphi_{p,q})_{p,q\in Q}$. For the sake of readability, we abbreviate $\mathrm{Trips}(\varphi_{p,q})$ as $T_{p,q}$. For each $p, q \in Q$, the set of trips $T_{p,q}$ is functional, so we can construct a *deterministic* pebble automaton $\mathcal{B}^{p,q}$ computing $T_{p,q}$, by Corollary 12.2.16. However, if we were to combine all these pebble automata in a pebble automaton $\mathcal{B}$ in the same way as described above, then the choice of the automaton $\mathcal{B}^{p,q}$ to be simulated (from a given starting configuration) would still be nondeterministic. Therefore, we modify the construction of $\mathcal{B}$ as follows.

Assume that the pebble automaton $\mathcal{B}$ is started in state $p \in Q$ at position i (with the pebble on position i as well) on the ω-extension $\widehat{u}$ of a word u. Since the transition relation Δ defined by the formulas $(\varphi_{p,q})_{p,q\in Q}$ is deterministic, there is exactly one state $q \in Q$ and one

number $j \in \mathbb{N}$ such that $(u, i, j) \in T_{p,q}$, so the automaton $\mathcal{B}$ only needs to simulate the corresponding automaton $\mathcal{B}^{p,q}$. We will now show that the automaton $\mathcal{B}$ can determine the correct state q deterministically.

Note that for each $p, q \in Q$, the ω-language

$$\mathrm{mark}(T_{p,q}) = \{\mathrm{mark}(\widehat{u}, i, j) \mid (u, i, j) \in T_{p,q}\}$$

is regular (by Proposition 12.2.8), and hence also the ω-language

$$L_{p,q}^{\exists j} := \{\mathrm{mark}(\widehat{u}, i) \mid \text{there is a } j \in \mathbb{N} \text{ such that } (u, i, j) \in T_{p,q}\}$$

is regular. Thus, we can construct for each $p, q \in Q$ a DFA $\mathcal{D}_{p,q}$ recognizing the shortest relevant prefixes $\mathrm{relPref}(L_{p,q}^{\exists j})$ (see Proposition 12.2.14).

When started in state $p \in Q$, with the pebble on the start position i, the pebble automaton $\mathcal{B}$ simulates for each $q \in Q$ the DFA $\mathcal{D}_{p,q}$ on the prefix of $\widehat{u}$ up to position $\max\{|u|, i\}$, treating the symbol $a \in \widehat{\Sigma}$ at the pebble position i as $(a, 1)$. If the current DFA $\mathcal{D}_{p,q}$ ends up in a final state, then $\mathcal{B}$ has found the correct state q, so it walks back to the pebble position i and simulates $\mathcal{B}^{p,q}$. Upon reaching a final state of $\mathcal{B}^{p,q}$, it switches to the final state $\widetilde{q}$.

The pebble automaton $\mathcal{B}$ constructed in this way computes Δ and is deterministic from each of its initial states $p \in Q$. □

12.4 Remarks on Expressive Power

In this section, we discuss the expressive power of ℕ-memory automata by giving examples and by drawing comparisons to related models.

12.4.1 *Examples*

We start with some simple examples of languages over the alphabet ℕ, which is encoded as $\{1\}^*$ in our framework (see Remark 12.1.3).

Example 12.4.1. The following languages are ℕ-memory-recognizable:

- $L_1 = \{1\ 2\ 3\ \ldots\ k \mid k \in \mathbb{N}_{\geq 1}\}$
- $L_2 = \{1\ w_1\ 2\ w_2\ 3\ w_3\ 4 \ldots w_{k-1}\ k \mid k \in \mathbb{N}_{\geq 1},\ w_i \in \mathbb{N}^*\}$
- $L_3 = \{w_1\ n\ w_2\ n\ w_3 \mid n \in \mathbb{N} \text{ is even}, w_1, w_2, w_3 \in \mathbb{N}^*\}$

The language L_1 from Example 12.4.1 is recognized by a deterministic ℕ-memory automaton that memorizes in each step the number that it

has just read and then compares it to the number encountered in the next step. It will stay in a final state as long as the numbers increase exactly by 1 in each step, and enter a non-final sink state otherwise.

The language L_2 is recognized in a similar way. In this case, the memorized position is only updated when its successor is encountered, and only at these points will the automaton enter a final state.

Moreover, the language L_3 can be recognized by a nondeterministic $\mathbb{N}$-memory automaton that memorizes the number at some nondeterministically chosen position of the input word and verifies that it is even. If the memorized number is encountered again at a later point, the automaton enters a final state and stays there until the end.

12.4.2 Comparison With Other Models

It is instructive to compare $\mathbb{N}$-memory automata with closely related models considered in recent literature. In particular, we mention here the $\mathcal{N}$-MSO-automata defined in [STW11; Bès08], the so-called strong automata and the progressive grid automata of [CST15], and the ordered data automata of [Tan12; Tan14]. An $\mathcal{N}$-MSO-automaton only has a finite state space. Transitions from a state p to a state q are specified by an MSO formula $\varphi_{p,q}(z)$ over the structure $\mathcal{N}$, which defines the input numbers that can lead from p to q. This model allows information flow from one input number to the next only by means of its states and thus cannot recognize a simple language such as $\{mm \mid m \in \mathbb{N}\}$.

In a strong automaton the transitions are specified in a more general way, by formulas $\varphi_{p,q}(z_1, z_2)$ imposing a condition on the current input number z_2 in relation to the previous input number z_1. This amounts to a restricted $\mathbb{N}$-memory automaton that always memorizes the current input number and compares it to the next one. The language L_1 from Example 12.4.1 is recognizable in this way, but L_2 and L_3 are not.

A progressive grid automaton can be regarded as an $\mathbb{N}$-memory automaton whose transition relation is not defined by a two-way pebble automaton but by a two-way automaton without a pebble, which weakens the model. For example, the $\mathbb{N}$-memory-recognizable language $\{m0m \mid m \in \mathbb{N}\}$ cannot be recognized by such an automaton (as shown formally in [CST15]), since the automaton will "forget" the memorized number when it verifies that the second symbol is 0.

Furthermore, we note that $\mathbb{N}$-memory automata are incomparable in expressive power to the ordered data automata of [Tan12; Tan14]. The language L_1 from Example 12.4.1 is recognized by the former but not by the latter, and the language $\{mnmn \mid m, n \in \mathbb{N}\}$ provides the converse.

12.4.3 Restriction to Finite Alphabets

Let us now study the restriction of ℕ-memory automata to finite alphabets. Formally, we consider ℕ-memory automata that only accept input words consisting of "single-symbol" micro words (i.e., micro words of length 1). Our automata are then expressively equivalent to the well-known model of one-counter automata.

one-counter automaton

Definition 12.4.2. A *one-counter automaton* $\mathcal{C}$ is a tuple $(Q, \Sigma, \Delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\Delta \subseteq Q \times \big(\Sigma \cup \{\varepsilon\}\big) \times \{0,1\} \times \{-1,0,1\} \times Q$ is a set of *rules*,
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of final states.

A configuration (p, i) of such a one-counter automaton $\mathcal{C}$ consists of a state $p \in Q$ and a *counter value* $i \in \mathbb{N}$. A transition of $\mathcal{C}$ is a tuple $\big((p, i), a, (q, j)\big)$ such that there exists a rule $(p, a, k, \ell, q) \in \Delta$ with $k = 0$ if and only if $i = 0$, and with $j = i + \ell$. The automaton $\mathcal{C}$ accepts an input word $w \in \Sigma^*$ if there is a sequence of transitions via w leading from the configuration $(q_0, 0)$ to some configuration (q, i) with $q \in F$. The language recognized by $\mathcal{C}$ is the set of words that are accepted by $\mathcal{C}$.

Theorem 12.4.3. Let $L \subseteq \Sigma^*$ be a language over a finite alphabet Σ. The following are equivalent:

(1) L is recognized by an ℕ-memory automaton.
(2) L is recognized by a one-counter automaton.

Analogously, the following are equivalent:

(1) L is recognized by a deterministic ℕ-memory automaton.
(2) L is recognized by a deterministic one-counter automaton.

Proof. First, we consider the translation from (deterministic) ℕ-memory automata to (deterministic) one-counter automata, and then the reverse direction.

From ℕ-memory to one-counter automata. The basic idea for transforming an ℕ-memory automaton recognizing L into an equivalent

one-counter automaton is to use the counter to represent the current memorized position of the $\mathbb{N}$-memory automaton. First, let us address a simple but incorrect approach: We might try to construct a one-counter automaton that simulates each step of the pebble automaton defining the transition relation of the $\mathbb{N}$-memory automaton, using the counter to keep track of the current position of the pebble automaton. However, this attempt will fail as the one-counter automaton has no means to also remember the position of the pebble.

We therefore choose a different approach: Instead of simulating each step of the pebble automaton, the one-counter automaton will simulate a transition $\big((p,i),a,(q,j)\big)$ of the $\mathbb{N}$-memory automaton by executing an a-transition followed by a sequence of ϵ-transitions (which read no input) that either only increase or only decrease the counter value i until a suitable value j is reached.

Let $\mathcal{A} = (Q, \Sigma^*, \Delta, q_0, F)$ be an $\mathbb{N}$-memory automaton recognizing L, where Δ is defined by a family of MSO formulas $(\varphi_{p,q})_{p,q\in Q}$. Since $\mathcal{A}$ only accepts words consisting of single-symbol micro words, we can ignore all transitions of $\mathcal{A}$ for longer micro words. For each micro word of the form $a \in \Sigma$, the transitions of $\mathcal{A}$ via a can be represented by the following MSO formulas, for $p, q \in Q$:

$$\psi_{p,a,q}(x,y) = \exists (Z_b)_{b\in\Sigma} \Big(Z_a(0) \wedge \forall z \big(z > 0 \to \bigwedge_{b\in\Sigma} \neg Z_b(z)\big) \wedge \varphi_{p,q}\big((Z_b)_{b\in\Sigma}, x, y\big)\Big)$$

Let $N = \{n_{p,a,q} \mid p, q \in Q,\ a \in \Sigma\}$ be the set of pumping moduli, as defined in Lemma 11.5.2, of all these formulas. (For the sake of readability, we write $n_{p,a,q}$ instead of $n_{\psi_{p,a,q}}$.)

We can now construct a one-counter automaton recognizing L in the following way. The one-counter automaton uses its counter to keep track of the current position memorized by the $\mathbb{N}$-memory automaton $\mathcal{A}$ and uses its state to keep track not only of the current state of $\mathcal{A}$ but also of the remainder of the current counter value modulo n, for each $n \in N$. After reading the current input symbol $a \in \Sigma$, the one-counter automaton simulates an applicable transition of $\mathcal{A}$ by a sequence of ε-transitions: Suppose that the current configuration of $\mathcal{A}$ is (p, i). The one-counter automaton remembers the remainders of the current counter value i (for all moduli $n \in N$) and starts decreasing or increasing (chosen nondeterministically) the counter using ε-transitions, keeping track of the remainders of the counter value (for all moduli $n \in N$). Once the counter reaches a value j such that (i, j) is in the relation defined by

the formula $\psi_{p,a,q}$ for some state $q \in Q$, the one-counter automaton can proceed with the simulation of the next step of $\mathcal{A}$, from state q.

By Lemma 11.5.2, whether or not (i, j) is in the relation defined by $\psi_{p,a,q}$ depends solely on the $\sim_{n_{p,a,q}}$-equivalence class of (i, j) and thus can be deduced from whether $i < j$, $i < n_{p,a,q}$, $j < n_{p,a,q}$, and/or $|i - j| < n_{p,a,q}$, and from the remainders of i and j modulo $n_{p,a,q}$.

The only required information that cannot be obtained directly from the state of the one-counter automaton is whether the old counter value i and/or the new counter value j are smaller than $n_{p,a,q}$. However, the automaton can verify at any time whether its counter value is smaller than $n_{p,a,q}$ by decreasing the counter, step by step, by the fixed amount $n_{p,a,q}$ (if possible) to check whether the value 0 is reached in the process, and then increase the counter again by the same amount to restore the original value. This completes the translation from nondeterministic $\mathbb{N}$-memory automata to nondeterministic one-counter automata.

If $\mathcal{A}$ is deterministic, then the decision to increase or decrease the counter value i in order to obtain j can be made deterministically, based on the available information about i, so we obtain a deterministic one-counter automaton.

From one-counter to $\mathbb{N}$-memory automata. Let $\mathcal{C}$ be a one-counter automaton recognizing L. W.l.o.g., we assume that $\mathcal{C}$ has no ε-transitions, i.e., every transition reads an input symbol (see [ABB97]). A transition of $\mathcal{C}$ via a symbol $a \in \Sigma$ has the form $\big((p, i), a, (q, j)\big)$, where p, q are states of $\mathcal{C}$ and $i, j \in \mathbb{N}$ are the values of the counter before and after the transition. We construct an $\mathbb{N}$-memory automaton $\mathcal{A}$ that has exactly the same transitions as the one-counter automaton $\mathcal{C}$ – with the counter values i, j being represented by the memorized positions of $\mathcal{A}$.

The desired transition relation can be defined by a pebble automaton: Starting in state p at position i, which is also marked by the pebble, the pebble automaton walks to the beginning of its input to read the input symbol $a \in \Sigma$ and to detect whether $i = 0$. Then it chooses an applicable rule (p, a, k, ℓ, q) of $\mathcal{C}$ to be simulated, returns to the pebble position i, moves the pebble upward or downward, depending on the counter adjustment $\ell \in \{-1, 0, 1\}$ of the rule, and terminates in the state $\widetilde{q}$.

It remains to be shown that we can construct a *deterministic* $\mathbb{N}$-memory automaton $\mathcal{A}$ if the one-counter automaton $\mathcal{C}$ is deterministic. In that case we cannot assume that $\mathcal{C}$ has no ε-transitions as this would restrict the expressive power of the model. However, we can assume that the states of $\mathcal{C}$ can be classified into states that only admit ε-transitions and states that only admit non-ε-transitions. The latter states are called

reading states, and we may assume that all final states are reading states (see [Sip12]). A run of $\mathcal{C}$ on a word $a_1 \dots a_n \in \Sigma^*$ then has the form

$$(q_0, 0) \xrightarrow{\varepsilon^*} (p_1, i_1) \xrightarrow{a_1} \xrightarrow{\varepsilon^*} (p_2, i_2) \xrightarrow{a_2} \xrightarrow{\varepsilon^*} \dots \xrightarrow{a_n} \xrightarrow{\varepsilon^*} (p_n, i_n),$$

where the states p_ℓ are reading states and $\xrightarrow{\varepsilon^*}$ indicates a (possibly empty) sequence of ε-transitions.

We use the initial state and the reading states of $\mathcal{C}$ as the states of the deterministic $\mathbb{N}$-memory automaton $\mathcal{A}$. The final states of $\mathcal{C}$ are also marked as final states of $\mathcal{A}$ – recall that all of them are reading states. In addition, we also designate the initial state q_0 as a final state of $\mathcal{A}$ if $\mathcal{C}$ accepts ε, which can be effectively determined.

Now instead of simulating a single transition of $\mathcal{C}$, we let the pebble automaton defining the transition relation of $\mathcal{A}$ simulate a sequence of transitions, which can be of the form $(q_0, 0) \xrightarrow{\varepsilon^*} \xrightarrow{a} \xrightarrow{\varepsilon^*} (p, j)$, where q_0 is the initial state and p is a reading state, or $(p, i) \xrightarrow{a} \xrightarrow{\varepsilon^*} (p', j)$, where p and p' are reading states. Since $\mathcal{C}$ is deterministic, this can be done in a deterministic way and hence we obtain a deterministic $\mathbb{N}$-memory automaton $\mathcal{A}$. □

12.5 Closure Properties of $\mathbb{N}$-Memory-Recognizable Languages

In this section, we study the closure properties of the languages recognized by deterministic and nondeterministic $\mathbb{N}$-memory automata.

We mentioned in Section 12.4.3 that in the special case of *finite* alphabets (where we only consider micro words of length 1), these languages coincide with those recognized by deterministic and nondeterministic one-counter automata, respectively. Therefore, some *negative* results (i.e., non-closure properties) presented below can be derived from the corresponding known results for (deterministic) one-counter automata. Nevertheless, in order to provide a better understanding of the capabilities of $\mathbb{N}$-memory automata, we give self-contained proofs without referring to one-counter automata.

Theorem 12.5.1. The class of $\mathbb{N}$-memory-recognizable languages is closed under union.

Proof. The union of the languages of two given $\mathbb{N}$-memory automata can be recognized by an $\mathbb{N}$-memory automaton that chooses nondeterministically which of the two given automata it will simulate.

Formally, let $\mathcal{A}_1 = (Q_1, \Sigma^*, \Delta_1, q_0^1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma^*, \Delta_2, q_0^2, F_2)$ be two $\mathbb{N}$-memory automata, where Δ_1 and Δ_2 are defined by families of MSO formulas $(\varphi^1_{p,q})_{p,q\in Q_1}$ and $(\varphi^2_{p,q})_{p,q\in Q_2}$, respectively. We define $\mathcal{A} = (Q, \Sigma^*, \Delta, q_0, F)$, where $Q = Q_1 \uplus Q_2 \uplus \{q_0\}$, $F = F_1 \uplus F_2$, and Δ is defined by $(\varphi_{p,q})_{p,q\in Q}$ with

$$\varphi_{p,q}(\overline{Z}, x, y)) = \begin{cases} \varphi^1_{q_0^1,q}(\overline{Z}, x, y)) & \text{if } p = q_0 \text{ and } q \in Q_1, \\ \varphi^2_{q_0^2,q}(\overline{Z}, x, y)) & \text{if } p = q_0 \text{ and } q \in Q_2, \\ \varphi^1_{p,q}(\overline{Z}, x, y)) & \text{if } p, q \in Q_1, \\ \varphi^2_{p,q}(\overline{Z}, x, y)) & \text{if } p, q \in Q_2, \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Then $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$. □

Theorem 12.5.2. The class of deterministically $\mathbb{N}$-memory-recognizable languages is *not* closed under union. This holds even for the alphabet $\mathbb{N}$.

Proof. Consider the two languages $L_1 = \{m_1 m_2 m_1 \mid m_1, m_2 \in \mathbb{N}\}$ and $L_2 = \{m_1 m_2 m_2 \mid m_1, m_2 \in \mathbb{N}\}$, which are both deterministically $\mathbb{N}$-memory-recognizable. Assume toward a contradiction that the language $L = L_1 \cup L_2$ is recognized by a deterministic $\mathbb{N}$-memory automaton $\mathcal{A} = (Q, \Sigma^*, \delta, q_0, F)$, where δ is defined by some family of MSO formulas $(\varphi_{p,q})_{p,q\in Q}$. Intuitively, for an input word $m_1 m_2 m_3$, this automaton would have to remember both m_1 and m_2 to verify that $m_3 = m_1$ or $m_3 = m_2$, which is not possible since it can only memorize one natural number.

Let us now formalize this intuition using a pumping argument. We define for each $p \in Q$ the following relation $R_p \subseteq \mathbb{N} \times \mathbb{N}$:

$$R_p = \big\{(i, n) \mid \text{from the configuration } (p, i), \\ \mathcal{A} \text{ accepts the word consisting} \\ \text{of the single symbol } n\big\}$$

This relation can be defined (in the structure $\mathcal{N}$) by a corresponding MSO formula ψ_p, using the formulas $\big(\varphi_{p,q}(z, x, y)\big)_{p,q\in Q}$ that define the transition function δ of $\mathcal{A}$:

$$\psi_p(x, z) = \exists y \left(\bigvee_{q\in F} \big(\varphi_{p,q}(z, x, y)\big) \right)$$

Let $n = \prod_{p \in Q} n_{\psi_p}$ be the product of the pumping moduli of these formulas as defined in Lemma 11.5.2. Now consider the unique run of $\mathcal{A}$ on the word w consisting of the two numbers $2n$ and $4n$, that is, $w = (2n)(4n)$. Let (p, i) be the configuration that is reached after reading w. The only words that can lead from that configuration (p, i) to a final state are $(2n)$ and $(4n)$, so we have $R_p \cap (\{i\} \times \mathbb{N}) = \{(i, 2n), (i, 4n)\}$.

We distinguish the following two cases for the memorized position i. If $i \geq 3n$ then $(i, 2n) \sim_{n_{\psi_p}} (i, n)$, by Definition 11.5.1. Thus we have $(i, n) \in R_p$ by Lemma 11.5.2, which is a contradiction. On the other hand, if $i < 3n$ then $(i, 4n) \sim_{n_{\psi_p}} (i, 5n)$, so Lemma 11.5.2 implies that $(i, 5n) \in R_p$ – again a contradiction. Thus, there is no deterministic $\mathbb{N}$-memory automaton recognizing $L = L_1 \cup L_2$. □

Corollary 12.5.3. The class of $\mathbb{N}$-memory-recognizable languages *strictly* includes the class of deterministically $\mathbb{N}$-memory-recognizable languages. In particular, the language

$$L = \{m_1 m_2 m_3 \mid m_1, m_2, m_3 \in \mathbb{N} \text{ with } m_1 = m_3 \text{ or } m_2 = m_3\}$$

from the proof of Theorem 12.5.2 is recognized by an $\mathbb{N}$-memory automaton, but not by a deterministic one.

Theorem 12.5.4. Neither the class of $\mathbb{N}$-memory-recognizable languages nor the class of deterministically $\mathbb{N}$-memory-recognizable languages is closed under intersection. This holds even for the alphabet $\mathbb{N}$.

Proof. Consider the two languages $L_1 = \{m_1 m_2 m_3 m_4 \in \mathbb{N}^* \mid m_1 = m_3\}$ and $L_2 = \{m_1 m_2 m_3 m_4 \in \mathbb{N}^* \mid m_2 = m_4\}$, both of which are deterministically $\mathbb{N}$-memory-recognizable. Assume toward a contradiction that their intersection $L = L_1 \cap L_2$ is recognized by an $\mathbb{N}$-memory automaton $\mathcal{A} = (Q, \Sigma^*, \Delta, q_0, F)$. Intuitively, for an input word $m_1 m_2 m_3 m_4$, this automaton would have to remember both m_1 and m_2 to verify that $m_1 = m_3$ and $m_2 = m_4$, which is not possible since it can only memorize one natural number.

For a formal proof, we apply a pumping argument, similar to the proof of Theorem 12.5.2. We define for each $p \in Q$ the following relation $R_p \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N}$:

$$R_p = \{(i, n_1, n_2) \mid \text{from the configuration } (p, i), \\ \mathcal{A} \text{ accepts the word } n_1 n_2\}$$

This relation can be defined (in the structure $\mathcal{N}$) by a corresponding MSO formula ψ_p, using the formulas $(\varphi_{p,q}(z,x,y))_{p,q\in Q}$ that define the transition relation Δ of $\mathcal{A}$:

$$\psi_p(x,z_1,z_2) = \exists y_1,y_2 \left(\bigvee_{\substack{q_1\in Q,\\ q_2\in F}} \left(\varphi_{p,q_1}(z_1,x_1,y_1) \wedge \varphi_{q_1,q_2}(z_2,y_1,y_2)\right)\right)$$

Based on R_p, we define for each $p \in Q$ the relations

$$R_p^1 = \left\{(i,n_1) \mid (i,n_1,n_2) \in R_p \text{ for some } n_2 \in \mathbb{N}\right\} \quad \text{and}$$
$$R_p^2 = \left\{(i,n_2) \mid (i,n_1,n_2) \in R_p \text{ for some } n_1 \in \mathbb{N}\right\},$$

which can be defined by the MSO formulas

$$\psi_p^1(x,z_1) = \exists z_2\ \psi_p(x,z_1,z_2) \quad \text{and}$$
$$\psi_p^2(x,z_2) = \exists z_1\ \psi_p(x,z_1,z_2),$$

respectively.

Let $n = \prod_{p\in Q} n_{\psi_p^1} \cdot n_{\psi_p^2}$ be the product of the pumping moduli of the latter formulas as defined in Lemma 11.5.2. Now consider the word $w = (2n)(4n)(2n)(4n)$. Since $w \in L$, there must be an accepting run of $\mathcal{A}$ on w. Let (p,i) be the configuration that is reached on that run after reading the first two symbols, i.e., after reading $(2n)(4n)$. The only word that can lead from that configuration (p,i) to a final state is again $(2n)(4n)$, so we have $R_p^1 \cap \big(\{i\} \times \mathbb{N}\big) = \{(i,2n)\}$ and $R_p^2 \cap \big(\{i\} \times \mathbb{N}\big) = \{(i,4n)\}$.

We distinguish two cases for the memorized position i. If $i \geq 3n$ then, by Definition 11.5.1,

$$(i,2n) \sim_{n_{\psi_p^1}} (i,n).$$

Thus we have $(i,n) \in R_p^1$ by Lemma 11.5.2, which is a contradiction. On the other hand, if $i < 3n$ then

$$(i,4n) \sim_{n_{\psi_p^2}} (i,5n),$$

so Lemma 11.5.2 implies that $(i,5n) \in R_p^2$ – again a contradiction. Thus, there is no $\mathbb{N}$-memory automaton recognizing $L = L_1 \cap L_2$. □

THEOREM 12.5.5. The class of $\mathbb{N}$-memory-recognizable languages is *not* closed under complement. This holds even for the alphabet $\mathbb{N}$.

Proof. If the class of $\mathbb{N}$-memory-recognizable languages was closed under complement, then closure under union (Theorem 12.5.1) together with De Morgan's laws would imply closure under intersection, contradicting Theorem 12.5.4. □

THEOREM 12.5.6. The class of deterministically $\mathbb{N}$-memory-recognizable languages is closed under complement.

Proof. Let $\mathcal{A} = (Q, \Sigma^*, \delta, q_0, F)$ be a deterministic $\mathbb{N}$-memory automaton. Then $\mathcal{A}' = (Q, \Sigma^*, \delta, q_0, Q \setminus F)$ recognizes $(\Sigma^*)^* \setminus L(\mathcal{A})$. □

THEOREM 12.5.7. The class of $\mathbb{N}$-memory-recognizable languages is closed under concatenation and iteration.

Proof. Given two $\mathbb{N}$-memory automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we can construct an $\mathbb{N}$-memory automaton $\mathcal{A}$ that recognizes $L(\mathcal{A}_1) \cdot L(\mathcal{A}_2)$ in the following way, just as in the case of NFAs: The automaton $\mathcal{A}$ first simulates $\mathcal{A}_1$. Whenever $\mathcal{A}_1$ enters a final state, $\mathcal{A}$ can choose nondeterministically either to continue the simulation of $\mathcal{A}_1$ or to start simulating $\mathcal{A}_2$ (treating the memorized position as 0 at that point). If the simulation of $\mathcal{A}_2$ reaches a final state at the end of the input word, then $\mathcal{A}$ accepts.

Analogously, we obtain an $\mathbb{N}$-memory automaton $\mathcal{A}$ recognizing $L(\mathcal{A}_1)^*$ by "restarting" the simulation of $\mathcal{A}_1$ over and over again, and by accepting if a final state of $\mathcal{A}_1$ is reached at the end. □

THEOREM 12.5.8. The class of deterministically $\mathbb{N}$-memory-recognizable languages is neither closed under concatenation nor under iteration. This holds even for the alphabet $\mathbb{N}$.

Proof. Consider $L = \{mm \mid m \in \mathbb{N}\} \cup \{mm'm \mid m, m' \in \mathbb{N}\}$, which is a deterministically $\mathbb{N}$-memory-recognizable language. Assume toward a contradiction that $L \cdot L$ is recognized by a deterministic $\mathbb{N}$-memory automaton $\mathcal{A}$. Note that a word of the form $m_1 m_1 m_1 m_2 m$, with $m_1, m_2, m \in \mathbb{N}$, is in $L \cdot L$ if and only if $m = m_1$ or $m = m_2$. Intuitively, the automaton $\mathcal{A}$ would have to remember both m_1 and m_2 to verify this, which is not possible since it can only memorize one natural number. More precisely, we obtain a contradiction by choosing m_1 and m_2 large

enough to allow for a pumping argument using Lemma 11.5.2, in the same way as in the proof of Theorem 12.5.2.

By the same argument, the iteration L^* of the language L is not deterministically ℕ-memory-recognizable. □

The closure properties of the languages recognized by (deterministic) ℕ-memory automata are summarized in Figure 10.

	det.	nondet.
union ($\cup$)	no	yes
intersection ($\cap$)	no	no
complement ($\neg$)	yes	no
concatenation ($\cdot$)	no	yes
iteration (*)	no	yes

Figure 10: Closure properties of ℕ-memory-recognizable languages.

	det.	nondet.
nonemptiness	yes	yes
membership	yes	yes
universality	yes	no
inclusion	no	no
empty intersection	no	no
equivalence	?	no
deterministic	—	no

Figure 11: Decidability results for ℕ-memory automata.

12.6 Decision Problems for ℕ-Memory Automata

In this section, we discuss the decidability of some decision problems for ℕ-memory automata. The results are summarized in Figure 11.

The undecidability results in this section can be obtained as a consequence of the corresponding undecidability results for (deterministic) one-counter automata, since the languages of finite words that are recognized by (deterministic) ℕ-memory automata restricted to a *finite* alphabet are precisely those recognized by (deterministic) one-counter automata (see Section 12.4.3). However, in order to provide a mostly

self-contained account, we prove these results without referring to one-counter automata.

THEOREM 12.6.1. The nonemptiness problem for $\mathbb{N}$-memory automata ("given $\mathcal{A}$: $L(\mathcal{A}) \neq \emptyset$?") is decidable.

Proof. Let $\mathcal{A} = (Q, \Sigma^*, \Delta, q_0, F)$ be the given $\mathbb{N}$-memory automaton, where Δ is defined by a family of MSO formulas $\big(\varphi_{p,q}(\overline{Z}, x, y)\big)_{p,q \in Q}$. We construct an MSO sentence φ such that $\mathcal{N} \models \varphi$ if and only if $L(\mathcal{A}) \neq \emptyset$. This property can then be effectively verified since the MSO theory of $\mathcal{N}$ is decidable (see [Büc66]).

We begin by defining a formula $\textsc{Run}_{p,q}(x, y)$ asserting the existence of a run of $\mathcal{A}$ (on some arbitrary input word) from the configuration consisting of the state p and the memorized position x to the configuration consisting of the state q and the memorized position y:

$$\textsc{Run}_{p,q}(x, y) = \forall (X_r)_{r \in Q} \Big(\big(X_p(x) \wedge \textsc{Closed}\big((X_r)_{r \in Q}\big) \big) \rightarrow X_q(y) \Big)$$

where

$$\textsc{Closed}\big((X_r)_{r \in Q}\big) = \bigwedge_{p,q \in Q} \forall x, y \left(\begin{bmatrix} X_p(x) \wedge \\ \exists \overline{Z}\ \varphi_{p,q}(\overline{Z}, x, y) \end{bmatrix} \rightarrow X_q(y) \right).$$

Now we obtain the desired MSO sentence φ as follows:

$$\varphi = \exists y \bigvee_{q \in F} \textsc{Run}_{q_0,q}(0, y) \qquad \square$$

THEOREM 12.6.2. The membership problem for $\mathbb{N}$-memory automata ("given $w \in (\Sigma^*)^*$ and $\mathcal{A}$: $w \in L(\mathcal{A})$?") is decidable.

Proof. We can construct an $\mathbb{N}$-memory automaton $\mathcal{A}'$ that behaves like the given automaton $\mathcal{A}$ but additionally checks whether the input matches the given word $w = u_1 \dots u_n \in (\Sigma^*)^*$ – and does not accept otherwise. Testing the automaton $\mathcal{A}'$ for nonemptiness then yields the answer to the question whether $w \in L(\mathcal{A})$. $\square$

In the case of deterministic $\mathbb{N}$-memory automata, the decidability of the nonemptiness problem (Theorem 12.6.1) implies the decidability of the universality problem, due to the fact that deterministic $\mathbb{N}$-memory automata can be effectively complemented (see Theorem 12.5.6):

Corollary 12.6.3. The universality problem for deterministic ℕ-memory automata ("given $\mathcal{A}$: $L(\mathcal{A}) = (\Sigma^*)^*$?") is decidable.

This, however, does not extend to nondeterministic ℕ-memory automata as we will see later. But first, let us address the empty intersection problem.

Theorem 12.6.4. The empty intersection problem for deterministic ℕ-memory automata ("given $\mathcal{A}_1$ and $\mathcal{A}_2$: $L(\mathcal{A}_1) \cap L(\mathcal{A}_2) = \emptyset$?") is undecidable. This holds even for the alphabet ℕ, and even if the intersection of the languages of the two automata is guaranteed to either be empty or consist of a single word.

To prove this theorem, we will use a reduction from the halting problem for 2-register machines.

2-register machine

Definition 12.6.5. A *2-register machine* $\mathcal{R}$ with registers x_0, x_1 is a finite sequence of instructions

$$\text{instr}_1;\ \text{instr}_2;\ \text{instr}_3;\ \ldots\ \text{instr}_k$$

where $\text{instr}_k = [\textsc{Halt}]$ and for each $\ell \in \{1, \ldots, k-1\}$, the instruction instr_ℓ is of one of the following types:

- $[\textsc{Inc}\ x_i]$ ("increment")
- $[\textsc{Dec}\ x_i]$ ("decrement")
- $[\textsc{Goto}\ m]$
- $[\textsc{If}\ x_i = 0\ \textsc{Then Goto}\ m]$

where $i \in \{0, 1\}$ and $m \in \{1, \ldots, k\}$.

A *configuration* of a 2-register machine $\mathcal{R}$ as defined above is a triple (ℓ, r_0, r_1) consisting of the current *instruction index* $\ell \in \{1, \ldots, k\}$, the current value $r_0 \in \mathbb{N}$ of register x_0 and the current value $r_1 \in \mathbb{N}$ of register x_1. To formalize the semantics of the 2-register machine $\mathcal{R}$, we define the *computation step relation* $\rightarrow_{\mathcal{R}} \subseteq \big(\{1, \ldots, k\} \times \mathbb{N} \times \mathbb{N}\big)^2$, where

$$(\ell, r_0, r_1) \rightarrow_{\mathcal{R}} (\ell', r_0', r_1')$$

if and only if one of the following holds:

- $\text{instr}_\ell = [\textsc{Inc}\ x_i]$ and $r_i' = r_i + 1,\ r_{1-i}' = r_{1-i},$ $\ell' = \ell + 1,$
- $\text{instr}_\ell = [\textsc{Dec}\ x_i]$ and $r_i' = \max(r_i - 1, 0),\ r_{1-i}' = r_{1-i},$ $\ell' = \ell + 1,$
- $\text{instr}_\ell = [\textsc{Goto}\ \ell']$ and $r_0' = r_0,\ r_1' = r_1,$ $\ell' = m,$
- $\text{instr}_\ell = [\textsc{If}\ x_i = 0\ \textsc{Then Goto}\ m]$ and $r_0' = r_0,\ r_1' = r_1,$ $\ell' = \begin{cases} m & \text{if } r_i = 0, \\ \ell + 1 & \text{otherwise.} \end{cases}$

Note that the relation $\to_{\mathcal{R}}$ is in fact functional.

The *computation* of $\mathcal{R}$ is the longest (possibly infinite) configuration sequence of the form

$$(\ell^1, r_0^1, r_1^1) \to_{\mathcal{R}} (\ell^2, r_0^2, r_1^2) \to_{\mathcal{R}} (\ell^3, r_0^3, r_1^3) \to_{\mathcal{R}} \dots$$

such that (ℓ^1, r_0^1, r_1^1) is the initial configuration $(1, 0, 0)$. This sequence is uniquely determined.

halting problem for 2-register machines

The *halting problem for 2-register machines* is to determine for a given 2-register machine $\mathcal{R}$ whether its computation is finite, i.e., whether $\mathcal{R}$ will eventually reach the $[\textsc{Halt}]$ instruction. This problem is well known to be undecidable (see [Min67]).

Proof of Theorem 12.6.4. We reduce the halting problem for 2-register machines to the empty intersection problem for deterministic $\mathbb{N}$-memory automata: For a given 2-register machine $\mathcal{R}$, we construct two deterministic $\mathbb{N}$-memory automata $\mathcal{A}_0$ and $\mathcal{A}_1$ over the alphabet $\mathbb{N}$ such that $L(\mathcal{A}_0) \cap L(\mathcal{A}_1) = \emptyset$ if and only if $\mathcal{R}$ reaches the $[\textsc{Halt}]$ instruction. (In fact, our construction will guarantee that $L(\mathcal{A}_0) \cap L(\mathcal{A}_1)$ is either empty or contains only a single word.) Since the halting problem is undecidable, so is the empty-intersection problem. This reduction, detailed below, is analogous to the well-known reduction to the empty intersection problem for deterministic one-counter automata (see [Val73]).

The computation $(\ell^1, r_0^1, r_1^1) \to_{\mathcal{R}} (\ell^2, r_0^2, r_1^2) \to_{\mathcal{R}} (\ell^3, r_0^3, r_1^3) \to_{\mathcal{R}} \dots$ of the machine $\mathcal{R}$ can be represented by the (possibly infinite) word

$$w_{\mathcal{R}} = n_0^1\, n_1^1\, n_0^2\, n_1^2\, n_0^3\, n_1^3 \dots$$

over the alphabet $\{0,1\} \subset \mathbb{N}$, where, for $i \in \{0,1\}$ and $j \geq 0$,

$$n_i^j = \begin{cases} 0 & \text{if } r_i^j = 0, \\ 1 & \text{if } r_i^j > 0. \end{cases}$$

We shall construct deterministic $\mathbb{N}$-memory automata $\mathcal{A}_0$, $\mathcal{A}_1$ such that

$$L(\mathcal{A}_0) \cap L(\mathcal{A}_1) = \begin{cases} \{w_{\mathcal{R}}\} & \text{if } \mathcal{R} \text{ eventually halts,} \\ \emptyset & \text{otherwise.} \end{cases}$$

Roughly speaking, given a finite word $m_0^1\, m_1^1\, m_0^2\, m_1^2\, m_0^3\, m_1^3 \ldots$, the automaton $\mathcal{A}_0$ simulates the evolution of the values of the register x_0 to check the "consistency" of $m_0^1, m_0^2, m_0^3, \ldots$, and $\mathcal{A}_1$ does the same for $m_1^1, m_1^2, m_1^3, \ldots$ with regard to the register x_1.

To prepare for a more precise description, consider a finite word of the form $w = m_0^1\, m_1^1\, m_0^2\, m_1^2 \ldots m_0^n\, m_1^n \in \{0,1\}^*$. This word induces a sequence of instruction indices $\ell^1, \ldots, \ell^n$: The first index is $\ell^1 = 1$, and for $j > 1$, we define ℓ^j as the uniquely determined index in $\{1, \ldots, k\}$ such that there exist $r_0, r_1 \in \mathbb{N}$ with $(\ell^{j-1}, m_0^{j-1}, m_1^{j-1}) \to_{\mathcal{R}} (\ell^j, r_0, r_1)$, provided that $\ell^{j-1} \in \{1, \ldots, k-1\}$ (otherwise we let $\ell^j = \perp$).

Intuitively, $\ell^1, \ldots, \ell^n$ is the sequence of instruction indices that arises if we ignore the actual register values produced by $\mathcal{R}$ and instead use the values m_0^j and m_1^j for the jth step of the computation. The values $m_0^j, m_1^j \in \{0,1\}$ can be regarded as representatives for arbitrary register values, since the next instruction does not depend on the specific values of the registers but only on a comparison with 0.

While reading the word w, the automaton $\mathcal{A}_0$ determines the instruction indices $\ell^1, \ldots, \ell^n$. Note that ℓ^j, for $j > 1$, only depends on ℓ^{j-1} and on the values $m_0^{j-1}, m_1^{j-1} \in \{0,1\}$. The automaton uses its memory to represent the value of the register x_0, and it simulates the effects of the instructions $\mathrm{instr}_{\ell^1}, \ldots, \mathrm{instr}_{\ell^{n-1}}$ on that value. For each $j \in \{1, \ldots, n\}$, the automaton verifies that m_0^j is 0 if and only if the register value obtained after the first $j-1$ instructions is 0.

The second automaton $\mathcal{A}_1$ is constructed analogously, it verifies the consistency of $m_1^1, \ldots, m_1^n$. In addition, $\mathcal{A}_1$ checks that the last instruction index ℓ^n is k (which means that the [HALT] instruction is reached), and that the length of the given input word w is even (which is a necessary technical condition for $w = w_{\mathcal{R}}$).

If $\mathcal{R}$ reaches the [HALT] instruction, then $w_{\mathcal{R}}$ is the one and only word of the form $m_0^1\, m_1^1\, m_0^2\, m_1^2 \ldots m_0^n\, m_1^n$ such that both $m_0^1, \ldots, m_0^n$ and $m_1^1, \ldots, m_1^n$ are consistent with the sequence of instructions induced by

the word and such that the last instruction is [HALT]. If $\mathcal{R}$ never halts, then there is no such word at all. Thus, we have $L(\mathcal{A}_0) \cap L(\mathcal{A}_1) = \emptyset$ if and only if $\mathcal{R}$ reaches the [HALT] instruction. □

As a first consequence of Theorem 12.6.4, we obtain the undecidability of the inclusion problem even for deterministic $\mathbb{N}$-memory automata.

THEOREM 12.6.6. The inclusion problem for deterministic $\mathbb{N}$-memory automata ("given $\mathcal{A}_1$ and $\mathcal{A}_2$: $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$?") is undecidable. This holds even for the alphabet $\mathbb{N}$.

Proof. We reduce the empty intersection problem for deterministic $\mathbb{N}$-memory automata to the inclusion problem. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the given deterministic $\mathbb{N}$-memory automata. By Theorem 12.5.6, we can construct a deterministic $\mathbb{N}$-memory automaton $\mathcal{A}_2'$ recognizing the complement of $L(\mathcal{A}_2)$. Then we have $L(\mathcal{A}_1) \cap L(\mathcal{A}_2) = \emptyset$ if and only if $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2')$. □

Theorem 12.6.4 also implies that the universality problem is undecidable for nondeterministic $\mathbb{N}$-memory automata, unlike for deterministic ones, as mentioned earlier:

THEOREM 12.6.7. The universality problem for $\mathbb{N}$-memory automata ("given $\mathcal{A}$: $L(\mathcal{A}) = (\Sigma^*)^*$?") is undecidable. This holds even for the alphabet $\mathbb{N}$, and even if the language of the automaton is guaranteed to be either $\mathbb{N}^*$ or of the form $\mathbb{N}^* \setminus \{w\}$ with $w \in \mathbb{N}^*$.

Proof. We can reduce the empty intersection problem for deterministic $\mathbb{N}$-memory automata to the universality problem for nondeterministic $\mathbb{N}$-memory automata as follows: Given two deterministic $\mathbb{N}$-memory automata $\mathcal{A}_1$ and $\mathcal{A}_2$ over the alphabet $\mathbb{N}$, we can construct (deterministic) automata $\mathcal{A}_1'$ and $\mathcal{A}_2'$ recognizing the complement of $L(\mathcal{A}_1)$ and of $L(\mathcal{A}_2)$, respectively (see Theorem 12.5.6). Utilizing nondeterminism, we can furthermore construct an $\mathbb{N}$-memory automaton $\mathcal{A}$ recognizing $L(\mathcal{A}_1') \cup L(\mathcal{A}_1')$ (see Theorem 12.5.1). Then we have $L(\mathcal{A}_1) \cap L(\mathcal{A}_1) = \mathbb{N}^* \setminus \big(L(\mathcal{A}_1') \cup L(\mathcal{A}_1')\big) = \mathbb{N}^* \setminus L(\mathcal{A})$, and therefore $L(\mathcal{A}_1) \cap L(\mathcal{A}_1) = \emptyset$ if and only if $L(\mathcal{A}) = \mathbb{N}^*$.

Since the intersection problem for deterministic $\mathbb{N}$-memory automata is undecidable even if $L(\mathcal{A}_1) \cap L(\mathcal{A}_1)$ is guaranteed to either be empty or consist of a single word (Theorem 12.6.4), the universality problem for nondeterministic $\mathbb{N}$-memory automata is undecidable even if $L(\mathcal{A})$ is guaranteed to be either $\mathbb{N}^*$ or of the form $\mathbb{N}^* \setminus \{w\}$ with $w \in \mathbb{N}^*$. □

In fact, the universality problem is already undecidable for the weaker model of progressive grid automata, as shown in [CST15] by a reduction from the halting problem for 2-register machines.

From Theorem 12.6.7, we can deduce that "determinizability" of a given nondeterministic $\mathbb{N}$-memory automaton is undecidable:

THEOREM 12.6.8. The problem to determine for a given nondeterministic $\mathbb{N}$-memory automaton whether there exists a deterministic $\mathbb{N}$-memory automaton recognizing the same language is undecidable. This holds even for the alphabet $\mathbb{N}$.

Proof. We reduce the universality problem for nondeterministic $\mathbb{N}$-memory automata to the determinizability problem. Let $\mathcal{A}$ be the given $\mathbb{N}$-memory automaton. Recall from Corollary 12.5.3 that the language

$$L = \{m_1 m_2 m_3 \mid m_1, m_2, m_3 \in \mathbb{N} \text{ with } m_1 = m_3 \text{ or } m_2 = m_3\}$$

is recognized by a nondeterministic $\mathbb{N}$-memory automaton, but not by a deterministic one. We can construct a nondeterministic $\mathbb{N}$-memory automaton $\mathcal{A}'$ recognizing the language

$$L(\mathcal{A}') = \big(L(\mathcal{A}) \cdot \mathbb{N} \cdot \mathbb{N} \cdot \mathbb{N}\big) \cup \big(\mathbb{N}^* \cdot L\big).$$

We claim that $L(\mathcal{A}) = \mathbb{N}^*$ if and only if $L(\mathcal{A}')$ is recognized by a deterministic $\mathbb{N}$-memory automaton.

If $L(\mathcal{A}) = \mathbb{N}^*$ then $L(\mathcal{A}')$ is the set of words over $\mathbb{N}$ of length at least 3, which is clearly deterministically $\mathbb{N}$-memory-recognizable.

Now suppose that $L(\mathcal{A}) \subsetneq \mathbb{N}^*$, and let $w_{\text{not}} \in \mathbb{N}^*$ be a word such that $w_{\text{not}} \notin L(\mathcal{A})$. Assume toward a contradiction that $L(\mathcal{A}')$ is recognized by a deterministic $\mathbb{N}$-memory automaton. We can modify this automaton in such a way that it also verifies that the input word is of the form $w_{\text{not}} \cdot m_1 m_2 m_3$ for some $m_1, m_2, m_3 \in \mathbb{N}$. We obtain a deterministic $\mathbb{N}$-memory automaton recognizing the language

$$L(\mathcal{A}') \cap \big(\{w_{\text{not}}\} \cdot \mathbb{N} \cdot \mathbb{N} \cdot \mathbb{N}\big) = \{w_{\text{not}}\} \cdot L.$$

But this language cannot be deterministically $\mathbb{N}$-memory recognizable, by the same argument as for the language L (see the proof of Theorem 12.5.2). Hence, the language $L(\mathcal{A}')$ cannot be recognized by a deterministic $\mathbb{N}$-memory automaton, which concludes the proof of the claim. □

The universality problem can also be reduced to the equivalence problem, as it amounts to a comparison with an automaton accepting all words, so we obtain the following result.

THEOREM 12.6.9. The equivalence problem for $\mathbb{N}$-memory automata ("given $\mathcal{A}_1$ and $\mathcal{A}_2$: $L(\mathcal{A}_1) = L(\mathcal{A}_2)$?") is undecidable. This holds even for the alphabet $\mathbb{N}$.

We leave it as an open question whether the equivalence problem is decidable in the deterministic case. For deterministic one-counter automata, which correspond to deterministic $\mathbb{N}$-memory automata over finite alphabets, this problem is decidable, as shown by Valiant and Paterson [VP75] in 1975. In fact, Sénizergues [Sén01] showed in 2001 that the equivalence problem is decidable even for deterministic pushdown automata, settling a long-standing open question posed in 1966 by Ginsburg and Greibach [GG66].

13

ℕ-Memory Automata on Infinite Words

In the previous chapter, we have defined ℕ-memory automata on finite words over infinite alphabets of the form Σ^* (for a finite set Σ). We now extend this automaton model to infinite words over such infinite alphabets.

This chapter is structured as follows: We define ℕ-*memory* ω-*automata* with various well-known acceptance conditions (Section 13.1) and we compare the expressive power of these automata (Section 13.2), analogously to classical ω-automata. We consider the closure properties of the class of ω-languages recognized by ℕ-memory ω-automata (Section 13.3), and address the decidability of some important decision problems (Section 13.4). We conclude this chapter with a closer look at the ω-languages recognized by ℕ-memory ω-automata over the alphabet ℕ, represented by $\{1\}^*$. More specifically, we examine the location of these ω-languages in the Borel hierarchy of the Baire space $\mathbb{N}^\omega$ (Section 13.5).

13.1 Definition of ℕ-Memory ω-Automata

To process ω-words over infinite alphabets of the form Σ^*, we define ℕ-*memory* ω-*automata*, equipped with one of several possible acceptance conditions, analogous to ω-automata over finite alphabets. In general, such automata have the following form.

ℕ-memory ω-automaton

Definition 13.1.1. An ℕ-*memory* ω-*automaton* $\mathcal{A}$ is a tuple of the form $(Q, \Sigma^*, \Delta, q_0, \mathrm{Acc})$, where

- Q is a finite set of states,
- Σ^* is an infinite alphabet, for a finite set Σ,
- $q_0 \in Q$ is the initial state,
- $\Delta \subseteq (Q \times \mathbb{N}) \times \Sigma^* \times (Q \times \mathbb{N})$ is a pebble-automatic transition relation (see Definition 12.3.2),
- Acc is the *acceptance component*, depending on the specific type of automaton, as defined below.

The automaton $\mathcal{A}$ is called *deterministic* if its transition relation Δ is deterministic. In that case, we may write Δ as a *transition function* $\delta\colon (Q \times \mathbb{N}) \times \Sigma^* \to (Q \times \mathbb{N})$, which can be extended to the domain $(Q \times \mathbb{N}) \times (\Sigma^*)^*$ in the usual way.

As in the case of $\mathbb{N}$-memory automata on finite words, a configuration of an $\mathbb{N}$-memory ω-automaton $\mathcal{A}$ is a pair (p, i) consisting of a state $p \in Q$ and a memorized position $i \in \mathbb{N}$. A *run* of $\mathcal{A}$ on an ω-word $\alpha = u_1u_2u_3\ldots$ with $u_\ell \in \Sigma^*$ (for $\ell \in \mathbb{N}_{\geq 1}$) is an infinite sequence of configurations

$$\varrho = (q_1, i_1)(q_2, i_2)(q_3, i_3)\ldots$$

with $(q_1, i_1) = (q_0, 0)$ such that for all $\ell \in \mathbb{N}_{\geq 1}$, there is a transition $\big((q_\ell, i_\ell), u_\ell, (q_{\ell+1}, i_{\ell+1})\big) \in \Delta$. The *acceptance condition* of the automaton, which determines the accepting runs, can be one of the following.

- *E-condition*, where the acceptance component Acc is a set $F \subseteq Q$:
  ϱ is accepting if there is an $\ell \in \mathbb{N}_{\geq 1}$ with $q_\ell \in F$.
- *A-condition*, where Acc is a set $F \subseteq Q$:
  ϱ is accepting if for all $\ell \in \mathbb{N}_{\geq 1}$, we have $q_\ell \in F$.
- *Büchi condition*, where Acc is a set $F \subseteq Q$:
  ϱ is accepting if there are infinitely many $\ell \in \mathbb{N}_{\geq 1}$ with $q_\ell \in F$.
- *co-Büchi condition*, where Acc is a set $F \subseteq Q$:
  ϱ is accepting if there are only finitely many $\ell \in \mathbb{N}_{\geq 1}$ with $q_\ell \in F$.
- *Muller condition*, where Acc is a set $\mathcal{F} \subseteq 2^Q$:
  ϱ is accepting if there is a set $F \in \mathcal{F}$ containing precisely those states that are visited infinitely often in ϱ.
- *Parity condition*, where Acc is a *priority function* $c\colon Q \to [k]$:
  ϱ is accepting if $\max\{c(q) \mid q \text{ is visited infinitely often in } \varrho\}$ is even.

Recall that $[k] = \{0, \ldots, k\}$.

Definition 13.1.2. Let $\mathcal{A}$ be an $\mathbb{N}$-memory ω-automaton over an alphabet Σ^* (with one of the above-mentioned acceptance conditions). The *ω-language recognized by $\mathcal{A}$* is the set

$$L(\mathcal{A}) = \{\alpha \in (\Sigma^*)^\omega \mid \text{there is an accepting run of } \mathcal{A} \text{ on } \alpha\}.$$

Let X be one of the acceptance condition types E, A, Büchi, co-Büchi, Muller, or parity. We refer to an $\mathbb{N}$-memory ω-automaton with

an acceptance condition of type X as an $\mathbb{N}$-*memory* X *automaton*. A language recognized by a (deterministic) $\mathbb{N}$-memory X automaton is called *(deterministically)* $\mathbb{N}$-*memory-X-recognizable*.

Example 13.1.3. Consider the following ω-language over the alphabet $\mathbb{N}$ (which is technically represented by $\{1\}^*$):

$$L = \{m_1 m_2 m_3 \ldots \in \mathbb{N}^\omega \mid m_1 m_2 m_3 \ldots \text{ is unbounded, i.e., } \{m_1, m_2, m_3, \ldots\} \text{ is an infinite set}\}$$

This ω-language is recognized by a deterministic $\mathbb{N}$-memory Büchi automaton that uses the memorized position to keep track of the maximum number encountered so far. Whenever this maximum increases, the automaton enters an accepting state. Thus, the unique run on an ω-word α visits infinitely often an accepting state if and only if α is unbounded.

13.2 Comparison of Acceptance Conditions

We now compare the expressive power of $\mathbb{N}$-memory ω-automata with different acceptance conditions. The results are obtained in the same way as for ω-automata over finite alphabets.

Proposition 13.2.1. Given a (deterministic) $\mathbb{N}$-memory E-automaton $\mathcal{A}$, we can construct both a (deterministic) $\mathbb{N}$-memory Büchi automaton and a (deterministic) $\mathbb{N}$-memory co-Büchi automaton each recognizing $L(\mathcal{A})$.

Proof. Consider an $\mathbb{N}$-memory E-automaton with a set of accepting states F. We construct an $\mathbb{N}$-memory Büchi automaton $\mathcal{A}'$ that simulates $\mathcal{A}$ until a state in F is reached, then switches to another copy of $\mathcal{A}$ where all states are marked as accepting states of the Büchi automaton. The automaton $\mathcal{A}'$ then recognizes $L(\mathcal{A})$ and this still holds if $\mathcal{A}'$ is interpreted as a co-Büchi automaton. □

Proposition 13.2.2. Given a (deterministic) $\mathbb{N}$-memory A-automaton $\mathcal{A}$, we can construct both a (deterministic) $\mathbb{N}$-memory Büchi automaton and a (deterministic) $\mathbb{N}$-memory co-Büchi automaton each recognizing $L(\mathcal{A})$.

Proof. Consider an $\mathbb{N}$-memory A-automaton with a set of accepting states F. We construct an $\mathbb{N}$-memory Büchi automaton $\mathcal{A}'$ that simulates $\mathcal{A}$, as long as only states in F are visited. If a state outside of

F is reached, the Büchi automaton switches to a non-accepting "sink state" – all other states are accepting. The Büchi automaton $\mathcal{A}'$ then recognizes $L(\mathcal{A})$ and this still holds if $\mathcal{A}'$ is interpreted as a co-Büchi automaton. □

Proposition 13.2.3. Given a (deterministic) ℕ-memory Büchi or co-Büchi automaton, we can construct a (deterministic) ℕ-memory parity automaton recognizing the same ω-language.

Proof. An ℕ-memory Büchi automaton with a set of accepting states F can be regarded as an ℕ-memory parity automaton where the states in F have priority 2 and the other states have priority 1.

Similarly, an ℕ-memory co-Büchi automaton can be viewed as an ℕ-memory parity automaton where the states in F have priority 0 and the other states have priority 1. □

Proposition 13.2.4. Given a (deterministic) ℕ-memory parity automaton, we can construct a (deterministic) ℕ-memory Muller automaton recognizing the same ω-language, and vice versa.

Proof. The priority function of an ℕ-memory parity automaton can be converted into an equivalent Muller acceptance component $\mathcal{F}$ containing those sets of states in which the maximum priority is even.

To transform an ℕ-memory Muller automaton into an ℕ-memory parity automaton, we augment the states by so-called *latest appearance records*, just as for standard ω-automata (see [Tho97] for details). □

Proposition 13.2.5. Given an ℕ-memory parity automaton, we can construct an ℕ-memory Büchi automaton that recognizes the same ω-language.

Proof. We can use the following construction known from ω-automata over finite alphabets. For a given ℕ-memory parity automaton $\mathcal{A}$ with priorities $\{0, \ldots, k\}$, the corresponding ℕ-memory Büchi automaton processes a given ω-word α as follows: It simulates a run of $\mathcal{A}$ on α and verifies that some even priority $e \in \{0, \ldots, k\}$ is seen infinitely often and that from some point onwards, no priority greater than e occurs. To do so, the Büchi automaton guesses nondeterministically such an even priority e and such a point in the run. At that point, the Büchi automaton switches to a copy of the automaton $\mathcal{A}$ where only transitions to states with a priority at most e are possible and where precisely the states with priority e are accepting. □

13.3 Closure Properties of ℕ-Memory-Recognizable ω-Languages

In this section, we investigate the closure properties of ℕ-memory ω-automata. We first prove the following two lemmas, which will allow us to transfer results from ℕ-memory automata over finite words to ℕ-memory ω-automata.

Lemma 13.3.1. Given an ℕ-memory parity automaton $\mathcal{A}$ and a state p of $\mathcal{A}$, we can construct an MSO formula $\textsc{AcceptingRun}_p(x)$ such that for all $i \in \mathbb{N}$,

$$\mathcal{N} \models \textsc{AcceptingRun}_p[i] \quad \text{iff} \quad \begin{array}{l}\text{there is an accepting run of } \mathcal{A} \\ \text{starting in the configuration } (p, i).\end{array}$$

Proof. Let $\mathcal{A} = (Q, \Sigma^*, \Delta, q_0, c)$ be the given ℕ-memory parity automaton with $c\colon Q \to [k]$, where Δ is defined by some family of MSO formulas $\left(\varphi_{p,q}(\overline{Z}, x, y)\right)_{p,q\in Q}$. A run of $\mathcal{A}$ that starts in (p, i) is accepting if and only if it has the following form, where $q \in Q$ is a state with an even priority $c(q)$:

$$(p,i)\ldots\underbrace{(q,i_1)\ldots(q,i_2)\ldots(q,i_3)\ldots(q,i_4)\ldots}_{\text{only states with priority at most } c(q)}$$

To express the existence of such a run, we use a slightly refined version of the formula $\textsc{Run}_{p,q}(x, y)$ presented in the proof of Theorem 12.6.1, taking into account the maximal admissible priority. More specifically, we define the formula $\textsc{Run}^e_{p,q}(x, y)$ with $e \in [k]$ asserting the existence of a run of $\mathcal{A}$ (on some arbitrary input word) from the configuration consisting of the state p and the memorized position x to the configuration consisting of the state q and the memorized position y, visiting only states whose priority is at most e:

$$\textsc{Run}^e_{p,q}(x, y) = \forall (X_r)_{r\in Q}\left(\left(X_p(x) \wedge \textsc{Closed}^e\left((X_r)_{r\in Q}\right)\right) \to X_q(y)\right)$$

with

$$\textsc{Closed}^e\left((X_r)_{r\in Q}\right) = \bigwedge_{\substack{p,q\in Q \\ c(p),c(q)\leq e}} \forall x, y \left(\left[\begin{array}{c} X_p(x) \wedge \\ \exists \overline{Z}\ \varphi_{p,q}(\overline{Z}, x, y)) \end{array}\right] \to X_q(y)\right).$$

The existence of the first part $(p, i) \dots (q, i_1)$ of a run as described above can be asserted using the formula $\textsc{Run}^k_{p,q}(x, y)$, that means, without restricting the priorities.

To describe the remaining part of the run, where q occurs repeatedly and the maximal priority is $c(q)$, we require that (q, i_1) is part of a set of configurations with state q such that from every configuration in that set, the automaton can reach another configuration in the same set visiting only states with a priority at most $c(q)$. This is expressed by the formula

$$\textsc{Recur}_q(y) = \exists X \left(X(y) \wedge \forall z \left(X(z) \rightarrow \exists z' \begin{bmatrix} X(z') \wedge \\ \textsc{Run}^{c(q)}_{q,q}(z, z') \end{bmatrix} \right) \right).$$

Now we can compose the desired MSO formula $\textsc{AcceptingRun}_p(x)$ as follows:

$$\textsc{AcceptingRun}_p(x) = \exists y \bigvee_{\substack{q \text{ with} \\ \text{even } c(q)}} \Big(\textsc{Run}^k_{p,q}(x, y) \wedge \textsc{Recur}_q(y) \Big).$$

□

Lemma 13.3.2. Let $W \subseteq \mathbb{N}^*_{\geq 1}$ and $L \subseteq \mathbb{N}^\omega$, with $L \neq \emptyset$. If the ω-language $W \cdot \{0\} \cdot L$ is (deterministically) $\mathbb{N}$-memory-parity-recognizable, then the language W is (deterministically) $\mathbb{N}$-memory-recognizable.

Proof. Let $\mathcal{A} = (Q, \mathbb{N}, \Delta, q_0, c)$ be an $\mathbb{N}$-memory parity automaton recognizing $W \cdot \{0\} \cdot L$, where Δ is defined by some family of MSO formulas $\big(\varphi_{p,q}(z, x, y)\big)_{p,q \in Q}$. Note that a finite word $w \in \mathbb{N}^*_{\geq 1}$ is in W if and only if there is a run of $\mathcal{A}$ on w ending in a configuration (p, i) with the property that there is an accepting run starting in (p, i) on some ω-word that begins with the symbol 0. Let us call such a configuration (p, i) an *accepting configuration*.

We obtain an automaton $\mathcal{A}'$ recognizing W from the ω-automaton $\mathcal{A}$ recognizing $W \cdot \{0\} \cdot L$ in the following way: We redirect every transition that enters an accepting configuration to a copy of that configuration where the state is marked as a final state, and we remove all transitions for the symbol 0 from the automaton. (If the initial configuration of $\mathcal{A}$ is accepting, then we also have to use a copy of q_0, marked as a final state, as the new initial state.)

To express that a configuration consisting of a state p and a memorized position x is an accepting configuration, we define the following MSO

formula $\textsc{AcceptingConfig}_p(x)$, where $\textsc{AcceptingRun}_q(y)$ is the formula obtained by Lemma 13.3.1:

$$\textsc{AcceptingConfig}_p(x) = \exists y \left(\bigvee_{q \in Q} \varphi_{p,q}(0, x, y) \wedge \textsc{AcceptingRun}_q(y) \right)$$

The desired $\mathbb{N}$-memory automaton $\mathcal{A}' = (Q', \mathbb{N}, \Delta', q_0', F)$ can now formally be constructed as follows:

- $Q' = \{0, 1\} \times Q$,
- Δ' is defined by the following family of formulas $(\varphi'_{p',q'})_{p',q' \in Q'}$, for $b \in \{0, 1\}$ and $p, q \in Q$:

$$\begin{aligned} \varphi'_{(b,p),(0,q)}(z, x, y) &= \neg\textsc{AcceptingConfig}_q(y) \wedge \\ &\quad \varphi_{p,q}(z, x, y) \wedge z \neq 0, \\ \varphi'_{(b,p),(1,q)}(z, x, y) &= \textsc{AcceptingConfig}_q(y) \wedge \\ &\quad \varphi_{p,q}(z, x, y) \wedge z \neq 0, \end{aligned}$$

- $q_0' = \begin{cases} (1, q_0) & \text{if } (q_0, 0) \text{ is an accepting configuration,} \\ (0, q_0) & \text{otherwise,} \end{cases}$

  which can in fact be determined effectively by checking whether $\mathcal{N} \models \textsc{AcceptingConfig}_{q_0}[0]$, as the MSO theory of $\mathcal{N}$ is decidable,
- $F = \{1\} \times Q$.

If the $\mathbb{N}$-memory parity automaton $\mathcal{A}$ is deterministic, then we can easily make the transition relation of the $\mathbb{N}$-memory automaton $\mathcal{A}'$ a complete function (and hence deterministic) by adding transitions via the symbol 0 to a non-final sink state. □

We are now ready to address the closure properties of $\mathbb{N}$-memory ω-automata.

Theorem 13.3.3. Let X be one of the acceptance condition types E, A, Büchi, co-Büchi, Muller, or parity.
The class of $\mathbb{N}$-memory-X-recognizable ω-languages is closed under union.

Proof. Given two $\mathbb{N}$-memory X automata, we can utilize nondeterminism to construct a third one recognizing the union of the ω-languages of the given automata. The construction is analogous to the one for $\mathbb{N}$-memory automata on finite words (see the proof of Theorem 12.5.1). □

Theorem 13.3.4. Let X be one of the acceptance condition types E, A, Büchi, co-Büchi, Muller, or parity.
The class of deterministically $\mathbb{N}$-memory-X-recognizable languages is *not* closed under union. This holds even for the alphabet $\mathbb{N}$.

Proof. We can lift the proof for the case of finite words to ω-words: Consider the languages (of finite words) $L_1 = \{m_1 m_2 m_1 \mid m_1, m_2 \in \mathbb{N}_{\geq 1}\}$ and $L_2 = \{m_1 m_2 m_2 \mid m_1, m_2 \in \mathbb{N}_{\geq 1}\}$. The ω-languages $L_1 \cdot \{0\} \cdot \mathbb{N}^\omega$ and $L_2 \cdot \{0\} \cdot \mathbb{N}^\omega$ are deterministically $\mathbb{N}$-memory-X-recognizable for all acceptance types X mentioned in the theorem. By the same argument as in the proof of Theorem 12.5.2, the language $L = L_1 \cup L_2 \subseteq \mathbb{N}_{\geq 1}^*$ is not deterministically $\mathbb{N}$-memory-recognizable. Thus, the ω-language $L_1 \cdot \{0\} \cdot \mathbb{N}^\omega \cup L_2 \cdot \{0\} \cdot \mathbb{N}^\omega = L \cdot \{0\} \cdot \mathbb{N}^\omega$ cannot be deterministically $\mathbb{N}$-memory-parity-recognizable, according to Lemma 13.3.2. □

Corollary 13.3.5. Let X be one of the acceptance condition types E, A, Büchi, co-Büchi, Muller, or parity. The class of deterministically $\mathbb{N}$-memory-X-recognizable ω-languages is strictly included in the class of $\mathbb{N}$-memory-X-recognizable languages.
Furthermore, the ω-language $L \cdot \{0\} \cdot \mathbb{N}^\omega$ with

$$L = \{m_1 m_2 m_3 \mid m_1, m_2, m_3 \in \mathbb{N}_{\geq 1} \text{ with } m_1 = m_3 \text{ or } m_2 = m_3\}$$

from the proof of Theorem 13.3.4 is $\mathbb{N}$-memory-E-recognizable and $\mathbb{N}$-memory-A-recognizable, but cannot be recognized by a deterministic $\mathbb{N}$-memory ω-automaton, not even with a parity acceptance condition.

Theorem 13.3.6. Let X be one of the acceptance condition types E, A, Büchi, co-Büchi, Muller, or parity. Neither the class of $\mathbb{N}$-memory-X-recognizable ω-languages nor the class of deterministically $\mathbb{N}$-memory-X-recognizable ω-languages is closed under intersection. This holds even for the alphabet $\mathbb{N}$.

Proof. We can transfer the corresponding result for finite words (Theorem 12.5.4) to ω-words. Consider the languages (of finite words)

$L_1 = \{m_1m_2m_3m_4 \in \mathbb{N}^*_{\geq 1} \mid m_1 = m_3\}$ and $L_2 = \{m_1m_2m_3m_4 \in \mathbb{N}^*_{\geq 1} \mid m_2 = m_4\}$. The ω-languages $L_1 \cdot \{0\} \cdot \mathbb{N}^\omega$ and $L_2 \cdot \{0\} \cdot \mathbb{N}^\omega$ are deterministically $\mathbb{N}$-memory-X-recognizable for all acceptance types X mentioned in the theorem. By the same argument as in the proof of Theorem 12.5.4, the language $L = L_1 \cap L_2 \subseteq \mathbb{N}^*_{\geq 1}$ is not $\mathbb{N}$-memory-recognizable. Thus, the ω-language $L_1 \cdot \{0\} \cdot \mathbb{N}^\omega \cap L_2 \cdot \{0\} \cdot \mathbb{N}^\omega = L \cdot \{0\} \cdot \mathbb{N}^\omega$ cannot be $\mathbb{N}$-memory-parity-recognizable, according to Lemma 13.3.2. □

Theorem 13.3.7. Let X be one of the acceptance condition types E, A, Büchi, co-Büchi, Muller, or parity. The class of $\mathbb{N}$-memory-X-recognizable ω-languages is *not* closed under complement. This holds even for the alphabet $\mathbb{N}$.

Proof. If the class of $\mathbb{N}$-memory-recognizable languages was closed under complement, then closure under union (Theorem 13.3.3) together with De Morgan's laws would imply closure under intersection, contradicting Theorem 13.3.6. □

For deterministic $\mathbb{N}$-memory ω-automata, the results depend on the acceptance condition, as for standard ω-automata on finite alphabets.

Theorem 13.3.8. The class of deterministically $\mathbb{N}$-memory-parity-recognizable ω-languages is closed under complement.

Proof. If we increase all priorities of a deterministic $\mathbb{N}$-memory parity automaton by 1, its unique run on a given ω-word becomes rejecting if it was accepting before, and vice versa. □

Theorem 13.3.9. Let X be one of the acceptance condition types E, A, Büchi, or co-Büchi. The class of deterministically $\mathbb{N}$-memory-X-recognizable ω-languages is *not* closed under complement. This holds even for the alphabet $\mathbb{N}$.

Proof. We can use essentially the same arguments as in the case of ω-automata over finite alphabets.

E-condition: While the ω-language $\mathbb{N}^* \cdot \mathbb{N}_{\geq 1} \cdot \mathbb{N}^\omega$ is deterministically $\mathbb{N}$-memory-E-recognizable, its complement $\{0^\omega\}$ is not.

A-condition: By exchanging the roles of the accepting and the non-accepting states in a deterministic $\mathbb{N}$-memory A-automaton $\mathcal{A}$, we

obtain an E-automaton recognizing the complement of $L(\mathcal{A})$, and vice versa. Therefore, the statement follows from the statement for the E-condition.

Büchi condition: The ω-language $(\mathbb{N}^* \cdot \mathbb{N}_{\geq 1})^\omega$ is deterministically $\mathbb{N}$-memory-Büchi-recognizable, its complement $\mathbb{N}^* \cdot \{0\}^\omega$ is not.

Co-Büchi condition: By exchanging the roles of accepting and non-accepting states in a deterministic $\mathbb{N}$-memory co-Büchi automaton $\mathcal{A}$, we obtain a Büchi automaton recognizing the complement of $L(\mathcal{A})$, and vice versa. Therefore, the statement follows from the statement for the Büchi condition. □

Proposition 13.3.10. There is a deterministically $\mathbb{N}$-memory-recognizable language W (of finite words) and a deterministically $\mathbb{N}$-memory-E- and -A-recognizable ω-language L such that $W \cdot L$ is not deterministically $\mathbb{N}$-memory-parity-recognizable.

Proof. The language $W = \{mm \mid m \in \mathbb{N}_{\geq 1}\} \cup \{mm'm \mid m, m' \in \mathbb{N}_{\geq 1}\}$ and the ω-language $L = W \cdot \{0\} \cdot \mathbb{N}^\omega$ satisfy the conditions of the proposition.

To see that $W \cdot L$ cannot be recognized by a deterministic $\mathbb{N}$-memory parity automaton, note that $W \cdot L = W \cdot W \cdot \{0\} \cdot \mathbb{N}^\omega$. By the same argument as in the proof of Theorem 12.5.8, the language $W \cdot W \subseteq \mathbb{N}_{\geq 1}^*$ is not deterministically $\mathbb{N}$-memory-recognizable. Thus, the ω-language $W \cdot L = W \cdot W \cdot \{0\} \cdot \mathbb{N}^\omega$ cannot be deterministically $\mathbb{N}$-memory-parity-recognizable, according to Lemma 13.3.2. □

Corollary 13.3.11. Let X be one of the acceptance condition types E, Büchi, co-Büchi, Muller, or parity. The class of deterministically $\mathbb{N}$-memory-X-recognizable ω-languages is not closed under concatenation with deterministically $\mathbb{N}$-memory-recognizable languages of finite words.

Theorem 13.3.12. For every $\mathbb{N}$-memory-recognizable language W of finite words, the ω-language W^ω is $\mathbb{N}$-memory-Büchi-recognizable.

Proof. Given an $\mathbb{N}$-memory automaton $\mathcal{A}$, we can construct an $\mathbb{N}$-memory Büchi automaton $\mathcal{A}'$ that recognizes $L(\mathcal{A})^\omega$ in the following way, as in the case of automata over finite alphabets: The automaton $\mathcal{A}'$

simulates $\mathcal{A}$. Whenever $\mathcal{A}$ reaches a final state, $\mathcal{A}'$ can choose nondeterministically to either continue or restart the simulation of $\mathcal{A}$ (treating the current memorized position as 0 in the latter case). The accepting states of the Büchi automaton $\mathcal{A}'$ are those at the beginning of a restart. □

Theorem 13.3.13. There is a deterministically $\mathbb{N}$-memory-recognizable language W of finite words such that W^ω is not deterministically $\mathbb{N}$-memory-parity-recognizable.

Proof. We let $U = \{mm \mid m \in \mathbb{N}_{\geq 1}\} \cup \{mm'm \mid m, m' \in \mathbb{N}_{\geq 1}\}$ and $W = U \cup \{0\}$. The language W (as well as U) is deterministically $\mathbb{N}$-memory-recognizable. However, by the same argument as in the proof of Theorem 12.5.8, the language U^* is not deterministically $\mathbb{N}$-memory-recognizable.

Now suppose that the ω-language W^ω is recognized by a deterministic $\mathbb{N}$-memory parity automaton. This automaton can easily be modified in such a way that it also verifies that the input word contains the symbol 0, so we obtain a deterministic $\mathbb{N}$-memory parity automaton recognizing $W^\omega \cap \mathbb{N}^* \cdot \{0\} \cdot \mathbb{N}^\omega = U^* \cdot \{0\} \cdot W^\omega$. Thus, it follows from Lemma 13.3.2 that the language U^* is deterministically $\mathbb{N}$-memory-recognizable, which is a contradiction. □

13.4 Decision Problems for $\mathbb{N}$-Memory ω-Automata

Theorem 13.4.1. The nonemptiness problem for $\mathbb{N}$-memory parity automata ("given $\mathcal{A}$: $L(\mathcal{A}) \neq \emptyset$?") is decidable.

Proof. Let $\mathcal{A}$ be the given $\mathbb{N}$-memory parity automaton with initial state q_0. We have already shown that we can construct an MSO formula $\textsc{AcceptingRun}_p(x)$ asserting the existence of an accepting run of $\mathcal{A}$ starting in state p and with the memorized position x (Lemma 13.3.1). In particular, we have $\mathcal{N} \models \textsc{AcceptingRun}_{q_0}[0]$ if and only if $L(\mathcal{A}) \neq \emptyset$. This property can be effectively verified since the MSO theory of $\mathcal{N}$ is decidable. □

For deterministic $\mathbb{N}$-memory parity automata, the decidability of the nonemptiness problem implies the decidability of the universality problem, since deterministic $\mathbb{N}$-memory parity automata can be effectively complemented (see Theorem 13.3.8):

Corollary 13.4.2. The universality problem for deterministic $\mathbb{N}$-memory parity automata ("given $\mathcal{A}$: $L(\mathcal{A}) = (\Sigma^*)^\omega$?") is decidable.

We will now transfer the undecidability results that we obtained for $\mathbb{N}$-memory automata on finite words to $\mathbb{N}$-memory ω-automata with an E-acceptance condition – this implies undecidability also for Büchi, co-Büchi, Muller and parity acceptance conditions. The basic idea is to transform languages $L \subseteq \mathbb{N}_{\geq 1}^*$ of finite words into ω-languages of the form $L \cdot \{0\} \cdot \mathbb{N}^\omega$. The "separating symbol" 0 guarantees a one-to-one correspondence between languages of finite words over $\mathbb{N}_{\geq 1}$ and ω-languages over $\mathbb{N}$.

Proposition 13.4.3. Given an $\mathbb{N}$-memory automaton $\mathcal{A}$ over the alphabet $\mathbb{N}$ recognizing a language (of finite words) $L(\mathcal{A}) \subseteq \mathbb{N}_{\geq 1}^*$, we can construct an $\mathbb{N}$-memory E-automaton $\mathcal{A}'$ recognizing the ω-language $L(\mathcal{A}) \cdot \{0\} \cdot \mathbb{N}^\omega$. If $\mathcal{A}$ is deterministic, then so is $\mathcal{A}'$.

Proof. To obtain $\mathcal{A}'$ from $\mathcal{A}$, we add a new "sink state" q_f with a self-loop for all memorized positions and all input numbers. This state q_f serves as the only accepting state of $\mathcal{A}'$. We add transitions from every final state of $\mathcal{A}$, for all memorized positions, via the number 0 to q_f. The resulting automaton $\mathcal{A}'$ accepts precisely the ω-words of the form $w0\beta$ with $w \in L(\mathcal{A})$ and $\beta \in \mathbb{N}^\omega$. In case $\mathcal{A}$ is deterministic, we can preserve determinism by removing all previously existing transitions via the number 0 from the final states of $\mathcal{A}$. Since $L(\mathcal{A}) \subseteq \mathbb{N}_{\geq 1}^*$, this does not alter the recognized language. □

Remark 13.4.4. All of the undecidability results in Section 12.6 were shown for $\mathbb{N}$-memory automata recognizing languages over the alphabet $\mathbb{N}$. However, these results also hold if the alphabet is restricted to $\mathbb{N}_{\geq 1} = \mathbb{N} \setminus \{0\}$, since every (deterministic) $\mathbb{N}$-memory automaton $\mathcal{A}$ can be transformed into another (deterministic) $\mathbb{N}$-memory automaton that recognizes the language $\{(m_1+1)\dots(m_n+1) \mid m_1 \dots m_n \in L(\mathcal{A})\} \subseteq \mathbb{N}_{\geq 1}^*$.

Theorem 13.4.5. The empty intersection problem for deterministic $\mathbb{N}$-memory E-automata ("given $\mathcal{A}_1$ and $\mathcal{A}_2$: $L(\mathcal{A}_1) \cap L(\mathcal{A}_2) = \emptyset$?") is undecidable. This holds even for the alphabet $\mathbb{N}$.

Proof. We apply a reduction from the empty intersection problem for deterministic $\mathbb{N}$-memory automata (on finite words), which is undecidable even for the alphabet $\mathbb{N}$ (Theorem 12.6.4).

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the given deterministic $\mathbb{N}$-memory automata on finite words over the alphabet $\mathbb{N}$. By Remark 13.4.4, we may assume that $L(\mathcal{A}_1), L(\mathcal{A}_2) \subseteq \mathbb{N}_{\geq 1}^*$. We can transform the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ into deterministic $\mathbb{N}$-memory E-automata $\mathcal{A}_1'$ and $\mathcal{A}_2'$ recognizing the ω-languages $L(\mathcal{A}_1') = L(\mathcal{A}_1) \cdot \{0\} \cdot \mathbb{N}^\omega$ and $L(\mathcal{A}_2') = L(\mathcal{A}_2) \cdot \{0\} \cdot \mathbb{N}^\omega$, respectively (see Proposition 13.4.3). Then we have $L(\mathcal{A}_1) \cap L(\mathcal{A}_2) = \emptyset$ if and only if $L(\mathcal{A}_1') \cap L(\mathcal{A}_2') = \emptyset$, which completes the reduction. □

Note that the reduction described in the proof of Theorem 13.4.5 also reduces the inclusion problem for deterministic $\mathbb{N}$-memory automata to the inclusion problem for deterministic $\mathbb{N}$-memory E-automata, since $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ if and only if $L(\mathcal{A}_1') \subseteq L(\mathcal{A}_2')$, yielding the following theorem.

THEOREM 13.4.6. The inclusion problem for deterministic $\mathbb{N}$-memory E-automata ("given $\mathcal{A}_1$ and $\mathcal{A}_2$: $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$?") is undecidable. This holds even for the alphabet $\mathbb{N}$.

In a similar way, we obtain the undecidability of the universality problem for nondeterministic $\mathbb{N}$-memory E-automata.

THEOREM 13.4.7. The universality problem for $\mathbb{N}$-memory E-automata ("given $\mathcal{A}$: $L(\mathcal{A}) = (\Sigma^*)^\omega$?") is undecidable. This holds even for the alphabet $\mathbb{N}$.

Proof. We give a reduction from the universality problem for nondeterministic $\mathbb{N}$-memory automata (on finite words), which is undecidable even for the alphabet $\mathbb{N}$ (Theorem 12.6.7).

Let $\mathcal{A}$ be the given $\mathbb{N}$-memory automaton. By Remark 13.4.4, we may assume that the alphabet of this automaton is restricted to $\mathbb{N}_{\geq 1}$. We can construct an $\mathbb{N}$-memory E-automaton $\mathcal{A}'$ recognizing the ω-language $L(\mathcal{A}') = L(\mathcal{A}) \cdot \{0\} \cdot \mathbb{N}^\omega$ (see Proposition 13.4.3). Note that $L(\mathcal{A}') \subsetneq \mathbb{N}^\omega$ since every ω-word in $L(\mathcal{A}')$ contains at least one 0. However, we can also build an $\mathbb{N}$-memory E-automaton $\mathcal{A}_{\geq 1}$ that accepts the words without a 0, that is, $L(\mathcal{A}_{\geq 1}) = \mathbb{N}_{\geq 1}^\omega$. By combining these automata, we obtain an $\mathbb{N}$-memory E-automaton $\mathcal{A}''$ recognizing $L(\mathcal{A}') \cup L(\mathcal{A}_{\geq 1})$. Then we have $L(\mathcal{A}) = \mathbb{N}_{\geq 1}^*$ if and only if $L(\mathcal{A}'') = L(\mathcal{A}') \cup L(\mathcal{A}_{\geq 1}) = \mathbb{N}^\omega$, which completes the reduction. □

By the same argument as in the case of finite words (see Theorem 12.6.9), the undecidability of the equivalence problems follows:

THEOREM 13.4.8. The equivalence problem for $\mathbb{N}$-memory E-automata ("given $\mathcal{A}_1$ and $\mathcal{A}_2$: $L(\mathcal{A}_1) = L(\mathcal{A}_2)$?") is undecidable. This holds even for the alphabet $\mathbb{N}$.

While we left open the question whether the equivalence problem is decidable for deterministic $\mathbb{N}$-memory automata on finite words (see the end of Section 12.6), we can in fact give a negative answer in the case of deterministic $\mathbb{N}$-memory Büchi automata, using a recent result by Böhm et al.

THEOREM 13.4.9. The equivalence problem for deterministic $\mathbb{N}$-memory Büchi automata ("given $\mathcal{A}_1$ and $\mathcal{A}_2$: $L(\mathcal{A}_1) = L(\mathcal{A}_2)$?") is undecidable. This holds even if the alphabet is restricted to a finite set.

Proof. Böhm et al. [BGH+17] prove that the equivalence problem is undecidable for deterministic one-counter Büchi automata. We have already shown that deterministic one-counter automata can be simulated by deterministic $\mathbb{N}$-memory automata over a finite alphabet (Theorem 12.4.3). The same argument holds for automata with a Büchi acceptance condition, yielding a reduction of the equivalence problem for deterministic one-counter Büchi automata to the equivalence problem for deterministic $\mathbb{N}$-memory Büchi automata, so the latter problem is undecidable as well. □

Finally, we address the "determinizability" problem.

THEOREM 13.4.10. The problem to determine for a given nondeterministic $\mathbb{N}$-memory E-automaton whether there exists a deterministic $\mathbb{N}$-memory E-automaton recognizing the same language is undecidable. This holds even for the alphabet $\mathbb{N}$.

Proof. We use a reduction from the determinizability problem for $\mathbb{N}$-memory automata (on finite words), which is undecidable even for the alphabet $\mathbb{N}$ (Theorem 12.6.8).

Let $\mathcal{A}$ be the given nondeterministic $\mathbb{N}$-memory automaton on finite words over the alphabet $\mathbb{N}$. By Remark 13.4.4, we may assume that $L(\mathcal{A}) \subseteq \mathbb{N}^*_{\geq 1}$. We can convert $\mathcal{A}$ into a nondeterministic $\mathbb{N}$-memory E-automaton $\mathcal{A}'$ recognizing the ω-language $L(\mathcal{A}') = L(\mathcal{A}) \cdot \{0\} \cdot \mathbb{N}^\omega$, as stated in Proposition 13.4.3. By the same proposition, if $L(\mathcal{A})$ is deterministically $\mathbb{N}$-memory-recognizable then $L(\mathcal{A}')$ is deterministically $\mathbb{N}$-memory-E-recognizable. The reverse direction follows from Lemma 13.3.2, which completes the reduction. □

13.5 Connection to the Borel Hierarchy

In set-theoretic terminology, ω-words over the finite alphabet $\{0,1\}$ are the elements of the *Cantor space* $\{0,1\}^\omega$, and ω-words over the infinite alphabet $\mathbb{N}$ are the elements of the *Baire space* $\mathbb{N}^\omega$. It is well known that over the alphabet $\{0,1\}$, the classes of ω-languages recognized by deterministic ω-automata with an E-, A- Büchi, or co-Büchi acceptance condition correspond to the first levels of the Borel hierarchy in the Cantor space, restricted to regular ω-languages (see [Wag79]). In this section, we consider the Borel hierarchy in the Baire space and examine its connection to the classes of ω-languages over the alphabet $\mathbb{N}$ that are recognized by deterministic $\mathbb{N}$-memory ω-automata.

Cantor space, Baire space

13.5.1 Basics on the Baire Space

Let us recall the topology of the Baire space $\mathbb{N}^\omega$ (see [Kec95; Mos09] for details). This is best visualized by the infinite $\mathbb{N}$-branching tree $\mathcal{T}_\mathbb{N}$, where each node $v \in \mathbb{N}^*$, starting with the root node ε, has the successors $v0, v1, v2, \ldots$. The elements of the Baire space are the infinite paths through this tree, and an ω-language $L \subseteq \mathbb{N}^\omega$ is a set of such paths.

A *basic open set* of the Baire space is an ω-language of the form $\{w\} \cdot \mathbb{N}^\omega$ with a common finite prefix $w \in \mathbb{N}^*$. Such an ω-language can be thought of as the set of all infinite paths of $\mathcal{T}_\mathbb{N}$ that go through the node w. The *open sets* of the Baire space are the ω-languages over $\mathbb{N}$ that can be obtained as a union of basic open sets. Thus, a set $L \subseteq \mathbb{N}^\omega$ is open if and only if it is of the form $L = W \cdot \mathbb{N}^\omega$ with $W \subseteq \mathbb{N}^*$. A set is called *closed* if it is the complement $\mathbb{N}^\omega \setminus L$ of an open set L.

basic open set

open set

closed set

The open and closed sets of the Cantor space are defined analogously, with $\mathbb{N}$ being replaced by $\{0,1\}$. The elements of the Cantor space can be understood as the infinite paths in the binary tree $\mathcal{T}_{\{0,1\}}$. Instead of $\{0,1\}$, we may also consider an arbirary finite set – the resulting space is homeomorphic (that is, topologically isomorphic) to the space $\{0,1\}^\omega$.

A small difference between the Baire space and the Cantor space appears in the characterization of the sets that are both open and closed, which are called *clopen sets*. In the Cantor space $\{0,1\}^\omega$, a set is clopen if and only if it is of the form $W \cdot \{0,1\}^\omega$ for some *finite* set $W \subseteq \{0,1\}^*$. This equivalence rests on an application of König's Lemma in the binary tree, which does no more hold in the $\mathbb{N}$-branching tree. In fact, in the Baire space, the set $\mathrm{Even} \cdot \mathbb{N}^\omega$ with $\mathrm{Even} = \{0,2,4,\ldots\}$ and its complement $\mathrm{Odd} \cdot \mathbb{N}^\omega$ with $\mathrm{Odd} = \{1,3,5,\ldots\}$ are both open and closed, but clearly neither of them is of the form $W \cdot \mathbb{N}^*$ with finite W.

clopen set

Borel hierarchy

The sets of the *Borel hierarchy*, both in the Baire space and in the Cantor space, are built from the open and closed sets by taking countable unions and countable intersections. We write Σ_1 and Π_1 to denote the class of open sets and the class of closed sets, respectively, and for every countable ordinal $\eta \geq 2$, we define the following classes of sets:

- Σ_η is the class of countable unions $\bigcup_{i\geq 0} L_i$ with $L_i \in \bigcup_{\mu<\eta} \Pi_\mu$.
- Π_η is the class of countable intersections $\bigcap_{i\geq 0} L_i$ with $L_i \in \bigcup_{\mu<\eta} \Sigma_\mu$.

Borel set

A set is said to be *Borel* if it belongs to some class Σ_η or Π_η (with $\eta \geq 1$). Both in the Baire space and in the Cantor space, the classes of the Borel hierarchy have the following properties: The class Σ_η consists precisely of the complements of the sets in Π_η. Furthermore, Σ_η and Π_η are both proper subclasses of $\Delta_{\eta+1} = \Sigma_{\eta+1} \cap \Pi_{\eta+1}$, which itself is properly contained in both $\Sigma_{\eta+1}$ and $\Pi_{\eta+1}$.

Another small difference between the Cantor space and the Baire space arises in the representation of *analytic sets*, which are obtained as projections of Borel sets: In the Baire space, the analytic sets coincide with the projections of Π_1-sets (that is, of closed sets), whereas in the Cantor space they are the projections of Π_2-sets.

13.5.2 ℕ-Memory ω-Automata and the Borel Hierarchy

The goal of this section is to locate ω-languages recognized by ℕ-memory ω-automata over the alphabet ℕ within the Borel hierarchy of the Baire space, depending on the acceptance condition of the automata. First, we consider the sets of "accepting sequences of states" specified by the different acceptance conditions. We use their location in the Borel hierarchy to draw conclusions about the recognized ω-languages.

Consider an ℕ-memory ω-automaton $\mathcal{A}$ over the alphabet ℕ with a (finite) state set Q. The acceptance condition of $\mathcal{A}$ defines a set of state sequences that are considered accepting. For example, if $\mathcal{A}$ is a Büchi automaton with a set of accepting states F, then the set of accepting state sequences is

$$\{q_1 q_2 q_3 \ldots \in Q^\omega \mid \text{for infinitely many } i \geq 1,\ q_i \in F\}.$$

It is clear that acceptance conditions of the types E, A, Büchi, and co-Büchi define sets of state sequences that are in the classes Σ_1, Π_1, Π_2, and Σ_2, respectively, of the Borel hierarchy of the space Q^ω. Similarly, the sets of state sequences defined by parity and Muller acceptance

conditions are Boolean combinations of sets in Σ_2. Let us now verify that the ω-language $L(\mathcal{A}) \subseteq \mathbb{N}^\omega$ recognized by a *deterministic* $\mathbb{N}$-memory ω-automaton $\mathcal{A}$ with one of the acceptance conditions above belongs to the corresponding Borel class, now in the Baire space.

A function $f : \mathbb{N}^\omega \to Q^\omega$ is *continuous* if, for all $\alpha \in \mathbb{N}^\omega$, each entry q_i of the sequence $f(\alpha) = q_1 q_2 q_3 \ldots$ is determined by a finite prefix of α. For a deterministic $\mathbb{N}$-memory ω-automaton $\mathcal{A}$, the function that associates with each ω-word $\alpha \in \mathbb{N}^\omega$ the corresponding run of $\mathcal{A}$ is clearly continuous. Hence we can apply the following well-known fact.

LEMMA 13.5.1. Let Q be a finite set and let $f : \mathbb{N}^\omega \to Q^\omega$ be a continuous function. Let $K \subseteq Q^\omega$. If K is in Σ_η (in Π_η) in the Cantor space Q^ω, then $f^{-1}(K)$ is also in Σ_η (in Π_η) in the Baire space $\mathbb{N}^\omega$.

Thus, we obtain the following results.

THEOREM 13.5.2. For every deterministic $\mathbb{N}$-memory ω-automaton $\mathcal{A}$ over the alphabet $\mathbb{N}$, equipped with an E-, A- Büchi, or co-Büchi acceptance condition, the ω-language $L(\mathcal{A}) \subseteq \mathbb{N}^\omega$ is in the class Σ_1, Π_1, Π_2, or Σ_2, respectively, of the Borel hierarchy of the Baire space.

THEOREM 13.5.3. For every deterministic $\mathbb{N}$-memory ω-automaton $\mathcal{A}$ over the alphabet $\mathbb{N}$, equipped with a parity or Muller acceptance condition, the ω-language $L(\mathcal{A}) \subseteq \mathbb{N}^\omega$ is a Boolean combination of ω-languages in the class Σ_2 of the Borel hierarchy of the Baire space.

We leave it as an open question whether the following problem is decidable:

Given a deterministic $\mathbb{N}$-memory parity automaton $\mathcal{A}$ over the alphabet $\mathbb{N}$, is the ω-language $L(\mathcal{A}) \subseteq \mathbb{N}^\omega$ already in class Σ_1, Π_1, Σ_2, or Π_2 of the Borel hierarchy of the Baire space?

In the Cantor space, a positive answer is provided by Landweber's Theorem [Lan69].

REMARK 13.5.4. By Martin's determinacy theorem [Mar75], Gale-Stewart games with Borel winning conditions are determined, which means that one of the players has a winning strategy. This result holds both

in the Cantor space and in the Baire space. Therefore, every Gale-Stewart game whose winning condition is defined by a deterministic $\mathbb{N}$-memory parity automaton is determined. In Chapter 15 we will present an algorithmic solution for such games.

Let us now turn to *nondeterministic* automata. We have already seen that for each of the considered acceptance types (E, A, Büchi, co-Büchi, Muller, and parity), nondeterministic $\mathbb{N}$-memory ω-automata are strictly more powerful than deterministic ones. The ω-language $L \cdot \{0\} \cdot \mathbb{N}^\omega$ with $L = \{m_1 m_2 m_3 \mid m_1, m_2, m_3 \in \mathbb{N}_{\geq 1} \text{ with } m_1 = m_3 \text{ or } m_2 = m_3\}$ given as a witness for this fact in Corollary 13.3.5 is located on the lowest level of the Borel hierarchy – it is a clopen set. However, nondeterminism does in fact increase the expressive power of $\mathbb{N}$-memory ω-automata beyond the Boolean closure of the class Σ_2.

Proposition 13.5.5. There is an $\mathbb{N}$-memory-Büchi-recognizable ω-language $L \subseteq \mathbb{N}^\omega$ that is in the Borel class Σ_3 of the Baire space but not in the Boolean closure of Σ_2.

As an example we take the ω-language $L = \bigcup_{m \in \mathbb{N}} (\mathbb{N}^* \cdot \{m\})^\omega$ of those sequences in which some number occurs infinitely often. A corresponding $\mathbb{N}$-memory Büchi automaton guesses nondeterministically a position where a number m occurs that reappears infinitely often later. The automaton can memorize the number m to verify later occurrences of the same number, and indicate each of these reoccurrences by switching to an accepting state. The Büchi condition then captures the definition of the language L. The proof that L is a "proper" Σ_3-set is skipped here; it is completely analogous to the proof of Proposition 4 in [CDT02].

14

ℕ-Memory Transducers

14.1 Overview

In this chapter, we introduce ℕ-*memory transducers* over input and output alphabets of the form Σ_{in}^* and Σ_{out}^*, for finite sets Σ_{in} and Σ_{out}. An ℕ-memory transducer is an ℕ-memory automaton with output, in the style of a "Moore automaton": It is equipped with an output function that assigns to each configuration $(p, i) \in Q \times \mathbb{N}$ of the transducer an output symbol $v \in \Sigma_{\text{out}}^*$. A *deterministic* ℕ-memory transducer thus induces a function $f : (\Sigma_{\text{in}}^*)^* \to \Sigma_{\text{out}}^*$ that maps a given input word to the output symbol of the configuration that is reached at the end of the corresponding run.

Analogous to the transition relation, which must be computable by a pebble automaton, the output function of an ℕ-memory transducer must be computable by a *pebble printer*. Pebble printers are defined in Section 14.2. In Section 14.3, we then provide a formal definition of ℕ-memory transducers, and we show that their output functions can equivalently be defined by MSO formulas.

14.2 Preliminaries: Vistas and Pebble Printers

Recall from Section 12.2 that the update of the memorized position in a transition of an ℕ-memory automaton via a micro word $u \in \Sigma_{\text{in}}^*$ can be regarded as a trip (u, i, j), which leads from the old position i to the new position j in the "landscape" given by u. Now consider a micro word $v \in \Sigma_{\text{out}}^*$ that is produced as output by an ℕ-memory transducer in a configuration (p, i). Extending our metaphor, we may view v as a picture of a landscape created by a painter standing at the vantage point i. Thus, we call the pair (i, v) a *vista*.

In this section, we define the class of regular sets of vistas, which coincide with those that can be defined by MSO formulas. Furthermore, we introduce pebble printers as a model of automata that compute vistas, and we show that for functional sets of vistas, pebble printers are expressively equivalent to MSO formulas. This serves as a preparation for the formal definition of ℕ-memory transducers in Section 14.3.

14.2.1 Regular Sets of Vistas

vista

Definition 14.2.1. A *vista* is a pair (i, u) consisting of a number $i \in \mathbb{N}$ and a finite word $u \in \Sigma^*$ over a finite alphabet Σ.

We will mostly be interested in functional sets of vistas.

functional set of vistas

Definition 14.2.2. A set of vistas $V \subseteq \mathbb{N} \times \Sigma^*$ is called *functional* if for every $i \in \mathbb{N}$, there is at most one $u \in \Sigma^*$ such that $(i, u) \in V$.

A vista (i, u) can be represented by the marked ω-word $\mathrm{mark}(\widehat{u}, i)$, which is obtained from the ω-extension of u by augmenting the symbol at position i with a marker (see Definition 12.2.4). Consequently, we can represent sets of vistas as languages of ω-words:

mark(V)

Notation 14.2.3. For a set of vistas $V \subseteq \mathbb{N} \times \Sigma^*$, we let

$$\mathrm{mark}(V) = \{\mathrm{mark}(\widehat{u}, i) \mid (i, u) \in V\}.$$

This allows us to define the notion of regularity for sets of vistas.

regular set of vistas

Definition 14.2.4. A set of vistas $V \subseteq \mathbb{N} \times \Sigma^*$ is called *regular* if the ω-language $\mathrm{mark}(V)$ is regular.

Analogously to the case of trips (see Section 12.2.1), we can represent sets of vistas using MSO formulas.

set of vistas induced by an MSO formula

Definition 14.2.5. Let Σ be a finite alphabet, and let $\varphi(x, \overline{Z})$ be an MSO formula over the structure $\mathcal{N}$, with $\overline{Z} = (Z_a)_{a \in \Sigma}$. The *set of vistas induced by* φ, denoted by $\mathrm{Vistas}(\varphi)$, is defined as follows:

Vistas(φ)

$$\mathrm{Vistas}(\varphi) = \left\{(i, u) \in \mathbb{N} \times \Sigma^* \mid \mathcal{N} \models \varphi[i, (K_a^u)_{a \in \Sigma}]\right\},$$

where for $u = a_1 \dots a_m \in \Sigma^*$ and $a \in \Sigma$, we let

$$K_a^u = \left\{k \in \{1, \dots, m\} \mid a_k = a\right\}.$$

An MSO formula $\varphi(x, \overline{Z})$ inducing a set of vistas V also defines an ω-language, namely $\mathrm{mark}(V)$. Since MSO formulas can define precisely the regular ω-languages (Büchi's Theorem [Büc66]), the sets of vistas induced by MSO formulas are the regular ones:

Proposition 14.2.6. A set of vistas $V \subseteq \mathbb{N} \times \Sigma^*$ is regular if and only if there is an MSO formula $\varphi(x, \overline{Z})$ with $\text{Vistas}(\varphi) = V$.

14.2.2 Pebble Printers

We now define a variation of pebble automata, which we will use to compute sets of vistas.

Definition 14.2.7. A *pebble printer* $\mathcal{C}$ is a tuple $(S, \Sigma, \Delta, S_0, S_f)$, where pebble printer

- S is a finite set of states,
- Σ is a finite output alphabet,
- $\Delta \subseteq (S \setminus S_f) \times \{\bot, \#\} \times \{0, 1\} \times S \times \text{Act}$ is the transition relation, where $\text{Act} = \{\uparrow, \downarrow, \diamond\} \cup \Sigma$ is the set of possible actions,
- $S_0 \subseteq S$ is the set of initial states, and
- $S_f \subseteq S$ is the set of final states.

Intuitively, a pebble printer can be regarded as a pebble automaton on the fixed input word $\bot\#^\omega$ with an additional write-only output tape, which will also be initialized with $\bot\#^\omega$. The printer always moves synchronously on both the input and the output tape. A transition $(s, c, b, s', d) \in \Delta$ with $d \in \{\uparrow, \downarrow, \diamond\}$ allows the printer to move up and down or place a pebble, in the same way as a pebble automaton (see Section 12.2.2). Additionally, by taking a transition with $d \in \Sigma$, the printer can write the symbol d on the output tape at the current position.

While the printer can detect whether the pebble is at the current position, it can never read the content of the output tape. However, it can read the content of the fixed input word $\bot\#^\omega$, which allows the printer to detect whether it is at the beginning of the input and hence of the output tape.

A pebble printer as defined above is called *deterministic* if it only has a single initial state and for every triple $(s, c, b) \in (S \setminus S_f) \times \{\bot, \#\} \times \{0, 1\}$, there is exactly one pair $(s', d) \in S \times \text{Act}$ such that $(s, c, b, s', d) \in \Delta$.

We now provide a formal definition of the behavior of a pebble printer. A *configuration* of a pebble printer $\mathcal{C} = (S, \Sigma, \Delta, S_0, S_f)$ is a tuple (s, h, ℓ, β), consisting of a state $s \in S$, the position $h \in \mathbb{N}$ of the printer, the position $\ell \in \mathbb{N}$ of the pebble, and the content $\beta \in (\Sigma \cup \{\bot, \#\})^\omega$ of the output tape.

A *run* of $\mathcal{C}$ is a finite or infinite sequence of configurations

$$(s_1, h_1, \ell_1, \beta_1)(s_2, h_2, \ell_2, \beta_2)(s_3, h_3, \ell_3, \beta_3)\dots$$

such that for every pair of consecutive configurations $(s_k, h_k, \ell_k, \beta_k)$ and $(s_{k+1}, h_{k+1}, \ell_{k+1}, \beta_{k+1})$, there is a transition $(s_k, c_k, b_k, s_{k+1}, d_k) \in \Delta$ with

$$c_k = \begin{cases} \bot & \text{if } h_k = 0, \\ \# & \text{if } h_k > 0, \end{cases} \qquad \text{and} \qquad b_k = \begin{cases} 1 & \text{if } h_k = \ell_k, \\ 0 & \text{otherwise,} \end{cases}$$

such that one of the following conditions is satisfied:

- $d_k = \uparrow$ and $h_{k+1} = h_k + 1,\ \ell_{k+1} = \ell_k,\ \beta_{k+1} = \beta_k$,
- $d_k = \downarrow$ and $h_k > 0,\ h_{k+1} = h_k - 1,\ \ell_{k+1} = \ell_k,\ \beta_{k+1} = \beta_k$,
- $d_k = \diamond$ and $\ell_{k+1} = h_{k+1} = h_k,\ \beta_{k+1} = \beta_k$,
- $d_k \in \Sigma$ and $h_{k+1} = h_k,\ \ell_{k+1} = \ell_k,\ \beta_{k+1} = \beta_k[h_k/d_k]$,

where $\beta_k[h_k/d_k]$ is the ω-word that is obtained from β_k by replacing the symbol at position h_k with the symbol d_k.

We can use pebble printers to compute vistas: A finite run that starts at position i and changes the content of the output tape from $\bot\#^\omega$ to $\bot u\#^\omega$ (for a finite word $u \in \Sigma^*$) yields the vista (i, u). By convention, we only consider runs that start and end in configurations in which the position of the pebble coincides with the position of the printer. This is formalized in the following definition.

set of vistas computed by a pebble printer

Vistas($\mathcal{C}$)

Definition 14.2.8. Let $\mathcal{C} = (S, \Sigma, \Delta, S_0, S_f)$ be a pebble printer. The *set of vistas computed by* $\mathcal{C}$, denoted by Vistas($\mathcal{C}$), is defined as follows:

$$\begin{aligned} \text{Vistas}(\mathcal{C}) = \big\{(i, u) \in \mathbb{N} \times \Sigma^* \mid \text{for some } s_0 \in S_0, s_f \in S_f, j \in \mathbb{N}, \\ \text{there is a run of } \mathcal{C} \text{ from the configuration} \\ (s_0, i, i, \bot\#^\omega) \text{ to the configuration } (s_f, j, j, \bot u\#^\omega)\big\} \end{aligned}$$

14.2.3 Pebble Printers vs. MSO Formulas

In this subsection, we compare the sets of vistas that can be computed by pebble printers and the regular sets of vistas, which can be defined by MSO formulas. In fact, if we only consider functional sets of vistas, pebble printers and MSO formulas turn out to be expressively equivalent.

We begin by showing that pebble printers can compute all regular sets of vistas.

Proposition 14.2.9. Every regular set of vistas $V \subseteq \mathbb{N} \times \Sigma^*$ can be computed by a pebble printer. Moreover, if V is also functional, then it can be computed by a deterministic pebble printer.

Proof. We prove the two parts of the proposition separately.

First part. Since V is a regular set of vistas, the ω-language $\mathrm{mark}(V)$ is regular (by definition). Furthermore, all ω-words in $\mathrm{mark}(V)$ end with the suffix $\#^\omega$. Thus, we can apply Proposition 12.2.14, which says that $\mathrm{relPref}(\mathrm{mark}(V))$ is a regular language of finite words. Let $\mathcal{D}$ be a DFA recognizing $\mathrm{relPref}(\mathrm{mark}(V))$. Note that for an ω-word $\alpha = \mathrm{mark}(\widehat{u}, i)$ representing a vista (i, u), the shortest relevant prefix $\mathrm{relPref}(\alpha)$ is the prefix that ends at the position $\max\{|u|, i\}$.

Constructing a nondeterministic pebble printer that computes V is easy: Starting at the initial position $i \in \mathbb{N}$, which is also marked by the pebble, the pebble printer moves down to the beginning of its output tape and then walks up again, simulating the run of the DFA $\mathcal{D}$ on a word that the pebble printer chooses nondeterministically, one symbol at a time. In each step, the printer writes the chosen symbol to the current position of the output tape (unless that symbol is $\bot$ or #). When the printer is at the position i marked by the pebble, it must simulate a transition of $\mathcal{D}$ on a nondeterministically chosen "marked symbol", i.e., on a symbol of the form $(a, 1)$. In that case, the symbol to be written is the corresponding "non-marked symbol" a. Whenever the simulation reaches a final state of $\mathcal{D}$, the pebble printer can choose to place the pebble and enter a final state, or to continue the simulation in order to produce a longer output word.

Second part. Now let the set of vistas V not only be regular but also functional. We will construct a *deterministic* pebble printer computing V. To that end, let $\mathcal{D}$ again be a DFA recognizing $\mathrm{relPref}(\mathrm{mark}(V))$. We call a state of $\mathcal{D}$ *productive* if there is a run from that state to a final state.

The desired deterministic pebble printer works as follows: Starting at the initial position $i \in \mathbb{N}$, which is also marked by the pebble, it first walks down to position 0 of its output tape. Then it walks upward again, simulating the DFA $\mathcal{D}$ on all words of length $i - 1$ simultaneously. In other words, while walking from position 0 toward position i, the pebble printer determines the sets $Q_0, Q_1, Q_2, \ldots, Q_{i-1}$ containing the states of $\mathcal{D}$ that are reachable via words of length $0, 1, 2, \ldots, i-1$, respectively (similar to a standard powerset construction). When the printer arrives at the position i, indicated by the pebble, it can then determine the set $\widehat{Q}_i$ of all states of $\mathcal{D}$ that can be reached from some state in Q_{i-1} by a

transition via some "marked" symbol, i.e., a symbol of the form $(a, 1)$. Note that $\widehat{Q}_i$ is the set of states of $\mathcal{D}$ that are reachable via some word of length i that ends with a marked symbol.

Since V is functional, $\widehat{Q}_i$ contains either exactly one productive state or no productive state at all. In the latter case, we let the pebble printer terminate at some position other than the pebble position, to indicate that there is no $u \in \Sigma^*$ with $(i, u) \in V$. On the other hand, if $\widehat{Q}_i$ contains a productive state q, then there is exactly one word $v_q \in \Sigma^*$, uniquely determined by q, that leads from q to some final state of $\mathcal{D}$. We let the pebble printer move upward from the position i, writing the output v_q, and then return to the position i marked by the pebble.

Now the pebble printer still has to write the part of the output from position i downward, which is given by the unique word of length i that leads to the state q. For this purpose, we let $\mathcal{D}^r_q$ be a DFA recognizing the reverse of the set of words that lead $\mathcal{D}$ from its initial state to the state q. At each of the positions $i, i-1, i-2 \ldots$, the pebble printer checks for each possible output symbol (and additionally, for the symbol # indicating positions without output) whether after fixing that output symbol, there would still be an accepting run of $\mathcal{D}^r_q$ for some choice of the remaining output symbols; this check is described below. If this is the case, then the printer writes the currently considered symbol (unless it is #), memorizes the state that is reached by $\mathcal{D}^r_q$ after reading that symbol, and moves the pebble one position downward to continue the process there. After arriving at position 0 and placing the pebble there, the pebble printer enters a final state.

To perform the check whether fixing a certain symbol $a \in \Sigma \cup \{\#\}$ still allows an accepting run of $\mathcal{D}^r_q$, the printer simulates an a-transition (or, for the position i, an $(a, 1)$-transition) of $\mathcal{D}^r_q$ from the currently memorized state and then continues simulating $\mathcal{D}^r_q$ simultaneously for all possible choices of the remaining output symbols (as in the simulation of $\mathcal{D}$ described above), moving downward after each step of the simulation. The check is successful if at position 0, some final state of $\mathcal{D}^r_q$ is reached. □

Since the proofs of Proposition 14.2.6 and Proposition 14.2.9 are constructive, we obtain the following corollary.

Corollary 14.2.10. For every MSO formula $\varphi(x, \overline{Z})$, we can construct a pebble printer $\mathcal{C}$ with $\mathrm{Vistas}(\mathcal{C}) = \mathrm{Vistas}(\varphi)$. Moreover, if $\mathrm{Vistas}(\varphi)$ is functional, then $\mathcal{C}$ can be chosen to be deterministic.

Now we address the reverse direction, from pebble printers to MSO formulas, but we only consider pebble printers that compute functional sets of trips. This includes all deterministic pebble printers, since they can produce at most one output word u for any given position $i \in \mathbb{N}$:

PROPOSITION 14.2.11. For every deterministic pebble printer $\mathcal{C}$, the set of vistas $\mathrm{Vistas}(\mathcal{C})$ is functional.

We show the following proposition.

PROPOSITION 14.2.12. For every pebble printer $\mathcal{C}$ computing a functional set of vistas $\mathrm{Vistas}(\mathcal{C})$, we can construct an MSO formula $\varphi(x, \overline{Z})$ with $\mathrm{Vistas}(\varphi) = \mathrm{Vistas}(\mathcal{C})$.

REMARK 14.2.13. We leave it as an open question whether *all* sets of vistas – even non-functional sets – computed by pebble printers (or a suitable restriction of that model) can be defined by MSO formulas.

Proof of Proposition 14.2.12. In a nutshell, we will express in MSO logic the existence of a run of $\mathcal{C}$ that computes a given vista. As in the case of pebble automata (see Section 12.2.4), we begin by constructing for each pair of states (s, s') of the pebble printer $\mathcal{C}$ an MSO formula $\textsc{UpDown}_{s,s'}(x, x', z)$ expressing that $\mathcal{C}$ has a transition from state s to s' that neither places the pebble nor writes any output but lets the automaton move from position x to x', with the pebble lying on position z. In contrast to pebble automata, the input is always the fixed ω-word $\bot\#^\omega$, so we encode it using quantified variables $Y_\bot$ and $Y_\#$ instead of free variables:

$$\begin{aligned}\textsc{UpDown}_{s,s'}(x, x', z) = \exists Y_\bot, Y_\# \Big(&\textsc{Input}(Y_\bot, Y_\#) \wedge \\ \Big(&\bigvee_{(s,c,0,s',\uparrow)\in\Delta} (Y_c(x) \wedge x \neq z \wedge x' = x + 1) \vee \\ &\bigvee_{(s,c,1,s',\uparrow)\in\Delta} (Y_c(x) \wedge x = z \wedge x' = x + 1) \vee \\ &\bigvee_{(s,c,0,s',\downarrow)\in\Delta} (Y_c(x) \wedge x \neq z \wedge x' = x - 1) \vee \\ &\bigvee_{(s,c,1,s',\downarrow)\in\Delta} (Y_c(x) \wedge x = z \wedge x' = x - 1) \quad \Big) \Big)\end{aligned}$$

with

$$\textsc{Input}(Y_\bot, Y_\#) = \forall y \left(Y_\bot(y) \leftrightarrow y = 0 \ \wedge \ Y_\#(y) \leftrightarrow y \neq 0 \right).$$

Similarly, for $a \in \Sigma$, we represent steps that write the output symbol a by the following formula:

$$\textsc{Write}^a_{s,s'}(x,z) = \exists Y_\bot, Y_\# \Big(\textsc{Input}(Y_\bot, Y_\#) \wedge \Big(\bigvee_{(s,c,0,s',a)\in\Delta} (Y_c(x) \wedge x \neq z) \quad \vee \bigvee_{(s,c,1,s',a)\in\Delta} (Y_c(x) \wedge x = z) \Big) \Big).$$

Next we define the reflexive transitive closure of the relation of all steps represented by the formulas $\textsc{UpDown}_{s,s'}$ and $\textsc{Write}^a_{s,s'}$. More precisely, the following formula $\textsc{FixedPebbleRun}_{s,s'}(x,x',z,Y)$ states that there is a sequence of up/down/write steps leading from state s at position x to state s' at position x' with the pebble lying on position z, with the additional requirement that no output must be written at the positions in the set Y. This latter condition will be important later to determine the last write operation for a given position.

$$\textsc{FixedPebbleRun}_{s,s'}(x,x',z,Y) = \forall (X_r)_{r\in S} \Big(\Big(X_s(x) \wedge \textsc{FixedClosed}\big((X_r)_{r\in S}, z, Y\big) \Big) \rightarrow X_{s'}(x') \Big)$$

with

$$\textsc{FixedClosed}\big((X_r)_{r\in S}, z, Y\big) = \bigwedge_{s,s'\in S} \forall x,x' \Big(\big(X_s(x) \wedge \textsc{UpDown}_{s,s'}(x,x',z) \big) \rightarrow X_{s'}(x') \Big) \wedge \bigwedge_{s,s'\in S} \forall x \Big(\big(X_s(x) \wedge \bigvee_{a\in\Sigma} \textsc{Write}^a_{s,s'}(x,z) \wedge \neg Y(x) \big) \rightarrow X_{s'}(x) \Big).$$

Using $\textsc{FixedPebbleRun}_{s,s'}$, we can now construct the following formula, which represents a sequence of transitions of $\mathcal{C}$ from one pebble placement to the next – that is, a sequence of transitions without a pebble placement, leading from position x to x', with the pebble lying on position x, followed by a placement of the pebble at position x'. Again, writing output is forbidden for the positions in Y.

$$\textsc{PebbleMove}_{s,s'}(x,x',Y) = \bigvee_{r\in S} \big(\textsc{FixedPebbleRun}_{s,r}(x,x',x,Y) \wedge \textsc{PutPebble}_{r,s'}(x',x) \big)$$

with

$$\textsc{PutPebble}_{r,s'}(x,z) = \exists Y_\bot, Y_\# \Big(\textsc{Input}(Y_\bot, Y_\#) \wedge \Big(\bigvee_{(r,c,0,s',\diamond)\in\Delta} (Y_c(x) \wedge x \neq z) \quad \vee \bigvee_{(r,c,1,s',\diamond)\in\Delta} (Y_c(x) \wedge x = z) \Big) \Big).$$

By defining the reflexive transitive closure of the "pebble movement relation" represented by the formulas $\textsc{PebbleMove}_{s,s'}$, we obtain the following formula, which expresses the existence of a run of $\mathcal{C}$ starting with the pebble on the start position x and ending with the pebble on the final position x', with the last transition being a pebble placement transition. Such a run may involve multiple intermediate placements of the pebble along the way. Once again, no output must be written at the positions specified by the set Y.

$$\textsc{PebbleMoveClosure}_{s,s'}(x, x', Y) = \forall (X_r)_{r \in S} \left(\Big(X_s(x) \wedge \textsc{MoveClosed}\big((X_r)_{r \in S}, Y\big) \Big) \rightarrow X_{s'}(x') \right)$$

with

$$\textsc{MoveClosed}\big((X_r)_{r \in S}, Y\big) = \bigwedge_{s,s' \in S} \forall x, x' \left(\big(X_s(x) \wedge \textsc{PebbleMove}_{s,s'}(x, x', Y) \big) \rightarrow X_{s'}(x') \right).$$

Next, we define a formula stating that there is a run of $\mathcal{C}$ that starts in state s at position x with the pebble on position y and ends in state s' at position x' with the pebble on position y'. In contrast to the corresponding formula for pebble automata, we allow the pebble to be on a different position than the printer at the beginning and end. This will be useful below. The set Y again indicates the positions where writing is forbidden.

$$\begin{aligned} \textsc{Run}_{s,s'}\big(x, y, x', y', Y\big) = {} & \big(y' = y \wedge \textsc{FixedPebbleRun}_{s,s'}(x, x', y, Y) \big) \vee \\ & \bigvee_{r_1, r_2, r_3 \in S} \exists z \, \big(\textsc{FixedPebbleRun}_{s,r_1}(x, z, y, Y) \wedge \\ & \qquad \textsc{PutPebble}_{r_1,r_2}(z, y) \wedge \\ & \qquad \textsc{PebbleMoveClosure}_{r_2,r_3}(z, y', Y) \wedge \\ & \qquad \textsc{FixedPebbleRun}_{r_3,s'}(y', x', y', Y) \big). \end{aligned}$$

We can now define $\textsc{RunOutput}_{s,s'}\big(x, x', (Z_a)_{a \in \Sigma}\big)$, which is the crucial formula stating that not only is there a run that starts at position x (with the pebble on the same position x) and ends at position x' (with the pebble on position x'), but also the output that is produced by that run matches the word encoded by the sets $(Z_a)_{a \in \Sigma}$. Note that the printer may overwrite its output multiple times, and the actual output at any given position is only determined by the last write step at that position. Therefore, the formula demands that for each position that occurs in

one of the sets $(Z_a)_{a\in\Sigma}$, there is a run that at some point writes the correct output for that position and never overwrites it afterward.

Note that it suffices to demand the existence of such a run separately for each output position since $\mathcal{C}$ computes a *functional* set of vistas: In that case, all runs that start in the same configuration and lead from an initial state $s \in S_0$ to a final state $s' \in S_f$ produce the same output. Hence, if one of these runs yields the correct output at a given position, then all of them do. (This argument only holds for $s \in S_0$ and $s' \in S_f$, but this is the only case that we will need.)

Additionally, the formula demands that no output is ever written at a position that is not mentioned in $(Z_a)_{a\in\Sigma}$. Again, it suffices to demand the existence of at least one run satisfying this condition – if such a run exists, then the condition holds for all runs.

Formally, we define

$$\begin{aligned}\textsc{RunOutput}_{s,s'}\big(x, x', (Z_a)_{a\in\Sigma}\big) = \exists Y \Bigg(&\forall z \left(Y(z) \leftrightarrow \bigwedge_{a\in\Sigma} \neg Z_a(z)\right) \wedge \\ &\textsc{Run}_{s,s'}(x, x, x', x', Y) \wedge \\ &\forall z \Bigg(\neg Y(z) \rightarrow \bigvee_{r_1,r_2\in S} \exists z' \Big(\textsc{Run}_{s,r_1}(x, x, z, z', Y) \wedge \\ &\qquad\qquad \bigvee_{a\in\Sigma} \Big(\textsc{Write}^a_{r_1,r_2}(z, z') \wedge Z_a(z)\Big) \wedge \\ &\qquad\qquad \textsc{Run}_{r_2,s'}(z, z', x', x', Y \cup \{z\})\Big)\Bigg)\Bigg).\end{aligned}$$

Finally, we obtain the desired MSO formula inducing the set of vistas computed by $\mathcal{C}$ by expressing the existence of a run as defined by $\textsc{RunOutput}_{s,s'}$ from an initial state to a final state, ending at an arbitrary position:

$$\varphi\big(x, (Z_a)_{a\in\Sigma}\big) = \bigvee_{\substack{s_0\in S_0,\\ s_f\in S_f}} \exists x' \, \textsc{RunOutput}_{s_0,s_f}\big(x, x', (Z_a)_{a\in\Sigma}\big). \qquad \square$$

Combining Corollary 14.2.10 and Proposition 14.2.12, we obtain the following result.

Proposition 14.2.14. Let $V \subseteq \mathbb{N} \times \Sigma^*$ be a functional set of vistas. The following are effectively equivalent:

(1) There is a deterministic pebble printer $\mathcal{C}$ with $\mathrm{Vistas}(\mathcal{C}) = V$.
(2) There is a pebble printer $\mathcal{C}$ with $\mathrm{Vistas}(\mathcal{C}) = V$.
(3) There is an MSO formula $\varphi(x, \overline{Z})$ with $\mathrm{Vistas}(\varphi) = V$.

14.3 Definition of ℕ-Memory Transducers

We define transducers over input and output alphabets of the form Σ_{in}^* and Σ_{out}^* by augmenting ℕ-memory automata over Σ_{in}^* with an output function computed by a pebble printer.

Definition 14.3.1. An *ℕ-memory transducer* $\mathcal{A}$ is a tuple of the form $(Q, \Sigma_{\text{in}}^*, \Sigma_{\text{out}}^*, \Delta, \mathit{out}, q_0)$, where

ℕ-memory transducer

- Q is a finite set of states,
- Σ_{in}^* and Σ_{out}^* are infinite alphabets, for finite sets Σ_{in} and Σ_{out},
- $\Delta \subseteq (Q \times \mathbb{N}) \times \Sigma_{\text{in}}^* \times (Q \times \mathbb{N})$ is a pebble-automatic transition relation,
- $\mathit{out}\colon Q \times \mathbb{N} \to \Sigma_{\text{out}}^*$ is a pebble-printable output function as defined below,
- $q_0 \in Q$ is the initial state.

Definition 14.3.2. An output function $\mathit{out}\colon Q \times \mathbb{N} \to \Sigma_{\text{out}}^*$ is *pebble-printable* if there is a pebble printer $\mathcal{C}$ of the form $(S, \Sigma_{\text{out}}, \Delta_{\mathcal{C}}, Q, S_f)$ that computes out in the following sense:

pebble-printable output function

$$\mathit{out}(p, i) = v \quad \Leftrightarrow \quad (i, v) \in \text{Vistas}(\mathcal{C}_p),$$

See Definition 14.2.8.

where $\mathcal{C}_p$ denotes the pebble printer $(S, \Sigma_{\text{out}}, \Delta_{\mathcal{C}}, \{p\}, S_f)$.

The configurations and runs of an ℕ-memory transducer are defined like those of an ℕ-memory automaton (see Section 12.3), and the function *out* specifies the output of the transducer in a given configuration.

We say that an ℕ-memory transducer is *deterministic* if it has a deterministic transition relation Δ, which can be viewed as a transition function $\delta\colon (Q \times \mathbb{N}) \times \Sigma_{\text{in}}^* \to Q \times \mathbb{N}$ in that case. We extend δ to the domain $(Q \times \mathbb{N}) \times (\Sigma_{\text{in}}^*)^*$ in the usual way, by letting $\delta\big((p, i), \varepsilon\big) = (p, i)$ and $\delta\big((p, i), u_1 \dots u_{n+1}\big) = \delta\big(\delta\big((p, i), u_1 \dots u_n\big), u_{n+1}\big)$.

Definition 14.3.3. Let $\mathcal{A} = (Q, \Sigma_{\text{in}}^*, \Sigma_{\text{out}}^*, \delta, \mathit{out}, q_0)$ be a deterministic ℕ-memory transducer over the alphabets Σ_{in}^* and Σ_{out}^*. We say that $\mathcal{A}$ *implements the function* $f_{\mathcal{A}}\colon (\Sigma_{in}^*)^* \to \Sigma_{out}^*$ with

function implemented by a deterministic ℕ-memory transducer

$$f_{\mathcal{A}}(u_1 \dots u_n) = \mathit{out}\big(\delta\big((q_0, 0), u_1 \dots u_n\big)\big).$$

As detailed in Section 12.3, we can use families of MSO formulas to define the transition relations of ℕ-memory automata – and, accordingly, of ℕ-memory transducers. Analogously, we will also use families of MSO formulas to define the output functions of ℕ-memory transducers.

MSO-family-definable output function

Definition 14.3.4. An output function $out\colon Q \times \mathbb{N} \to \Sigma_{\text{out}}^*$ is *MSO-family-definable* if it is defined by a family of MSO formulas $\big(\psi_p(x, \overline{Z})\big)_{p \in Q}$ with $\overline{Z} = (Z_a)_{a \in \Sigma_{\text{out}}}$, meaning that

$$out(p, i) = v \quad \Leftrightarrow \quad (i, v) \in \text{Vistas}(\psi_p).$$

See Definition 14.2.5.

By the following theorem, the two representations of output functions, namely pebble printers and MSO formulas, are interchangeable.

Theorem 14.3.5. Let $out\colon Q \times \mathbb{N} \to \Sigma_{\text{out}}^*$ be an output function. The following are effectively equivalent:

(1) The function *out* is pebble-automatic.
(2) The function *out* is MSO-family-definable.

Proof. This follows from the fact that the functional sets of vistas induced by MSO formulas are exactly those computed by deterministic pebble printers. For a more precise argument, we address the two directions separately.

(1) ⇒ (2): Let *out* be computed by a pebble printer $\mathcal{C}$. For $p \in Q$, consider the pebble printer $\mathcal{C}_p$ obtained from $\mathcal{C}$ by making p the only initial state. Note that $\text{Vistas}(\mathcal{C}_p)$ is functional – otherwise, *out* would not be a well-defined function. By Proposition 14.2.14, we can construct an MSO formula $\psi_p(x, \overline{Z})$ such that $\text{Vistas}(\psi_p) = \text{Vistas}(\mathcal{C}_p)$. Hence we obtain a family of MSO formulas $(\psi_p)_{p \in Q}$ inducing the same output function *out* as the pebble printer $\mathcal{C}$.

(2) ⇒ (1): Let *out* be induced by a family of MSO formulas $(\psi_p)_{p \in Q}$. Note that $\text{Vistas}(\psi_p)$ is functional for each $p \in Q$ – otherwise, *out* would not be a well-defined function. By Corollary 14.2.10, we can construct for each $p \in Q$ a deterministic pebble printer $\mathcal{C}^p = (S^p, \Sigma_{\text{out}}, \Delta^p, \{s_0^p\}, S_f^p)$ such that $\text{Vistas}(\mathcal{C}^p) = \text{Vistas}(\psi_p)$. We combine all these pebble printers in a single pebble printer $\mathcal{C} = (S, \Sigma_{\text{out}}, \Delta_{\mathcal{C}}, Q, S_f)$ that behaves like $\mathcal{C}^p$

when it is started in state $p \in Q$. Formally, we set $S = Q \uplus \biguplus_{p \in Q} S^p$, and $S_f = \biguplus_{p \in Q} S^p_f$, and

$$\Delta_{\mathcal{C}} \;=\; \bigcup_{p \in Q} \Delta^p \;\cup\; \bigcup_{p \in Q} \big\{(p, c, b, s, d) \mid (s^p_0, c, b, s, d) \in \Delta^p\big\}.$$

The pebble printer $\mathcal{C}$ induces the same output function *out* as the formulas $(\psi_p)_{p \in Q}$. □

Note that each pebble-printable output function can be computed by a pebble printer that is deterministic from each of its initial states, since the pebble printer $\mathcal{C}$ constructed in the proof above always has that property.

15

Solving Games over Infinite Alphabets

15.1 Main Result and Overview

In this chapter, we present the main result of Part II of this thesis: an algorithmic solution of Gale-Stewart games with winning conditions defined by deterministic $\mathbb{N}$-memory parity automata.

Gale-Stewart game $\mathcal{G}(L)$

Recall that in a Gale-Stewart game, Player I and Player II alternately choose symbols from a fixed alphabet. In our case, we are dealing with infinite alphabets of the form Σ^*, where Σ is a finite set, so the symbols that can be chosen by the two players are finite words. A *play* of the game is therefore a sequence $\alpha \in (\Sigma^*)^\omega$ of finite words, and the winning condition is an ω-language $L \subseteq (\Sigma^*)^\omega$. A play α is won by Player II if $\alpha \in L$, otherwise it is won by Player I. We write $\mathcal{G}(L)$ to denote the Gale-Stewart game with the winning condition L.

play

strategy

In such a game over the alphabet Σ^*, a *strategy* for Player I is a function $\tau_{\mathrm{I}} \colon (\Sigma^*)^* \to \Sigma^*$, and a strategy for Player II is a function $\tau_{\mathrm{II}} \colon (\Sigma^*)^+ \to \Sigma^*$.

winning strategy

We call τ_{II} a *winning strategy* for Player II in the game $\mathcal{G}(L)$ if Player II is guaranteed to win when using this strategy, that is, if $\alpha \in L$ for every play $\alpha = u_1 v_1 u_2 v_2 \ldots \in (\Sigma^*)^\omega$ with $v_n = \tau_{\mathrm{II}}(u_1 u_2 \ldots u_n)$ for all $n \in \mathbb{N}_{\geq 1}$. Analogously, τ_{I} is a winning strategy for Player I if $\alpha \notin L$ for every play $\alpha = u_1 v_1 u_2 v_2 \ldots \in (\Sigma^*)^\omega$ with $u_n = \tau_{\mathrm{I}}(v_1 v_2 \ldots v_{n-1})$ for all $n \in \mathbb{N}_{\geq 1}$. If a player has a winning strategy, then we say that he wins the game.

Our main result can be formulated as follows.

Theorem 15.1.1. Given a deterministic $\mathbb{N}$-memory parity automaton $\mathcal{A}$ recognizing an ω-language $L(\mathcal{A}) \subseteq (\Sigma^*)^\omega$, we can

- decide who wins the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$, and
- construct a deterministic $\mathbb{N}$-memory transducer implementing a winning strategy for the winner.

Note that all deterministically $\mathbb{N}$-memory-parity-recognizable winning conditions are Borel sets, and hence the games considered here are determined, meaning that one of the two players has a winning strategy (see Remark 13.5.4).

To prove Theorem 15.1.1, we proceed in three steps, following a pattern known from the classical solution of Church's Synthesis Problem for finite alphabets.

1. Convert the given automaton $\mathcal{A}$ into a *parity game* $G_{\mathcal{A}}$, which is essentially played on the "transition graph" of $\mathcal{A}$. In contrast to the classical setting, the game graph is infinite here.
2. Determine the winner of the parity game $G_{\mathcal{A}}$ and thus of the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$.
3. Compute a positional winning strategy for the parity game $G_{\mathcal{A}}$, and transform that strategy into a transducer implementing a winning strategy for $\mathcal{G}(L(\mathcal{A}))$.

The basics on parity games are presented in Section 15.2. In Section 15.3, we define the parity game $G_{\mathcal{A}}$ associated with a given deterministic $\mathbb{N}$-memory parity automaton $\mathcal{A}$, and we show that $G_{\mathcal{A}}$ is MSO-definable in a product structure of the form $Q \times \mathcal{N}$, for a finite set Q.

We then describe in Section 15.4 how to determine the winner of the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$, using the fact that the so-called winning regions of the parity game $G_{\mathcal{A}}$ are MSO-definable. In Section 15.5, we show how to construct an $\mathbb{N}$-memory transducer implementing a winning strategy for the Gale-Stewart game, based on an MSO-definable positional winning strategy for the parity game. The underlying theorems about MSO-definable parity games will be proved separately in Chapter 16.

15.2 Preliminaries: Games on Graphs

As described above, we will solve Gale-Stewart games over infinite alphabets by solving corresponding games on graphs.

game graph

Definition 15.2.1. A *game graph* is a tuple (V, V_0, V_1, E) where the (possibly infinite) set of *positions* $V = V_0 \dot\cup V_1$ is partitioned into positions owned by Player 0 and Player 1, respectively, and $E \subseteq V \times V$ is a set of edges. To avoid technical difficulties, we require that for all $u \in V$, there is at least one outgoing edge $(u, v) \in E$.

parity game, priority function

Definition 15.2.2. A *parity game* $G = (V, V_0, V_1, E, c)$ consists of a game graph (V, V_0, V_1, E) and a *priority function* $c \colon V \to [k]$ for some $k \in \mathbb{N}$.

Remark 15.2.3. Recall that $[k] = \{0, \dots, k\}$. In a logical context, we will often view a parity game as a relational structure $(V, V_0, V_1, E, (C_\ell)_{\ell \in [k]})$, where the priority function $c\colon V \to [k]$ of the game is represented by *priority sets* $C_\ell = \{v \in V \mid c(v) = \ell\}$. We sometimes call such a structure a *game structure*.

priority set, game structure

A parity game $G = (V, V_0, V_1, E, c)$ between Player 0 and Player 1 is played as follows, starting at some given initial position: The player who owns the current position u selects an outgoing edge $(u, v) \in E$, and the game proceeds from the target position v. This continues ad infinitum, resulting in a sequence of positions $\varrho = v_0 v_1 v_2 \dots$ with $(v_i, v_{i+1}) \in E$ for all $i \in \mathbb{N}$, called a *play*. A play ϱ of the game G is won by Player 0 if

play

$$\max\{c(v) \mid \text{position } v \text{ is visited infinitely often in } \varrho\}$$

is *even*, otherwise it is won by Player 1.

A *strategy* for Player $i \in \{0, 1\}$ on the game graph (V, V_0, V_1, E) is a function $\sigma\colon V^* V_i \to V$ assigning to each finite play prefix $v_0 \dots v_n$ with $v_n \in V_i$ a position $v_{n+1} = \sigma(v_0 \dots v_n)$ to which the player should move next, satisfying $(v_n, v_{n+1}) \in E$.

strategy

The strategy σ is called *positional* if it only depends on the last position, that is, if $\sigma(wv) = \sigma(w'v)$ for all $w, w' \in V^*$ and $v \in V_i$. In that case, we simply treat the strategy as a function $\sigma\colon V_i \to V$.

positional strategy

A strategy σ for Player i is called a *winning strategy* in the parity game G from a position $v_0 \in V$ if Player i wins all plays starting at v_0 when playing according to σ.

winning strategy

The *winning region* of Player i in the game G is the set $W_i \subseteq V$ of those positions from which Player i has a winning strategy.

winning region

A strategy σ for Player i in G is called a *uniform winning strategy* if it is a winning strategy from all positions in the winning region of Player i.

uniform winning strategy

It is well known that parity games are determined with positional winning strategies [EJ91], that is, from each position, exactly one of the players has a positional winning strategy. Moreover, each player has a uniform winning strategy on his winning region (see [Zie98]).

15.3 From ℕ-Memory Parity Automata to Parity Games

We will now define the transformation of a given deterministic ℕ-memory parity automaton $\mathcal{A}$ recognizing $L(\mathcal{A}) \subseteq (\Sigma^*)^\omega$ into a parity game $G_\mathcal{A}$. The positions of that game will be the configurations of the automaton and the edges will correspond to the transitions of the automaton. Note that for each play of the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$, there is exactly

one corresponding run of the automaton $\mathcal{A}$ – and hence exactly one play of the parity game $G_{\mathcal{A}}$.

To reflect the alternation between the two players of the Gale-Stewart game also in the parity game, we require that the state set Q of the automaton is partitioned into sets Q_0 and Q_1 such that the initial state is in Q_1 and all transitions from Q_1 lead to Q_0 and vice versa. The given automaton can always be brought into this form using two copies of its original state set. Formally, given an automaton $(P, \Sigma^*, \delta, p_0, c)$, where δ is defined by a family of MSO formulas $(\varphi_{p,p'})_{p,p' \in P}$, we can construct an equivalent automaton $(Q_0 \cup Q_1, \Sigma^*, \delta_{\text{new}}, q_0, c_{\text{new}})$ with the required property as follows: We let $Q_0 = \{0\} \times P$ and $Q_1 = \{1\} \times P$, the initial state is $q_0 = (1, p_0)$, the priorities are given by $c_{\text{new}}\big((b,p)\big) = c(p)$, and δ_{new} is defined by the following formulas, for $(b,p), (b',p') \in Q_0 \cup Q_1$:

$$\varphi^{\text{new}}_{(b,p),(b',p')}(\overline{Z}, x, y) = \begin{cases} \varphi_{p,p'}(\overline{Z}, x, y) & \text{if } b \neq b', \\ \texttt{false} & \text{otherwise.} \end{cases}$$

From now on, we will therefore assume that $\mathcal{A}$ has the required format.

$\mathbb{N}$-memory parity game

Definition 15.3.1. A given deterministic $\mathbb{N}$-memory parity automaton $\mathcal{A} = (Q, \Sigma^*, \delta, q_0, c_{\mathcal{A}})$, equipped with a partition $Q = Q_0 \uplus Q_1$ of its set of states, induces the parity game $G_{\mathcal{A}} = (V, V_0, V_1, E, c)$ with

- $V = Q \times \mathbb{N}$,
- $V_0 = Q_0 \times \mathbb{N}$,
- $V_1 = Q_1 \times \mathbb{N}$,
- $E = \big\{\big((p,i),(q,j)\big) \mid \delta\big((p,i),u\big) = (q,j) \text{ for some } u \in \Sigma^*\big\}$,
- $c \colon V \to [k]$ with $c\big((p,i)\big) = c_{\mathcal{A}}(p)$.

We call such a game $G_{\mathcal{A}}$ an $\mathbb{N}$-*memory parity game*.

As indicated above, plays in $\mathcal{G}(L(\mathcal{A}))$ can be translated to plays in $G_{\mathcal{A}}$. For every play $\alpha = u_1 v_1 u_2 v_2 \ldots \in (\Sigma^*)^\omega$ in $\mathcal{G}(L(\mathcal{A}))$, there is a corresponding run $\varrho = (q_0, i_0)(q_1, i_1)(q_2, i_2)\ldots$ of $\mathcal{A}$ on α (uniquely defined, since $\mathcal{A}$ is deterministic). By definition of $G_{\mathcal{A}}$, this run can be viewed as a play in $G_{\mathcal{A}}$. The play α in $\mathcal{G}(L(\mathcal{A}))$ is won by Player II if and only if the corresponding run ϱ is accepting – that is, if and only if the play ϱ in $G_{\mathcal{A}}$ is won by Player 0.

This allows us to transform winning strategies for Player 0 or Player 1 in the parity game $G_{\mathcal{A}}$ into winning strategies for Player II or Player I,

respectively, in the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$. We treat here the case of a winning strategy for Player 1, the case for Player 0 is analogous. More specifically, let $\sigma\colon (Q_1 \times \mathbb{N}) \to (Q_0 \times \mathbb{N})$ be a positional winning strategy for Player 1 in $G_{\mathcal{A}}$. First, we fix a function $\mathit{microword}_\sigma\colon (Q_1 \times \mathbb{N}) \to \Sigma^*$ that maps a given configuration (p, i) to a micro word u such that $\delta\big((p,i),u\big) = \sigma\big((p,i)\big)$. We may choose u to be the least micro word with that property, with respect to the length-lexicographic order.

Note: The length-lexicographic order is a well-order, i.e., every nonempty set of micro words has a least element.

Now we can define a strategy $\tau\colon (\Sigma^*)^* \to \Sigma^*$ for Player I in $\mathcal{G}(L(\mathcal{A}))$ as follows, for $v_1, \ldots, v_n \in \Sigma^*$:

$$\tau(v_1 \ldots v_n) = \mathit{microword}_\sigma\big(\,\delta\big((q_0,0),\, u_1 v_1 u_2 v_2 \ldots u_n v_n\big)\,\big)$$

where $u_1 = \tau(\varepsilon)$ and $u_\ell = \tau(v_1 v_2 \ldots v_{\ell-1})$ for $\ell \in \{2, \ldots, n\}$.

To see that this yields a winning strategy for Player I in $\mathcal{G}(L(\mathcal{A}))$, consider a play $\alpha = u_1 v_1 u_2 v_2 \ldots \in (\Sigma^*)^\omega$ of $\mathcal{G}(L(\mathcal{A}))$ that adheres to the strategy τ. The unique run ϱ of $\mathcal{A}$ on α can be regarded as a play in $G_{\mathcal{A}}$, and by construction of τ, this play ϱ conforms to the original strategy σ. Since σ is a winning strategy for Player 1, ϱ violates the parity condition of the game $G_{\mathcal{A}}$ and hence of $\mathcal{A}$, so the play α is rejected by $\mathcal{A}$ and therefore won by Player I.

Analogously, winning strategies for Player 0 in $G_{\mathcal{A}}$ can be converted into winning strategies for Player II in $\mathcal{G}(L(\mathcal{A}))$, so we obtain the following proposition.

Proposition 15.3.2. Let $\mathcal{A}$ be a deterministic ℕ-memory parity automaton with initial state q_0. Player I wins the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$ if and only if Player 1 wins the ℕ-memory parity game $G_{\mathcal{A}}$ from the initial position $(q_0, 0)$.

Since the game structure $G_{\mathcal{A}}$ is derived from an ℕ-memory automaton, it can be defined using MSO formulas over the structure $Q \times \mathcal{N}$ (see Definition 11.3.1), which will be instrumental in solving the game.

Proposition 15.3.3. Let $G_{\mathcal{A}}$ be the ℕ-memory parity game induced by a deterministic ℕ-memory parity automaton $\mathcal{A}$ with a state set Q. The game structure $G_{\mathcal{A}}$ is MSO-definable in $Q \times \mathcal{N}$.

Proof. Let $G_{\mathcal{A}} = (Q \times \mathbb{N}, Q_0 \times \mathbb{N}, Q_1 \times \mathbb{N}, E, (C_\ell)_{i \in [n]})$ be the game structure induced by the deterministic ℕ-memory parity automaton $\mathcal{A}$. Let $\big(\varphi_{p,q}(\overline{Z}, x, y)\big)_{p,q \in Q}$ be a family of MSO formulas that defines the transition function of $\mathcal{A}$.

The edge relation E of $G_{\mathcal{A}}$ is MSO-family-definable in $\mathcal{N}$ by the formulas $\varphi'_{p,q}(x,y) = \exists \overline{Z}\ \varphi_{p,q}(\overline{Z},x,y)$, for $p,q \in Q$. By Proposition 11.3.3, the relation E is therefore MSO-definable in $Q \times \mathcal{N}$.

The sets $V_0 = Q_0 \times \mathbb{N}$ and $V_1 = Q_1 \times \mathbb{N}$ can be defined in $Q \times \mathcal{N}$ by $\varphi_{V_0}(x) = \bigvee_{p \in Q_0} P_p(x)$ and $\varphi_{V_1}(x) = \bigvee_{p \in Q_1} P_p(x)$, respectively.

Similarly, each of the sets $C_\ell = \{(p,i) \in Q \times \mathbb{N} \mid c(p) = \ell\}$ can be defined in $Q \times \mathcal{N}$ by an MSO formula $\varphi_{C_\ell}(x) = \bigvee_{p \in c_{\mathcal{A}}^{-1}(\ell)} P_p(x)$. □

15.4 Determining the Winner

In this section, we show that the winner of a Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$ defined by a deterministic $\mathbb{N}$-memory automaton $\mathcal{A}$ can be determined effectively. According to Proposition 15.3.2, the winner of $\mathcal{G}(L(\mathcal{A}))$ can be found by determining the winner of the corresponding $\mathbb{N}$-memory parity game $G_{\mathcal{A}}$ from position $(q_0, 0)$, so it suffices to show that the latter can be done effectively. Therefore, we now focus on the parity game $G_{\mathcal{A}} = (Q \times \mathbb{N}, V_0, V_1, E, c)$.

We use the following theorem, which we shall prove in Chapter 16. Recall that for a finite set Γ, we write $\mathcal{T}_\Gamma$ to denote the infinite Γ-branching tree, with the set of nodes Γ^* (Definition 11.4.1).

Theorem 16.1.1. Let Q and Γ be finite sets. Let G be a parity game over the set of positions $Q \times \Gamma^*$ such that the structure G is MSO-definable in $Q \times \mathcal{T}_\Gamma$. The winning regions of both players are MSO-definable in $Q \times \mathcal{T}_\Gamma$, and we can construct corresponding MSO formulas.

The structure $\mathcal{N} = (\mathbb{N}, \mathrm{Succ})$ is isomorphic to the Γ-branching tree $\mathcal{T}_\Gamma$ for $\Gamma = \{1\}$ (see Remark 11.4.2), so we have the following corollary.

Corollary 15.4.1. Let Q be a finite set. Let G be a parity game over the set of positions $Q \times \mathbb{N}$ such that the structure G is MSO-definable in $Q \times \mathcal{N}$. The winning regions of both players are MSO-definable in $Q \times \mathcal{N}$, and we can construct corresponding MSO formulas.

By Proposition 15.3.3, the structure $G_{\mathcal{A}}$ is MSO-definable in $Q \times \mathcal{N}$. Thus, we can construct an MSO formula $\varphi_0(x)$ over the structure $Q \times \mathcal{N}$ that defines the winning region of Player 0 for the game $G_{\mathcal{A}}$. We can then determine the winner of the game $G_{\mathcal{A}}$ from the initial position $(q_0, 0)$ by checking whether

$$Q \times \mathcal{N} \models \exists x \big(\varphi_{\mathrm{init}}(x) \wedge \varphi_0(x)\big),$$

where $\varphi_{\text{init}}(x) = P_{q_0}(x) \wedge \neg\exists y \left(\text{Succ}'(y, x)\right)$ defines the position $(q_0, 0)$.

As we have seen in Section 11.4, the MSO theory of $Q \times \mathcal{N}$ is decidable (Proposition 11.4.4), so this check can be performed algorithmically. This concludes the proof of the first part of Theorem 15.1.1, stating that the winner of the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$ can be determined effectively.

15.5 Constructing a Transducer

We now turn to the second part of Theorem 15.1.1. Given a deterministic $\mathbb{N}$-memory parity automaton $\mathcal{A}$, we want to construct a deterministic $\mathbb{N}$-memory transducer implementing a winning strategy for the winner of the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$. We proceed in two steps:

1. Compute an MSO-definable positional winning strategy for the respective player in the $\mathbb{N}$-memory parity game $G_{\mathcal{A}}$.
2. Transform this winning strategy for $G_{\mathcal{A}}$ into a winning strategy for $\mathcal{G}(L(\mathcal{A}))$ in the form of a transducer.

For step 1, we use the following theorem, which is proved in Chapter 16.

Theorem 16.1.3. Let Q and Γ be finite sets. Let G be a parity game over the set of positions $Q \times \Gamma^*$ such that the structure G is MSO-definable in $Q \times \mathcal{T}_\Gamma$. We can construct for each player a uniform positional winning strategy (on his winning region) that is MSO-definable in $Q \times \mathcal{T}_\Gamma$.

Recall that the structure $\mathcal{N}$ is isomorphic to $\mathcal{T}_\Gamma$ for $\Gamma = \{1\}$ (see Remark 11.4.2), so we obtain the following corollary.

Corollary 15.5.1. Let Q be a finite set. Let G be a parity game over the set of positions $Q \times \mathbb{N}$ such that the structure G is MSO-definable in $Q \times \mathcal{N}$. We can construct for each player a uniform positional winning strategy (on his winning region) that is MSO-definable in $Q \times \mathcal{N}$.

Since the game structure $G_{\mathcal{A}}$ is MSO-definable in $Q \times \mathcal{N}$ (Proposition 15.3.3), we can thus construct an MSO formula $\varphi^{\text{strat}}(x, y)$, over the structure $Q \times \mathcal{N}$, that defines a uniform positional winning strategy for Player 0 in the game $G_{\mathcal{A}}$. In particular, this is a winning strategy from the initial position $(q_0, 0)$.

Now we address step 2, the construction of an $\mathbb{N}$-memory transducer from this MSO-definable winning strategy, which concludes the proof of Theorem 15.1.1.

Proposition 15.5.2. Let $\mathcal{A}$ be a deterministic $\mathbb{N}$-memory parity automaton with initial state q_0. Given an MSO-definable positional winning strategy for Player 1 (Player 0) in the $\mathbb{N}$-memory parity game $G_{\mathcal{A}}$ from the initial position $(q_0, 0)$, we can construct a deterministic $\mathbb{N}$-memory transducer implementing a winning strategy for Player I (Player II, respectively) in the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$.

Proof. Intuitively, the transducer will read the sequence of micro words chosen by the opponent in the game $\mathcal{G}(L(\mathcal{A}))$ and simulate the corresponding play of $G_{\mathcal{A}}$ according to the given winning strategy. It will output in each step a micro word that is "consistent" with that winning strategy. We first consider the construction of a transducer for Player I from an MSO-definable winning strategy for Player 1, as it allows for a slightly simpler exposition. The case of Player II is addressed afterward.

Reminder: the strategy for Player I. Let $\mathcal{A} = (Q, \Sigma^*, \delta, q_0, c_{\mathcal{A}})$ be the deterministic $\mathbb{N}$-memory parity automaton inducing the parity game $G_{\mathcal{A}}$, with $Q = Q_0 \cup Q_1$. Let $\sigma\colon (Q_1 \times \mathbb{N}) \to (Q_0 \times \mathbb{N})$ be an MSO-definable positional winning strategy for Player 1 from $(q_0, 0)$. As detailed in Section 15.3, this strategy σ induces the following winning strategy $\tau\colon (\Sigma^*)^* \to \Sigma^*$ for Player I in the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$:

$$\tau(v_1 \dots v_n) = \mathit{microword}_\sigma\big(\,\delta\big((q_0, 0),\, u_1 v_1 u_2 v_2 \dots u_n v_n\big)\,\big)$$

where $u_1 = \tau(\varepsilon)$ and $u_\ell = \tau(v_1 v_2 \dots v_{\ell-1})$ for $\ell \in \{2, \dots, n\}$. Recall that $\mathit{microword}_\sigma$ maps a given configuration $(p, i) \in Q_1 \times \mathbb{N}$ to a micro word u such that $\delta\big((p, i), u\big) = \sigma\big((p, i)\big)$, specifically the least such u with respect to the length-lexicographic order.

The transducer for Player I. We will now construct a deterministic $\mathbb{N}$-memory transducer $\mathcal{B}$ implementing this strategy τ for Player I. While reading a sequence of micro words $v_1 v_2 v_3 \dots$, this transducer simulates $\mathcal{A}$ on the sequence $u_1 v_1 u_2 v_2 \dots$ with u_ℓ as defined above, and the output in a configuration (p, i) is given by $\mathit{microword}_\sigma\big((p, i)\big)$. Note that a single step of the transducer via a micro word v_ℓ emulates two steps of $\mathcal{A}$, via u_ℓ and v_ℓ, as illustrated in Figure 12.

For the formal construction of the transducer, let $\varphi^{\mathrm{strat}}(x, y)$ be an MSO formula over $Q \times \mathcal{N}$ defining the strategy σ for the game $G_{\mathcal{A}}$. We first apply Proposition 11.3.3 to translate $\varphi^{\mathrm{strat}}(x, y)$ into a family of MSO formulas $\big(\varphi^{\mathrm{strat}}_{p,q}(x, y)\big)_{p,q \in Q}$ defining σ in the structure $\mathcal{N}$. Furthermore, let $\big(\varphi_{p,q}(\overline{Z}, x, y)\big)_{p,q \in Q}$ be a family of MSO formulas defining the transition function δ of $\mathcal{A}$.

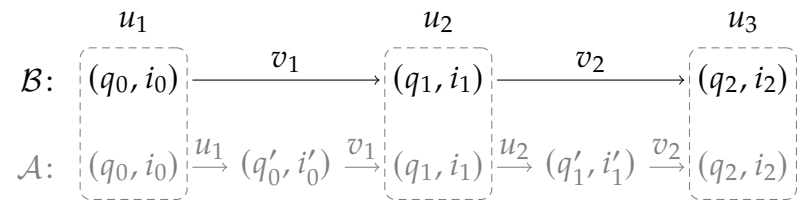


FIGURE 12: Illustration of a run of the transducer $\mathcal{B}$ implementing a strategy for Player I. Output is indicated above the configurations. The simulated run of the automaton $\mathcal{A}$ is shown in gray.

We can now define the desired $\mathbb{N}$-memory transducer as

$$\mathcal{B} = \big(Q_1, \Sigma^*, \Sigma^*, \delta_{\mathcal{B}}, out_{\mathcal{B}}, q_0\big),$$

where $\delta_{\mathcal{B}}$ is defined by the family of MSO formulas $(\psi^{\text{trans}}_{p,q})_{p,q\in Q_1}$ with

$$\psi^{\text{trans}}_{p,q}(\overline{Z}, x, y) = \bigvee_{p'\in Q_0} \exists x' \, \big(\varphi^{\text{strat}}_{p,p'}(x, x') \wedge \varphi_{p',q}(\overline{Z}, x', y)\big),$$

and $out_{\mathcal{B}} = microword_\sigma$ is defined by $(\psi^{\text{out}}_p)_{p\in Q_1}$ with

$$\psi^{\text{out}}_p(x, \overline{Z}) = \bigvee_{p'\in Q_0} \exists x' \left(\varphi^{\text{strat}}_{p,p'}(x, x') \wedge \textsc{LeastWord}_{p,p'}(\overline{Z}, x, x')\right).$$

The auxiliary formula $\textsc{LeastWord}_{p,p'}(\overline{Z}, x, x')$ expresses that the micro word $u \in \Sigma^*$ represented by $\overline{Z} = (Z_a)_{a\in\Sigma}$ is the least micro word, with respect to the length-lexicographic order, such that $\varphi_{p,p'}(\overline{Z}, x, x')$ is satisfied. It can be defined as follows, where $\prec$ is an arbitrarily chosen total order on the elements of the finite set Σ:

$$\begin{aligned}
&\textsc{LeastWord}_{p,p'}\big((Z_a)_{a\in\Sigma}), x, x'\big) = \varphi_{p,p'}\big((Z_a)_{a\in\Sigma}), x, x'\big) \wedge \\
&\quad \forall (Z'_a)_{a\in\Sigma} \Bigg(\varphi_{p,p'}\big((Z'_a)_{a\in\Sigma}), x, x'\big) \rightarrow \\
&\qquad \exists z \Bigg(\forall z' \Big(z' < z \rightarrow \bigwedge_{a\in\Sigma} Z_a(z') \leftrightarrow Z'_a(z')\Big) \wedge \\
&\qquad\quad \Big(\bigwedge_{a\in\Sigma} \neg Z_a(z) \;\vee \bigvee_{\substack{a,b\in\Sigma,\\ a\prec b}} \big(Z_a(z) \wedge Z'_b(z)\big)\Big)\Bigg)\Bigg).
\end{aligned}$$

This concludes the construction of the transducer $\mathcal{B}$ implementing a winning strategy for Player I in the Gale-Stewart game $\mathcal{G}(L(\mathcal{A}))$.

The case of Player II. Let us now address the other case, where Player 0 has an MSO-definable winning strategy in $G_{\mathcal{A}}$ from $(q_0, 0)$ and we want to build a transducer implementing a winning strategy for Player II in $\mathcal{G}(L(\mathcal{A}))$.

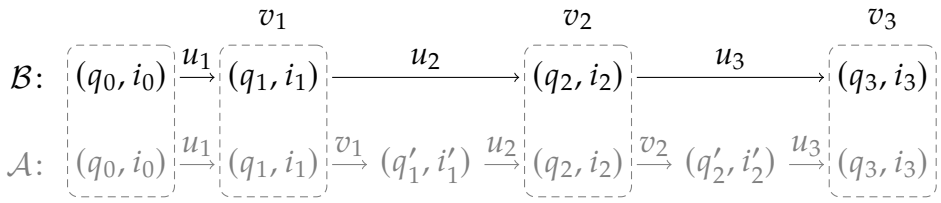


Figure 13: Illustration of a run of the transducer $\mathcal{B}$ implementing a strategy for Player II. Output is indicated above the configurations. The simulated run of the automaton $\mathcal{A}$ is shown in gray.

The construction is analogous to the one for Player I, except that in this case, the first output needs to be produced only after the first input has already been read. Hence the transducer starts by emulating a *single* step of $\mathcal{A}$ via the first input and then continues as described above. This is illustrated in Figure 13. The output of the transducer in the initial configuration is simply ignored. □

16

MSO-Definable Games and Strategies

16.1 Results and Overview

The subject of this chapter are parity games (see Definition 15.2.2) with positions of the form $(q, w) \in Q \times \Gamma^*$ for finite sets Q and Γ. In particular, we consider parity games over the set of positions $Q \times \Gamma^*$ that are MSO-definable in $Q \times \mathcal{T}_\Gamma$, where $\mathcal{T}_\Gamma$ is the infinite Γ-branching tree (see Definition 11.4.1). We will show that in such games, the winning regions of the two players are MSO-definable and we can construct MSO-definable uniform positional winning strategies. These results have already been announced in Section 15.4 and Section 15.5, respectively.

First, let us prove the MSO-definability of the winning regions:

Theorem 16.1.1. Let Q and Γ be finite sets. Let G be a parity game over the set of positions $Q \times \Gamma^*$ such that the structure G is MSO-definable in $Q \times \mathcal{T}_\Gamma$. The winning regions of both players are MSO-definable in $Q \times \mathcal{T}_\Gamma$, and we can construct corresponding MSO formulas.

In [Walo2] it is shown that the winning regions of the two players in any parity game (not necessarily an MSO-definable one) can be defined by MSO formulas over the structure G:

Proposition 16.1.2 ([Walo2]). For a given number of priorities $k \in \mathbb{N}$, we can construct MSO formulas $\varphi_0(x)$ and $\varphi_1(x)$ such that for every parity game G with k priorities, the winning regions of the two players are defined in the structure G by $\varphi_0(x)$ and $\varphi_1(x)$, respectively.

For a parity game G that is MSO-definable in $Q \times \mathcal{T}_\Gamma$, we can transform $\varphi_0(x)$ and $\varphi_1(x)$ into MSO formulas $\psi_0(x)$ and $\psi_1(x)$ defining the winning regions in the structure $Q \times \mathcal{T}_\Gamma$ instead of G (by Proposition 11.2.3). This concludes the proof of Theorem 16.1.1.

Now we turn to the main result of this chapter, concerning MSO-definable winning strategies.

THEOREM 16.1.3. Let Q and Γ be finite sets. Let G be a parity game over the set of positions $Q \times \Gamma^*$ such that the structure G is MSO-definable in $Q \times \mathcal{T}_\Gamma$. We can construct for each player a uniform positional winning strategy (on his winning region) that is MSO-definable in $Q \times \mathcal{T}_\Gamma$.

REMARK ON TERMINOLOGY 16.1.4. Throughout this chapter, we are interested in game structures and relations (in particular, strategies) that are MSO-definable in the structure $Q \times \mathcal{T}_\Gamma$. In this context, for the sake of readability, we will use the term *"MSO-definable"* as an abbreviation for *"MSO-definable in* $Q \times \mathcal{T}_\Gamma$*"*, unless a different structure is explicitly specified.

In the following sections, we present a proof of Theorem 16.1.3.

- We start, in Section 16.2, by constructing MSO-definable winning strategies in parity games on simpler graphs, namely *pushdown graphs*. We recall a result by Kupferman, Piterman and Vardi [KPV10], which allows us to construct winning strategies in such games in the form of finite automata with output, and adapt it to obtain a strategy defined by an MSO formula.
- In Section 16.3, we then show how to construct MSO-definable strategies for parity games on more general graphs, namely *prefix-recognizable graphs*. To that end, we reduce these prefix-recognizable parity games to pushdown parity games, using a technique due to Cachat [Cac03].
- Finally, in Section 16.4, we adopt a proof by Blumensath [Blu01] to show that the game graphs of MSO-definable games in the sense of Theorem 16.1.3 are in fact prefix-recognizable, so we can indeed construct MSO-definable winning strategies for these games.

This approach is illustrated in Figure 14.

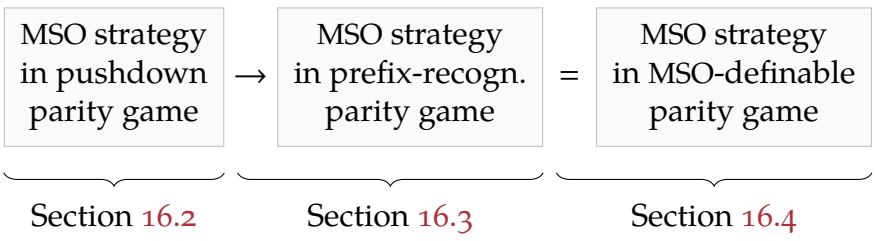


FIGURE 14: Outline of our approach to MSO-definable strategies.

16.2 Parity Games on Pushdown Graphs

In this section, we show that we can construct MSO-definable winning strategies in parity games on pushdown graphs. Subsection 16.2.1 provides the basic concepts. In Subsection 16.2.2, we present a construction of a tree automaton recognizing the uniform positional winning strategies of a given player in a pushdown parity game. This is a special case of a more general result due to Kupferman, Piterman and Vardi [KPV10]. Finally, in Subsection 16.2.3, we show how an MSO-definable winning strategy can be obtained from the tree automaton of the previous subsection.

16.2.1 Definitions and Basic Results

Definition 16.2.1. A *pushdown system* $\mathcal{P}$ is a tuple (Q, Γ, R), where

pushdown system

- Q is a finite set of *control states*,
- Γ is a finite *stack alphabet*,
- $R \subseteq Q \times \widehat{\Gamma} \times \Gamma^* \times Q$ is a finite set of *pushdown rules*, where $\widehat{\Gamma} = \Gamma \cup \{\bot\}$ includes the special stack bottom symbol $\bot \notin \Gamma$.

pushdown rule

A *configuration* of $\mathcal{P}$ is a pair $(p, u) \in Q \times \Gamma^*$ consisting of a control state p and a *stack content* u.

stack content

We often use the notation $p \xrightarrow{a/v} q$ for a pushdown rule $(p, a, v, q) \in R$.

$p \xrightarrow{a/v} q$

Definition 16.2.2. A pushdown system $\mathcal{P} = (Q, \Gamma, R)$ induces the graph (V, E), known as the configuration graph of $\mathcal{P}$, where

pushdown graph

- $V = Q \times \Gamma^*$ and
- $E = \big\{\big((p, wa), (q, wv)\big) \mid (p \xrightarrow{a/v} q) \in R,\ w \in \Gamma^*,\ a \in \Gamma\big\}$
  $\cup \big\{\big((p, \varepsilon), (q, v)\big) \mid (p \xrightarrow{\bot/v} q) \in R\big\}$.

Such a graph is called a *pushdown graph*.

Note that the stack bottom symbol $\bot$ is not included in the stack alphabet Γ and hence does not occur in the nodes of the configuration graph. The symbol $\bot$ is used merely for the technical purpose of indicating pushdown rules that can be applied when the stack is empty. We employ this slightly non-standard definition because we want to

define relations and functions (in particular, strategies) over the set of configurations using MSO formulas over the structure $Q \times \mathcal{T}_\Gamma$. Adding the purely technical symbol $\bot$ to the stack alphabet Γ would "distort" this structure.

Also note that the "top" of a stack content $u \in \Gamma^*$, i.e., the place where it can be modified, is at the end of u, not at the beginning (in contrast to the usual definition). This matches the definition of the Γ-branching tree $\mathcal{T}_\Gamma$, where the successors of a node $u \in \Gamma^*$ are obtained by appending symbols at the end of u.

We will sometimes use pushdown rules of the form $p \xrightarrow{\varepsilon/v} q$, inducing for each $w \in \Gamma^*$ an edge $\big((p,w),(q,wv)\big)$ in the configuration graph. This is only a matter of convenience, since such a rule is easily simulated by the rules $p \xrightarrow{\bot/v} q$ and $p \xrightarrow{a/av} q$ for all $a \in \Gamma$.

pushdown parity game

Definition 16.2.3. A parity game $G = (V, V_0, V_1, E, c)$ is called a *pushdown parity game* if

- (V, E) is a pushdown graph over $V = Q \times \Gamma^*$,
- the sets of positions $V_0 = Q_0 \times \Gamma^*$ and $V_1 = Q_1 \times \Gamma^*$, are induced by a partition $Q = Q_0 \uplus Q_1$, and
- the priority function $c \colon V \to [k]$ is induced by a priority function $c_Q \colon Q \to [k]$ on the control states, that is, $c\big((p,u)\big) = c_Q(p)$.

Proposition 16.2.4. Let $G = (V, V_0, V_1, E, (C_\ell)_{\ell \in [k]})$ be a pushdown parity game with $V = Q \times \Gamma^*$. The structure G is MSO-definable in $Q \times \mathcal{T}_\Gamma$.

Proof. Clearly, the sets $V_0 = Q_0 \times \Gamma^*$, $V_1 = Q_1 \times \Gamma^*$, and $(C_\ell)_{\ell \in [k]}$ are MSO-definable in $Q \times \mathcal{T}_\Gamma$. This is analogous to the proof of Proposition 15.3.3.

Let (Q, Γ, R) be the pushdown system inducing the graph (V, E). The edge relation E can be defined by the MSO formula

$$\varphi_E(x,y) = \bigvee_{(p,a,v,q) \in R} \Big(P_p(x) \wedge \exists z \, \big(\mathrm{Eq}_{\mathcal{T}_\Gamma}(x,z) \wedge P_q(z) \wedge \varphi^{\mathrm{replace}}_{a/v}(z,y) \big) \Big)$$

with

$$\varphi^{\mathrm{replace}}_{a/b_1 \dots b_n}(z,y) = \exists z_0, \dots, z_n \, \Big(\varphi^{\mathrm{pop}}_a(z,z_0) \wedge z_n = y \wedge \bigwedge_{i=1}^{n} \mathrm{Succ}'_{b_i}(z_{i-1}, z_i) \Big)$$

and

$$\varphi_a^{\mathrm{pop}}(z, z_0) = \begin{cases} \mathrm{Succ}'_a(z_0, z) & \text{if } a \in \Gamma, \\ z_0 = z \wedge \neg\exists x \left(\bigvee_{b \in \Gamma} \mathrm{Succ}'_b(x, z)\right) & \text{if } a = \bot. \end{cases}$$

□

By Proposition 16.1.2, the winning region of each player in a parity game G can be defined by an MSO formula over the structure G. We have just shown that if G is a *pushdown* parity game, then the structure G, in turn, is MSO definable in $Q \times \mathcal{T}_\Gamma$. Thus, by Proposition 11.2.3, we obtain the following corollary.

Corollary 16.2.5. For every pushdown parity game over the set of positions $Q \times \Gamma^*$, the winning regions of the two players are MSO-definable in $Q \times \mathcal{T}_\Gamma$, and we can construct corresponding MSO formulas.

16.2.2 Tree Automata Recognizing Winning Strategies

In this subsection, we show that the set of uniform positional winning strategies for a player in a pushdown parity game can be recognized by a parity tree automaton on labeled infinite trees. We begin by defining labeled infinite trees and corresponding tree automata.

Definition 16.2.6. Let Σ and Γ be finite sets. A Σ-*labeled* Γ-*branching tree* is a labeling $t : \Gamma^* \to \Sigma$ of the nodes of the Γ-branching tree $\mathcal{T}_\Gamma$.

Σ-labeled Γ-branching tree

Definition 16.2.7. Let Σ and Γ be finite sets. A *one-way parity tree automaton* $\mathcal{A}$, also simply called a *parity tree automaton*, over Σ-labeled Γ-branching trees is a tuple $(S, \Sigma, \Delta, s_0, c)$, where

(one-way) parity tree automaton

- S is a finite set of states,
- $\Delta \subseteq S \times \Sigma \times S^{|\Gamma|}$ is the transition relation,
- $s_0 \in S$ is the initial state, and
- $c : S \to [k]$ is the priority function.

A *run* of a parity tree automaton $\mathcal{A}$ on a Σ-labeled Γ-branching tree t is an S-labeled Γ-branching tree ϱ such that $\varrho(\varepsilon) = s_0$ and $\big(\varrho(u), t(u), (\varrho(ua))_{a \in \Gamma}\big) \in \Delta$ for all $u \in \Gamma^*$. A run ϱ is accepting if each

infinite path of ϱ satisfies the parity condition, that is, if for all $\alpha \in \Gamma^\omega$, the number

$$\max\{c(s) \mid s \in S, \text{ and for infinitely many prefixes } u \text{ of } \alpha, \text{ we have } \varrho(u) = s\}$$

is even.

The set of Σ-labeled Γ-branching trees *recognized* by the parity tree automaton $\mathcal{A}$ is

$$L(\mathcal{A}) = \{t \mid \text{there is an accepting run of } \mathcal{A} \text{ on } t\}.$$

universal two-way parity tree automaton

Definition 16.2.8. Let Σ and Γ be finite sets. A *universal two-way parity tree automaton* $\mathcal{U}$ over Σ-labeled Γ-branching trees is a tuple $(S, \Sigma, \delta, s_0, c)$, where

- S is a finite set of states,
- $\delta\colon S \times \Sigma \to 2^{\mathrm{Dir} \times S}$ is the transition function, where $\mathrm{Dir} = \Gamma \cup \{\uparrow, \varepsilon\}$,
- $s_0 \in S$ is the initial state, and
- $c\colon S \to [k]$ is the priority function.

Intuitively, a universal two-way parity tree automaton $\mathcal{U}$ walks around in a given tree and can, in each step, spawn additional runs by sending multiple copies of itself in any directions. We use pairs of the form $(s, u) \in S \times \Gamma^*$ to indicate the current state and position of the automaton.

Formally, a *run* of $\mathcal{U}$ on a Σ-labeled Γ-branching tree t is a maximal (i.e., either infinite or non-prolongable) sequence

$$\varrho = (s_1, u_1)(s_2, u_2)(s_3, u_3)\dots$$

with $(s_1, u_1) = (s_0, \varepsilon)$, such that for each $i > 1$, one of the following holds:

- $(\uparrow, s_i) \in \delta\big(s_{i-1}, t(u_{i-1})\big)$ and $u_i a = u_{i-1}$ for some $a \in \Gamma$, or
- $(a, s_i) \in \delta\big(s_{i-1}, t(u_{i-1})\big)$ and $u_i = u_{i-1}a$, with $a \in \Gamma \cup \{\varepsilon\}$.

A run ϱ is called accepting if it satisfies the parity condition, i.e., if $\max\{c(s) \mid s \in S, \text{ and } s \text{ occurs infinitely often in } \varrho\}$ is even. The set of Σ-labeled Γ-branching trees *recognized* by the automaton $\mathcal{U}$ is

$$L(\mathcal{U}) = \{t \mid \textit{all} \text{ runs of } \mathcal{U} \text{ on } t \text{ are accepting}\}.$$

Now let us define a representation of positional strategies in pushdown parity games in the form of labeled trees.

strategy tree

Definition 16.2.9. Consider a game graph induced by a pushdown system (Q, Γ, R) and a partition $Q = Q_0 \uplus Q_1$. Let $\sigma \colon Q_i \times \Gamma^* \to Q \times \Gamma^*$ be a positional strategy for Player i on that game graph. This strategy can alternatively be written as a function $f_\sigma \colon Q_i \times \Gamma^* \to R$ that assigns to each configuration (p, u) of Player i the unique pushdown rule $r \in R$ that transforms (p, u) into $\sigma\big((p, u)\big)$.
The *strategy tree* representing the strategy σ is the 2^R-labeled Γ-branching tree t_σ with $t_\sigma(u) = \{f_\sigma(p, u) \mid p \in Q_i\}$.

In the remainder of this subsection, we make no distinction between a strategy σ and the corresponding strategy tree t_σ.

As a special case of a more general result by Kupferman, Piterman, and Vardi [KPV10] (see also Piterman's thesis [Pito4]), we have the following proposition.

Proposition 16.2.10 ([KPV10]). Given a pushdown parity game with a distinguished initial position, we can construct for each player a parity tree automaton recognizing the set of his positional winning strategies from that initial position.

We present here a simplified proof for this special case.

Proof. Let G be the given game, which is induced by a pushdown system $\mathcal{P} = (Q, \Gamma, R)$ along with a partition $Q = Q_0 \uplus Q_1$ and a priority function $c_Q \colon Q \to [k]$. We will construct a parity tree automaton recognizing the positional winning strategies for Player 0 in the game G from the given initial position. The construction for Player 1 is analogous.

Step 1. First, we only consider trees that represent valid strategies, meaning that for every node $u \in \Gamma^*$, the corresponding label $R' \subseteq R$ contains for each $p \in Q_0$ exactly one pushdown rule that is applicable in the configuration (p, u). We will address the "invalid" trees in step 2.

Furthermore, we augment the labels of the tree nodes to indicate the topmost symbol of the stack content represented by the respective node. More precisely, the augmented label for a node of the form ua, with $u \in \Gamma^*$ and $a \in \Gamma$, comprises not only a set of rules R' but also the symbol a – and for the root node ε the symbol $\bot$. Thus, we are dealing with labeled trees of the form $t \colon \Gamma^* \to 2^R \times \widehat{\Gamma}$ with $\widehat{\Gamma} = \Gamma \uplus \{\bot\}$. The

correctness of the additional component of the labels will be verified in step 2.

We construct a universal two-way parity tree automaton $\mathcal{U}$ that checks the strategy induced by a given tree with such augmented labels. The automaton $\mathcal{U}$ walks to the tree node that corresponds to the stack content of the given initial configuration. From there, it simulates all plays starting in the initial configuration that conform to the strategy indicated by the labels in the tree. If all these plays satisfy the parity winning condition, the tree will be accepted.

During the simulation of a play, the stack content $u \in \Gamma^*$ of the current game position (p, u) is indicated by the current position of the automaton in the tree, and the current control state p is tracked in the state of the automaton. More precisely, if the current control state p belongs to Player 0, i.e., if $p \in Q_0$, then $\mathcal{U}$ consults the label of the current tree node u to determine the pushdown rule to be applied and moves to the tree node u' resulting from the application of that rule. If the control state p belongs to the opponent, i.e., if $p \in Q_1$, then $\mathcal{U}$ simulates *all* applicable pushdown rules by sending copies of itself to all resulting nodes u'. Note that the automaton can determine the applicable rules due to the augmented labeling, which indicates the topmost stack symbol.

Formally, the universal two-way parity tree automaton $\mathcal{U}$ is defined as $(S, 2^R \times \widehat{\Gamma}, \delta, s_0, c')$, consisting of the following components, where (p_0, u_0) is the given initial position of the game:

- $S = Q \times \Gamma^m$, where $m = \max\big(\{|u_0|\} \cup \{|v| \mid (p \xrightarrow{a/v} q) \in R\}\big)$,
- For $p \in Q$, $R' \subseteq R$, $a \in \widehat{\Gamma}$, $b \in \Gamma$, $w \in \Gamma^{<m}$:
  - $\delta\big((p, \varepsilon), (R', a)\big) = \begin{cases} \big\{(\overline{\uparrow}, (q, v)) \mid (p \xrightarrow{a/v} q) \in R'\big\} & \text{if } p \in Q_0, \\ \big\{(\overline{\uparrow}, (q, v)) \mid (p \xrightarrow{a/v} q) \in R\big\} & \text{if } p \in Q_1, \end{cases}$

    where $\overline{\uparrow} = \uparrow$ if $a \neq \bot$ and $\overline{\uparrow} = \varepsilon$ if $a = \bot$,
  - $\delta\big((p, bw), (R', a)\big) = \big\{\big(b, (p, w)\big)\big\}$,
- $s_0 = (p_0, u_0)$,
- $c'\big((p, w)\big) = c_Q(p)$, for $(p, w) \in S$.

Step 2. We construct a universal two-way parity tree automaton $\mathcal{U}'$ that behaves like $\mathcal{U}$ but additionally spawns runs along all paths of the given tree, verifying that the tree represents a valid strategy and that the additional labels indicating the topmost stack symbol are correct (as described in step 1). All trees violating these criteria are rejected.

Step 3. We convert the universal two-way tree automaton $\mathcal{U}'$ into a (nondeterministic) one-way tree automaton $\mathcal{A}$, also with a parity condition, recognizing the same set of trees. This construction is given in [Var98] even for alternating two-way automata, a detailed description can also be found in [Cac02].

Step 4. Finally, from the automaton $\mathcal{A}$ we can derive the desired (nondeterministic) parity tree automaton, which recognizes the positional winning strategies for Player 0, by projection on the component of the labels that indicates the selected pushdown rules, i.e., by removing the additional component that indicates the topmost stack symbol. □

Note that the strategies accepted by the tree automaton from Proposition 16.2.10 are winning strategies from the given initial position, but they need not be *uniform* winning strategies. However, we can adapt the construction to recognize the uniform winning strategies.

Proposition 16.2.11. Given a pushdown parity game, we can construct for each player a parity tree automaton recognizing the set of his uniform positional winning strategies.

Proof. We consider here the case of Player 0, the construction for Player 1 is analogous. We adapt the construction from the proof of Proposition 16.2.10 by replacing the universal two-way tree automaton $\mathcal{U}$, which checks whether a given strategy is a winning strategy, with a slightly modified automaton $\widehat{\mathcal{U}}$.

To that end, let Q be the set of control states and let Γ be the stack alphabet of the pushdown system inducing the graph of the game. By Corollary 16.2.5, the winning region W_0 of Player 0 is MSO-definable in $Q \times \mathcal{T}_\Gamma$ and hence MSO-family-definable in $\mathcal{T}_\Gamma$ (Proposition 11.3.3). Thus, for each $p \in Q$, the stack contents $u \in \Gamma^*$ with $(p, u) \in W_0$ constitute a regular language, recognized by some DFA $\mathcal{D}_p$ (see Proposition 11.4.3).

Now we construct the following universal two-way tree automaton $\widehat{\mathcal{U}}$ replacing the automaton $\mathcal{U}$ in the proof of Proposition 16.2.10: Instead of checking the strategy induced by a given tree from a fixed initial position (p_0, u_0), the automaton $\widehat{\mathcal{U}}$ sends copies along all paths of the tree, each simulating simultaneously all the DFAs $(\mathcal{D}_p)_{p \in Q}$. Whenever a final state of a DFA $\mathcal{D}_p$ is reached upon arriving at a tree node $u \in \Gamma^*$, we know that the configuration (p, u) is in the winning region W_0, so the tree automaton $\widehat{\mathcal{U}}$ starts a copy of $\mathcal{U}$ to check the strategy for all plays that start at the position (p, u). Hence, $\widehat{\mathcal{U}}$ will only accept if the strategy is winning from all positions in the winning region. □

16.2.3 MSO-Definable Strategies in Pushdown Parity Games

We will now show that we can extract an MSO-definable winning strategy from the tree automaton obtained by Proposition 16.2.11.

If the set of trees (i.e., of winning strategies) recognized by the tree automaton is not empty, then it contains a so-called *regular* tree (see [Rab72]), which can be represented by a DFA that reads a given tree node $u \in \Gamma^*$ and outputs the corresponding label. The emptiness test for parity tree automata can easily be extended to yield such a DFA, representing a winning strategy. Adopting the terminology of Carayol and Hague [CH14], we call such a DFA a *strategy automaton*.

strategy automaton

Definition 16.2.12. A *strategy automaton* $\mathcal{A}$ for a pushdown system $\mathcal{P} = (Q, \Gamma, R)$ is a tuple $(S, \Gamma, s_0, \delta, \text{Rules})$, where

- S is a finite set of states,
- Γ is the finite stack alphabet of the pushdown system, serving as the input alphabet of the automaton,
- $s_0 \in S$ is the initial state,
- $\delta\colon S \times \Gamma \to S$ is the transition function, and
- $\text{Rules}\colon S \to 2^R$ is an output function mapping each state to a set of pushdown rules.

A given strategy automaton $\mathcal{A} = (S, \Gamma, s_0, \delta, \text{Rules})$ represents the tree $t_{\mathcal{A}}\colon \Gamma^* \to 2^R$ defined by $t_{\mathcal{A}}(u) = \text{Rules}\big(\delta(s_0, u)\big)$, where δ is extended from single input symbols to words in the usual way.

If $t_{\mathcal{A}}$ is the strategy tree of some positional strategy σ, i.e., if $t_{\mathcal{A}} = t_\sigma$, then we say that $\mathcal{A}$ *induces the strategy* σ.

Corollary 16.2.13. Given a pushdown parity game, we can construct for each player a strategy automaton inducing a uniform positional winning strategy on his winning region.

Once we have constructed a strategy automaton, we can transform it into an MSO-definable strategy:

Proposition 16.2.14. Let $\mathcal{A}$ be a strategy automaton inducing a positional strategy σ on a pushdown game graph. Then the strategy σ is MSO-definable, and we can construct a corresponding MSO formula.

Proof. In a nutshell, the statement follows from the expressive equivalence of finite automata and MSO logic, combined with the fact that the effect of a given pushdown rule can be described by an MSO formula.

For a more precise argument, let $\mathcal{P} = (Q, \Gamma, R)$ be the pushdown system inducing the game graph, with $Q = Q_0 \cup Q_1$. As a first step, for each pushdown rule $r = (p_1 \xrightarrow{a/v} p_2) \in R$, we consider the DFA $\mathcal{D}_r^{\text{stacks}} = (S, \Gamma, s_0, \delta, F)$ that is obtained from the strategy automaton $\mathcal{A} = (S, \Gamma, s_0, \delta, \text{Rules})$ by choosing $F = \{s \in S \mid r \in \text{Rules}(s)\}$ as the set of final states. Note that the language $L(\mathcal{D}_r^{\text{stacks}})$ recognized by this DFA is the set of *stack contents* u such that the rule to be applied in the configuration (p_1, u) is r, according to the strategy automaton $\mathcal{A}$. By Proposition 11.4.3, we can derive from $\mathcal{D}_r^{\text{stacks}}$ an MSO formula $\varphi_r^{\text{stacks}}(x)$ that defines $L(\mathcal{D}_r^{\text{stacks}})$ in the tree $\mathcal{T}_\Gamma$.

The set of *configurations* in which the rule $r = (p_1 \xrightarrow{a/v} p_2)$ should be applied is hence defined in $\mathcal{T}_\Gamma$ by the family of MSO formulas $\left(\varphi_{r,p}^{\text{configs}}(x)\right)_{p \in Q}$ with

$$\varphi_{r,p}^{\text{configs}}(x) = \begin{cases} \varphi_r^{\text{stacks}}(x) & \text{if } p = p_1, \\ \texttt{false} & \text{otherwise.} \end{cases}$$

By Proposition 11.3.3, this family of formulas can be converted into a formula $\varphi_r^{\text{configs}}(x)$ defining the same set of configurations in $Q \times \mathcal{T}_\Gamma$.

We can now represent the strategy $\sigma \colon Q_i \times \Gamma^* \to Q \times \Gamma^*$ induced by the strategy automaton $\mathcal{A}$ by the following MSO formula $\varphi(x, y)$, which asserts that there is a pushdown rule $p \xrightarrow{a/v} q$ that is selected by $\mathcal{A}$ for the configuration x and yields the configuration y:

$$\varphi(x, y) = \bigvee_{(p,a,v,q) \in R} \Big(\varphi_{(p,a,v,q)}^{\text{configs}}(x) \wedge \exists z \left(\text{Eq}_{\mathcal{T}_\Gamma}(x, z) \wedge P_q(z) \wedge \varphi_{a/v}^{\text{replace}}(z, y) \right) \Big)$$

with $\varphi_{a/v}^{\text{replace}}(z, y)$ as defined in the proof of Proposition 16.2.4. □

Combining the construction of a strategy automaton (Corollary 11.3.7) and the translation to MSO logic (Proposition 16.2.14), we obtain the following result.

Proposition 16.2.15. Given a pushdown parity game G, we can construct for each player an MSO-definable uniform positional winning strategy on his winning region.

16.3 Parity Games on Prefix-Recognizable Graphs

In games on pushdown graphs as discussed in the previous section, each move by one of the two players changes the content of the stack only in a bounded way – at most one symbol is removed from the stack, and a word of bounded length is added. Ultimately, however, we are interested in games with MSO-definable edge relations, and the moves in such games may involve unbounded changes. In this section, we therefore consider games on prefix-recognizable graphs, a generalization of pushdown graphs that allows such changes. As we will see in Section 16.4, MSO-definable graphs are in fact prefix-recognizable.

In Subsection 16.3.1, we provide the definition of prefix-recognizable parity games over a set of positions of the form $Q \times \Gamma^*$, for finite sets Q and Γ. We then describe in Subsection 16.3.2 a "reduction" from prefix-recognizable parity games to pushdown parity games, which was originally given by Cachat [Cac03]. Based on this reduction, we show in Subsection 16.3.3 that MSO-definable winning strategies in a given prefix-recognizable parity game can be obtained from such strategies in the corresponding pushdown parity game. We also prove that this result does not only apply to games in which the owner and the priority of a position $(p, u) \in Q \times \Gamma^*$ are determined by p, but can even be extended to games with MSO-definable ownership and priority sets.

16.3.1 Definitions

We begin with the definition of prefix-recognizable systems, a generalization of pushdown systems.

prefix-recognizable system

Definition 16.3.1. A *prefix-recognizable system* $\mathcal{R}$ is a pair (Γ, H), where Γ is a finite alphabet and $H \subseteq \mathrm{Reg}(\Gamma) \times \mathrm{Reg}(\Gamma) \times \mathrm{Reg}(\Gamma)$ is a finite set of *rewrite rules*, where $\mathrm{Reg}(\Gamma)$ is the set of regular languages over the alphabet Γ.

rewrite rule

A rewrite rule (U, V, W) consisting of regular languages $U, V, W \subseteq \Gamma^*$ allows us to replace a suffix $u \in U$ of a word of the form wu, with $w \in W$, by a new suffix $v \in V$. More formally, the rule (U, V, W) induces the following relation, denoted by $W(U \times V)$:

$W(U \times V)$

$$W(U \times V) = \{(wu, wv) \mid w \in W,\ u \in U,\ v \in V\} \subseteq \Gamma^* \times \Gamma^*$$

Note that applying a rewrite rule modifies a suffix of a word, not a prefix, so the term "prefix-recognizable system" may appear as a misnomer. Indeed, the induced relation is usually defined in the reverse order as $(U \times V)W$. However, as in [Blu01], we use the definition of $W(U \times V)$ above since it aligns with the definition of the tree $\mathcal{T}_\Gamma$, where the successors of a node $w \in \Gamma^*$ are obtained by appending symbols at the end of w.

Definition 16.3.2. A prefix-recognizable system $\mathcal{R} = (\Gamma, H)$ induces the relation $D_\mathcal{R} \subseteq \Gamma^* \times \Gamma^*$ defined by

prefix-recognizable relation

$$D_\mathcal{R} = \bigcup_{(U,V,W) \in H} W(U \times V).$$

Such a relation is called a *prefix-recognizable relation*.

Usually, a graph is called a prefix-recognizable graph if it has a prefix-recognizable edge relation over a universe of the form Γ^* (see [Blu01]). However, to obtain a consistent format for all the game graphs considered in this thesis, we adapt this definition to graphs over $Q \times \Gamma^*$, for a finite set of "control states" Q.

Definition 16.3.3. A family of prefix-recognizable systems $(\mathcal{R}_{p,q})_{p,q \in Q}$, for some finite set Q, induces the graph (V, E) with

prefix-recognizable graph

- $V = Q \times \Gamma^*$ and
- $E = \bigcup_{p,q \in Q} \{((p,u),(q,v)) \mid (u,v) \in D_{\mathcal{R}_{p,q}}\}$.

We call such a graph a *prefix-recognizable graph*.

We are interested in parity games that are played on such prefix-recognizable graphs.

Definition 16.3.4. A parity game $G = (V, V_0, V_1, E, (C_\ell)_{\ell \in [k]})$ is called a *prefix-recognizable parity game* if

prefix-recognizable parity game

- (V, E) is a prefix-recognizable graph graph over $V = Q \times \Gamma^*$,
- the sets of positions $V_0 = Q_0 \times \Gamma^*$ and $V_1 = Q_1 \times \Gamma^*$ are induced by a partition $Q = Q_0 \uplus Q_1$, and
- the priority function $c \colon V \to [k]$ is induced by a priority function $c_Q \colon Q \to [k]$ on Q, that is, $c((p,u)) = c_Q(p)$.

16.3.2 Reducing Prefix-Recognizable Games to Pushdown Games

Let G be a prefix-recognizable parity game. Our goal is to construct MSO-definable winning strategies for the two players on their winning regions. To that end, we transform the game into a pushdown parity game in such a way that we can transfer winning strategies from the pushdown game to the prefix-recognizable game. This transformation was originally given by Cachat [Cac03]. In this subsection, we present a slight adaptation of his construction to our setting, where the positions of the prefix-recognizable game involve not only a word but also a control state. As we will see in Subsection 16.3.3, we can obtain MSO-definable winning strategies in the prefix-recognizable game from MSO-definable winning strategies in the pushdown game.

The game graph of G is induced by a family of prefix-recognizable systems $(\mathcal{R}_{p,q} = (\Gamma, H_{p,q}))_{p,q \in Q}$, where Q is a finite set partitioned into Q_0 and Q_1. For each rewrite rule (U, V, W) occurring in any of the prefix-recognizable systems $\mathcal{R}_{p,q}$, we let $\mathcal{B}_U = (S_U, \Gamma, s_0^U, \delta_U, F_U)$, $\mathcal{B}_V = (S_V, \Gamma, s_0^V, \delta_V, F_V)$, and $\mathcal{B}_W = (S_W, \Gamma, s_0^W, \delta_W, F_W)$ be three DFAs recognizing U^r, V, and W^r, respectively. Here, $U^r = \{a_n \dots a_1 \mid a_1 \dots a_n \in U\}$ is the reverse of the language U (which is regular for a regular U), and W^r is defined analogously. The reason for this treatment of U and W will be given below.

A move from a position $(p, wu) \in Q \times \Gamma^*$ to a position $(q, wv) \in Q \times \Gamma^*$ in the game G corresponds to an application of some rewrite rule $t = (U, V, W) \in H_{p,q}$ with $u \in U$, $v \in V$, $w \in W$. We construct a pushdown parity game G' where each such move is simulated by a sequence of applications of pushdown rules, each adding or removing at most one symbol to/from the stack. This sequence consists of a pop phase, where the current player removes the suffix u from the stack, and a push phase, where he adds the new suffix v. After the pop phase, the other player has the opportunity to verify that the remaining prefix w is indeed in W. More precisely, an application of the rewrite rule $(U, V, W) \in H_{p,q}$ is simulated as follows:

1. First, in the *pop phase*, the player controlling the current state p can remove a suffix u of the current stack content, one symbol at a time. To determine whether this suffix is in the language U, the DFA $\mathcal{B}_U$ is simulated on that suffix step by step while it is being removed, and the current state of $\mathcal{B}_U$ is tracked in the control state of the pushdown system. (Note that the suffix is processed in reverse, which is why we chose $\mathcal{B}_U$ to recognize U^R instead of U.) Whenever a final state of $\mathcal{B}_U$ is reached, meaning that a suffix

in U has been removed, the player may decide to end the pop phase and hand control over to his opponent.

2. This initiates the *verification phase*, which serves to ensure that the remaining stack content w is a word in W as required for a valid application of the rewrite rule (U, V, W).

   In this phase, the opponent is in control and has two options: On the one hand, he can contest the claim that the stack content is in W. In that case, he will empty the stack one symbol at a time, simultaneously simulating the DFA $\mathcal{B}_W$ on the sequence of removed symbols (i.e., the reverse of w). If the state that is reached when the stack is empty is not a final state, he wins, otherwise he loses (by reaching a position with a self-loop and a suitable priority). On the other hand, the opponent can instead choose to believe that the stack content is in W and immediately transfer control back to the original player.

3. If the opponent chose not to check the remaining prefix, the *push phase* begins. Now the original player can add a new suffix v to the stack, again one symbol at a time, while simulating the DFA $\mathcal{B}_V$ on the word that is added this way. Whenever a final state of $\mathcal{B}_V$ is reached, indicating that the new suffix is a word in V, the player can switch to the control state q, thereby completing the simulation of the rewrite rule $(U, V, W) \in H_{p,q}$.

Formally, the game graph of the game G' is defined by the pushdown system $\mathcal{P} = (P, \Gamma, R)$, where the stack alphabet Γ is simply adopted from the prefix-recognizable systems and P is partitioned as $P = P_0 \cup P_1$, where, for $i \in \{0, 1\}$,

$$P_i = Q_i \cup P_i^{\text{pop}} \cup P_i^{\text{verify?}} \cup P_i^{\text{verify}} \cup P_i^{\text{push}} \cup \{\text{win}_i\}$$

with

$$\begin{aligned}
P_i^{\text{pop}} &= \{(p, q, t, \text{pop}, s) \mid p \in Q_i,\ q \in Q,\\
&\qquad\qquad t = (U, V, W) \in H_{p,q},\ s \in S_U\},\\
P_i^{\text{verify?}} &= \{(p, q, t, \text{verify?}, s) \mid p \in Q_{1-i},\ q \in Q,\\
&\qquad\qquad t \in H_{p,q}\},\\
P_i^{\text{verify}} &= \{(p, q, t, \text{verify}, s) \mid p \in Q_{1-i},\ q \in Q,\\
&\qquad\qquad t = (U, V, W) \in H_{p,q},\ s \in S_W\},\\
P_i^{\text{push}} &= \{(p, q, t, \text{push}, s) \mid p \in Q_i,\ q \in Q,\\
&\qquad\qquad t = (U, V, W) \in H_{p,q},\ s \in S_V\}.
\end{aligned}$$

The set of pushdown rules is

$$R = \bigcup_{\substack{p,q \in Q,\\ t \in H_{p,q}}} R_{p,q,t}$$

where $R_{p,q,t}$ is the set of pushdown rules to simulate applications of the rewrite rule $t = (U, V, W) \in H_{p,q}$. More precisely, $R_{p,q,t}$ contains the following pushdown rules, where $i \in \{0, 1\}$ such that $p \in Q_i$:

$$\begin{aligned}
p &\xrightarrow{\varepsilon/\varepsilon} (p, q, t, \mathrm{pop}, s_0^U) && \\
(p, q, t, \mathrm{pop}, s) &\xrightarrow{a/\varepsilon} (p, q, t, \mathrm{pop}, \delta_U(s, a)) && \text{for } a \in \Gamma, s \in S_U \\
(p, q, t, \mathrm{pop}, s_f) &\xrightarrow{\varepsilon/\varepsilon} (p, q, t, \mathrm{verify?}) && \text{for } s_f \in F_U \\
(p, q, t, \mathrm{verify?}) &\xrightarrow{\varepsilon/\varepsilon} (p, q, t, \mathrm{verify}, s_0^W) && \\
(p, q, t, \mathrm{verify?}) &\xrightarrow{\varepsilon/\varepsilon} (p, q, t, \mathrm{push}, s_0^V) && \\
(p, q, t, \mathrm{verify}, s) &\xrightarrow{a/\varepsilon} (p, q, t, \mathrm{verify}, \delta_W(s, a)) && \text{for } a \in \Gamma, s \in S_W \\
(p, q, t, \mathrm{verify}, s) &\xrightarrow{\bot/\varepsilon} \mathrm{win}_{1-i} && \text{for } s \in S_W \setminus F_W \\
(p, q, t, \mathrm{verify}, s_f) &\xrightarrow{\bot/\varepsilon} \mathrm{win}_i && \text{for } s_f \in F_W \\
(p, q, t, \mathrm{push}, s) &\xrightarrow{\varepsilon/a} (p, q, t, \mathrm{push}, \delta_V(s, a)) && \text{for } a \in \Gamma, s \in S_V \\
(p, q, t, \mathrm{push}, s_f) &\xrightarrow{\varepsilon/\varepsilon} q && \text{for } s_f \in F_V \\
\mathrm{win}_i &\xrightarrow{\varepsilon/\varepsilon} \mathrm{win}_i && \\
\mathrm{win}_{1-i} &\xrightarrow{\varepsilon/\varepsilon} \mathrm{win}_{1-i} &&
\end{aligned}$$

Finally, we provide the winning condition for the pushdown parity game G' by defining a function c_P that assigns priorities to the control states of the pushdown system $\mathcal{P}$. Let $c_Q \colon Q \to [k]$ be the priority function for the states Q of the prefix-recognizable game G. We define $c_P \colon P \to [k+2]$ as follows:

$$c_P(\widehat{p}) = \begin{cases} c_Q(\widehat{p}) + 2 & \text{if } \widehat{p} \in Q, \\ 0 & \text{if } \widehat{p} = \mathrm{win}_0 \text{ or } \widehat{p} \in P_1 \setminus \big(Q \cup \{\mathrm{win}_1\}\big), \\ 1 & \text{if } \widehat{p} = \mathrm{win}_1 \text{ or } \widehat{p} \in P_0 \setminus \big(Q \cup \{\mathrm{win}_0\}\big). \end{cases}$$

The reasoning behind the definition of c_P is the following. For positions that also exist in the original prefix-recognizable game G, i.e., positions with control states in Q, the priorities are "shifted up" by 2 compared to G. This preserves the order of these priorities and

ensures that they are larger than the priorities 0 and 1 used for the additional positions (those with control states in $P \setminus Q$). Thus, in a play ϱ' of the pushdown game that simulates a play ϱ of the original game, the maximum priority that is seen infinitely often is only determined by positions that also exist in the original game, not by the auxiliary positions visited in between, so the winner of ϱ and ϱ' is the same.

The choice of priorities for win_0 and win_1 guarantees that looping in one of these states is winning for the respective player.

To justify the priorities of the remaining positions, note that the pushdown game contains a "loophole" that allows the players to avoid simulating the original game: In a push phase, the current player can refuse to end that phase and instead just keep adding symbols to the stack. To prevent the player to win by exploiting this loophole, the priority for the positions of the push phase is 1 for Player 0 and 0 for Player 1. We choose the priorities for the positions of the pop and verification phase in the same way; however, since these positions cannot be visited infinitely often without also visiting infinitely often positions that exist in the original game, an arbitrary priority in $\{0, 1\}$ could be used.

Proposition 16.3.5. Let G be a prefix-recognizable parity game, let G' be the corresponding pushdown parity game as defined above, and let (p_0, u_0) be a position of G. Player 0 has a winning strategy from (p_0, u_0) in G if and only if he has a winning strategy from (p_0, u_0) in G'.

Proof. Since parity games are determined with positional winning strategies, it suffices to show that for each player, a positional winning strategy in the pushdown parity game G' can be transformed into a positional winning strategy in the prefix-recognizable parity game G. We consider here the case of Player 0; the case of Player 1 is analogous.

Assume that Player 0 has a positional winning strategy σ' from $(p_0, u_0) \in Q \times \Gamma^*$ in the pushdown game G'. We will now define a positional winning strategy σ for Player 0 in G based on σ'. Consider a position $(p, u) \in Q \times \Gamma^*$ of G. This position also exists in G', and from (p, u), the winning strategy σ' in G' selects a sequence of edges of the pop phase. A sequence of pop operations cannot be infinite, so it must finally lead to a position that allows the other player to choose to either verify the remaining prefix or yield control back to the original player.

From the position that is reached in the latter case, the strategy σ' again selects a sequence of edges, belonging to the push phase. This sequence cannot be infinite, since by our choice of the priorities, such

an infinite sequence would yield a losing play for the current player (that is, Player 0). Thus, the sequence leads to a uniquely determined position of the form $(q, v) \in Q \times \Gamma^*$.

Note that there is an edge from (p, u) to (q, v) in the game G, since the steps selected in the pop and push phase correspond to an existing rewrite rule – otherwise Player 1 could have won by verifying the remaining suffix, contradicting the assumption that σ' is a winning strategy. In the strategy σ for the game G, we let Player 0 move from (p, u) to (q, v).

If Player 0 uses the strategy σ defined in this way, every play ϱ in G starting in (p_0, u_0) corresponds to a play ϱ' in G' that simulates the moves in ϱ – and that conforms with Player 0's winning strategy σ' in G'. Since Player 0 wins ϱ', he therefore also wins ϱ. □

16.3.3 MSO-Definable Strategies in Prefix-Recognizable Parity Games

In the previous subsection, we have seen that winning strategies for prefix-recognizable parity games can be obtained from winning strategies in corresponding pushdown parity games. We now want to strengthen this result by showing that we can in fact obtain MSO-definable winning strategies in this way.

Proposition 16.3.6. Given a prefix-recognizable parity game, we can construct for each player an MSO-definable uniform positional winning strategy on his winning region.

Proof. We address here the construction of a winning strategy for Player 0, the case of Player 1 is analogous.

Let G be the given prefix-recognizable parity game, and let G' be the pushdown parity game obtained from G as described in Subsection 16.3.2. By Proposition 16.2.15, we can construct an MSO-definable uniform positional winning strategy σ' for Player 0 in G'. Recall the corresponding winning strategy σ for Player 0 in G (which is again positional and uniform) as described in the proof of Proposition 16.3.5: For a position $(p, u) \in Q \times \Gamma^*$ of G, it selects the successor $(q, v) \in Q \times \Gamma^*$ that is reached in G' from (p, u) by following the sequence of pop and push moves defined by Player 0's winning strategy σ' there.

Now let $\varphi'(x, y)$ be an MSO formula defining σ' in the structure $P \times \mathcal{T}_\Gamma$, where $P \supseteq Q$ is the set of control states of the pushdown game G'. We want to transform $\varphi'(x, y)$ into another MSO formula $\varphi(x, y)$ defining the winning strategy σ for the game G in $Q \times \mathcal{T}_\Gamma$. To that end, it suffices

to construct an MSO formula $\psi(x, y)$ that defines σ in $P \times \mathcal{T}_\Gamma$. Since $Q \subseteq P$, we can then apply Proposition 11.3.4 to obtain the desired formula $\varphi(x, y)$.

To build the formula $\psi(x, y)$, we essentially define the transitive closure of the intermediate moves that are selected by the strategy σ' defined by $\varphi'(x, y)$, skipping the verification phase (which is controlled by the opponent):

$$\psi(x,y) = \bigvee_{p,q \in Q} \big(P_p(x) \wedge P_q(y)\big) \wedge \forall X \left(\Big(X(x) \wedge \textsc{Closed}(X)\Big) \rightarrow \exists z \left(X(z) \wedge \varphi'(z,y)\right)\right)$$

where

$$\textsc{Closed}(X) = \forall z, z' \left(\left(X(z) \wedge \begin{bmatrix} \textsc{Move}(z,z') \vee \\ \textsc{SkipVerify}(z,z') \end{bmatrix}\right) \rightarrow X(z')\right)$$

and

$$\textsc{Move}(z,z') = \varphi'(z,z') \wedge \bigvee_{\widehat{p} \in P \setminus Q} P_{\widehat{p}}(z'),$$

$$\textsc{SkipVerify}(z,z') = \mathrm{Eq}_{\mathcal{T}_\Gamma}(z,z') \wedge \bigvee_{(p,q,t,\text{verify?}) \in P} \begin{bmatrix} P_{(p,q,t,\text{verify?})}(z) \wedge \\ P_{(p,q,t,\text{push})}(z') \end{bmatrix}.$$

□

Recall that our definition of prefix-recognizable parity games (Definition 16.3.4) requires that the owner and the priority of a position $(p, u) \in Q \times \Gamma^*$ are determined only by p. However, we can extend our results even to a slightly generalized version of such games.

Definition 16.3.7. A parity game $G = (V, V_0, V_1, E, (C_\ell)_{\ell \in [k]})$ is called a *generalized prefix-recognizable parity game* if

generalized prefix-recognizable parity game

- (V, E) is a prefix-recognizable graph over $V = Q \times \Gamma^*$,
- the sets of positions $V_0, V_1 \subseteq Q \times \Gamma^*$ are MSO-definable, and
- the priority sets $C_\ell \subseteq Q \times \Gamma^*$ are MSO-definable.

Proposition 16.3.8. Given a generalized prefix-recognizable parity game, we can construct for each player an MSO-definable uniform positional winning strategy on his winning region.

Proof. We reduce the given generalized game G to a "standard" prefix-recognizable parity game $\widehat{G}$, where the owner and priority of a position (p, u) is determined by p, and transfer the winning strategy for $\widehat{G}$ back to G. The basic idea is to represent a position $(p, u) \in Q \times \Gamma^*$ of G by a position $((p, i, \ell), u)$ of $\widehat{G}$, where $i \in \{0, 1\}$ and $\ell \in [k]$ explicitly indicate the owner and priority of (p, u). To force each player to select only *consistent* positions, with the correct i and ℓ, we make sure that otherwise his opponent can win.

Reducing the game. Let $G = (V, V_0, V_1, E, (C_\ell)_{\ell \in [k]})$ be the given generalized prefix-recognizable parity game. The graph (V, E), with V of the form $Q \times \Gamma^*$, is induced by a family of prefix-recognizable systems $(\mathcal{R}_{p,q} = (\Gamma, H_{p,q}))_{p,q \in Q}$.

To specify the positions of the game $\widehat{G}$, we define the set $\widehat{Q} = P_0 \cup P_1 \cup P_0^{\mathrm{chk}} \cup P_1^{\mathrm{chk}} \cup \{\mathrm{win}_0, \mathrm{win}_1\}$, where $P_i = Q \times \{i\} \times [k]$ and $P_i^{\mathrm{chk}} = \{\mathrm{chk}_i\} \times Q \times \{0, 1\} \times [k]$. We partition $\widehat{Q}$ into $\widehat{Q}_0$ and $\widehat{Q}_1$ with $\widehat{Q}_i = P_i \cup P_i^{\mathrm{chk}} \cup \{\mathrm{win}_i\}$. The priorities are determined by the function $c_{\widehat{Q}} \colon \widehat{Q} \to [k]$ defined by

$$c_{\widehat{Q}}(\widehat{p}) = \begin{cases} \ell & \text{if } \widehat{p} = (p, i, \ell), \\ 0 & \text{if } \widehat{p} = (\mathrm{chk}_i, q, j, m), \\ i & \text{if } \widehat{p} = \mathrm{win}_i. \end{cases}$$

An edge in G from (p, u) to (q, v) has multiple counterparts in $\widehat{G}$, leading from position $\big((p, i, \ell), u\big)$ to $\big((\mathrm{chk}_{1-i}, q, j, m), v\big)$, for $i, j \in \{0, 1\}$, $\ell, m \in [k]$. After Player i selects such an edge in $\widehat{G}$, his opponent can react: If the target position is not consistent, the opponent can go to a position where he will always win. Otherwise the opponent can only move to $\big((q, j, m), v\big)$ and the play continues there. Thus, to have a chance to win, the players must never move to an inconsistent position.

Formally, the game graph of $\widehat{G}$ is induced by the family of prefix-recognizable systems $(\widehat{\mathcal{R}}_{\widehat{p},\widehat{q}} = (\Gamma, \widehat{H}_{\widehat{p},\widehat{q}}))_{\widehat{p},\widehat{q} \in \widehat{Q}}$ whose sets of rewrite rules are defined by

$$\widehat{H}_{\widehat{p},\widehat{q}} = \begin{cases} H_{p,q} & \text{if } \widehat{p} = (p, i, \ell),\ \widehat{q} = (\mathrm{chk}_{1-i}, q, j, m), \\ \big\{\big(\{\varepsilon\}, \{\varepsilon\}, \Gamma^* \setminus \big(V_j^q \cap C_m^q\big)\big)\big\} & \text{if } \widehat{p} = (\mathrm{chk}_i, q, j, m),\ \widehat{q} = \mathrm{win}_i, \\ \big\{\big(\{\varepsilon\}, \{\varepsilon\}, \Gamma^*\big)\big\} & \text{if } \widehat{p} = (\mathrm{chk}_i, q, j, m),\ \widehat{q} = (q, j, m), \\ \emptyset & \text{otherwise,} \end{cases}$$

where $V_j^q = \{u \in \Gamma^* \mid (q, u) \in V_j\}$ and $C_m^q = \{u \in \Gamma^* \mid (q, u) \in C_m^q\}$.

Note that $V_j, C_m \subseteq Q \times \Gamma^*$ are MSO-definable in $Q \times \mathcal{T}_\Gamma$ and hence MSO-family-definable in $\mathcal{T}_\Gamma$ (Proposition 11.3.3). Thus, the languages V_j^q and C_m^q are regular for each $q \in Q$ (see Proposition 11.4.3), so the sets of rewrite rules $\widehat{H}_{\widehat{p},\widehat{q}}$ are well-defined.

Transferring the winning strategy. By Proposition 16.3.6, we can construct an MSO-definable uniform positional winning strategy $\widehat{\sigma}$ for Player $i \in \{0,1\}$ in the game $\widehat{G}$. This strategy $\widehat{\sigma}$ induces a uniform positional winning strategy σ for Player i in G, which is defined as follows, for $(p,u) \in V_i$ with $(p,u) \in C_\ell$:

$$\sigma\big((p,u)\big) = (q,v) \quad \Leftrightarrow \quad \widehat{\sigma}\big(((p,i,\ell),u)\big) = ((\mathrm{chk}_{1-i},q,j,m),v) \text{ for some } j \in \{0,1\},\ m \in [k].$$

We will now show that this strategy σ is MSO-definable. First, we can define the strategy $\widehat{\sigma}$ for the game $\widehat{G}$ by a family of MSO formulas $(\widehat{\varphi}_{\widehat{p},\widehat{q}})_{\widehat{p},\widehat{q}\in\widehat{Q}}$ (by Proposition 11.3.3). Furthermore, since the sets $V_i, C_\ell \subseteq Q \times \Gamma^*$ of the generalized game G are MSO-definable in $Q \times \mathcal{T}_\Gamma$, we can construct families of MSO formulas $\big(\psi_p^{V_i}(x)\big)_{p\in Q}$ and $\big(\psi_p^{C_\ell}(x)\big)_{p\in Q}$ defining these sets in $\mathcal{T}_\Gamma$.

The winning strategy σ for Player i in the game G can now be defined in $\mathcal{T}_\Gamma$ by the family of MSO formulas $(\varphi_{p,q})_{p,q\in Q}$ with

$$\varphi_{p,q}(x,y) = \psi_p^{V_i}(x) \ \wedge \bigvee_{\substack{\ell,m\in[k],\\ j\in\{0,1\}}} \psi_p^{C_\ell}(x) \wedge \widehat{\varphi}_{(p,i,\ell),(\mathrm{chk}_{1-i},q,j,m)}(x,y).$$

This family of formulas can finally be transformed into an MSO formula $\varphi(x,y)$ that defines the winning strategy σ in the structure $Q \times \mathcal{T}_\Gamma$ (by Proposition 11.3.3). □

16.4 MSO-Definable Parity Games

Using the results of the previous section, we can finally prove the main result of this chapter, namely Theorem 16.1.3, which states that we can construct MSO-definable uniform positional winning strategies in every parity game over $Q \times \Gamma^*$ that is MSO-definable in $Q \times \mathcal{T}_\Gamma$.

According to Proposition 16.3.8, we can construct such winning strategies in parity games on prefix-recognizable graphs with MSO-definable ownership partitions and priority sets. Theorem 16.1.3 then follows from the fact that every graph over $Q \times \Gamma^*$ that is MSO-definable in $Q \times \mathcal{T}_\Gamma$ is in fact prefix-recognizable:

Proposition 16.4.1. Let Q and Γ be finite sets. Let (V, E) be a graph over $V = Q \times \Gamma^*$ such that E is MSO-definable (in $Q \times \mathcal{T}_\Gamma$). Then (V, E) is a prefix-recognizable graph, and we can construct a corresponding family of prefix-recognizable systems.

Let us now prove this proposition. Since the edge relation E is MSO-definable, it can (by Proposition 11.3.3) be defined in $\mathcal{T}_\Gamma$ by a family of MSO formulas $(\varphi_{p,q}(x,y))_{p,q\in Q}$. In other words, we have $E = \bigcup_{p,q\in Q}\{((p,u),(q,v)) \mid (u,v) \in D_{p,q}\}$, where $D_{p,q} \subseteq \Gamma^* \times \Gamma^*$ is the relation defined by $\varphi_{p,q}(x,y)$. Thus, it suffices to show that for every $p, q \in Q$, the relation $D_{p,q}$ is a prefix-recognizable relation.

Proposition 16.4.2 ([Bar97; Blu01]). Let $D \subseteq \Gamma^* \times \Gamma^*$ be a binary relation, where Γ is a finite set. If D is MSO-definable in $\mathcal{T}_\Gamma$, then D is a prefix-recognizable relation and we can construct a corresponding prefix-recognizable system.

Remark 16.4.3. Conversely, every prefix-recognizable relation is MSO-definable. See [Blu01] for details.

Proposition 16.4.2 was originally shown by Barthelmann [Bar97]. We present here a proof using an approach by Blumensath [Blu01], based on tree automata on labeled Γ-branching trees (see Definition 16.2.7).

Proof of Proposition 16.4.2. Let $\varphi(x,y)$ be an MSO formula defining the relation $D \subseteq \Gamma^* \times \Gamma^*$ in $\mathcal{T}_\Gamma$. We can translate $\varphi(x,y)$ into a corresponding parity tree automaton $\mathcal{A} = (S, \{0,1\}^2, \Delta, s_0, c)$ recognizing the following set of $\{0,1\}^2$-labeled Γ-branching trees (see [Rab68]):

$$L(\mathcal{A}) = \{t_{w_1,w_2} \mid (w_1,w_2) \in D\},$$

where t_{w_1,w_2} is the tree in which the nodes w_1 and w_2 are marked. This tree is formally defined by

$$t_{w_1,w_2}(u) = (b_1,b_2) \quad \text{with} \quad b_j = \begin{cases} 1 & \text{if } u = w_j, \\ 0 & \text{otherwise.} \end{cases}$$

For a state $s \in S$, we write $\mathcal{A}_s$ to denote the parity tree automaton that is obtained from $\mathcal{A}$ by making s the initial state. To simplify the following constructions, we assume that every state s of $\mathcal{A}$ is productive, meaning that $L(\mathcal{A}_s) \neq \emptyset$. Otherwise, we can effectively remove all non-productive states.

Furthermore, we define the following subsets of the state set of $\mathcal{A}$, where $\mathrm{Lab}(t) \subseteq \{0,1\}^2$ is the set of all labels occurring in the labeled tree t.

$$\begin{aligned} S_1 &= \big\{ s \in S \mid \mathcal{A}_s \text{ accepts some } t \text{ with } \mathrm{Lab}(t) = \{(1,0),(0,0)\} \big\} \\ S_2 &= \big\{ s \in S \mid \mathcal{A}_s \text{ accepts some } t \text{ with } \mathrm{Lab}(t) = \{(0,1),(0,0)\} \big\} \\ S_{1,2} &= \big\{ s \in S \mid \mathcal{A}_s \text{ accepts some } t \text{ with } \mathrm{Lab}(t) = \{(1,1),(0,0)\} \\ &\qquad \text{or } \mathrm{Lab}(t) = \{(1,0),(0,1),(0,0)\} \big\} \end{aligned}$$

Note that these sets can be effectively constructed.

Every pair of words (i.e., tree nodes) $(w_1, w_2) \in \Gamma^* \times \Gamma^*$ can be written as (wu, wv) such that w is the longest common prefix of the two words and $u, v \in \Gamma^*$ are suitable suffixes. We will now construct for each transition $\tau \in \Delta$ of the tree automaton $\mathcal{A}$ three regular languages $W_\tau, U_\tau, V_\tau \subseteq \Gamma^*$ such that for every pair $(wu, wv) \in \Gamma^* \times \Gamma^*$, we have: $w \in W_\tau$, $u \in U_\tau$, $v \in V_\tau$ if and only if w is the longest common prefix of wu and wv and there is an accepting run of $\mathcal{A}$ on $t_{wu,wv}$ that uses the transition τ at the node w. Then we have

$$\begin{aligned} D &= \big\{ (w_1, w_2) \in \Gamma^* \times \Gamma^* \mid t_{w_1,w_2} \in L(\mathcal{A}) \big\} \\ &= \bigcup_{\tau \in \Delta} W_\tau (U_\tau \times V_\tau), \end{aligned}$$

so D is indeed a prefix-recognizable relation, induced by the prefix-recognizable system (Γ, H) with $H = \{(U_\tau, V_\tau, W_\tau) \mid \tau \in \Delta\}$.

Let $\tau = \big(\widehat{s}, (b_1, b_2), (\widehat{s}_a)_{a \in \Gamma}\big) \in \Delta$ be a transition of the tree automaton $\mathcal{A}$. The language W_τ is defined by an NFA that processes a given input word $w \in \Gamma^*$ by simulating $\mathcal{A}$ on a (fictitious) tree of the form $t_{wu,wv}$, which means that the marked nodes are in the subtree rooted at the node w. More specifically, the NFA simulates $\mathcal{A}$ along the path that leads to the node w, and it accepts w if the resulting state is $\widehat{s}$. Note that the simulated part of the run can be extended to some complete accepting run of $\mathcal{A}$ since all states of $\mathcal{A}$ are productive. Formally, the NFA is defined as $(S, \Gamma, \Delta^{W_\tau}, s_0, \{\widehat{s}\})$ with

$$\Delta^{W_\tau} = \big\{ (s, b, s_b) \mid \big(s, (0,0), (s_a)_{a \in \Gamma}\big) \in \Delta,\ b \in \Gamma \text{ with } s_b \in S_{1,2} \big\}.$$

The language U_τ for $\tau = \big(\widehat{s}, (b_1, b_2), (\widehat{s}_a)_{a \in \Gamma}\big)$ is defined by an NFA that, given an input word $u \in \Gamma^*$, simulates $\mathcal{A}_{\widehat{s}}$ on a tree of the form $t_{u,v}$ such that the longest common prefix of u and v is ε. More precisely, the NFA simulates $\mathcal{A}_{\widehat{s}}$ along the path that leads to the node u, with the

additional requirement that the first transition used by $\mathcal{A}_{\widehat{s}}$ is τ. Note that since all states of $\mathcal{A}$ are productive, the simulated part of the run can be extended to some complete accepting run of $\mathcal{A}_{\widehat{s}}$. The NFA accepts u if it does not get stuck, that is, if such an accepting run of $\mathcal{A}_{\widehat{s}}$ exists. The NFA is formally defined as $(S \cup \{s_0^{U_\tau}, s_f^{U_\tau}\}, \Gamma, \Delta^{U_\tau}, s_0^{U_\tau}, F^{U_\tau})$ with

Note: We use ε-transitions to simplify the construction.

$$\begin{aligned}\Delta^{U_\tau} = {} & \left\{(s_0^{U_\tau}, b, \widehat{s}_b) \mid b \in \Gamma \text{ with } \widehat{s}_b \in S_1\right\} \\ & \cup \left\{(s, b, s_b) \mid \big(s, (0,0), (s_a)_{a\in\Gamma}\big) \in \Delta,\, b \in \Gamma \text{ with } s_b \in S_1\right\} \\ & \cup \left\{(s, \varepsilon, s_f^{U_\tau}) \mid \big(s, (1,0), (s_a)_{a\in\Gamma}\big) \in \Delta\right\}\end{aligned}$$

and

$$F^{U_\tau} = \begin{cases} \{s_0^{U_\tau}, s_f^{U_\tau}\} & \text{if } (b_1, b_2) \in \{(1,0),(1,1)\}, \\ \{s_f^{U_\tau}\} & \text{otherwise.} \end{cases}$$

Finally, the third language V_τ is defined analogously to U_τ, using the set of states S_2 instead of S_1. □

This completes the proof of Proposition 16.4.1 and hence also the proof of Theorem 16.1.3.

17

Conclusion on Games over Infinite Alphabets

In Part II of this thesis, we gave an algorithmic solution for a certain class of Gale-Stewart games over infinite alphabets of the form Σ^* for a finite set Σ. In particular, this applies to the alphabet $\mathbb{N}$, encoded as $\{1\}^*$.

As a first step, we defined a natural model of automata over such alphabets, called $\mathbb{N}$-memory automata, and a corresponding model of automata with output, called $\mathbb{N}$-memory transducers. We developed the basic theory of $\mathbb{N}$-memory automata, including the study of closure properties and decision problems, both on finite and on infinite words.

For Gale-Stewart games with winning conditions defined by deterministic $\mathbb{N}$-memory parity automata, we showed that the winner can be decided and that a winning strategy can be constructed in the form of an $\mathbb{N}$-memory transducer. Our approach is based on a solution of parity games on graphs, a pattern that is known from solutions of Church's Synthesis Problem over finite alphabets – in our case, however, the game graphs are infinite. More specifically, we showed that "MSO-definable" parity games admit the construction of winning strategies represented by MSO formulas.

The presented procedure gives rise to some open questions:

- Can the transducer be constructed in a more direct way, bypassing the route through MSO logic?
- What is the complexity of our algorithm and, more generally, what is the complexity of Church's Synthesis Problem over infinite alphabets as formulated in this thesis?
- Can we strengthen the model of $\mathbb{N}$-memory automata while still preserving the decidability of the synthesis problem?

We should mention that our results can in fact be generalized in the following way. In the framework of $\mathbb{N}$-memory automata, the symbols in the alphabet are finite words, i.e., labelings of a finite "initial segment" of the linearly ordered natural numbers. This concept can be extended to alphabets containing finite trees, i.e., labelings of a finite "initial subtree"

of the Γ-branching tree, for some fixed finite set Γ. An automaton working on a sequence of such finite trees could then memorize a node $u \in \Gamma^*$ of the Γ-branching tree rather than a number $i \in \mathbb{N}$, so we might call it a Γ^*-memory automaton. We may define the transitions relation in terms of MSO-definable sets of trips on trees (rather than words), or equivalently using pebble automata with invisible pebbles as introduced by Engelfriet, Hoogeboom, and Samwel [EHS07].

We can then carry over most of the results from $\mathbb{N}$-memory automata to Γ^*-memory automata. In particular, this includes the solvability of Gale-Stewart games with winning conditions defined by deterministic Γ^*-memory parity automata, with the following caveat: In the case of alphabets of words, we obtained the output function of the transducer implementing a winning strategy by choosing a single output word from a set of valid options. To that end, we used the length-lexicographic well-order on the set of words, which can be defined by an MSO formula (see the proof of Proposition 15.5.2). We cannot apply the same approach for alphabets of trees, since there is no MSO-definable choice function on the nodes of the Γ-branching tree, as shown by Carayol, Löding, and Niwiński [CLN+10]. We can, however, construct a transducer with an output *relation*, representing a "nondeterministic" strategy.

Finally, there is a range of further open issues concerning the model of $\mathbb{N}$-memory automata in its own right:

- Can the classes of ω-languages recognized by deterministic or nondeterministic $\mathbb{N}$-memory parity automata be characterized using logical formalisms? This appears to be a difficult task, as these classes of ω-languages have only poor logical closure properties (see Section 13.3). Alternatively, one may explore whether there are meaningful subclasses of these ω-languages that can be captured in terms of a suitable logic.

- Over finite alphabets, the exact position of the ω-language recognized by a given parity or Muller automaton within the Borel hierarchy can be effectively decided, according to Landweber's Theorem [Lan69]. Does this also hold for $\mathbb{N}$-memory parity or Muller automata (see Section 13.5)?

- The definition of $\mathbb{N}$-memory automata may be generalized by introducing acceptance conditions that depend not only on the states, but rather on the configurations of the respective automaton. What are the implications of such generalizations regarding the expressive power and algorithmic properties of the automata?

Bibliography

[ABB97] J.-M. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 111–174. Springer, 1997. doi: 10.1007/978-3-642-59136-5_3.

[AMM12] B. Aminof, F. Mogavero, and A. Murano. Synthesis of hierarchical systems. In *Proceedings of the 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, volume 7253 of *Lecture Notes in Computer Science*, pages 42–60. Springer, 2012. doi: 10.1007/978-3-642-35743-5_4.

[Bar97] K. Barthelmann. On Equational Simple Graphs. Technical report, Universität Mainz, Institut für Informatik, 1997.

[BC10] A. Blumensath and B. Courcelle. On the monadic second-order transduction hierarchy. *Logical Methods in Computer Science*, 6(2), 2010. doi: 10.2168/LMCS-6(2:2)2010.

[BCL08] A. Blumensath, T. Colcombet, and C. Löding. Logical theories and compatible operations. In J. Flum, E. Grädel, and T. Wilke, editors, *Logic and Automata: History and Perspectives*, volume 2 of *Texts in Logic and Games*, pages 73–106. Amsterdam University Press, 2008.

[BDM+11] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Transactions on Computational Logic*, 12(4):27, 2011. doi: 10.1145/1970398.1970403.

[Bès08] A. Bès. An application of the Feferman-Vaught theorem to automata and logics for words over an infinite alphabet. *Logical Methods in Computer Science*, 4(1), 2008. doi: 10.2168/LMCS-4(1:8)2008.

[BGH+17] S. Böhm, S. Göller, S. Halfon, and P. Hofman. On Büchi one-counter automata. In *Proceedings of the 34th Symposium on Theoretical Aspects of Computer Science (STACS 2017)*, volume 66 of *Leibniz International Proceedings in Informat-*

ics, 14:1–14:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/LIPIcs.STACS.2017.14.

[BGJ+07] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: hardware from PSL. *Electronic Notes in Theoretical Computer Science*, 190(4):3–16, 2007. DOI: 10.1016/j.entcs.2007.09.004.

[BH67] M. Blum and C. Hewitt. Automata on a 2-dimensional tape. In *Proceedings of the 8th Annual Symposium on Switching and Automata Theory (SWAT 1967)*, pages 155–160. IEEE Computer Society, 1967. DOI: 10.1109/FOCS.1967.6.

[BKL11] M. Bojańczyk, B. Klin, and S. Lasota. Automata with group actions. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS 2011)*, pages 355–364. IEEE Computer Society, 2011. DOI: 10.1109/LICS.2011.48.

[BKL14] M. Bojańczyk, B. Klin, and S. Lasota. Automata theory in nominal sets. *Logical Methods in Computer Science*, 10(3), 2014. DOI: 10.2168/LMCS-10(3:4)2014.

[BL69] J. R. Büchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969. DOI: 10.2307/1994916.

[BLT17] B. Brütsch, P. Landwehr, and W. Thomas. $\mathbb{N}$-memory automata over the alphabet $\mathbb{N}$. In *Proceedings of the 11th International Conference on Language and Automata Theory and Applications (LATA 2017)*, volume 10168 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2017. DOI: 10.1007/978-3-319-53733-7_6.

[Blu01] A. Blumensath. Prefix-Recognisable Graphs and Monadic Second-Order Logic. Technical report AIB-06-2001, RWTH Aachen University, 2001. URL: http://aib.informatik.rwth-aachen.de/2001/2001-06.ps.gz.

[BMS+06] M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS 2006)*, pages 7–16. IEEE Computer Society, 2006. DOI: 10.1109/LICS.2006.51.

[Bod98] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1–2):1–45, 1998. doi: 10.1016/S0304-3975(97)00228-4.

[Brü15] B. Brütsch. Synthesizing structured reactive programs via deterministic tree automata. *Information and Computation*, 242:108–127, 2015. doi: 10.1016/j.ic.2015.03.013.

[BS07] H. Björklund and T. Schwentick. On notions of regularity for data languages. In *Proceedings of the 16th International Symposium on Fundamentals of Computation Theory (FCT 2007)*, volume 4639 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2007. doi: 10.1007/978-3-540-74240-1_9.

[BS10] H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*. Fundamentals of Computation Theory, 411(4):702–715, 2010. doi: 10.1016/j.tcs.2009.10.009.

[BT16] B. Brütsch and W. Thomas. Playing games in the Baire space. In *Proceedings of the Cassting Workshop on Games for the Synthesis of Complex Systems (CASSTING 2016)*, volume 220 of *EPTCS*, pages 13–25, 2016. doi: 10.4204/EPTCS.220.2.

[Büc66] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1–11. Elsevier, 1966. doi: 10.1016/S0049-237X(09)70564-6.

[Cac02] T. Cachat. Two-way tree automata solving pushdown games. In E. Grädel, W. Thomas, and T. Wilke, editors, *Automata, Logics, and Infinite Games*. Volume 2500, Lecture Notes in Computer Science, pages 303–317. Springer, 2002. doi: 10.1007/3-540-36387-4_17.

[Cac03] T. Cachat. *Games on Pushdown Graphs and Extensions*. PhD thesis, RWTH Aachen University, 2003. url: https://nbn-resolving.org/urn:nbn:de:hbz:82-opus-9576.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007. url: http://tata.gforge.inria.fr/.

[CDT02] T. Cachat, J. Duparc, and W. Thomas. Solving pushdown games with a Σ_3 winning condition. In *Proceedings of the 16th International Workshop/11th Annual Conference of the EACSL on Computer Science Logic (CSL 2002)*, volume 2471 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2002. DOI: 10.1007/3-540-45793-3_22.

[CFK+15] C. Carapelle, S. Feng, A. Kartzow, and M. Lohrey. Satisfiability of ECTL* with tree constraints. In *Proceedings of the 10th International Computer Science Symposium in Russia (CSR 2015): Computer Science – Theory and Applications*, volume 9139 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2015. DOI: 10.1007/978-3-319-20297-6_7.

[CH14] A. Carayol and M. Hague. Regular strategies in pushdown reachability games. In *Proceedings of the 8th International Workshop on Reachability Problems (RP 2014)*, volume 8762 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2014. DOI: 10.1007/978-3-319-11439-2_5.

[Chu57] A. Church. Application of recursive arithmetic to the problem of circuit synthesis. *Summaries of the Summer Institute of Symbolic Logic*, 1:3–50, 1957.

[Chu63] A. Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians 1962*, pages 23–35. Institut Mittag-Leffler, Djursholm, 1963.

[CLN+10] A. Carayol, C. Löding, D. Niwinski, and I. Walukiewicz. Choice functions and well-orderings over the infinite binary tree. *Open Mathematics*, 8(4), 2010. DOI: 10.2478/s11533-010-0046-z.

[CST15] C. Czyba, C. Spinrath, and W. Thomas. Finite automata over infinite alphabets: two models with transitions for local change. In *Proceedings of the 19th International Conference on Developments in Language Theory (DLT 2015)*, volume 9168 of *Lecture Notes in Computer Science*, pages 203–214. Springer, 2015. DOI: 10.1007/978-3-319-21500-6_16.

[Dav64] M. Davis. Infinite games of perfect information. In *Advances in Game Theory*. Volume 52, number 55 in Annals of Mathematics Studies. Princeton University Press, 1964. DOI: 10.1515/9781400882014-008.

[Dil89] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertation. MIT Press, 1989. DOI: 10.7551/mitpress/6874.001.0001.

[EFT94] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer, 2nd edition, 1994. DOI: 10.1007/978-1-4757-2355-7.

[EHB99] J. Engelfriet, H. J. Hoogeboom, and J.-P. v. Best. Trips on trees. *Acta Cybernetica*, 14(1):51–64, 1999. URL: http://www.inf.u-szeged.hu/actacybernetica/edb/vol14n1/Engelfriet_1999_ActaCybernetica.xml.

[EHS07] J. Engelfriet, H. J. Hoogeboom, and B. Samwel. XML transformation by tree-walking transducers with invisible pebbles. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2007)*, pages 63–72. ACM, 2007. DOI: 10.1145/1265530.1265540.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS 1991)*, pages 368–377. IEEE Computer Society, 1991. DOI: 10.1109/SFCS.1991.185392.

[FHL16] D. Figueira, P. Hofman, and S. Lasota. Relating timed and register automata. *Mathematical Structures in Computer Science*, 26(6):993–1021, 2016. DOI: 10.1017/S0960129514000322.

[FPS12] J. Fearnley, D. Peled, and S. Schewe. Synthesis of succinct systems. In *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis (ATVA 2012)*, volume 7561 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2012. DOI: 10.1007/978-3-642-33386-6_18.

[FPS15] J. Fearnley, D. Peled, and S. Schewe. Synthesis of succinct systems. *Journal of Computer and System Sciences*, 81(7):1171–1193, 2015. DOI: 10.1016/j.jcss.2015.02.005.

[Gel12] M. Gelderie. Strategy machines and their complexity. In *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS 2012)*, volume 7464 of *Lecture Notes in Computer Science*, pages 431–442. Springer, 2012. DOI: 10.1007/978-3-642-32589-2_39.

[Gel14] M. Gelderie. *Strategy Machines: Representation and Complexity of Strategies in Infinite Games*. PhD thesis, RWTH Aachen University, 2014. URL: https://nbn-resolving.org/urn:nbn:de:hbz:82-opus-50253.

[GG66] S. Ginsburg and S. Greibach. Deterministic context free languages. *Information and Control*, 9(6):620–648, 1966. DOI: 10.1016/S0019-9958(66)80019-0.

[GL12] E. Grädel and S. Lessenich. Banach-Mazur games with simple winning strategies. In *Proceedings of the 26th International Workshop/21st Annual Conference of the EACSL on Computer Science Logic (CSL 2012)*, volume 16 of *Leibniz International Proceedings in Informatics*, pages 305–319. Schloss Dagstuhl–Leibniz – Zentrum für Informatik, 2012. DOI: 10.4230/LIPIcs.CSL.2012.305.

[GPV+96] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th International Symposium on Protocol Specification, Testing and Verification (PSTV 1995)*, IFIP Advances in Information and Communication Technology, pages 3–18. Springer, 1996. DOI: 10.1007/978-0-387-34892-6_1.

[GS53] D. Gale and F. M. Stewart. Infinite games with perfect information. In *Contributions to the Theory of Games*. Volume 2, number 28 in Annals of Mathematics Studies. Princeton University Press, 1953. DOI: 10.1515/9781400881970-014.

[HKT10] M. Holtmann, L. Kaiser, and W. Thomas. Degrees of lookahead in regular infinite games. In *Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS 2010)*. Volume 6014, Lecture Notes in Computer Science, pages 252–266. Springer, 2010. DOI: 10.1007/978-3-642-12032-9_18.

[Kec95] A. S. Kechris. *Classical Descriptive Set Theory*, volume 156 of *Graduate Texts in Mathematics*. Springer, 1995. DOI: 10.1007/978-1-4612-4190-4.

[KF94] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994. DOI: 10.1016/0304-3975(94)90242-9.

[KPV10] O. Kupferman, N. Piterman, and M. Y. Vardi. An automata-theoretic approach to infinite-state systems. In Z. Manna and D. A. Peled, editors, *Time for Verification*. Volume 6200, Lecture Notes in Computer Science, pages 202–259. Springer, 2010. DOI: 10.1007/978-3-642-13754-9_11.

[KV99] O. Kupferman and M. Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245–263, 1999. DOI: 10.2307/421091.

[Lan69] L. H. Landweber. Decision problems for ω-automata. *Mathematical Systems Theory*, 3(4):376–384, 1969. DOI: 10.1007/BF01691063.

[LV09] Y. Lustig and M. Y. Vardi. Synthesis from component libraries. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS 2009)*, volume 5504 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2009. DOI: 10.1007/978-3-642-00596-1_28.

[Mad11] P. Madhusudan. Synthesizing reactive programs. In *Proceedings of the 25th International Workshop/20th Annual Conference of the EACSL on Computer Science Logic (CSL 2011)*, volume 12 of *Leibniz International Proceedings in Informatics*, pages 428–442. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. DOI: 10.4230/LIPIcs.CSL.2011.428.

[Man10] A. Manuel. Two variables and two successors. In *Proceedings of the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS 2010)*, volume 6281 of *Lecture Notes in Computer Science*, pages 513–524. Springer, 2010. DOI: 10.1007/978-3-642-15155-2_45.

[Mar75] D. A. Martin. Borel determinacy. *Annals of Mathematics*, 102(2):363–371, 1975. DOI: 10.2307/1971035.

[Min67] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., 1967. ISBN: 978-0-13-165563-8.

[Mos09] Y. N. Moschovakis. *Descriptive Set Theory*, volume 155 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2nd edition, 2009. DOI: 10.1090/surv/155.

[NSV04] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5(3):403–435, 2004. DOI: 10.1145/1013560.1013562.

[Par72] J. B. Paris. ZF $\models \Sigma^0_4$ determinateness. *The Journal of Symbolic Logic*, 37(4):661–667, 1972. DOI: 10.2307/2272410.

[Pit04] N. Piterman. *Verification of Infinite-State Systems*. PhD thesis, Weizmann Institute of Science, 2004.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1989)*, pages 179–190. ACM, 1989. DOI: 10.1145/75277.75293.

[Rab68] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Bulletin of the American Mathematical Society*, 74(5):1025–1029, 1968. DOI: 10.1090/S0002-9904-1968-12122-6.

[Rab72] M. O. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *CBMS Regional Conference Series in Mathematics*. American Mathematical Society, 1972. DOI: 10.1090/cbms/013.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[Seg06] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *Proceedings of the 20th International Workshop/15th Annual Conference of the EACSL on Computer Science Logic (CSL 2006)*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2006. DOI: 10.1007/11874683_3.

[Sén01] G. Sénizergues. L(A)=L(B)? decidability results from complete formal systems. *Theoretical Computer Science*, 251(1):1–166, 2001. DOI: 10.1016/S0304-3975(00)00285-1.

[Sip12] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012. ISBN: 978-1-133-18779-0.

[SK06] L. Sunil Chandran and T. Kavitha. The treewidth and pathwidth of hypercubes. *Discrete Mathematics*, 306(3):359–365, 2006. DOI: 10.1016/j.disc.2005.12.011.

[ST93] P. Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B*, 58(1):22–33, 1993. DOI: 10.1006/jctb.1993.1027.

[STW11] A. Spelten, W. Thomas, and S. Winter. Trees over infinite structures and path logics with synchronization. In *Proceedings of the 13th International Workshop on Verification of Infinite-State Systems (INFINITY 2011)*, volume 73 of *EPTCS*, pages 20–34, 2011. DOI: 10.4204/EPTCS.73.5.

[Tan10] T. Tan. On pebble automata for data languages with decidable emptiness problem. *Journal of Computer and System Sciences*, 76(8):778–791, 2010. DOI: 10.1016/j.jcss.2010.03.004.

[Tan12] T. Tan. An automata model for trees with ordered data values. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012)*, pages 586–595. IEEE Computer Society, 2012. DOI: 10.1109/LICS.2012.69.

[Tan14] T. Tan. Extending two-variable logic on data trees with order on data values and its automata. *ACM Transactions on Computational Logic*, 15(1):8, 2014. DOI: 10.1145/2559945.

[Tho97] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 389–455. Springer, 1997. DOI: 10.1007/978-3-642-59126-6_7.

[Tho98] M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142(2):159–181, 1998. DOI: 10.1006/inco.1997.2697.

[Val73] L. Valiant. *Decision Procedures for Families of Deterministic Pushdown Automata*. PhD thesis, University of Warwick, 1973. URL: http://webcat.warwick.ac.uk/record=b1746283~S15.

[Var98] M. Y. Vardi. Reasoning about the past with two-way automata. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP 1998)*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer, 1998. DOI: 10.1007/BFb0055090.

[VP75] L. G. Valiant and M. S. Paterson. Deterministic one-counter automata. *Journal of Computer and System Sciences*, 10(3):340–350, 1975. DOI: 10.1016/S0022-0000(75)80005-5.

[Wag79] K. Wagner. On ω-regular sets. *Information and Control*, 43(2):123–177, 1979. DOI: 10.1016/S0019-9958(79)90653-3.

[Wal01] I. Walukiewicz. Pushdown processes: games and model-checking. *Information and Computation*, 164(2):234–263, 2001. DOI: 10.1006/inco.2000.2894.

[Wal02] I. Walukiewicz. Monadic second-order logic on tree-like structures. *Theoretical Computer Science*, 275(1-2):311–346, 2002. DOI: 10.1016/S0304-3975(01)00185-2.

[Wol55] P. Wolfe. The strict determinateness of certain infinite games. *Pacific Journal of Mathematics*, 5(5):841–847, 1955. DOI: 10.2140/pjm.1955.5.841.

[Zie98] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998. DOI: 10.1016/S0304-3975(98)00009-7.

Notation

$\sim_n$	n-equivalence relation, 73
X^*	set of words over the alphabet X, 11
X^+	set of nonempty words over the alphabet X, 11
$X^{\leq k}$	set of words over X of length at most k, 11
$X^{<k}$	set of words over X of length less than k, 11
X^ω	set of ω-words over the alphabet X, 11
$\lvert M\rvert$	number of elements of the set M, 11
$\lvert w\rvert$	length of the word w, 11
$[\![e]\!](\sigma)$	value of expression e for the variable valuation σ, 20
$[k]$	set of natural numbers from 0 to k, 11
$[V(G)]^{\leq k}$	set of subsets of $V(G)$ that contain at most k nodes, 53
2^M	powerset of the set M, 11
$\overline{a}$	tuple of the form $(a_1, \ldots, a_k)$, 11
$\mathcal{B}_{\text{bad}}$	Büchi automaton recognizing bad infinite traces, 35
Dom_t	set of nodes (domain) of the tree t, 27
ε	empty word, 11
$\text{FinSig}(p, \mathcal{B}_{\text{bad}})$	finite co-execution signature of the program p, 38
$\text{FinTraces}(p)$	set of finite traces of the program p, 24
$\mathcal{G}(L)$	Gale-Stewart game with the winning condition L, 143
$G_1 \preccurlyeq G_2$	(G_1, id) is stutter-simulated by (G_2, ℓ) for some ℓ, 49
G_p	transition graph of the program p, 46
H_d	hypercube of dimension d, 50
id	identity function, 49
$\text{InfSig}(p, \mathcal{B}_{\text{bad}})$	infinite co-execution signature of the program p, 38
$\text{InfTraces}(p)$	set of infinite traces of the program p, 24
$\text{io}(\alpha)$	I/O sequence of the trace α, 24
$\text{io}(p)$	set of I/O sequences of the program p, 24

L^*	iteration of the language L, 11
$\mathcal{M}(p)$	IOI machine associated with the program p, 21
mark(T)	set of marked ω-words for the set of trips T, 78
mark$(\widehat{u}, i)$	marked ω-word associated with u and i, 78
mark$(\widehat{u}, i, j)$	marked ω-word associated with u, i, and j, 78
mark(V)	set of marked ω-words for the set of vistas V, 130
$\mathbb{N}$	set of natural numbers, 11
$\mathbb{N}_{\geq 1}$	set of natural numbers without 0, 11
$\mathcal{N}$	structure $(\mathbb{N}, \mathrm{Succ})$, 68
$p \xrightarrow{a/v} q$	pushdown rule, 155
$Q \times \mathcal{S}$	product of the structure $\mathcal{S}$ with the finite set Q, 68
relPref(α)	shortest relevant prefix of the ω-word α, 82
relPref(L)	set of shortest relevant prefixes of the ω-words in L, 82
S_v	word set determined by the input sequence v, 49
$\sigma[z/a]$	variable valuation σ with variable z set to value a, 20
Σ, Γ	finite alphabets, 11
$\widehat{\Sigma}$	in Part I: union of the alphabets $\widehat{\Sigma}_\mathrm{I}$, $\widehat{\Sigma}_\mathrm{O}$, $\widehat{\Sigma}_\tau$, 20
	in Part II: extended alphabet $\Sigma \cup \{\bot, \#\}$, 78
$\Sigma_\mathrm{I}, \Sigma_\mathrm{O}$	input/output alphabets for structured programs, 19
$\widehat{\Sigma}_\mathrm{I}, \widehat{\Sigma}_\mathrm{O}$	augmented input/output alphabets, 20
$\widehat{\Sigma}_\tau$	alphabet $\{\tau\}$, used to indicate assignment steps, 20
$\mathcal{T}_\Gamma$	infinite Γ-branching tree, 72
T_Σ	set of finite trees over the alphabet Σ, 27
tree(p)	tree representation of the program p, 28
Trips$(\mathcal{B})$	set of trips computed by the pebble automaton $\mathcal{B}$, 80
Trips(φ)	set of trips induced by the formula φ, 79
tw(G)	tree-width of the graph G, 53
$\widehat{u}$	ω-extension $\bot u \#^\omega$ of the word u, 78
Vistas$(\mathcal{C})$	set of vistas computed by the pebble printer $\mathcal{C}$, 132
Vistas(φ)	set of vistas induced by the formula φ, 130
$W(U \times V)$	relation induced by the rewrite rule (U, V, W), 164
W^ω	infinite iteration of the language W, 12

Index