Certificates for automata in a hostile environment

Sebastian Muskalla

Note

This thesis has **not** been reviewed and approved by the Carl-Friedrich-Gauß-Fakultät of the Technische Universität Braunschweig.

This version of the thesis is **not** identical to the one that has been submitted to the Carl-Friedrich-Gauß-Fakultät of the Technische Universität Braunschweig with the goal of obtaining a PhD (Dr. rer. nat.).

2022

Abstract

The automated verification of the runtime behavior of a program with respect to a specification is a difficult problem, as demonstrated by undecidability results due to Turing, Rice, and others. The automata-theoretic approach to verification consists of abstracting the program into an automaton, a restricted model of computation, while preserving the part of the program behavior that is relevant to the specification. The field of automata theory provides various classes of such *automata* and studies the trade-off between their expressiveness and the decidability and computational complexity of their algorithmic problems. When taking a language-theoretic approach, one associates to an automaton the sets of words it can generate and studies algorithmic problems in which the task is to decide properties of these languages.

Decision procedures for automata resp. their languages can be used in verification by seeing the input automaton in isolation as a perfect model of the system that should be verified. This view, however, has two shortcomings. The first one is that because an automaton is an abstraction of the real system, one call to a decision procedure is oftentimes insufficient. Typically, a multitude of such procedure calls is needed, e.g. when using a refinement loop that improves the abstraction in each iteration, or when dealing with the verification of a concurrent system in which each component is considered separately. Overcoming this deficiency requires procedures that in addition to the Boolean answer to a decision problem also return a *certificate*, an easily verifiable proof justifying the yes/no answer. In a setting in which a decision procedure is invoked multiple times, the certificates produced by earlier calls can serve as an additional input for subsequent calls of the procedure, facilitating the verification process. The second shortcoming is that in many cases, the automaton under consideration is not isolated. It interacts with an *environment* that is *hostile* with respect to our goal of verifying the system. This hostile environment may stem from user input, communication with components not modeled as part of the system, unreliable network communication, or it may simply be the result of discarding some parts of the system when abstracting it into an automaton.

Our claim is that in order to enable the automata-theoretic approach to verification, decision procedures for *automata* should provide *certificates* and take the *hostile environment* into account. This thesis provides such procedures for three different settings.

In the first setting, we assume that communication with the system under consideration is done using an unreliable network that is either lossy or gainy. The observable output of the system is a subsequence resp. supersequence of the real output, which can be modeled as the downward resp. upward of the language of the system. It is known that such a *language closure* always stems from the simple class of regular languages, which in particular means that a representation for it can serve as a certificate. However, computing this representation is not trivial depending on the class of automata the initial system stems from. In this thesis, we consider Petri

nets with coverability as the acceptance condition, a model that is well-known to be suitable as a representation for concurrent systems. We prove a collection of results that show how to construct representations of the downward and upward closures with optimal size and within optimal time for both Petri nets and restrictions thereof.

The second setting is the compositional verification of a concurrent system. This approach tries to verify each component of the system on its own, avoiding the state explosion problem that plagues the verification of concurrent systems. When focusing on a single component, the rest of the components form an environment that has to be taken into account. We argue that the assume-guarantee approach to compositional verification is closely related to the problem of *regular separability*. Two languages are regularly separable if there is a regular language containing one and being disjoint from the other. This regular separator serves as a certificate for intersection-emptiness and it can be used as an overapproximation of the language it contains. We show that in the case of languages of well-structured transition system (WSTSes), a generalization of the aforementioned class of Petri net coverability languages, any two disjoint WSTS languages are regularly separable. From our proof, we obtain a construction of the separator.

Finally, we consider *games* played on game arenas induced by automata. These games model situations in which two kinds of nondeterminism influence the behavior of the system. Usually, one type of nondeterminism is favorable to the goal of verifying the system, while the other type represents the hostile environment. This situation comes up e.g. when verifying branching systems or when solving synthesis problems. We present an approach to solving such games that is based on effective denotational semantics. That is, we turn the automaton into a system of equations whose least solution provides the winner of the game. Additionally, the winning strategy for the game, which can be seen as a certificate, can also be read off from the least solution. We design algorithms that are based on effective denotational semantics for various kinds of games induced by automata, including games defined by context-free grammars and higher-order recursion schemes. Lastly, we study the frontier of the decidability of games on arenas induced by valence systems over graph monoids and establish that context-free games are the only type of these games that can be solved.

Zusammenfassung

Die automatisierte Verifikation des Laufzeitverhaltens eines Programms entsprechend einer Spezifikation ist ein kompliziertes Problem, wie durch die Unentscheidbarkeitsresultate von Turing, Rice und anderen gezeigt wurde. Die automatentheoretische Herangehensweise an dieses Problem besteht darin, das Programm zu einem Automaten, einem eingeschränkten Berechnungsmodell, zu abstrahieren, dabei allerdings den Teil des Programmverhaltens, der für die Spezifikation relevant ist, zu erhalten. Das Gebiet der Automatentheorie stellt eine Reihe solcher *Automaten* zur Verfügung und untersucht den Zielkonflikt zwischen dem Erreichen möglichst hoher Ausdruckskraft und der Entscheidbarkeit und Berechnungskomplexität ihrer algorithmischen Probleme. Wenn ein sprachtheoretischer Ansatz gewählt wird, assoziiert man zu einem Automaten die Menge der von ihm generierten Wörter und untersucht algorithmische Probleme, bei denen es das Ziel ist, Eigenschaften dieser Sprachen zu entscheiden.

Entscheidungsverfahren für Automaten bzw. für ihre Sprachen können in der Verifikation genutzt werden, indem man den gegebenen Automaten als ein perfektes Modell für das System sieht, welches verifiziert werden soll. Dieser Ansatz hat jedoch zwei Unzulänglichkeiten. Die Erste ist, dass ein einzelner Aufruf einer Entscheidungsprozedur meist nicht ausreicht, da ein Automat lediglich eine Abstraktion des zu verifizierenden Systems ist. Typischerweise ist eine Vielzahl von Prozeduraufrufen nötig, z.B. wenn eine Schleife genutzt wird, die die Abstraktion in jeder Iteration verfeinert, oder wenn ein nebenläufiges System verifiziert wird, bei dem jede Komponente einzeln betrachtet wird. Das Überwinden dieser Unzulänglichkeit erfordert Prozeduren, die zusätzlich zu ihrem Bool'schen Ergebnis auch ein *Zertifikat* liefern, also einen leicht überprüfbaren Beweis als Begründung für die Ja/Nein-Antwort. In einem Szenario, in dem eine Entscheidungsprozedur mehrfach aufgerufen wird, können die Zertifikate, die von früheren Aufrufen generiert werden, als zusätzliches Argument für weitere Aufrufe genutzt werden, um den Verifikationsprozess zu erleichtern. Die zweite Schwäche ist, dass der Automat meist nicht isoliert betrachtet werden kann. Er interagiert möglicherweise mit einer *Umgebung*, die sich gegenüber dem Verifikationsziel *feindlich* verhält. Diese Umgebung kann aus Benutzereingaben, der Kommunikation mit externen Komponenten, die im System nicht modelliert sind, unzuverlässiger Kommunikation über ein Netzwerk oder einfach daraus resultieren, dass bei der Abstraktion des Systems in einen Automaten manche Aspekte verloren gegangen sind.

Unsere These ist, dass die Ermöglichung der automatentheoretischen Herangehensweise an die Verifikation von Programmen Entscheidungsprozeduren für *Automaten* benötigt, die *Zertifikate* liefern und die die *feindliche Umgebung* berücksichtigen. Die hier vorliegende Arbeit stellt solche Prozeduren für drei verschiedenen Szenarien zur Verfügung.

Im ersten Szenario gehen wir davon aus, dass die Kommunikation mit dem zu betrachtenden System über ein unzuverlässiges Netzwerk abgewickelt wird, welches verlustbehaftet ist. Die

beobachtbare Ausgabe des Systems ist eine Teilsequenz der tatsächlichen Ausgabe, was als Abschluss der Sprache des Systems nach unten modelliert werden kann. Analog dazu kann man eine Situation betrachten, in der die beobachtbare Ausgabe des Systems die tatsächliche Ausgabe als eine Teilsequenz enthält, was dem Abschluss der Sprache nach oben entspricht. Es ist bekannt, dass diese *Sprachabschlüsse* immer aus der einfachen Klasse der regulären Sprachen stammen, was insbesondere bedeutet, dass ein sie repräsentierender Automat als Zertifikat geeignet ist. Einen solchen Automaten zu berechnen ist jedoch ein nicht-triviales Problem, abhängig davon, aus welcher Klasse die ursprüngliche Sprache kommt. In dieser Arbeit betrachten wir Petri-Netze mit Überdeckbarkeit als Akzeptanzbedingung, eine Klasse von Automaten, welche bekannt dafür ist, für die Modellierung nebenläufiger Systeme geeignet zu sein. Wir beweisen eine Reihe von Resultaten, durch die wir zeigen, wie im Fall von Petri-Netzen Automaten optimaler Größe, die den Sprachabschluss nach unten bzw. oben repräsentieren, mit optimalem Zeitverbrauch berechnet werden können.

Das zweite Szenario ist die kompositionelle Verifikation nebenläufiger Systeme. Diese Herangehensweise besteht darin, jede Komponente eines solchen Systems isoliert zu verifizieren, um damit das Problem der Zustandsraumexplosion zu vermeiden. Beim Betrachten einer einzelnen Komponente bilden die anderen Komponenten eine feindliche Umgebung, die berücksichtigt werden muss. Wir argumentieren, dass der sogenannte Assume-Guarantee-Ansatz für kompositionelle Verifikation mit dem *regulären Separierbarkeitsproblem* verwandt ist. Zwei Sprachen sind regulär separierbar, wenn es eine reguläre Sprache gibt, die eine der Sprachen beinhaltet, aber von der anderen disjunkt ist. Dieser reguläre Separator dient als Zertifikat für die Leerheit des Schnitts der Sprachen und er kann als Überapproximation für die Sprache, die er beinhaltet, verwendet werden. Wir zeigen, dass für die Sprachen von wohlstrukturierten Transitionssystemen (WSTS), eine Verallgemeinerung der oben genannten Petri-Netz-Überdeckbarkeitssprachen, gilt, dass zwei disjunkte WSTS-Sprachen immer regulär separierbar sind. Aus unserem Beweis resultiert eine Konstruktion für den Separator.

Im letzten Szenario betrachten wir *Spiele*, die auf von Automaten induzierten Spielbrettern gespielt werden. Diese Spiele modellieren Situationen, in denen zwei Arten von Nichtdeterminismus das Verhalten des Systems beeinflussen. Typischerweise ist dabei eine Art des Nichtdeterminismus hilfreich bei der Verifizierung des Systems, während die andere die feindliche Umgebung repräsentiert. Diese Situation entsteht z.B. bei der Verifikation verzweigender Systeme und bei Syntheseproblemen. Wir stellen eine Herangehensweise zum Lösen solcher Spiele vor, die auf effektiver denotationeller Semantik beruht. Das bedeutet, dass wir den Automaten in ein Gleichungssystem übersetzen, dessen kleinste Lösung den Gewinner des Spiels liefert. Zusätzlich kann die Gewinnstrategie, welche als Zertifikat dient, auch von dieser kleinsten Lösung abgelesen werden. Wir entwerfen auf effektiver denotationeller Semantik basierende Algorithmen für mehrere durch Automaten induzierte Spiele, darunter Spiele, die durch kontextfreie Systeme sowie durch Rekursionsschemata höherer Ordnung definiert werden. Schlussendlich untersuchen wir die Grenze der Entscheidbarkeit von Spielen, die von Valenzsystemen über Graphmonoiden induziert sind, und zeigen, dass nur bei kontextfreien Spielen der Gewinner berechnet werden kann.

Contents

Abstract 5

Zusammenfassung 7

I. Introduction 11

1 Introduction . . . 13
2 Outline . . . 49

II. Models of computation 51

3 Preliminaries . . . 55
4 Labeled transition systems and finite-state automata . . . 73
5 Grammar-based models . . . 89
6 Petri nets and well-structured transition systems . . . 109

III. Closures of Petri net languages 139

7 Closures of Petri net languages . . . 143
8 Upward closures . . . 151
9 Downward closures . . . 175
10 Being upward/downward closed and regular containment . . . 205

IV. Separability 211

11 Separability and regular separability . . . 215
12 WSTS expressiveness . . . 223
13 Regular separability for WSTSes . . . 235
14 Bounds on the separator size for Petri nets . . . 255

V. Games 271

15 Games with perfect information . . . 275
16 Effective denotational semantics . . . 285
17 Context-free games . . . 313
18 Higher-order games . . . 375
19 Valence games . . . 433

VI. Conclusion 473

20 Contributions . . . 475
21 Future work . . . 479

Bibliography 483

The title page of each part provides a table of contents listing the sections within each chapter.

Part I.

Introduction

Contents

1 Introduction 13

1.1 Compositional verification and separability 28

1.2 Unreliable communication and language closures 35

1.3 Games 40

2 Outline 49

1 Introduction

Contents

1.1 Compositional verification and separability 28
1.2 Unreliable communication and language closures 35
1.3 Games 40

Which problems can be solved by computers – and how? This question lies at the heart of computer science. *Theoretical computer science* studies which problems can be solved *in principle*. To approach this difficult question, we start from the related area of *verification*. Verification problems consist of checking whether a given system – usually a program, specified by its source code – is correct with respect to a certain specification. Interest in verification is also motivated by its practical importance, given the ubiquity of computers, in particular their usage in applications like aviation where failure may lead to lethal accidents. The deep link between verification and theoretical computer science comes from the theme of *self-application*: To understand which problems can be solved by computers, one tries to understand which properties of computers can be decided by other computers. The principle of self-application has a history reaching back to the beginning of the axiomatization of mathematics and theoretical computer science, demonstrating the usefulness of the approach. It is used in Russel's paradox [Rus03], in the proof of Gödel's incompleteness theorem [Göd31], and in the proof of Turing's famous result that the halting problem is undecidable [Tur36].

In fact, Turing's result can be understood as a contribution to the area of verification. He essentially has shown that it is impossible to algorithmically check the termination of a given (imperative) program. Similarly, Church's earlier undecidability result [Chu36] for checking the equivalence of λ-calculus terms can be understood in terms of functional programs: It is not decidable if two given programs have the same runtime behavior. Finally, Rice's theorem [Ric53] shows that the problems studied by Church and Turing are not undecidable because of their exceptional hardness. Rather, these are problems for which the proof of undecidability is not too difficult. Any other problem in the area of verification is just as undecidable: A computer cannot decide non-trivial properties of other computer systems in general.

This undecidability conflicts with the practical interest in verification, caused by the need for correct programs in safety-critical applications. A number of workarounds have risen to fame, including the unit tests and path coverage tests that are the current industry standard [Bei90; MSB12]. However, these tests only show the absence of incorrect behavior in certain executions selected by a human. In order to prove the absence of bugs in *all* executions, researchers in theoretical computer science have developed models that make it easier for humans to prove correctness by hand, like abstract state machines [Gur00; Bör97], and they have also developed

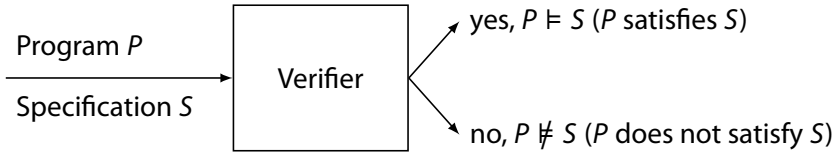


Figure 1.0.a: An ideal verifier.

methods for computer-aided verification. For example, a method based on Hoare logic [Hoa69] may complete a proof of correctness after certain parts of it, so-called loop invariants, have been specified by a human.

Regardless of the advances in these areas, the fully automatic verification of programs remains the ultimate goal: A computer system that checks all executions of a program for correctness within finite time, without requiring the time investment by or the ingenuity of a human. Figure 1.0.a depicts such an ideal verifier. The undecidability results by Church, Turing, and Rice prove that it cannot exist, but luckily, they leave two loopholes that one can exploit. The first one is that the results state that verification cannot be solved in general, i.e. for all input programs. The second loophole is that the undecidability results apply to a scenario that is symmetric: Both the input and the solver come from the class of general computer programs.

The first loophole means that it may be possible to construct an algorithm that is able to determine the correctness of *some* input programs in finite time, while it may fail to do so for others. In fact, many verification problems are semi-decidable, e.g. for some verification problem, it may be possible to always find a violation of the specification within finite time if the input program is indeed incorrect. An algorithm for the problem that can always disprove correctness in finite time while also being able to prove correctness for some programs does not violate the aforementioned undecidability results while potentially being very useful in practice.

The second loophole means that if we assume that we want to solve a verification problem using a general computer, but the input programs come from a restricted class of programs, the undecidability results do not apply. This has led to various classes of systems, so-called *automata*, being studied in the context of verification.

Automata

We use automata as a broad term for all kinds of computational devices that have a finite description. This includes both models for general computers or computer programs, like Turing machines, and weaker models. If we study verification problems for weaker models, i.e. with the assumption that the program that is the input to the verification problem is from a certain class of automata, we can hope for the problem to be decidable. Automata theory, the subarea of theoretical computer science which studies automata, provides a long list of these models that differ from each other mainly in two ways. The first is how expressive they are, i.e. how close they are to being able to model a real computer. The second is how amenable they are to automatic verification, i.e. which verification problems can be solved algorithmically if we

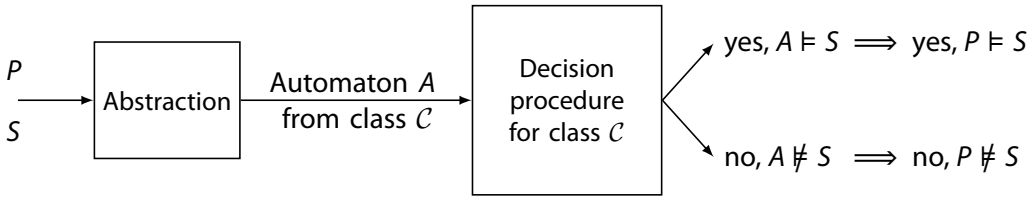


Figure 1.0.b: An ideal verifier using the automata-theoretic approach.

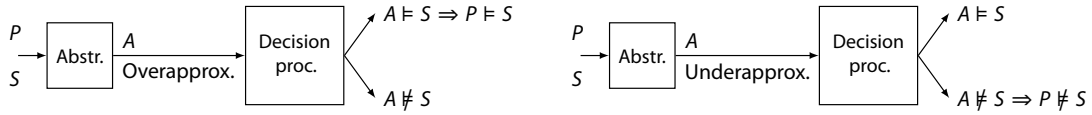
assume the input programs come from that class, and if they can be solved, how efficiently. As expected, there is a tradeoff between the two aspects: Instances of a very simple model can be analyzed easily, but they are not very expressive. Turing machines, a complex type of automata, are able to model any real-world computer program, but the undecidability results apply and automatic verification is impossible. We will provide more examples for automata models later.

Combined, the two aforementioned loopholes enable the automata-theoretic approach to verification. Given a verification problem, one constructs an algorithm that works in two steps. The first step is an *abstraction* step that transforms the input program into an instance of an automata model. In the second step, one invokes an algorithm for the equivalent verification problem for that class of automata. The hope is that the output of the algorithm deciding the verification problem for the automaton coincides with the desired answer to the verification problem for the given input program.

A schematic ideal verifier using this automata-theoretic approach is depicted in Figure 1.0.b. Unfortunately, just like the more general ideal verifier, it cannot exist. Assume we choose a class of automata that is expressive enough to capture the behavior of real computer programs as the target of the abstraction. Then the decision problem for this class of automata that corresponds to the undecidable verification problem that we want to solve is also undecidable. If we choose a strictly weaker class, the corresponding decision problem might become decidable. However, in that case, the abstraction will lead to a loss of information. We cannot guarantee that the answer to the decision problem, i.e. whether the automaton satisfies the specification, is equal to the answer to the verification problem, i.e. whether the program satisfies the specification.

One way to circumvent this problem is to choose the abstraction carefully. This includes choosing the right class of automata as the target for the abstraction. We will come back to this aspect at the end of this section when we mention various models of automata and their strengths and weaknesses. For now, we focus on a more general approach that works for various classes of automata. The idea is to choose the abstraction so that it either underapproximates or overapproximates the given program. These are ways to abstract a program into an automaton from a restricted class, thus avoiding undecidability, while still retaining some information.

An *underapproximation* yields an automaton so that any execution of the automaton represents a valid execution of the program, but not every execution of the program is necessarily reflected in the behavior of the automaton. Consequently, an execution of the automaton violating the specification is also an execution of the program violating it. In summary, we obtain that if



i) Using overapproximation. ii) Using underapproximation.

Figure 1.0.c: Verifiers using the automata-theoretic approach as in Figure 1.0.b, but the abstraction approximates the behavior of the input program. We may avoid undecidability, but only in one of the two cases, the answer provided by the decision procedure has mandatory implications for the correctness of the input program.

the automaton violates the specification, so does the program. If the automaton satisfies the specification, we do not know whether the initial program is indeed correct or whether it was incorrect, but the abstraction has led to losing the violating executions. Hence, underapproximations are useful for finding bugs, but a priori they are not suitable for proving correctness.

An *overapproximation* leads to an automaton with a larger set of possible executions: Any execution of the program is represented by an execution of the automaton, but the automaton may have additional *spurious executions*. If the automaton is then shown to be correct, all its executions – including all executions of the original program – satisfy the specification. If the automaton is incorrect, we do not know whether this comes from the program being incorrect or whether it is caused by a spurious execution introduced by the abstraction. Hence, overapproximations are suitable for proving correctness, but in order to handle incorrect input programs, we would need some way to deal with spurious violations of the specification. Both concepts – underapproximations and overapproximations – are depicted in Figure 1.0.c. In the rest of this chapter, we will focus on examples using overapproximations.

One way to obtain an overapproximation of a program is by using a control flow abstraction. This means we abstract a program into an automaton that just models the control flow, while we discard the data values. Conditional branching in the program that jumps to one of several branches depending on a data value is replaced by the nondeterministic choice among these branches in the automaton. Consequently, even if the input program was deterministic, the result of the control-flow abstraction is typically a nondeterministic automaton whose behavior is an overapproximation of the program. The overapproximation can be made more precise using *predicate abstraction* [GS97]. We assume that we have a finite collection of predicates, functions that map the state space of the program (including the data values) to Boolean values. In addition to the control flow of the program, the automaton that is the result of applying predicate abstraction also keeps track of the values of these predicates. We can resolve conditional branching deterministically whenever the description of the data values provided by the predicates is sufficiently precise. Otherwise, we still rely on nondeterminism.[1] The result is still

[1] For example, consider a program storing an integer n. Instead of storing n, a predicate abstraction may just store the parity of n. When the program branches depending on whether n is even, we can resolve this choice. When it branches depending on whether n equals 0, we have to use nondeterminism. Similarly, when an instruction increments n by one, we can update the state of automaton deterministically because an increment will flip the parity of n. When an instruction halves n, we have to use nondeterminism as we do not know the resulting parity.

an overapproximation, but depending on the choice of the predicates, it can be much more precise than a simple control flow abstraction.

The concept of approximations gives us a template for the construction of verification procedures that are useful, although not very sophisticated. (We will discuss more involved concepts that can deal with e.g. the shortcomings of overapproximations later.) In order to instantiate this template, one needs abstractions and decision procedures for automata models. Practical research in verification has come up with a wide selection of abstractions that transform programs into instances of various automata models. While we mention a few of them below, abstractions are not the focus of this thesis.

The goal of this thesis is to enrich the toolkit of automata theory by providing procedures that solve decision problems related to automata with optimal resource consumption. We observe that in order to be useful in practice, a verification procedure that uses automata often needs *to produce and use certificates* and to *interact with the environment*. Hence, the decision procedures that we contribute should take these aspects into account. In the rest of this section, we will explain what we mean by certificates and the environment. We will then give an introductory example, demonstrating the usefulness of these concepts.

Certificates

Before discussing why certificates are useful in the context of the automata-theoretic approach to verification, we will give a general definition. Assume we have an instance of a decision problem and the information whether it is a yes- or a no-instance. A certificate is additional information that proves that the Boolean yes/no answer is indeed correct. Typically, it is easier to verify the answer using the certificate than to compute the answer without prior knowledge.

An easy example is intersection-emptiness problems. Assume we are given two (representations of) sets. It is usually easier to verify that a given element is contained in both sets than to compute that the two sets have a non-empty intersection. Hence, an element of the intersection can serve as a certificate for the non-emptiness of the intersection of two sets. Verification problems can often be seen as intersection-emptiness problems: A program represents a set of possible executions while a specification gives rise to a set of executions that violate the specification. If the two sets are disjoint, all possible executions are valid and the program is correct. If the two sets have a non-empty intersection, the program is incorrect. An element of the intersection as a certificate for non-emptiness is an execution of the program that violates the specification. Hence, it is indeed a certificate for the incorrectness of the program. Producing certificates in the case of an empty intersection resp. a program that is correct is a more involved problem that we will discuss in the next section of this chapter, Section 1.1.

It is noteworthy that the modern definition of the important complexity class NP [Coo00] also uses certificates. Traditionally, NP has been defined as the problem solvable by nondeterministic Turing machines (NTMs) in polynomial time. NTMs are a model that is often perceived to

be hard to understand, in particular because NTMs cannot be implemented in practice. To circumvent this problem, NP can equivalently be defined as the class of decision problems such that each yes-instance has a certificate that allows a deterministic algorithm to verify that it is indeed a yes-instance (with the constraint that the size of the certificate and the running time of the verifier are polynomial).

There are multiple reasons for being interested in algorithms that do not only compute a yes/no answer for a decision problem that they solve, but also a certificate. Firstly, it makes the algorithms accountable: Instead of blindly trusting the result produced by a procedure, the certificate can be used to check its correctness. This makes it easier to develop advanced and highly optimized algorithms that are correct by using a well-understood and less-optimized procedure to check the certificates during the development process. For example, in the SAT competition, a competition in which state-of-the-art algorithms for the satisfiability problem of propositional logic are benchmarked, the algorithms are required to provide certificates both for yes- and for no-instances [HIJSB18].

Secondly, there are cases in which it is not the Boolean answer to the decision problem that is relevant in practice, but the certificate for that answer. This is true in particular for verification problems: If the answer to a verification problem is negative and the program is incorrect, the developers of the program will most likely require a violating execution to be able to find, understand, and resolve a bug in the program. In the last section of this chapter, we will briefly discuss decision problems related to synthesis for which a certificate for the answer, a so-called *strategy*, is needed in order to actually perform the synthesis task.

Verification procedures that use the automata-theoretic approach may invoke a decision procedure not once, but multiple times. For example, each invocation may correspond to a different part of the system that should be verified. The first call produces a certificate and subsequent calls will take the certificates produced by earlier calls as additional inputs. We will discuss compositional verification as an example for this concept in detail later. Alternatively, a procedure may work with a sequence of refinements of the abstraction of the system, calling the decision procedure once for each refinement. To refine an abstraction in that sequence in order to obtain the next one, one calls the decision procedure and then uses the resulting certificate. We briefly discuss the CEGAR loop as an example.

Counter-example guided abstraction refinement (CEGAR) [CGJLV00] is a technique that was first developed in the context of *abstract interpretation*. Abstract interpretation [CC77] is an approach to verification that is in a certain sense the antithesis to the automata-theoretic approach. The automata-theoretic approach takes the input program, abstracts it, and then computes precise information about the abstraction. Abstract interpretation takes the unmodified program and computes imprecise or abstract information about it, in the hope that this information is sufficient to settle the answer to the verification problem. In spite of this contrast, many techniques can be brought from one world to the other. For example, the CEGAR approach was brought to the world of automata theory by Podelski and his coauthors, see e.g. [HHP10]. We

have previously mentioned that an overapproximation of a program is useful for verification in the sense that if the overapproximation is correct, then so is the initial program. If the overapproximation is incorrect, we will have to determine whether this is due to an execution of the initial program that violates the specification, or just due to a spurious execution introduced by the overapproximation. The CEGAR approach consists of constructing an initial overapproximation, and then executing the following steps in a loop: We verify the current overapproximation by calling an automata-theoretic decision procedure. If the overapproximation is correct, then so is the program and the procedure terminates. Otherwise, we assume that the decision procedure yields an incorrect execution as the certificate. We check whether the violating execution is indeed an execution of the given program. In contrast to the verification problem, this is a decidable question. If the violating execution is a real execution of the program, the program is incorrect and the procedure terminates. Otherwise, the violating execution is a spurious counterexample, and we can obtain a proof for its spuriousness as a second certificate. We then use this proof as a guide to construct the description of a set of spurious counterexamples that includes the one that we obtained earlier. We construct a new overapproximation whose set of possible executions is the old set of possible executions minus this set, then restart the loop.

Discussing the details of the procedure, e.g. how we check whether an execution is spurious and how the refinement works in detail, is beyond our scope. If the basic CEGAR principle outlined here is combined with some implementation tricks, one ends up with a procedure that has been shown to perform well in practice [HCDGN+17]. The loop is not guaranteed to terminate, meaning that the CEGAR approach does not solve the verification problem within finite time in all cases and hence its existence does not contradict the undecidability results. Even if the program is correct, we may not be able to find an overapproximation that is precise enough to prove correctness within finite time. Similarly, even an incorrect program that actually has a violating execution may still cause us to consider spurious executions in the loop ad infinitum. However, for many programs of practical interest, the procedure will within finite time either prove that the program is correct by finding a suitable overapproximation or prove incorrectness by finding a violating execution. Certificates are crucial in two steps of the loop in order to deal with spurious counterexamples: We need a certificate to decide whether a violation that we have found is spurious, and we need a certificate for the refinement of the approximation. Hence, certificates are absolutely essential to make the CEGAR approach work.

The environment

When we apply an automata-theoretic decision procedure to an abstraction of a system, there is a part of the system that is not reflected in the automaton because it has been lost during the abstraction. We call this part of the system the *environment*. It is *hostile* to the verification process because it is the essence of what may prevent the result of the decision procedure from coinciding with the desired answer to the original verification problem. In order to succeed at verification, an algorithm has to interact with the hostile environment at some point. Only if

we make sure that every possible behavior of the environment is taken into account, we can guarantee that the output of the decision procedure is correct.

In an algorithm that is based on overapproximation, the environment corresponds to the set of spurious executions. These are the executions that are present in the automaton but not in the initial program. If the abstraction that is obtained by overapproximating is correct, one can deduce the correctness of the program; the environment has been taken care of by the overapproximation. But if the abstraction is incorrect, interacting with the environment is necessary in order to have a chance at completing the verification task. In the aforementioned CEGAR approach, if the algorithm finds that the current overapproximation violates the specification, it queries the environment in order to check whether the violating execution is spurious. If it is, we also obtain from the environment a proof of spuriousness that is then used for refinement. Only by using the environment, we can obtain a powerful tool that can deal with real programs even if the initial overapproximation is not correct.

The initial system that the verification procedure is starting with is usually not living in a vacuum, but also in what is often called an environment. For example, a program may accept user inputs or it may communicate with a reactive system like a database server. These external dependencies are typically not fully specified in a description of the system. Our notion of environment may seem to be in conflict with this more common type of environment. However, we argue in the following that the two notions of environment are essentially the same.

Firstly, the two types of environments play the same role. In both cases, we are interested in the correctness of a system, but our algorithm has to work with an abstract description of the system that is missing some parts. It does not matter whether the description is incomplete because we have applied an abstraction to a more comprehensive model of the system, or whether the description is missing some external dependencies that were never explicitly modeled to begin with.

Secondly, the two types of environments usually manifest themselves in the same way in the abstraction. Hence, they can also be dealt with in the same way. For example, it is common to model external dependencies using nondeterministic branching: A program location at which the program queries a database server can be seen a nondeterministic choice between various transitions, one for each possible result. Nondeterminism also comes into play when we apply an abstraction. As we have explained earlier, an abstraction that removes a data value means that conditional branching in the program that depends on this data value has to be turned into nondeterministic branching in the automaton. In both cases, the abstraction overapproximates the behavior of the original program. This imprecision is handled by seeing the nondeterminism as hostile to the verification task. A nondeterministic system is usually only considered to be correct if all of its executions that may result from resolving the nondeterminism at runtime are correct. When we find a violating execution, we may interact with the environment to find out whether the execution is spurious. If the nondeterminism comes from an external dependency like a database server, we might check a specification of the database server to find out if

the behavior in the spurious execution is actually possible. If the nondeterminism comes from the abstraction that we had applied earlier, we might check whether the execution is a valid execution of the original program, like in the CEGAR loop. In the rest of this thesis, we will not conceptually distinguish the two types of environment.

We give some additional examples on how a verification algorithm may take the environment into account. Each of these examples corresponds to one of the three main topics that this thesis is concerned with. In each example, the way in which the environment is specified and hence also the way in which it is interacted with is different.

The first example is the case of a system with which we interact through unreliable communication. We have a description of the system, but what we are actually interested in is its visible behavior. Hence, we may see the unreliability of the communication channel as an environment that has to be taken into account. We do not have a precise description of the environment and will assume that it is either lossy, i.e. it only removes messages from the communication channel, or gainy, i.e. it only adds messages. This allows us to compute a description of the behavior of the overall system (i.e. after taking the communication into account) that is larger but conceptually simpler than the description we started with. We discuss the details in the second section of this chapter, Section 1.2, and the corresponding theory in Part III. of this thesis.

In our second example, we consider games, i.e. systems with two types of nondeterminism. One type of nondeterminism is uncontrollable *demonic* nondeterminism which corresponds to the environment and is hostile to the verification task. Additionally, there is controllable *angelic* nondeterminism. The latter can be instantiated at runtime to react to the nondeterministic choices of the environment in order to obtain an execution satisfying a specification. The goal is to find an instantiation that always reacts correctly. This setting appears in some verification tasks, e.g. when both the system and the specification are modeled as nondeterministic automata. Such a system is correct if for each of its branches, there is a matching branch of the specification. Additionally, two types of nondeterminism are needed to model program synthesis. We discuss the details in the third section of this chapter, Section 1.3, and the corresponding theory in Part V. of this thesis.

Our final example is concerned with a compositional approach to the verification of concurrent systems. The goal is to verify the system component-wise. When focusing on a single component, the rest of the components become an environment that has to be taken into account. We explain the details in the following.

Compositional verification

We consider concurrent systems, systems that consist of multiple components that run concurrently and may communicate with each other during runtime. The verification of such systems is one of the most challenging tasks in the area of verification. At the same time, the ubiquity of e.g. distributed systems means that this task is of the utmost importance.

For the sake of simplicity, we will in the following consider a concurrent system whose components are given as finite(-state) automata. Finite automata are among the simplest and most restricted types of automata. A finite automaton consists of a list of finitely many states and a list of the transitions between these states. At each point during runtime, the automaton is in a specific state. Other than that, it has no storage that it can manipulate. With this set of features, a finite automaton can model a program that only uses bounded storage by encoding the stored data as part of the state. Finite automata support neither recursion nor concurrency. The advantage of using such a simple model is that the undecidability results do not apply. For input programs modeled by finite automata, virtually all verification problems are decidable.

While a single finite automaton is simple, a concurrent system whose components are finite automata is not. Just as other problems in the area of verification of concurrent systems, verifying such a system suffers from the so-called *state explosion problem*. It would be possible to construct a single finite automaton as a representation for the whole system. However, since finite automata lack inherent support for concurrency, this leads to an automaton with a huge state space. Each global state of that automaton is a combination of local states, one for each component, and the global state space is the set of all such combinations. Assume we add components to the system or enlarge existing ones. The effect on the size of its description, given as a collection of automata, is additive. The size of the global state space, however, grows multiplicatively. Any verification procedure that explicitly explores this global state space usually has a worst-case running time that is exponential in the number of components.

Results in computational complexity theory have led us to believe that it is highly unlikely that this worst-case lower bound can be improved. In particular, algorithms that do not construct a single finite automaton as a representation of the system, but instead use a model that has inherent support for concurrency like Petri nets, are not any more efficient. This brings us back to the first loophole that we considered when we wanted to solve undecidable verification problems. An undecidable problem cannot be solved in full generality, but a procedure may be able to determine the status of some instances within finite time. Similarly, a problem with a high computational complexity cannot be solved efficiently in the worst case. However, an algorithm may be able to deal with a large class of inputs that are relevant in practice quickly. This hope is fueled by advances with respect to similar problems. For example, satisfiability checking for propositional logic is strongly believed to be impossible with less than exponential time consumption in general. Nevertheless, there are state-of-the-art tools that can solve instances of practical interest with millions of variables within minutes, see e.g. [LFLMLL18].

Several approaches that try to enable efficient verification procedures for concurrent systems are collectively referred to as *compositional verification* [OG76; Lam77; MC81; Pnu84]. They all try to verify each component of a concurrent system on its own, without ever fully exploring the global state space. If this succeeds, the running time should scale polynomially with the size of the description of the system (including the number of components), instead of exponentially.

Under the presence of some form of communication between the components, however, it is typically impossible to simply consider each component individually.

When we focus on a single component of a concurrent system, the rest of the components turn into an environment. The effect of the environment on that component usually cannot be disregarded, it has to be taken into account. In order to avoid the aforementioned state explosion problem, we do not want to explicitly construct a description of the environment. The hope is that an abstraction of the environment is sufficient to capture the interference caused by it with respect to the verification of the single component that we consider. When we verify that component, the abstraction is a specification for the behavior of the hostile environment that we can rely on. However, this results in an additional proof obligation: We have to establish that the environment indeed satisfies its specification.

We argue that the specification of the environment plays the role of a certificate. If the system at hand consists of two components, we start by focusing on one component. The goal is to establish a specification that is satisfied by any system containing that component, independent of the interference caused by the other component. Then, one shows that the whole system is correct by considering the other component. In order to deal with communication between components, we rely on the specification that has already been established. For a system with multiple components, one can use an iterative approach. One considers the components one by one, often starting with the component that is influenced the least by the others. The algorithm produces a sequence of certificates in the form of specifications for parts of the system until it is finally able to prove the desired property of the whole system.

There is a clear trade-off between the size of the abstraction and the ease of using it to establish correctness. A precise abstraction resembles the system more closely, which can make it easier to both establish its correctness and to use it to complete the verification of the system. However, the size heavily impacts the running time of the verification procedure. In general, there is no guarantee that there is an abstraction of a component that is precise enough to prove correctness while having a substantially smaller description. Hence, the existing approaches for compositional verification do not contradict the belief that the efficient verification of concurrent systems is impossible in general. However, the method has been shown to work well in practice. On the theoretical side, when the components are finite automata, there are algorithms that can find the smallest abstraction of a component that is sufficient to prove correctness based on Angluin's L^* algorithm [Ang87; PGBCB08]. When the components are modeled as automata from a less restrictive class, the problem becomes more involved. We will discuss the details in the next section of this chapter, Section 1.1.

Automata theory

We conclude this section by coming back to the area of automata theory. We name some classes of automata and explain their strengths and weaknesses. This will also explain how an automata

model that is suitable as the target of the abstraction in the automata-theoretic approach to verification can be chosen.

We have already mentioned that there are types of automata that are able to model the behavior of a general program, with Turing machines being the most important example. One believes that these *Turing-complete* models are at the frontier of computability: Each automaton from such a model can be simulated by a real computer, and every algorithm that can be implemented in the physical word, e.g. in the form of a computer program, can be modeled as such an automaton. Unfortunately, each of these models suffers from the undecidability results mentioned at the beginning of this section. This means that every verification problem in which the goal is deciding a non-trivial property of the behavior of an automaton from a Turing-complete class is undecidable and hence cannot be solved algorithmically in full generality.

We have already briefly discussed finite automata. This model is very restricted and hence unable to model most real computer programs. At the same time, virtually all verification problems which take finite automata as inputs are decidable. Between finite automata and Turing machines, there is a whole hierarchy of models that are more expressive than finite automata while still allowing for some verification problems to be decidable. We mostly focus on two branches of this hierarchy in the rest of the thesis: Context-free models like pushdown automata and their extended and restricted variants, and Petri nets and related models.

Pushdown automata are essentially finite automata that maintain a stack to store information during runtime. To avoid Turing-completeness, this stack can only be accessed in a LIFO (last in, first out) manner. It is well-understood that pushdown automata are suitable to model programs with recursion by using the stack as a call-stack. However, pushdown automata cannot model unbounded storage and they have no inherent support for concurrency. The stack makes them suitable to model the *control flow* of programs in many classical imperative programming languages, e.g. C [ISO90], while lacking support for modern features like generics and templates [Ale01] and higher-order functions [Pey03].

For pushdown automata, many verification problems like safety, i.e. whether a certain error state can be avoided, or liveness, i.e. whether a certain good state can be reached indefinitely often, are decidable. However, some problems that are decidable for finite automata are undecidable for pushdown automata. For example, it is not decidable whether two pushdown automata can be synchronized in a way such that an error state is reached in both automata. This corresponds to the fact that pushdown automata cannot deal with concurrency; the combination of two pushdown automata, or alternatively, a pushdown automaton that maintains two stacks, is a Turing-complete model.

In contrast to pushdown automata, Petri nets can deal with concurrency but not with recursion (and neither of the two models can deal with unbounded storage). A Petri net is essentially a finite automaton in which a potentially unbounded number of threads run simultaneously. These threads can spawn, terminate, and synchronize during runtime. Similar to pushdown

automata, basic verification problems like safety verification are decidable for Petri nets. Unlike pushdown automata however, for which safety verification can be done in polynomial time, *coverability*, the algorithmic problem for Petri nets corresponding to safety verification, is much more expensive and needs at least exponential time in general.[1]

At the beginning of this section, we have argued that the ultimate goal is to design a verification algorithm that works without human input. The automata-theoretic approach seems to miss that goal, since it appears that we have to choose an automata model as the target of the abstraction by hand. However, as we just have discussed, each class of automata has a general set of strengths and weaknesses, e.g. Petri nets are useful to model concurrency. This allows us to pick an automata model for a verification task based on the type of program and the type of specification, without actually looking at the details of the implementation. It also means that a verification tool using the automata-theoretic approach that has been developed with a specific program in mind is typically also useful for a wide class of similar programs. If the chosen automaton model is not suitable to model certain features of the original program (e.g. neither Petri nets nor pushdown automata can deal with unbounded storage), using an iterative approach like the CEGAR loop may help to alleviate the problem.

In addition to the automata-theoretic approach to verification, there is another noteworthy link between automata theory and verification. The verification problems for various automata are important examples of problems that are complete for certain classes of computational complexity. This essentially means that a certain type of automaton with unrestricted resource consumption typically corresponds to an unrestricted program with a certain bounded resource consumption. For example, whether a pushdown automaton reaches a certain error state can be decided in polynomial time. In turn, any Turing-machine that has a polynomially bounded time consumption can be turned into a pushdown automaton so that the Turing-machine can reach an error state if and only if the pushdown automaton can. When we introduce a decision problem that is based on automata in the rest of thesis, we will generally also provide a lower bound for the computational complexity of solving that problem. When we propose an algorithm solving the problem, we will analyze its resource consumption and determine whether it matches the lower bound.

The language-theoretic approach

Automata theory is closely connected to taking a *language-theoretic approach* to verification. The basic idea is to associate to a program or system a finite set of atomic actions that the program may execute. An execution of the system corresponds to a sequence of such actions, and the system itself gives rise to a set of sequences, one for each possible execution. In the following, we will call the actions *letters*, the set of possible actions an *alphabet*, the sequences of actions *words*, and sets of sequences *languages*. How we translate actions into letters depends

[1] In fact the problem is EXPSPACE-complete [Lip76], meaning that solving it in full generality needs at least exponential space in the worst case. If the commonly held beliefs about the relations between various complexity classes are true (EXPSPACE ≠ EXP), this means an algorithm solving it will need at least super-exponential time.

on the verification task at hand. For example, letters may correspond to assembly commands, packages sent over a network, or read and write accesses to memory locations.

In the same way that a program gives rise to a language, a specification defines a language of legal executions and its complement, the language of illegal executions. Verifying a program typically amounts to checking an inclusion: The set of possible executions should be a subset of the set of legal executions. Equivalently, the intersection between the possible and the illegal executions should be empty.

The crucial advantage provided by the language-theoretic approach is a layer of abstraction on top of the internal workings of a system. For example, it allows us to consider specifications that are independent of the internal design of a system, e.g. the reachability of a special error state. Instead, the specification talks just about the visible behavior of the system. This allows us to specify the behavior that we want to consider illegal and then verify that it does not occur, without relying on the implementation of the error handling inside the system being correct.

In the following, we will use the operator $\mathcal{L}(-)$ to associate a language to a system or specification. The verification problem for a system A and a specification φ is typically equivalent to the question of whether the inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(\varphi)$ holds. We assume that there is a common alphabet Σ so that both $\mathcal{L}(A)$ and $\mathcal{L}(\varphi)$ are subsets of Σ^*, the set of all finite words formed using letters from Σ. Note that in principle, it is possible to also consider languages of infinite words, corresponding to non-terminating executions, as well as languages of trees, corresponding to executions that incorporate branching behavior. However, for the sake of simplicity, we will focus on language of finite words, corresponding to terminating executions, for now.

We have argued before that automata models are useful to represent classes of systems with restricted expressiveness. Combining an automaton model with a notion of *acceptance* yields the notion of the language of an automaton, the set of all words generated by the accepting executions of that automaton. Each class of automata gives rise to a class of languages that they describe. Properties of the automata typically correspond to properties of their languages. The aforementioned finite automata lead to a class of languages called the *regular languages*. As we pointed out earlier, deciding properties of finite automata is simple. Correspondingly, the same applies to regular languages, assuming they are given in the form of finite automata generating them. Additionally, regular languages are very well-behaved when it comes to closure properties, e.g. the intersection of regular languages is again a regular language.

Other classes of automata that are more expressive define classes of languages that are larger but for which it is more difficult to decide their properties. Pushdown automata define the class of context-free languages, a class of languages that is particularly useful to model the syntax of programming languages. For Petri nets, different acceptance conditions will lead to different classes of languages. In this thesis, we almost exclusively consider so-called coverability as the condition for when an execution is accepting; we will refer to the class of languages associated

to coverability as Petri net languages. Both the context-free and the Petri net languages are superclasses of the regular languages.

Certificates for automata in a hostile environment

The rest of this introductory chapter is concerned with discussing each of the three aforementioned examples in more detail: Compositional verification, unreliable communication, and games. In each case, we will give more information on why the environment is important and on the role that certificates play. It turns out that each example is associated with an interesting class of automata-theoretic decisions problems. The Parts III. to V. of this thesis are concerned with studying the theory behind these decision problems. Whenever these problems have already been studied and solved for certain classes of automata, e.g. finite automata, our goal is to provide a solution for more expressive classes of automata. For each problem we develop a rich theory, provide lower bounds for the computational complexity, present an algorithm solving it, and show that our algorithm has optimal resource consumption.

1.1 Compositional verification and separability

In this section, we continue to consider the compositional verification of concurrent systems. We have already explained why the verification of concurrent system is essential, but also rather difficult. We came to the conclusion that in order to tame the complexity of the problem, a compositional approach is necessary. In the following, we make our explanation more formal. We will then explain how separability problems are related to the theoretic foundations of compositional verification.

A language-theoretic approach to compositional verification

We start by explaining how the language of a concurrent system can be constructed, given the languages of its components. Let $A = A_1 \parallel A_2 \parallel \ldots \parallel A_k$ be a concurrent system that is obtained as the result of the parallel composition of the components A_1 to A_k. We assume that each component A_i has an associated alphabet Σ_i, a set of letters or atomic commands that this component may execute. We consider a simple model in which two components A_i, A_j synchronize on the actions in the shared alphabet $\Sigma_i \cap \Sigma_j$, while actions that are not shared can be executed independently. We mimic this model by defining the parallel composition $\mathcal{L}_i \parallel \mathcal{L}_j$ of two languages $\mathcal{L}_i$ and $\mathcal{L}_j$ over Σ_i and Σ_j as the set of words over $\Sigma_i \cup \Sigma_j$ such that if we project a word to one of the alphabets (by removing all letters not contained in that alphabet), we obtain a word in the corresponding language. With this definition, the language of the parallel composition of components is the parallel composition of the languages of the components, $\mathcal{L}(A_1 \parallel A_2) = \mathcal{L}(A_1) \parallel \mathcal{L}(A_2)$. Given that the parallel composition operator is both associative and commutative, extending this to systems with more than two components is straightforward.

There are two special cases of the parallel composition of languages that are noteworthy. The first is the case that the alphabets are equal, $\Sigma_1 = \Sigma_2$. In this case, the projections leave the words unchanged, and the parallel composition is equal to the intersection $\mathcal{L}_1 \cap \mathcal{L}_2$. The second interesting case is that the two alphabets are disjoint, $\Sigma_1 \cap \Sigma_2 = \varnothing$. Then, the parallel composition is equal to the so-called shuffle of the languages, the set of all interleavings of words from $\mathcal{L}_1$ and $\mathcal{L}_2$, where two words from each of the languages are interleaved in an arbitrary order, but the order of letters within each of the two words is preserved.

One popular approach to composition verification is assume-guarantee reasoning [Lam77; Jon83]. Our presentation follows [GNP18], but we have adapted it to take a more language-theoretic approach. Assume-guarantee reasoning is based on triples of the form $\langle\psi\rangle A \langle\varphi\rangle$, where A is a system and ψ and φ are specifications. Such a triple is valid if any concurrent system containing A as a component that satisfies ψ also satisfies φ. The name comes from the fact that under the assumption of ψ, one can guarantee φ.

To express an assume-guarantee triple on the level of languages, we make use of the fact that the language of a system containing A as a component can be written as $\mathcal{L} \parallel \mathcal{L}(A)$ for some

suitable language $\mathcal{L}$. As explained above, satisfying a specification can be expressed as an inclusion. We have to deal with the problem that the specification and the system may not use the same alphabet. Under the assumption that $\Sigma_\varphi \subseteq \Sigma_A$, i.e. the alphabet associated to the system is an extension of the alphabet used by the specification, we write $\mathcal{L}(A) \sqsubseteq \mathcal{L}(\varphi)$ to denote the inclusion $\pi_{\Sigma_\varphi}(\mathcal{L}(A)) \subseteq \mathcal{L}(\varphi)$, where $\pi_{\Sigma_\varphi}(-)$ is the projection that removes from a word all letters not contained in Σ_φ With this notation at hand, we obtain that the triple $\langle\psi\rangle A\langle\varphi\rangle$ is valid if and only if for every language $\mathcal{L}$, $\mathcal{L} \parallel \mathcal{L}(A) \sqsubseteq \mathcal{L}(\psi)$ implies $\mathcal{L} \parallel \mathcal{L}(A) \sqsubseteq \mathcal{L}(\varphi)$.

One can check whether a triple $\langle\psi\rangle A\langle\varphi\rangle$ is valid by checking if the language $\mathcal{L}(\psi) \parallel \mathcal{L}(A) \parallel \overline{\mathcal{L}(\varphi)}$ is empty [PGBCB08]. Here, $\overline{\mathcal{L}(\varphi)}$ is the complement of $\mathcal{L}(\varphi)$, i.e. the set of all words over Σ_φ not contained in $\mathcal{L}(\varphi)$. As a start, we will assume that both the specifications and the systems under consideration are given as finite-state automata resp. regular languages. For specifications, this makes sense: Regular languages are not only the class of languages associated to finite automata, but also expressive enough to model common specification logics like MSO and LTL [Pnu77]. For systems, limiting ourselves to finite automata is a heavy restriction. We will come back to this aspect later. Under the assumption that all components of a triple $\langle\psi\rangle A\langle\varphi\rangle$ are given as finite-state automata, checking the validity of a triple can be automated. To this end, one can compute an automaton for $\mathcal{L}(\psi) \parallel \mathcal{L}(A) \parallel \overline{\mathcal{L}(\varphi)}$ and verify that its language is empty by conducting a reachability check. However, computing this automaton may be very expensive. Recall that the system A is typically a concurrent system that is not given explicitly, but as a collection of its components. If we construct a finite automaton representing A, the whole procedure will suffer from the state explosion problem detailed in the previous section.

A proof rule for compositional verification

This is where compositional verification comes to our aid. The compositional approach to checking the validity of assume-guarantee triples is based on proof rules. The goal is to establish the validity of an assume-guarantee triple for a complex system under the premise that the validity of assume-guarantee triples for simpler systems have already been shown. Here, we focus on the following simple but powerful proof rule:

$$\frac{\langle\text{true}\rangle A\langle\psi\rangle \qquad \langle\psi\rangle B\langle\varphi\rangle}{\langle\text{true}\rangle A \parallel B\langle\varphi\rangle\,.}$$

This rule states that if the two premises, namely the validity of the triples $\langle\text{true}\rangle A\langle\psi\rangle$ and $\langle\psi\rangle B\langle\varphi\rangle$ have been proven, then we obtain the validity of the triple $\langle\text{true}\rangle A \parallel B\langle\varphi\rangle$. Phrased differently, in order to establish that $A \parallel B$ satisfies φ in any environment and without precondition[1], it is sufficient to show that A satisfies ψ in any environment and that B satisfies φ in any environment in which it satisfies ψ. The rule enables compositional verification: It decomposes checking the

[1] We see true as the specification with respect to which every execution is legal, i.e. as the language of all words over a suitable alphabet resp. a finite automaton generating this language.

validity of a triple for a parallel composition into two triples, one for each of the two components. It is not hard to see that the rule is sound, meaning that the premises indeed imply the conclusion. Assume that the two premises hold, and let $\mathcal{L}$ be any language. We have to show that $\mathcal{L} \parallel \mathcal{L}(A \parallel B) \sqsubseteq \mathcal{L}(\varphi)$. We use the first premise, instantiating the environment for A with $\mathcal{L} \parallel \mathcal{L}(B)$, and obtain that the composition satisfies ψ, $(\mathcal{L} \parallel \mathcal{L}(B)) \parallel \mathcal{L}(A) \sqsubseteq \mathcal{L}(\psi)$. Using the associativity and commutativity of parallel composition, the language $(\mathcal{L} \parallel \mathcal{L}(B)) \parallel \mathcal{L}(A)$ can be rewritten as $(\mathcal{L} \parallel \mathcal{L}(A)) \parallel \mathcal{L}(B)$. We may see $\mathcal{L} \parallel \mathcal{L}(A)$ as an environment for B with which together it satisfies ψ. This allows us to apply the second premise, yielding that the composition also satisfies φ. Finally, we rewrite $(\mathcal{L} \parallel \mathcal{L}(A)) \parallel \mathcal{L}(B)$ as $\mathcal{L} \parallel \mathcal{L}(A \parallel B)$, using commutativity and the correspondence between the parallel composition of languages and the parallel composition of automata. We obtain $\mathcal{L} \parallel \mathcal{L}(A \parallel B) \sqsubseteq \mathcal{L}(\varphi)$ as desired.

Surprisingly, the rule is not just sound, but also complete. To make this notion precise, note that the specification ψ occurring in the premises does not occur in the conclusion. Hence, if we start with the goal of proving $\langle \text{true} \rangle A \parallel B \langle \varphi \rangle$, we are free to choose the specification ψ. As long as our choice of ψ leads to the triples in the premises being valid, it is suitable for instantiating the proof rule. This means that ψ plays the role of a certificate. Given an appropriate ψ, it is easier to check $\langle \text{true} \rangle A \parallel B \langle \varphi \rangle$ by using the proof rule and checking the triples $\langle \text{true} \rangle A \langle \psi \rangle$ and $\langle \psi \rangle B \langle \varphi \rangle$ than it would be to check the triple for the composition itself. Also note that conceptually, ψ is an abstraction of the behavior of A in an arbitrary environment. On the one hand, this abstraction should be coarse in the sense that the finite automaton representing it is smaller than A. The difficulty of automatically checking the validity of a triple as explained above does not only scale with the size of the system, but also with the size of the two specifications. Hence, the smaller the representation of ψ is, the easier it will be to check the triples $\langle \text{true} \rangle A \langle \psi \rangle$ and $\langle \psi \rangle B \langle \varphi \rangle$. On the other hand, the abstraction has to be precise enough so that the two triples are actually valid. In a sense, ψ is the abstract description of A that we use to verify B. If it is too imprecise, we will not be able to complete our goal of establishing the property φ.

When we say that a proof rule is complete, we mean that whenever the conclusion is true, then there is an instantiation of the premises that allows us to apply the rule. In our example, this means that if $\langle \text{true} \rangle A \parallel B \langle \varphi \rangle$ is true, then there is a specification ψ so that the triples $\langle \text{true} \rangle A \langle \psi \rangle$ and $\langle \psi \rangle B \langle \varphi \rangle$ from the premises are valid. In fact, this specification can be chosen to be simply A.[1] Note that this completeness only applies under the assumption that both the components and the specifications are given in the form of finite automata. The fact that our simple proof rule is complete means that conceptually, compositional verification is always possible. However, the existence of an appropriate specification ψ does not mean that it is easy to find one that is not unsuitably large. Recall that the validity of a triple $\langle \psi \rangle B \langle \varphi \rangle$ is equivalent to the emptiness of $\mathcal{L}(\psi) \parallel \mathcal{L}(B) \parallel \overline{\mathcal{L}(\varphi)}$. Instantiating this result for $\langle \psi \rangle B \langle \varphi \rangle$ and for $\langle \text{true} \rangle A \parallel B \langle \varphi \rangle$ yields that

[1] Assume that $\langle \text{true} \rangle A \parallel B \langle \varphi \rangle$ is valid. We claim that then $\langle \text{true} \rangle A \langle A \rangle$ and $\langle A \rangle B \langle \varphi \rangle$ are valid. For the validity of $\langle \text{true} \rangle A \langle A \rangle$, note that the composition $\mathcal{L} \parallel \mathcal{L}(A)$ of A with any language $\mathcal{L}$ will satisfy $\mathcal{L} \parallel \mathcal{L}(A) \sqsubseteq \mathcal{L}(A)$ by the definition of the parallel composition. For the validity of $\langle A \rangle B \langle \varphi \rangle$, let $\mathcal{L}$ be any language so that $\mathcal{L} \parallel \mathcal{L}(B) \sqsubseteq \mathcal{L}(A)$. Take any word $w \in \mathcal{L} \parallel \mathcal{L}(B)$ and note that we have both $\pi_{\Sigma_B}(w) \in \mathcal{L}(B)$ and $\pi_{\Sigma_A}(w) \in \mathcal{L}(A)$. Hence, $\pi_{\Sigma_B \cup \Sigma_A}(w)$ is an element of $\mathcal{L}(A \parallel B)$. Since we assumed $\langle \text{true} \rangle A \parallel B \langle \varphi \rangle$ to be valid, this implies $\pi_{\Sigma_\varphi}(\pi_{\Sigma_B \cup \Sigma_A}(w)) = \pi_{\Sigma_\varphi}(w) \in \mathcal{L}(\varphi)$.

if ψ is similar in size to A, then checking $\langle\psi\rangle B\langle\varphi\rangle$ is not easier than checking $\langle\text{true}\rangle A \parallel B\langle\varphi\rangle$. Hence, the proof rule only leads to a substantial reduction in running time if we are able to find a specification ψ that is substantially smaller than A. While the existence of a specification ψ that leads to the premises becoming true is guaranteed, the existence of a small representation for a suitable ψ is not guaranteed. In fact, complexity-theoretic reasons have led us to believe that it cannot exist in some cases.

The commonly held belief is that the more loosely two components are coupled in a concurrent system, the easier it is to find a specification for their interface that is sufficient for verification. In the literature, numerous approaches to compositional verification based on assume-guarantee reasoning have been developed. Notably, there is a learning algorithm that tries to compute the certificate ψ on the fly [CGP03; PGBCB08]. If the triple that should ultimately be proved is indeed valid, this algorithm is guaranteed to terminate with the smallest deterministic automaton representing ψ so that the above proof rule can be applied. Additionally, there is research on proof rules that work e.g. for more than just two components [PGBCB08].

Compositional verification and separability

All the aforementioned results focus on the simple case that the components and specifications are given as finite automata. Our goal in the following will be to go beyond finite automata and regular languages. Fundamentally, the question we want to answer is whether compositional verification is possible for more general types of systems. To this end, we apply a sequence of simplifications to the problem at hand. The result is a theory that allows us to clearly formulate the question of whether compositional verification is possible. While the resulting theory will not immediately lead to an approach to compositional verification that is usable in practice, it can hopefully be extended towards one in the future.

We focus on a triple $\langle\text{true}\rangle A \parallel B\langle\varphi\rangle$. The first simplification that we apply is that we discard the idea of having an arbitrary environment. We are interested in the system $A \parallel B$, and whenever we focus on one of the two components, the other one will be the environment. Consequently, we can discard the universal quantification over a language in the definition of the validity of a triple. Instead of asking whether $\mathcal{L} \parallel \mathcal{L}(A) \parallel \mathcal{L}(B) \sqsubseteq \mathcal{L}(\varphi)$ for all languages $\mathcal{L}$, we simply want to check $\mathcal{L}(A) \parallel \mathcal{L}(B) \sqsubseteq \mathcal{L}(\varphi)$.

The second simplification is that we assume that A, B, and φ all use the same alphabet Σ. This means that A and B synchronize on every action, and not just on the actions represented by letters in a subset of their alphabets. Luckily, this property can usually be enforced rather easily without changing the behavior of the system. For each letter of B that was not in the shared alphabet, we introduce appropriate transitions in A that correspond to this letter without actually changing the internal state of A. The resulting system behaves as A does, even when composed with B, but uses an extended alphabet. We apply the same procedure to B and φ to make sure that all three objects use the same alphabet. As mentioned before, the parallel composition

of the languages of systems that use the same alphabet is simply the intersection of these languages. Hence, $\mathcal{L}(A) \parallel \mathcal{L}(B)$ equals $\mathcal{L}(A) \cap \mathcal{L}(B)$. Additionally, we can replace our variant $\sqsubseteq$ of the subset relation that involves a projection by the normal inclusion relation $\subseteq$. In total, our goal is now checking $\mathcal{L}(A) \cap \mathcal{L}(B) \subseteq \mathcal{L}(\varphi)$.

Finally, we transform this expression into a form that is even simpler. The inclusion $\mathcal{L}(A) \cap \mathcal{L}(B) \subseteq \mathcal{L}(\varphi)$ holds if and only if $(\mathcal{L}(A) \cap \mathcal{L}(B)) \cap \overline{\mathcal{L}(\varphi)}$ is empty, where $\overline{\mathcal{L}(\varphi)}$ is the complement language of $\mathcal{L}(\varphi)$. We rewrite this to $(\mathcal{L}(A) \cap \overline{\mathcal{L}(\varphi)}) \cap \mathcal{L}(B)$. Recall that the language $\mathcal{L}(\varphi)$ is regular – while we are interested in systems that are more expressive than finite automata, regular languages are usually sufficient to represent the specifications we are interested in. By the closure properties of regular languages, also the complement language $\overline{\mathcal{L}(\varphi)}$ is regular. Additionally, most other language classes are well-behaved when it comes to intersections with regular languages: Intersecting a language from such a class with a regular language leads to a language that is again in that class.[1] For example, given a pushdown automaton and a finite automaton, it is easy to construct a pushdown automaton whose language is the intersection of the two associated languages. The same is true for Petri nets. Ultimately, this means that we can construct a system A' whose language is $\mathcal{L}(A) \cap \overline{\mathcal{L}(\varphi)}$. After applying this construction, checking $\mathcal{L}(A) \cap \mathcal{L}(B) \subseteq \mathcal{L}(\varphi)$ amounts to checking $\mathcal{L}(A') \cap \mathcal{L}(B) = \varnothing$.

We have argued before that checking whether a system satisfies a specification means checking whether the intersection of the possible and the illegal executions is empty. In that case, we usually consider the intersection of a complicated language, the language of possible executions of a system, with a simple language, the language of illegal executions according to a specification. In the case of checking $\mathcal{L}(A') \cap \mathcal{L}(B) = \varnothing$, we are dealing with a setting that is more involved. The language of A' is the set of executions that are possible with respect to component A while being illegal with respect to specification φ. Both $\mathcal{L}(A')$ and $\mathcal{L}(B)$ are languages of systems that are more complicated than finite automata.

How can we check $\mathcal{L}(A) \cap \mathcal{L}(B) = \varnothing$? (Here and in the following, we drop the notation A' for the sake of simplicity, since $\mathcal{L}(A)$ and $\mathcal{L}(A')$ come from the same class of languages.) Explicitly constructing an automaton whose language is $\mathcal{L}(A) \cap \mathcal{L}(B)$ is not a good idea. In some cases, it may not even be possible. For example, the intersection of two languages of pushdown automata is not the language of a pushdown automaton in general. Even if it is possible, it will suffer from the same state explosion problem that we have explained in the case of finite automata. To solve this problem, we need a compositional approach.

We propose the following *separability proof rule* as a possible solution:

$$\frac{\mathcal{L}(A) \subseteq \mathcal{R} \qquad \mathcal{R} \cap \mathcal{L}(B) = \varnothing}{\mathcal{L}(A) \cap \mathcal{L}(B) = \varnothing\ .}$$

[1] Using the nomenclature from the theory of abstract families of languages (AFLs) [Ber79], many classes of languages are *trios*, which in particular means that they are closed under intersection with regular languages. This applies to both the context-free [Ber79] and the Petri net languages [Jan87].

Let us first argue that this rule is analogous to the compositional proof rule for assume-guarantee triples. In the conclusion, the validity of the triple $\langle \text{true} \rangle A \parallel B \langle \varphi \rangle$ has been replaced by $\mathcal{L}(A) \cap \mathcal{L}(B) = \varnothing$ as explained. Similarly, in the second premise the property of satisfying a specification φ has been replaced with the emptiness of a language. The most interesting part is the language $\mathcal{R}$, which replaces the specification ψ in both the first and the second premise.

In order to understand the role of $\mathcal{R}$, we prove the soundness of the proof rule. Assume there is a language $\mathcal{R}$ so that both premises are true. This means that by the first premise, $\mathcal{R}$ is bigger than $\mathcal{L}(A)$. Additionally, $\mathcal{R}$ and $\mathcal{L}(B)$ are disjoint by the second premises. But if $\mathcal{L}(B)$ has an empty intersection with a language that is bigger than $\mathcal{L}(A)$, then also $\mathcal{L}(A)$ and $\mathcal{L}(B)$ need to be disjoint and the conclusion is true. A language $\mathcal{R}$ that makes the premises become true is called a *separator*. It is a certificate for the disjointness of $\mathcal{L}(A)$ and $\mathcal{L}(B)$.

The question that remains is whether the separability proof rule is complete. Is it true that whenever two languages are disjoint, a separator has to exist? The answer to this question depends on the language classes that we consider. If $\mathcal{L}(A)$, $\mathcal{L}(B)$, and $\mathcal{R}$ come from the same class of languages, completeness obviously holds. If $\mathcal{L}(A)$ and $\mathcal{L}(B)$ are disjoint, then $\mathcal{L}(A)$ itself is a separator. However, in that case, we have gained nothing. While the first premise trivially holds, checking the second premise simply amounts to checking the desired conclusion. Even in the case that all languages come from the same class, however, it would be interesting to ask whether there is a separator that has a smaller description than A. This is closely related to the observations we made in the case of finite automata.

Regular separability

We will focus on the case that the separator is required to come from a class that is less expressive than the languages $\mathcal{L}(A)$ and $\mathcal{L}(B)$. To be precise, we will exclusively consider regular separators in this thesis. This is motivated by two facts. The first fact is the aforementioned correspondence of regular languages to specification logics. Secondly, regular languages are very well-behaved when it comes to closure properties and algorithmics. This means that for a candidate separator that is a regular language, it is highly likely that the two premises of the separability proof rule can indeed be automatically checked. This goes along with our theme of being interested in certificates that are easier to verify than the initial problem is to solve.

In the following, we consider *regular separability*. Is it true for a class of languages that whenever two languages are disjoint, a regular separator exists? If that is not true, can one decide whether a separator exists? If a separator exists, can it be computed? For context-free languages, the class of languages defined by pushdown automata, it turns out that separability is difficult. Given two languages and a regular candidate separator, we can automatically check whether the two premises of the separability proof rule hold. In that case, the soundness of the rules implies that the two given context-free languages are disjoint. However, it is easy to construct two context-free languages so that no regular separator can exist. Additionally, the problem of deciding whether a separator exists has turned out to be undecidable [SW76]. In

the literature, it has been established that this behavior does not only apply to context-free languages, but also to restricted versions of context-free languages that are only minor extensions of the regular languages, e.g. the languages of one-counter automata [CL17] and visibly pushdown automata [Kop16].

Outlook

In Part IV. of this thesis, we focus on languages of well-structured transition systems (WSTSes). This class of languages is a superclass of the class of Petri net languages. We will show that under mild conditions, the separability proof rule is complete for WSTS languages. Whenever two WSTS languages are disjoint, there is a regular separator as a certificate for this disjointness.

To be more precise, our main result is that any two disjoint WSTS languages, one of them the language of a deterministic WSTS, are regularly separable. Requiring one of the WSTSes to be deterministic seems like a substantial restriction. However, we will establish a hierarchy of subclasses of languages within the class of WSTS languages and show that any WSTS whose transition relation is finitely branching or whose underlying order is ω^2 can be determinized. Almost all WSTSes that are of practical relevance satisfy at least one of these two requirements. Hence, our result on the separability of WSTS languages can be applied to almost all WSTSes.

Additionally, one can extract from the proof of the main result an algorithm for computing a finite automaton representing the regular separator. We will demonstrate this in the case of Petri net languages, obtaining a construction for the separator and an upper bound on its size, which we will accompany by a lower bound.

1.2 Unreliable communication and language closures

In this section, we pick up one of our earlier examples and discuss it in detail. The setting we consider is the verification of a system that we communicate with over an unreliable communication channel. Firstly, we elaborate on what we mean by unreliable communication. Secondly, we present a theoretical model that allows us to make this notion precise. This will lead us to formalizing two computational problems that are crucial for the verification tasks. Finally, we also explain how the environment and certificates come into play.

Examples of gainy and lossy communication

Assume we have a system S that we are communicating with. The communication is conducted via a communication channel C, e.g. a network connection. What we observe is not the messages sent by system S, but the messages we receive via channel C. If we assume that C is *perfect*, these two things coincide and we can simply omit C. The case that we are interested in is that C is unreliable. We consider two settings: a *lossy* channel and a *gainy* channel.

If the channel is lossy, a message that is sent by S may not appear at the other end of C. The visible behavior of S that we observe through C is only a fragment of the real behavior of S. This setting can occur under various circumstances. The most obvious case is communication via network infrastructure. Assume that C is a network connection over which S sends messages in the form of packets. It is common that packets can be lost, e.g. as the result of an outage in the network infrastructure. This infrastructure is typically not part of our model of S. We assume that we are dealing with a system that is so low-level that we cannot simply rely on a protocol like TCP [Pos81] to ensure that all packets arrive at the destination eventually. In other words, we verify the transportation layer of the system and not just the application layer. In our model, each packet sent by S either arrives or it does not. (Note that the model we present here cannot deal with packets not being received in the correct order. We will comment on this aspect later.) We may want to verify that the behavior of S is correct, even under the assumption that some packets are lost.

In the following, we give a second example in which lossiness plays a role even though the communication is perfect. Assume that S is a component of a concurrent system that writes to shared memory. We assume that the memory accesses are perfect: All write accesses to memory executed by S actually take place, and they do so in the correct order. However, we assume that we observe this memory location from the perspective of another component of the concurrent system. If we poll the memory location without synchronizing with S in some way, we may not see all writes that have been executed by S. If between reading the memory location twice, two write accesses of S take place, we will only see the second write access; the first one will be lost. Similarly, if we start and stop polling the memory at some point in time, we will not see any write accesses by S that happen outside this time interval. In summary, this type of communication shows the same lossy behavior as in the first example.

Gainy communication works in the opposite way: Instead of just seeing the messages sent by S, we potentially observe more messages. We observe a sequence of messages that contains the messages sent by S, but also additional ones. Such a setting can occur if we observe a network endpoint with which several systems communicate. Usually, these systems are distinguishable, but they may not be, e.g. if an attacker tries to impersonate S. Similarly, we might observe a memory location to which several other threads write, but we are only provided with a model for one of these threads. Note that in order to avoid having both lossy and gainy effects at the same time, we will have to assume that we synchronize with the writing threads so that we are notified of every write access.

In all of these examples, we face the same challenge: We are given a description of S which allows us to deduce properties of the behavior of S. But the goal is to verify the behavior of S that is visible through C, i.e. we should decide a property of the behavior of C. For now, we will assume that the only information about C that is available to us is whether C is lossy or gainy.

Language closures

Our goal is to devise a method to compute a description of the behavior of C given a description of S. In order to tackle this problem, we start by presenting a theoretical model for lossy and gainy behavior. We use a language-theoretic approach. We assume that all messages that we observe are from some message alphabet Σ. The sequence of messages sent by S in an execution corresponds to a word $w \in \Sigma^*$ of messages, and there is a language $\mathcal{L}(S) \subseteq \Sigma^*$ of all such possible sequences. What we observe is not $\mathcal{L}(S)$, but rather $\mathcal{L}(C)$, the possible sequences of messages we receive via the communication channel C. In the case of lossiness, we have that every sequence in $\mathcal{L}(C)$ is obtained from a sequence in $\mathcal{L}(S)$ by losing messages. To formalize this, we introduce the well-known subword ordering: A word v is a subword of w if v is obtained from w by deleting letters. For example, *radar* is a subword of *abracadabra*.

If C is a lossy communication channel, then every word in $\mathcal{L}(C)$ is a subword of a word from $\mathcal{L}(S)$. To obtain the behavior of C, one can form the *language closure* of $\mathcal{L}(S)$ with respect to the subword ordering. This means we consider all words that are subwords of words from $\mathcal{L}(S)$. The resulting set is denoted as the *downward closure* $\mathcal{L}(S)\downarrow$. Note that as the notion of closure suggests, this set is indeed a superset of $\mathcal{L}(S)$, $\mathcal{L}(S) \subseteq \mathcal{L}(S)\downarrow$, because every word is a subword of itself. Under the assumption of lossy communication, this set $\mathcal{L}(S)\downarrow$ is exactly $\mathcal{L}(C)$, the set of possible sequences of messages that we observe via the communication channel.

In a gainy setting, we simply flip the direction of the subword order: Word w is a superword of v if v is a subword of w, or equivalently, if w is obtained from v by inserting letters. The *upward closure* $\mathcal{L}(S)\uparrow$ of $\mathcal{L}(S)$ is obtained as the set of words that are a superword of a word from $\mathcal{L}(S)$. As with the downward closure, we have that the upward closure is a superset, $\mathcal{L}(S) \subseteq \mathcal{L}(S)\uparrow$. If the communication is gainy, $\mathcal{L}(S)\uparrow$ is exactly the set of possible channel contents of C.

Note that we cannot deal with a channel that is both lossy and gainy at the same time. Both the downward closure of the upward closure $\mathcal{L}(S)\uparrow\downarrow$ and the upward closure of the downward

closure $\mathcal{L}(S)\downarrow\uparrow$ equal the set Σ^* of all possible sequences of messages if $\mathcal{L}(S)$ is non-empty. We also require that the messages appear in the communication channel in the same order in which they have been sent by S. If we assume that the messages appear in an arbitrary order, we would need to modify the model. One possibility would be to apply a commutative closure to S, an operation that we will briefly discuss in Section 6.4.

The theoretical model leads to two computational problems that can be formalized easily: Given S (and hence a description of $\mathcal{L}(S)$), compute a description of $\mathcal{L}(S)\downarrow$ resp. $\mathcal{L}(S)\uparrow$. In order to decide a property of the visible behavior of S through C, solving one of these two problems should be the first step. After a description of the appropriate language closure has been obtained, it can serve as the input for decision procedures that settle the answer to the original verification problem. The latter is beyond our scope here, we focus on the computation of the language closures.

There is a well-known theoretical result that comes to our aid with respect to these language closures: For any language $\mathcal{L}(S)$, both the downward and the upward closure are always regular [Hai69], meaning they can be represented by a finite automaton. Not only are finite automata a very simple way of representing a language, but that also means that once we obtain a description, it is easy to apply decision procedures. Also, regular languages are very well-behaved with respect to closure properties, e.g. intersections. This means if we have more information about C beyond the property of it being lossy or gainy, it should be easy to incorporate this information after a finite automaton representing the closure has been computed. In summary, if we are able to obtain a finite automaton as a description for a language closure, we will have a good basis for the rest of the verification task.

The fact that the closures are always regular means that closures also have another use case: The downward resp. upward closure of a language is a regular overapproximation of that language. This means when we want to abstract a system (resp. its language) to a simpler class of automata (resp. the corresponding class of languages), then choosing these closures as abstraction is a possibility. For example, approximations based on the downward closure have been used in [LCMM12] to design an iterative procedure that can solve the undecidable problem of intersection-emptiness for context-free languages in some cases.

The approach of abstracting a system into its downward or upward closure is particularly useful if the system that we start with already incorporates lossy or gainy aspects as detailed in the earlier examples. In this case, we can hope that the closure is either equal to the language of the system itself or at least the difference is rather small. The latter means that if the original system is indeed correct, then it should be very likely that proving the correctness by considering a language closure as overapproximation will succeed.

Unfortunately, the proof of the regularity of the language closures is non-constructive. This means that we know that a finite automaton representing the closure exists, but we do not

have a way to compute it in general. Given the undecidability results that we mentioned earlier, this should not be surprising. By Rice's theorem [Ric53], deciding any non-trivial property of Turing machine languages is impossible, including their emptiness. Assume we start with a system S that is given as a Turing machine. The language of S is empty if and only if its downward closure is. If there were a way to compute a finite automaton representing $\mathcal{L}(S)\downarrow$, we could check the emptiness of $\mathcal{L}(S)$ by checking whether $\mathcal{L}(S)\downarrow$ is empty. The latter boils down to checking the emptiness of the language of a finite automaton, which is a simple task. This approach would decide the emptiness problem for Turing-machine languages, a contradiction to Rice's undecidability result. The same line of argumentation works for any Turing-complete model.

Luckily, for many types of automata that are not Turing-complete, effective procedures for the computability of the upward and downward closures have been found. For example, it is possible to transform a given pushdown automaton into finite automata whose languages are the downward resp. upward closures of the pushdown automaton. However, there are also models that are not Turing-complete and for which e.g. safety verification is decidable, but the closures cannot be computed [May03]. We will give an overview of the related work in Section 7.2.

Outlook

In this thesis, we will focus on Petri nets and BPP nets, a restricted subclass thereof. As mentioned earlier, these models are very useful when it comes to modelling concurrent systems. Given a Petri net S, it is obvious how to construct Petri nets whose languages are the downward resp. upward closures of $\mathcal{L}(S)$. However, we would like to obtain not a Petri net, but rather a finite automaton as a representation for the downward resp. upward closure. The computation of the language closures for Petri nets has not received much attention in the literature yet. There has been work by Habermehl, Meyer, and Wimmel [HMW10] that resulted in a procedure that computes the downward closure. We will present algorithms for the computation of the upward closure of Petri net languages as well as for the computation of both the upward and the downward closure of BPP net languages. While the general algorithm for Petri nets could be applied to compute the closures of BPP nets, our specialized procedures exploit the properties of BPP nets to achieve a much better running time.

Our proposed algorithms for the computation of the closures extend various well-known techniques from the literature. For example, we tweak the definitions used by Rackoff [Rac78] in his proof that Petri net coverability is EXPSPACE-hard in order to obtain a bound on the length of the computations that are needed to generate all minimal words in the language of the net. With this bound at hand, we can construct an automaton whose language is equal to the upward closure of the Petri net language. In the case of BPP nets, we use a similar approach, but the required bound on the length of computations results from using unfoldings [EH08]. On the one hand, the results in Part III. of the thesis often rely on the versatility of results from the literature. On the other hand, in order to be able to use these results for the computation of the closures, we have to extend them and incorporate fresh ideas.

We conclude this section by explaining how both the environment and certificates come into play in the setting that we consider. At the beginning, we have argued that the challenging aspect is that we are given a description of just the system, but our goal is to decide a property of the behavior of the system including the communication channel. This means that the communication channel itself forms an environment according to our definition: It is the difference between the real system behavior that we should verify and the model of the system that we are given. The theory that we have presented allows us to deal with this environment, even with no information about the channel beyond the fact that it is lossy or gainy.

It remains to show which role certificates play. As we have seen, deciding a property of the behavior of a system that we interact with via an unreliable communication channel boils down to deciding a property of a downward or upward closure. For a downward- or upward-closed language having or not having a certain property, the finite automaton representing that regular language can serve as a certificate. Finite automata are indeed good certificates in principle, as it is typically rather simple to check properties of their languages. Unfortunately, the latter task scales with the size of the automaton, i.e. it becomes harder the larger the automaton is. The procedures that we will provide later to compute automata describing the language closures can result in very large automata, e.g. automata that are doubly exponentially larger than the initial system. Unfortunately, this cannot be avoided: We will prove results in the areas of descriptive complexity showing that the size of the automata yielded by our algorithms is optimal. To circumvent this problem, we suggest an approach that is based on using so-called *simple-regular expressions (SREs)* as certificates for properties of the language closures. SREs are particularly simple representations for certain regular languages. While not every regular language can be represented by an SRE, SREs are expressive enough to represent downward- and upward-closed languages [ACBJ04]. It turns our that proving that the language of an SRE is included in the downward or upward closure of a Petri net language is much easier than computing a representation of the language closure itself. Hence, SREs can serve as a certificate for properties of downward- or upward-closed languages, since both checking that the SRE is included in the language closure and checking properties of the SRE is relatively easy. The latter comes from the fact that SREs represent regular languages, but an SRE representing just a part of the downward or upward closure is hopefully much smaller than a full representation of the closure.

Finally, we will also consider the problem of checking whether the language of a given Petri net is downward or upward closed. To settle this problem, we will in particular show how to check whether the language of a Petri net contains the language of a finite automaton. The decidability of this *regular containment problem* is not only of independent interest, but it also means that we can solve the following task: Given a Petri net and an automaton whose language should be equal to the language of the net, we can check that this equality indeed holds. In particular, when we are given a Petri net modelling a system that incorporates gainy or lossy aspects and we want to verify that its language is downward or upward closed, then we can compute a finite automaton representing the language closure using the aforementioned results and use the decidability of regular containment to verify that the two languages are equal.

1.3 Games

The last part of this thesis is concerned with games. In this section, we start by giving a brief introduction. We then present three examples for games that are useful in theoretical computer science. We discuss the resulted presented in this thesis that allow us to solve such games.

Games with perfect information

A *game* is a system in which several independent entities influence the behavior. The system constitutes a game arena, the entities are called players, and an execution of the system is referred to as a play. As the name suggests, what is colloquially referred to as a game can be seen as a game according to this definition. This applies to team sports like soccer, board games like chess, and video games of all kinds. The interest in games in science mostly comes from their applications that are less obvious. In mathematics, there is interest in properties like the determinacy of games, and the most-famous result in this area, the Borel determinacy theorem [Mar75], has applications in set theory. In economics, games are used to model capitalist systems, with the players representing the participants in a free market [LR94].

We will focus on explaining why games are studied in the context of theoretical computer science. In both this explanation and the rest of this thesis, we will focus on *games with perfect information*. In these games, the players know the rules of the game (including the game arena, i.e. the system the game is played on) and whenever they have to make a decision, they are aware of the history, the sequence of decisions that have been made by themselves and the other players up to this point. Perfect information also means that the players make their choices one after another instead of concurrently. We consider games with a winning condition, a special type of zero-sum games [Rag94]. This means the set of maximal plays, the plays that cannot be extended anymore, is partitioned into those that satisfy the condition and those that do not. There is a coalition of players that tries to influence the play so that it satisfies the winning condition, while the rest of the players, the opposition, works against that goal. In a perfect-information game, it is sufficient to consider two players with exactly one of them trying to satisfy the winning condition.

Strategies, parity tree automata, and game semantics

We give three examples of perfect-information games that are important in computer science. We start by considering systems with multiple types of nondeterminism. At the beginning of this chapter, we have already mentioned that many system models feature nondeterminism. This nondeterminism is typically not a genuine part of the system at hand, but it has been introduced to simplify behavior that should not or cannot be modelled explicitly. If just one type of nondeterminism is present, we usually assume that the nondeterminism works in our favor, or that all of it works against us. A reachability problem for a nondeterministic system can be phrased as “Can the nondeterminism be resolved in a way that leads to reaching the target?”,

seeing the nondeterminism as angelic. The complement problem of unreachability can be formulated as "Can the target be avoided, no matter how nondeterminism is resolved?", using the opposite concept of demonic nondeterminism. However, a system may have multiple types of nondeterminism with some of them being angelic, i.e. working in favor of a property we are trying to establish, while others are demonic and work against it. In such a case, it is natural to see the system as a game with the various types of nondeterminism being the players.

We consider the acceptance problem of parity tree automata [Zie98] as an example. A tree is an object whose branches represent various executions of a system. The branching behavior can be seen as one type of nondeterminism. The automaton provides a second type of nondeterminism that works in the opposite way. To be precise, a parity tree automaton accepts an infinite binary tree if there is a run of the automaton so that a so-called *parity condition* is satisfied for all branches of the tree. The nondeterminism coming from the branching of the tree is demonic in the context of the acceptance problem, because to achieve acceptance, the run needs to satisfy the parity condition on every single branch of the tree. The nondeterministic automaton itself provides a second type of nondeterminism that is angelic, because the existence of one valid run is sufficient to prove acceptance. Given an infinite tree and an automaton, the question of whether the tree is accepted can be seen as a game [Zie98]. One of the players represents the branching in the tree; her job is to pick a branch of the tree. The other player represents the automaton and picks a run. The two players alternate in making their choices, resulting in a branch of the tree and a run of the automaton on that branch. The goal of the player representing the automaton is to pick a run so that it satisfies the parity condition.

To be able to continue our explanation, we need to make clear what it means to solve a game. We ask whether one of the players has a systematic way of playing that guarantees that any resulting play will satisfy (resp. not satisfy) the winning condition, no matter which choices the other player makes. A systematic way of playing is called a strategy, and if it achieves the aforementioned objective, it is called a winning strategy. A player is the winner of a game if she has a winning strategy, and solving a game means computing which player wins.[1]

In the game we described above, the player representing the automaton wins the game if and only if the tree is accepted by the automaton, and her winning strategy corresponds to a run of the automaton that satisfies the parity condition on all branches. This has three consequences: Firstly, assuming we are given a finite description of a tree, we can decide whether the three is accepted by a parity tree automaton by solving a game. Secondly, the game can be modified so that instead of checking whether a given tree is accepted, it checks whether there is a tree that is accepted, thus solving the language-emptiness problem for parity tree automata. Finally, one can use games in order to prove Rabin's tree theorem [Rab68; Zie98], a deep and important

[1] It is easy to see that it is impossible for both players to have a winning strategy for a given game: If both players follow their strategy, the result is a unique play that either satisfies the winning condition or it does not – at most one of the two strategies is a winning strategy. However, it is not at all clear that at least one player has a winning strategy. We call games with the property that exactly one player has a winning strategy *determined*. For now, it shall suffice to say that all games that we consider in the following have been proven to be determined.

result in automata theory showing that the languages of parity tree automata are closed under complementation. To this end, one constructs an automaton that accepts the complement of the language of a given parity tree automaton by checking that the player representing the automaton does not have a winning strategy in the game corresponding to a given tree.

Our second example is *game semantics* [LL78]. In many verification problems, the specification is given as a formula in a certain type of logic. For some of these types of logic, it is instructive to see the logical formulas themselves as games. For example, the modal μ-calculus [BW18] features fixed-point operators. These operators can be used together with a subformula to assign to a variable the greatest or least set of system states that satisfy the subformula. When checking whether a system satisfies a specification expressed in the modal μ-calculus, these fixed points have to be computed. Instead of using a deterministic up-front computation of the sets, game semantics provides an elegant alternative. One can construct a game that features a guess-and-check mechanism. Whenever a fixed-point quantifier is encountered, one of the players suggests the value of the set. Then the other player can either verify the suggestion, or she can use the suggested value. This mechanism is designed so that the player suggesting the sets loses the game if she makes an incorrect suggestion. Hence, computing her winning strategy means implicitly computing the values for the fixed points.

Game semantics for μ-calculus on finite structures can be extended to show deep results for automata-theoretic games as we will consider them in this thesis. For example, Walukiewicz's reduction [Wal01] is a famous result that was developed in the context of showing the satisfiability of μ-calculus formulas on pushdown systems. It proves that games on infinite game arenas defined by pushdown automata can be reduced to games on finite arenas with an exponential blowup. We will come back to this result in Section 17.6.

Synthesis

Our final example is the area of synthesis. We have argued that the verification of programs is an important subject. It seems canonical to ask whether instead of verifying a given program that has been written by humans, one can let a computer generate a correct program from the given specification. This idea is known under the name of *(program) synthesis* [Chu63; MW80]. If one could implement it in practice, not only would it allow for the automatic generation of programs with very little time investments by humans, but it would also solve the problem of verification. If the synthesis tool is correct, i.e. given a specification, it always returns a program that indeed satisfies the specification, the programs that it produces can be assumed to be correct and no effort has to be put into verifying them.

For certain types of systems, like Boolean circuits, synthesis tools have been developed with moderate success [JP21]. Unfortunately, the synthesis of complex programs suffers from various problems. One conceptual problem is that synthesis does not save as much human time investment as it may seem. The effort is simply shifted from writing the program to writing the formal specification. Each aspect of the desired system has to be specified extensively to ensure

that the resulting program has the expected behavior. If a mistake has been made when writing the specification, the resulting flaw of the program may be hard to identify. Another fundamental problem of synthesis is the computational complexity of synthesis problems. Firstly, many of these problems are in high complexity classes, e.g. synthesizing a program from a specification may take time doubly exponential in the size of the specification and the program may be just as large [MSS06]. Secondly, unlike for other problems where the worst-case behavior rarely occurs in practice, synthesis problems are known to actually show this terrible complexity on practical examples [Var18].

We try to circumvent both problems by considering *syntax-guided synthesis* [ABDFG+15; Pad21]. In addition to a specification, a syntax-guided synthesis problem also consists of a *program template*, which is essentially a program with missing parts, e.g. it may be a collection of modules that need to be connected appropriately. The goal is to fill in the missing parts of the program template such that the resulting program satisfies the specification. The hope is that this version of the synthesis problem is simpler to handle. The synthesis tool having less freedom because it only has to generate code at specified locations in the program means that there is a better chance of obtaining a manageable running time. The fact that a large part of the desired program is given in the form of the program template means that we are not as heavily reliant on providing a detailed specification for all aspects of the program as in the version of the synthesis problem that takes no program template. Note that the synthesis task will only succeed if it is in fact possible to instantiate the program template so that the resulting program satisfies the specification. This means that conceptually, the synthesis task includes the verification of the given program template.

Synthesis as a game

In the following, we model a synthesis problem as a game. The game has two players, one for the environment and one for the program synthesizer. In the parts of the program template that are fully specified, the environment is in control. This means for any nondeterministic choice in that part of the program, the player representing the environment resolves it. We have argued earlier how such nondeterministic choices may be introduced to model dependencies on external resources or as the result of applying an abstraction. When the program arrives at a part of the template that should be filled in by the synthesis tool, the other player is in control. A play is won by the player representing the program synthesis if and only if it satisfies the specification. In that case, her winning strategy corresponds to the solution to the synthesis problem.

In order to make this explanation more precise, we will give a more concrete example. We focus on a simple case of program templates in which the task is to synthesize conditional expressions. This means our template may contain conditionals that are of the form `if (???) then { ... } else { ... }`. The goal is to replace `???` so that the resulting program satisfies the specification. We present context-free games as a model for this synthesis task. In a context-free game, the game arena is potentially infinite, but it is given by a

finite description in the form of a context-free system. We have already briefly mentioned pushdown automata as one example for context-free systems. Here, it will be easier to consider context-free grammars, an equivalent model. In the derivation of a word using a context-free grammar, nonterminals are replaced using production rules until we obtain a word that only contains non-replaceable terminal symbols. As mentioned before, context-free grammars are sufficient to model the control-flow of recursive programs. This will make it easy to translate the given program template into a context-free game.

As in the other two sections, we prefer to take a language-theoretic approach. This means that our specification is given in the form of a language, and the plays of the game produce words that may or may not be in that language. The language defines the winning condition: The goal of the environment player is obtaining a play, i.e. an execution of the program, that corresponds to a word not in that language. The goal of the synthesis player representing the environment is the opposite, she wants to enforce that if the play of the game produces a word, then that word is contained in the language. Here, we consider regular languages of finite words, represented by finite automata, as target languages. Note that by choosing a larger class of languages, e.g. context-free languages, the problem would become undecidable.

Assume in the program template, some procedure `p()` is specified by the source code `if (x) then { f(); } else { g(); }`. Note that in this case, the conditional expression is explicitly given and no synthesis has to be performed. In the context-free grammar, we would introduce nonterminal symbols for every program location, e.g. for `p()`, `f()`, and `g()`. In the concrete example, we would translate the program code into the two rules `p()` $\rightarrow$ read(x, true).`f()` and `p()` $\rightarrow$ read(x, false).`g()`. Note that read(x, true) and read(x, false) are terminal symbols, symbols that cannot be replaced, that signal that the current value of variable x should be true resp. false. Finally, we give ownership of `p()` to the environment player. This means that when the nonterminal `g()` has to be replaced during a play of the game, this player can choose whether to apply the first or the second rule, replacing `p()` with the right-hand side of the chosen rule. This is an example for a concept that we have mentioned earlier: We have replaced a deterministic choice in the program by nondeterministic branching. On the one hand, this allows us to simply model the control flow of the program in the context-free grammar without having to keep track of the data values. On the other hand, it means that we need to somehow enforce that the environment player cannot simply create an invalid execution by picking the wrong branch, e.g. by picking the rule `p()` $\rightarrow$ read(x, false).`g()` when the current value of x is actually true.

Here, the language-theoretic approach comes in handy. Recall that our specification is given as a target language of valid executions. In order to solve the above problem, we can assume that the executions that are impossible to occur in reality because they violate the consistency of data values are added to the target language. Technically, this means we construct a finite automaton that uses two states to keep track of whether the value of x is true or false. The automaton accepts an execution if it contains an invalid data access, e.g. if we have an occurrence

of read(x, false) even though in the current state, the value of x is true. The original specification can be extended by joining the corresponding automata. Let us consider again the choice that the environment player has to make in the game when replacing nonterminal p() in a play. If she deliberately picks the wrong rule, i.e. the rule that does not correspond to the current value of variable x, the automaton will definitely accept any finite word that is produced by the play. Hence, such a word is in the target language and the environment player loses the play. Hence, her only chance to win will be to pick the rule that corresponds to the current value of x.

Note that the above concept only works for variables that can only take a small range of values, e.g. a Boolean variable as in our example. For a data value that is e.g. a 32-bit integer, the automaton we need to keep track of all possible values would have 2^{32} states. In such a case, it would be better to initially not keep track of the data value at all. We can then use an iterative process like the CEGAR-loop [CGJLV00; HHP10] to make sure that if the synthesis algorithm comes up with a program, that program is indeed correct with respect to handling this data value.

We have extensively discussed the environment and the corresponding player. Note that in contrast to the other two main topics of this thesis, this time, the behavior of the environment is explicitly specified in our model. The difference here is that we are able to react to it. Let us now consider the player representing the program synthesis. When discussing her behavior in the game, we will also see how certificates come into play.

Assume that the source code for p() is `if (???) then { f(); } else { g(); }`. Note that this time, the conditional expression is not given; it should be filled in by the synthesis algorithm. We model this by giving ownership of p() to the synthesis player and by having the rules p() → f() and p() → g(). This means whenever in a play nonterminal p() has to be replaced, the synthesis player can choose freely between replacing it by f() or replacing it by g(). Correspondingly, in an execution of the system, whenever procedure p() is called, the synthesis tool can decide whether to go to the if-branch and call procedure f() or whether to go to the else branch and call procedure g(). The absence of any kind of conditional expression in the rules that we have designed to model the source code of p() may be surprising. We will come back to this aspect soon.

After translating a program template and a specification into a context-free game with a regular target language, one can solve that game. The synthesis problem can be solved if and only if the synthesis player wins the game. However, this information is insufficient: In order to actually solve the synthesis task, we need to come up with an instantiation of the program template. To this end, we consider a winning strategy for the synthesis player. This strategy is a description of how she has to behave in order to ensure that she wins the game. The strategy also acts as a certificate for the fact that she is the winner. Verifying that a given strategy is winning is usually much simpler than computing a winning strategy. For example, for context-free games with membership in a regular target language as the winning condition, a given strategy can be used to transform the context-free game into a normal context-free grammar. Checking that

the strategy is winning amounts to checking that the resulting grammar produces a language that is a subset of the regular target language. Hence, we also say that the winning condition of the game is regular inclusion and call this type of game an inclusion game.

Let us come back to the instantiation of the template, i.e. synthesizing the conditional expression in the source code for `p()`. Note that the goal here is not to make a static choice between the if- and the else-branch, which would amount to either putting true or false as the conditional. Instead, every time procedure `p()` is encountered, the synthesis player can decide which branch to pick. This means that a winning strategy for her will pick one of the two rules depending on the history of the play, i.e. what has happened so far in the execution of the system. For context-free games, one can prove that if a winning strategy exists, then there is one that can be implemented by a so-called strategy automaton. If we have this automaton available at runtime, we can query its state to decide which rule to pick.

In order to implement this, we have to compute a strategy automaton that represents a winning strategy for the synthesis player. Afterwards, the program template is modified to keep track of the state of the strategy automaton. Assume the automaton has states from the set $\{1, \ldots, n\}$. The program is modified by introducing corresponding Boolean variables $\mathtt{a}_1, \ldots, \mathtt{a}_n$ so that during runtime, $\mathtt{a}_i$ is true if and only if the strategy automaton is currently in state $\mathtt{a}_i$. When we encounter procedure `p()`, the strategy will tell us whether to use the if- or the else-branch. Correspondingly, there is a list of states $i_1, i_2, \ldots, i_k \in \{1, \ldots, n\}$ of that automaton in which the if-branch should be taken, while the else branch should be taken in all other states. The conditional expression that we need to synthesize is $\mathtt{a}_{i_1}$ `or` $\mathtt{a}_{i_2}$ `or ... or` $\mathtt{a}_{i_k}$, and hence the completed source-code for procedure `p()` is `if (`$\mathtt{a}_{i_1}$ `or` $\mathtt{a}_{i_2}$ `or ... or` $\mathtt{a}_{i_k}$`) then { f(); } else { g(); }`. Note that even if the program template contains multiple conditional expressions that should be synthesized, one strategy automaton is sufficient to represent the solution to the synthesis problem. But for each conditional expression that has to be generated, the states of the automaton that correspond to taking the if-branch may differ.

In summary, we have translated a program template and a specification into a context-free inclusion game. One player represents the environment. Whenever the program template contains a conditional with a given conditional expression, she is allowed to resolve nondeterministically which branch to pick, but the winning condition of the game will force her to respect the actual data values of the variables. The other player represents the synthesis task. By solving the game, we check whether it is possible to complete the synthesis task or whether any instantiation of the template will result in a program that violates the specification. If the synthesis player wins the game, synthesis is possible. We compute a winning strategy for that player as a certificate for the fact that she wins. This winning strategy can be represented in the form of a strategy automaton which in turn allows us to actually complete the synthesis task.

Outlook

Our main goal in Part V. of this thesis will be finding a procedure efficiently solving context-free inclusion games. Before we do so, we explain *effective denotational semantics* [Aeh07; Sum77; SW15b], the approach that we will take to solve context-free inclusion games and several other problems in that part of the thesis. This technique is based on translating a verification problem into a system of equations and then computing its least solution, from which the answer to the verification problem can be read off. We show how this method can be applied to the regular inclusion problem for context-free grammars as demonstrated by Holík and Meyer [HM15] and we extend that work to the case of languages of infinite words.

Then, we use the same approach in a more complex setting by considering context-free games with membership in a regular language of finite words as the winning condition. With the help of a novel composition operator, these games can be translated into a system of equations. Their least solutions do not only provide the winner of the game, they can also be used to construct strategy automata for the winning strategies as certificates. We compare our method to existing methods from the literature that could be adapted to solve this type of game, including Walukiewicz's aforementioned reduction [Wal01]. Additionally, we consider two extensions of our method. The first one consists of having a regular language of infinite words as the target language, which is useful to model reactive systems with non-terminating executions.

The second extension is considering game arenas described by higher-order recursion schemes, a generalization of context-free grammars. Solving these games using effective denotational semantics requires a major amount of work, but leads to several interesting by-products. We will establish both a template model for interpreting systems of equations defined by higher-order recursion schemes and a framework that can be used to transfer properties of the least solution with respect to one model to the least solution with respect to another mode. The template model mechanism and the framework then enable us to solve games defined by higher-order recursion schemes.

Finally, we will study the frontier of the decidability of games. Synthesis problems based on program templates can be seen as a generalization of verification. Indeed, if the given program template actually is a complete program, then synthesis simply amounts to verifying that program. Correspondingly, solving a game is more involved than solving e.g. a reachability problem for a nondeterministic system. Hence, for any model that is Turing-complete, solving the associated type of games is certainly just as undecidable as verification is. However, there are models like Petri nets for which reachability problems can be solved, but it turns out that the corresponding games are undecidable, i.e. we cannot compute their winner in general. We will use valence systems over graph monoids, a model of automata that generalizes many well-known models including both Petri nets and pushdown automata, to give a complete classification of the models for which games are decidable. We show that among all automata models that can be represented as a valence system over graph monoids, reachability games can essentially be only solved in the aforementioned case of context-free games.

2 Outline

We give a brief overview of the structure of this thesis. The thesis is partitioned into six parts, the first of which is the introduction that this chapter concludes.

Part II. consists of preliminaries. It introduces the models of computation on which the results in the rest of the thesis are based. It mostly presents results that have been established in the literature without contributions from the author of this thesis.

Part III. presents our results on the closures of Petri net coverability languages. These correspond to Section 1.2 of the introduction. We study the computation of both the downward and the upward closure of Petri net languages as well as restricted versions of these problems. We establish upper bounds on the size of the closures by providing algorithms, typically matching the lower bounds that we also present and prove. The content of this part is based on the publication [AMMS17].

Part IV. is concerned with results related to the regular separability of WSTS languages. We have given a brief introduction to this topic and highlighted its relevance in Section 1.1 of the introduction. We first establish some basic results on various classes of WSTS languages. Then, we prove the main result, showing that disjoint WSTS languages are always separable under certain mild conditions. Finally, we give an upper bound and lower bound for the size of the construction in the case of Petri net coverability languages. The content of this part is based on the publication [CLMMKS18].

Part V. presents our results on solving games. We have explained how games are related to problems in verification and synthesis in Section 1.3 of the introduction. After providing basic definitions for games, we describe a fixed-point based approach to solving verification problems. This approach will be our vehicle for solving games defined by context-free grammars and higher-order recursion schemes afterwards. We conclude the part by exploring the boundaries of the decidability of games. To this end, we use valence systems over graph monoids as a model that provides a unified theory for various types of automata. The content of this part is based on the publications [HMM16; HMM17; MMZ18; MMN17].

Part VI. forms the conclusion. Firstly, we summarize the contributions of this thesis. We make clear for each of the results whether it has been published before. We also detail who, in addition to the author of this thesis, has contributed to establishing the result. Secondly, we give an outlook on potential future work. This includes extensions of the theoretical results as well as a brief discussion on how the results in this thesis could be applied in practice.

We will give a more detailed overview at the beginning of each part of the thesis.

Part II.

Models of computation

Contents

3 Preliminaries 55
- 3.1 Basics . . . 55
- 3.2 (Alternating) Turing machines . . . 58
- 3.3 Classes of computational complexity . . . 62
- 3.4 Formulas in propositional logic and Presburger arithmetic . . . 69

4 Labeled transition systems and finite-state automata 73
- 4.1 (Labeled) Transition systems . . . 73
- 4.2 Finite-state automata . . . 77
- 4.3 Descriptive complexity . . . 79
- 4.4 ω-regular languages . . . 82
- 4.5 The transition monoid . . . 85

5 Grammar-based models 89
- 5.1 Context-free grammars . . . 89
- 5.2 ω-languages of context-free grammars . . . 95
- 5.3 Higher-ordered recursion schemes . . . 103

6 Petri nets and well-structured transition systems 109
- 6.1 Unlabeled Petri nets . . . 109
- 6.2 Algorithmic problems for Petri nets . . . 113
- 6.3 Petri net coverability languages . . . 118
- 6.4 BPP nets . . . 123
- 6.5 Well-structured transition systems . . . 132

Part II.
Models of computation

This part contains some preliminaries: We formally define the models of computation on which the problems that we will study in the rest of this thesis are based.

Outline

We start by introducing some basic notation. Then, we define Turing machines and formulas in propositional logic and Presburger arithmetic; while they are not a main object of study in this thesis, they are needed for proving some complexity results.

In Chapter 4, we define labeled transition systems as state-based language-generating mechanisms. In particular, we focus on the class of finite automata.

Chapter 5 is concerned with context-free grammars and higher-order recursion schemes as grammar-based language-generating mechanisms.

Finally, Chapter 6 introduces Petri nets, a model for concurrent systems, and an extension thereof in the form of well-structured transition systems.

Some concepts that are only used in a single part of the thesis will be defined later: board games with perfect information in Chapter 15, fixed-point based techniques for systems of equations in Chapter 16, and valence systems in Chapter 19.

Sources and publications

This part contains almost no contributions by the author of the thesis; it merely presents commonly used models following standard textbooks. References will be given in the main text. There are two exceptions: In Section 5.2, we present a novel way in which context-free grammars can be used to define languages of infinite words. This is taken from the publication [MMN17]. In Section 6.4, we show that the word problem for BPP nets is NP-complete. This result is not contained in the publication [AMMS17], but its proof is an adaption of the proof of Theorem 24 from the full version [AMMS17a] of the paper. We will discuss the contributions to these publications by the author of the thesis in more detail in Chapter 20.

3 Preliminaries

Contents

3.1 Basics 55
3.2 (Alternating) Turing machines 58
3.3 Classes of computational complexity 62
3.4 Formulas in propositional logic and Presburger arithmetic 69

After defining some basic notation, we give a brief introduction to (alternating) Turing machines and the robust complexity classes derived from them. We introduce formulas in propositional logic and Presburger arithmetic and discuss the complexity of their satisfiability problems.

3.1 Basics

We start by fixing some notation that will be used throughout the thesis. Whenever we use a symbol in the definition of a class of objects, e.g. Σ for alphabets, all later occurrences of that symbol shall denote an arbitrary object from that class, even if it is not explicitly stated, e.g. Σ always denotes some alphabet in the rest of this thesis. We provide additional explanation whenever it is needed to avoid ambiguity.

Sets and functions

For a set X, we denote by $\mathcal{P}(X) = \{Y \mid Y \subseteq X\}$ its *powerset*, the set of its subsets. Its *cardinality* is $|\mathcal{P}(X)| = 2^{|X|}$. We use $M \uplus N$ to denote the *disjoint union* of M and N, i.e. its value is $M \cup N$, but we additionally express that the sets are disjoint ($M \cap N = \varnothing$ holds). For sets X, Y, we see $X \to Y$ as the set of functions from X to Y. Nevertheless, we write $f\colon X \to Y$ as usual to denote $f \in (X \to Y)$. If $f\colon X \to Y$ is a function and $X' \subseteq X$ is a subset, we denote by $f_{\restriction X'}\colon X' \to Y$ its *restriction* to X'.

Numbers

We use $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ to denote the *integers* and $\mathbb{N} = \{0, 1, 2, \ldots\}$ to denote the *natural numbers*, the non-negative integers (including 0). For numbers $i, j \in \mathbb{Z}$, $[i, j] = \{k \in \mathbb{Z} \mid k \geqslant i, k \leqslant j\}$ is the *(closed) interval* from i to j.

Whenever convenient, we see $n \in \mathbb{N}$ as the set $\{0, \ldots, n-1\}$ of cardinality n. For the set with two elements, we also write $\mathbb{B} = \{0, 1\}$ and identify its elements 0 and 1 with the truth values false and true, respectively. We may see an element of the powerset $\mathcal{P}(X)$ as a function with signature $X \to \mathbb{B}$, the so-called *characteristic function* that specifies for each element of X whether it is contained in the subset.

For a set X and $k \in \mathbb{N}$, X^k is the k-fold Cartesian product of X with itself, and its element are tuples $(x_1, \ldots, x_k)$ of dimension k. The special case of $k = 0$ yields the singleton set $X^0 = \{()\}$ that only consists of the empty tuple. We call such tuples *vectors* (of dimension k), even if X^k is not a vector space. We sometimes write $\vec{v}$ to make clear that symbol v denotes a vector. In particular, we use $\vec{0} \in \mathbb{Z}^k$ for the vector of suitable dimension with all components 0. We may see vectors $v \in \mathbb{N}^k$ as functions $v\colon [1, k] \to \mathbb{N}$. Similarly, we may denote by X^Y the set of functions with signature $Y \to X$ whenever we want to see such functions as vectors.

For a finite set $M \subseteq \mathbb{Z}$ of numbers, we use $||M||_\infty = \max_{m \in M} |m|$ for the *infinity norm* of M, the maximum of the absolute values of the numbers in M. We extend this notation to vectors by seeing them as a set of their entries. Similarly, if $f\colon N \to \mathbb{Z}$ is a function whose *range* $f(N) = \{f(n) \mid n \in N\} \subseteq \mathbb{Z}$ is a finite set of numbers, we use $||f||_\infty$ to denote $||f(N)||_\infty$.

For a finite set $M \subseteq \mathbb{Z}$ of numbers, we use $||M||_1 = \sum_{m \in M} |m|$ for the ℓ_1-*norm* of M, the sum of the absolute values of its elements. Similarly, the ℓ_1-norm of a vector $v \in \mathbb{Z}^k$ is $||v||_1 = \sum_{i \in [1,k]} |v_i|$.

Orders

For a relation $\leqslant \subseteq X \times X$ we denote by $\leqslant^{-1}$ its *opposite*, the relation with $x \leqslant^{-1} y$ iff $y \leqslant x$. Whenever suitable, we simply use the inverted symbol for the opposite, e.g. $\geqslant$ and $>$ for the opposites of $\leqslant$ and $<$, respectively.

Recall that a *quasi-order* $\leqslant$ on some set X is a relation $\leqslant \subseteq X \times X$ that is reflexive and transitive. It is called a *partial order* if it also is antisymmetric.

For a quasi-order $\leqslant$ we use $<$ to denote the irreflexive order defined by $x < y$ iff $x \leqslant y$ and $y \not\leqslant x$. Note that if $\leqslant$ is a partial order, this definition of $<$ coincides with the usual one: $x < y$ iff $x \leqslant y$ and $x \neq y$.

Given two ordered sets $(X_1, \leqslant_1), (X_2, \leqslant_2)$, we denote by $\leqslant_\times$ the *product order* on $X_1 \times X_2$ in which a tuple is bigger if it is bigger in each component:

$$(x_1, x_2) \leqslant_\times (y_1, y_2) \quad \text{iff} \quad x_1 \leqslant_1 y_1 \text{ and } x_2 \leqslant y_2 \,.$$

If $\leqslant_1$ and $\leqslant_2$ are quasi-orders resp. partial orders, then so is $\leqslant_\times$.

The most common use case will be that we take the product of multiple copies of the same ordered set $(X, \leqslant)$, e.g. in the case of $\mathbb{N}^k$. In this case, we simply denote the product order by $\leqslant$. This means we implicitly generalize the order on a set to vectors over that set. We do the same for other operations on the set, e.g. we generalize the addition and subtraction of numbers to (component-wise) addition and subtraction of vectors.

A subset Y of a quasi-ordered set $(X, \leqslant)$ is a *chain* if its elements are pairwise comparable. It is an *antichain* if its elements are pairwise incomparable. An infinite *ascending chain* is a sequence $(x_i)_{i \in \mathbb{N}}$ of elements of X such that $x_i \leqslant x_{i+1}$ for all $i \in \mathbb{N}$. As the name suggests,

the set of elements $\{x_i \mid i \in \mathbb{N}\}$ occurring in such a sequence is indeed a chain. *Infinite strictly ascending*, *infinite descending*, and *infinite strictly descending chains* as well as their finite versions are defined similarly in the expected way.

Finite words

An *alphabet* Σ is a finite set, its elements are called *letters* or *symbols*. In the case of an alphabet, we write tuples $w \in \Sigma^k$ as $w_1 \dots w_k$ and call them *(finite) words* of *length* $|w| = k$. The empty tuple is called the *empty word* $\varepsilon \in \Sigma^0$; it is the unique word of length $|\varepsilon| = 0$. The set $\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma^i$ is the *set of all finite words*. A *(formal) language (of finite words)* is a subset $\mathcal{L} \subseteq \Sigma^*$.

Finite words w, v can be *concatenated* in the expected way, which we denote by $w.v$ or simply wv. Concatenation is lifted to languages by applying it element-wise. For a language $\mathcal{L}$, we inductively define $\mathcal{L}^0 = \{\varepsilon\}$, $\mathcal{L}^{i+1} = \mathcal{L}.\mathcal{L}^i$. The *Kleene star* $\mathcal{L}^* = \bigcup_{i \in \mathbb{N}} \mathcal{L}^i$, the positive hull $\mathcal{L}^+ = \bigcup_{i>0} \mathcal{L}^i$, and the complement (relative to Σ^*) $\overline{\mathcal{L}} = \Sigma^* \setminus \mathcal{L}$ are defined as usual.

Infinite words

We define $\omega = (\mathbb{N}, \leqslant)$ as the standard order $\leqslant$ on the natural numbers $\mathbb{N}$. The set Σ^ω is the set of all functions $f : \mathbb{N} \to \Sigma$. We represent such a function by the infinite sequence of its function values, i.e. we write $w = w_1 w_2 w_3 \dots$ to denote the function f defined by $i \mapsto w_{i-1}$. Accordingly, we call such functions *infinite words* and subsets of Σ^ω *(formal) languages of infinite words*, or *ω-languages* for short.

A finite word on the left and an infinite word on the right can be concatenated to obtain an infinite word, similar for the corresponding types of languages. For a language $\mathcal{L} \subseteq \Sigma^*$ of finite words, we define its *ω-iteration* $\mathcal{L}^\omega$ to be the set of all infinite concatenations $w^{(1)}.w^{(2)} \dots \in \Sigma^\omega$ with $w^{(i)} \in \mathcal{L} \setminus \{\varepsilon\}$ for all $i \in \mathbb{N}$. Note that we exclude ε to ensure that the result is an infinite word.

Notation

We mostly use $w = w_1 \dots w_n$ for the decomposition of a word into its letters, i.e. $w_i \in \Sigma$; we resort to using different notation, e.g. $w = w^{(1)} \dots w^{(n)}$ with $w^{(n)} \in \Sigma^*$, for a decomposition into infixes. We often use w to denote the singleton language $\{w\}$, and write e.g. w^ω for $\{w\}^\omega$.

3.2 (Alternating) Turing machines

We define alternating Turing machines as a general form of computational device and nondeterministic and deterministic Turing machines as special cases. The presentation loosely follows [Koz06]. In contrast to the restricted models of computation that we will define later, a Turing machine [Tur36] is a computational device that has unrestricted access to an unbounded storage. Alternating Turing machines [CKS81] additionally provide both universal and existential nondeterminism.

Formally, an *alternating Turing machine (ATM)* is a tuple $M = (Q_\forall, Q_\exists, q_{\text{final}}, q_{\text{init}}, \Sigma, \delta)$ where $Q = Q_\forall \uplus Q_\exists \uplus \{q_{\text{final}}\}$ is a finite set of *control states*, partitioned into the *universal states* $Q_\forall$, the *existential states* $Q_\exists$, and the *final state* q_{final}. The *initial control state* is $q_{\text{init}} \in Q$. The alphabet Σ is the *input alphabet* of the machine. Finally, $\delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ is the *transition relation*, where $\Gamma = \Sigma \uplus \{\textvisiblespace\}$ is the *tape alphabet* of the machine, consisting of the input alphabet and a special *blank symbol* $\textvisiblespace \notin \Sigma$, and $\{L, R\}$ is the set of *directions*. We write $(q, a) \to (p, b, d)$ instead of $(q, a, p, b, d) \in \delta$.

A configuration of an ATM has the shape $(w, q, a.v) \in \Gamma^* \times Q \times \Gamma^+$, oftentimes written as $w\ q\ a.v$. It consists of a control state, a finite tape content, and the position of the read/write head on the tape. To be precise, the tape content is infinite, but only finitely many cells are not filled with the blank symbol $\textvisiblespace$; we assume that $\textvisiblespace$ symbols have been exhaustively removed from both ends of the tape content, identifying $w\ q\ a.v$ with $\textvisiblespace^\omega.w\ q\ a.v.\textvisiblespace^\omega$.

The transition relation δ of an ATM induces a transition relation T on configurations in the expected way: We have $w\ q\ a.v \to w'\ q'\ a'.v'$ if either $(q, a) \to (q', b, L)$ and $w = w'.a'$, $v' = b.v$ or $(q, a) \to (q', b, R)$ and $w' = w.b$, $v' = a'.v$.

The semantics of alternating Turing machines is defined in terms of trees. The unique *computation tree* of an ATM M for input $x \in \Sigma^*$ is a tree labeled by configurations, defined inductively as follows: (1) The root node is labeled by $\varepsilon\ q_{\text{init}}\ x$, and (2) if the tree has a node labeled by $w\ q\ v$ with $q \neq q_{\text{final}}$, then for each configuration $w'\ q'\ v'$, with $w\ q\ v \to w'\ q'\ v'$, this node has exactly one child labeled with that configuration.

Note that a computation tree may be infinite. Leaves of the tree are either labeled by configurations with control state q_{final}, or by *rejecting* configurations in which no transition is applicable. We say that ATM M *halts* on input x if the computation tree for x is finite. We say that ATM M is a *decider*, or *total*, if it halts on all inputs.

To define the language of an ATM M, it remains to introduce the acceptance condition. A leaf of the computation tree is accepting if the control state (of the configuration labeling it) is q_{final}. An inner node is accepting if its control state is from $Q_\exists$ and it has an accepting child, or if the control state is from $Q_\forall$ and all its children are accepting. A computation tree is accepting if its root node is accepting. In this case, input x is contained in the language $\mathcal{L}(M) \subseteq \Sigma^*$. Phrased

differently, the language $\mathcal{L}(M)$ *recognized* by ATM M is the set of all words from Σ^* for which the computation tree is accepting. An ATM that is a total *decides* its language (or, more specifically, the membership problem for that language) in that one can construct the computation tree associated to an input in finite time and check whether it is accepting.

Remark
In the literature, e.g. in [Koz06], leaves of the tree are often considered to be accepting if they are labeled by a universal control state. This is motivated by seeing these nodes as empty conjunctions, and it simplifies some constructions for ATMs. As we do not need said constructions, we forgo defining such leaves as accepting, which saves us from some case distinctions.

Decision problems and word problems

Theoretical computer science is often concerned with studying decisions problems. These problems consist of a set of instances $\mathcal{I}$ and a partition $\mathcal{I} = \mathcal{I}_{\text{yes}} \uplus \mathcal{I}_{\text{no}}$ into the yes-instances $\mathcal{I}_{\text{yes}}$ and the no-instances $\mathcal{I}_{\text{no}}$. Intuitively, the question is, given an instance $i \in \mathcal{I}$ of such a problem, to determine whether it is a yes- or a no-instance. We often present decision problems in the following form.

Decision problem for $\mathcal{I} = \mathcal{I}_{\text{yes}} \uplus \mathcal{I}_{\text{no}}$
Given: Instance $i \in \mathcal{I}$.
Question: $i \in \mathcal{I}_{\text{yes}}$?

When we talk about the *decidability* of such a problem, we are asking whether a computer can solve it. In order to formalize this concept, one observes that for most decision problems of practical interest, it is possible to encode instances $i \in \mathcal{I}$ as words over some suitable alphabet. This means the set of instances is the set Σ^* of words over the alphabet, and the set of yes-instances as a language $\mathcal{L} \subseteq \Sigma^*$. With this encoding, a decision problem can be seen as a *word problem*, the problem of deciding whether a given word is in a language.

Word problem for $\mathcal{L} \subseteq \Sigma^*$
Given: Word $w \in \Sigma^*$.
Question: $w \in \mathcal{L}$?

We call the word problem for a language $\mathcal{L}$ *decidable* if $\mathcal{L}$ is the language of an ATM that is a decider. Intuitively, that ATM indeed decides the language. Given an input word, we can in finite time construct the computation tree of the ATM for the input and then read off whether it is accepting. If so, the input word is indeed a member of the language of the ATM.

A general decision problem is called decidable if the word problem that results from choosing an appropriate encoding is decidable. In particular, we can formalize decision problems that expect ATMs as inputs as word problems and study their decidability. Encoding an ATM into e.g. a binary or number representation is a process called Gödel numbering, named after a similar technique that was developed by Gödel for the proof of his incompleteness theorems.

The halting problem is one such example for a problem that expects a Turing machine as input.

Halting problem

Given: ATM M, input x for M.

Question: Does M halt on x, i.e. is the computation tree of M for input x finite?

A variant of this halting problem is the famous problem for which Turing [Tur36] showed that it is undecidable. It is not too hard to obtain an ATM that recognize the language associated to a suitable encoding of the halting problem: This ATM has to simulate the given ATM on the given input. Our definition of acceptance makes sure that if the input is accepted, then a finite subtree of the computation tree is sufficient to prove acceptance. Hence, the simulation can accept all yes-instances of the halting problem with the finite time. No-instances are hard to detect within finite time. If the given machine M does not halt on the given input, then the simulation of that machine will not halt either. In fact, Turing result show that it is impossible to construct a decider for the halting problem.

Restricted variants of Turing machines

An ATM is called an *(existentially) nondeterministic Turing machine (NTM)* if $Q_\forall = \varnothing$. In this case, an input is accepted if the associated computation tree contains an accepting leaf.

An ATM is called a *deterministic Turing machine (DTM)* if δ is unique in that for each tuple $(q, a) \in Q \times \Gamma$, there is at most one transition $(q, a) \to (q', a', d)$ in δ. In this case, each configuration has at most one successor, and the computation tree is actually a sequence of configurations. Whether the control states are universal or existential does not matter in this case; we may assume that every DTM is an NTM.

It is well-known that NTMs can be converted in polynomial-time into language equivalent DTMs. While the translation runs in polynomial time, the resulting DTM may have increased resource consumption over the original NTM (a notion that we will make precise in the next section). Similarly, ATMs that are deciders can be converted to DTMs, but this does not hold true for arbitrary ATMs: The class of languages that can be recognized by ATMs is a strict superclass of the class of *recursively enumerable* or *semi-decidable* languages RE, the languages recognized by DTMs and NTMs. For example, RE is not closed under complement[1] while the class of ATM-recognizable languages is.

[1] The famous halting problem [Tur36] is undecidable but semi-decidable. If RE were closed under complementation, its complement would be semi-decidable too, implying that the problem itself is decidable.

So far, we have seen Turing machines as devices for deciding acceptance. We may see a decider as a device that computes the (total) characteristic function of its language $\chi_{\mathcal{L}(M)} : \Sigma^* \to \mathbb{B}$ with $\chi_{\mathcal{L}(M)}(x) = \text{true}$ iff $x \in \mathcal{L}(M)$. To conclude this section, we define Turing machines that compute functions with non-Boolean function values. To this end, we restrict ourselves to the class of deterministic Turing machines that accept every input.[1] By our definition of acceptance, this in particular means that they halt on every input after finitely many steps. Such a DTM computes function $f\colon \Sigma^* \to \Sigma^*$ if on input x, the (unique) leaf of the computation tree is labeled by a configuration $w\ q_{\text{final}}\ v$ such that $w.v = f(x)$ (with all occurrences of the blank symbol $\sqcup$ removed on both ends of the tape content). A function f that is computed by some machine M is called *computable*.

Alternative models

There is a variety of other models that are equally powerful as Turing machines. Formally, a model is called Turing-complete if each Turing machine can be simulated by an instance of the model and vice versa. All such models share the property of Turing-machines that any non-trivial verification problem – like *state reachability*, *configuration reachability* or *halting* – are undecidable. Sometimes, it is technically easier to show the undecidability of a problem using one of the alternative models. In this thesis, we will consider *counter machines* in several proofs. However, we delay giving a formal definition to the end of Section 6.1 when we have more of the required notation at hand.

[1] Note that it is not possible to algorithmically check whether a given DTM satisfies that condition.

3.3 Classes of computational complexity

We define the complexity classes, classes of languages that are defined by having Turing machines deciding the membership problem for these languages within certain resource bounds. The presentation loosely follows standard textbooks, e.g. [Koz06].

We are exclusively interested in the resources time and space. The *space consumption* of ATM M on input x, $\mathrm{Space}_M(x)$ is the maximum length of $|w.v|$ in any configuration $w\ q\ v$ occurring in the configuration tree of M for x (with symbol $\sqcup$ truncated exhaustively on both ends of the tape content). The *time consumption* $\mathrm{Time}_M(x)$ is the height of the computation tree, the number of transitions from the initial configuration to the most distant leaf of the tree. Both $\mathrm{Space}_M(x)$ and $\mathrm{Time}_M(x)$ may be infinite, but whenever M halts on x, they are guaranteed to be finite: Indeed, $\mathrm{Space}_M(x) \leqslant \mathrm{Time}_M(x) + |x|$ holds. In the following, we only consider deciders, which implies that $\mathrm{Space}_M(x)$ and $\mathrm{Time}_M(x)$ are finite natural numbers for any input x. Hence, Space_M and Time_M are functions with signature $\Sigma^* \to \mathbb{N}$ in this case. We generalize Space_M and Time_M to functions with signature $\mathbb{N} \to \mathbb{N}$ by considering the worst-case input: We define $\mathrm{Space}_M(n) = \max\{\mathrm{Space}_M(x) \mid x \in \Sigma^*, |x| = n\}$, similarly for $\mathrm{Time}_M(n)$.

For a function $f\colon \mathbb{N} \to \mathbb{N}$, we define $\mathrm{ASPACE}(f)$ to be the class of all languages $\mathcal{L} \subseteq \Sigma^*$ such that there is an alternating Turing machine deciding it with space consumption f, i.e. there is an ATM M (with input alphabet Σ) that is a decider with $\mathrm{Space}_M = f$ such that $\mathcal{L}(M) = \mathcal{L}$ holds. We generalize the definition to classes of languages $F \subseteq \mathbb{N} \to \mathbb{N}$ by taking the union, i.e. $\mathrm{ASPACE}(F) = \bigcup_{f \in F} \mathrm{ASPACE}(f)$. We will mostly use this definition in the case $F = \mathcal{O}(f)$ for some function f, where $\mathcal{O}(f)$ is the set of functions that are asymptotically bounded by f from above, $\mathcal{O}(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{N}, \exists n_0 \in \mathbb{N}\colon \forall n \geqslant n_0\colon g(n) \leqslant c \cdot f(n)\}$. Similarly, we define $\mathrm{ATIME}(f)$ to be the class of languages that can be decided with a Turing machine with time consumption f. We define $\mathrm{NSPACE}(f)$ and $\mathrm{NTIME}(f)$, as well as $\mathrm{DSPACE}(f)$ and $\mathrm{DTIME}(f)$ by replacing ATM in the definition of ASPACE and ATIME by NTM resp. DTM.

3.3.1 Example

The class of *decidable* or *recursive languages* REC is

$$\mathrm{REC} = \mathrm{ATIME}(\mathbb{N} \to \mathbb{N}) = \mathrm{NTIME}(\mathbb{N} \to \mathbb{N}) = \mathrm{DTIME}(\mathbb{N} \to \mathbb{N})\ ,$$

the set of languages that can be decided by (alternating / nondeterministic / deterministic) Turing machines with arbitrarily high but finite time consumption.

With the notation at hand, we can now define the *robust complexity classes*,

$$\mathrm{P} = \mathrm{PTIME} = \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}\left(\mathcal{O}\left(n^k\right)\right) \qquad \mathrm{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{NSPACE}\left(\mathcal{O}\left(n^k\right)\right)$$
$$\mathrm{EXP} = \mathrm{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}\left(2^{\mathcal{O}\left(n^k\right)}\right) \qquad \mathrm{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}\left(2^{\mathcal{O}\left(n^k\right)}\right)$$

the classes of languages decidable using polynomial time resp. polynomial space resp. exponential time resp. exponential space by deterministic Turing machines. The classes NP (or NPTIME), NPSPACE, NEXP (or NEXPTIME), and NEXPSPACE are defined by replacing DTIME and DSPACE by NTIME and NSPACE, respectively. Similarly, we can define APTIME, APSPACE, AEXPTIME and AEXPSPACE using ATIME and ASPACE.

These classes are considered *robust* since they are invariant to minor changes of the computational model, e.g. equipping Turing machines with additional tapes. The *Church-Turing thesis*, first stated in 1943 by Kleene [Kle43], claims that any decision problem that can be solved by some (physically implementable) model of computation can be solved by a deterministic Turing machine. Its *strong* version additionally states that this deterministic Turing machine is slower than the other model of computation only by a polynomial factor. There are two problems with that thesis: The first are the recent advantages in quantum computing that have lead to an ongoing discussion of whether the strong version of the Church-Turing thesis is true [BV93; KML07]. The second is the problem of defining precisely what it means to be *computable* by a physically implementable device. Different versions of the thesis use different wordings to express this, but they share the problem that without a much deeper understanding of physics, it is impossible to prove any such thesis. Luckily, we can use that effective polynomial-time equivalence has been shown for many random access machines [CR72], a model that is very close to sequential imperative programs. Hence, to show that a problem is in a certain complexity class, it will be sufficient to provide pseudocode for an algorithm solving the problem within that time. We then know that it is possible to construct a Turing machine solving the problem that is slower only by a polynomial factor.

Relations between the complexity classes

Let us put the classes defined by alternating Turing machines aside for a bit. The following relationships between the other complexity classes are known:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP} \subseteq \mathrm{EXPSPACE} = \mathrm{NEXPSPACE}\,.$$

We give a brief explanation for the inclusions: We have already observed that $\mathrm{Space}_M(x) \leqslant \mathrm{Time}_M(x)+|x|$; thus, $\mathrm{P} \subseteq \mathrm{PSPACE}$, $\mathrm{NP} \subseteq \mathrm{NPSPACE}$ and so on. Savitch's result [Sav70] states that nondeterministic machines can be converted to deterministic machines while squaring the space consumption, yielding PSPACE = NPSPACE and EXPSPACE = NEXPSPACE. This also proves NP $\subseteq$ PSPACE and NEXP $\subseteq$ NEXPSPACE, but these inclusions could actually be shown with a simpler proof that does not invoke Savitch's theorem. If a machine defines a

language in PSPACE, its computation tree can only contain configurations of bounded length. The number of such configurations is at most exponential in the input, so it is possible to deterministically simulate the machine with a time consumption that is exponentially higher. We obtain the final relation PSPACE $\subseteq$ EXP. One could state more general versions of these results (in which some of them have additional prerequisites, e.g. time-constructability), but doing so is beyond the scope of the thesis.

For all inclusions among neighbors in the above chain, it is unknown whether they are strict.[1] It is known that having exponentially more space or time strictly increases the computational power, i.e. P $\subsetneq$ EXP, PSPACE $\subsetneq$ EXPSPACE and so on. The relationship between time and space and between determinism and nondeterminism in the case of time, however, is not well understood. In fact, answering these questions is universally considered to be the most important open problem of computer science. This in particular holds true for P $\subseteq$ NP, often phrased as P $\stackrel{?}{=}$ NP. Without going into any details, we briefly mention that P is considered the class of *tractable problems*, problems that can be solved efficiently, while NP is the class of problems where a proposed solution (a *certificate*) can be verified efficiently. Answering P $\stackrel{?}{=}$ NP thus not only has practical implications (since many important practical problems from NP, e.g. optimization problems, are not known to be in P), but also philosophical ones [For13].

Reductions and hardness

At the moment, it seems out of reach to show absolute hardness, i.e. proving that some computational problems from NP or PSPACE are not in P. Instead, one defines a notion of relative hardness: A problem is hard for a class if an efficient solution for that problem leads to an efficient solution to all problems in the class.

To formalize the concept, we introduce *polytime reductions*. A polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ is a function that is computed by a DTM running in time $\mathcal{O}(n^k)$ for some constant k. A *polynomial-time* or *polytime reduction* from some language $\mathcal{L} \subseteq \Sigma^*$ to $\mathcal{L}' \subseteq \Sigma'^*$ is a polytime-computable function $f\colon \Sigma''^* \to \Sigma''^*$ (with $\Sigma, \Sigma' \subseteq \Sigma''$) such that $x \in \mathcal{L}$ if and only if $f(x) \in \mathcal{L}'$. We write $\mathcal{L} \leqslant_{\text{poly}} \mathcal{L}'$ if such a reduction exists. Intuitively, this means $\mathcal{L}$ is *easier* to decide than $\mathcal{L}'$: If $\mathcal{L}'$ can be decided in time $t\colon \mathbb{N} \to \mathbb{N}$, then $\mathcal{L}$ can be decided in time $t \circ \mathcal{O}(n^k)$ by first applying the reduction and then using the decider for $\mathcal{L}'$. Formally, for any of the classes $\mathcal{C}$ introduced above, the following *robustness against polytime reductions holds*: If $\mathcal{L} \in \mathcal{C}$ and $\mathcal{L}' \leqslant_{\text{poly}} \mathcal{L}$, then $\mathcal{L}' \in \mathcal{C}$.

With the notation at hand, we can formalize the concept of relative hardness. We say that a problem $\mathcal{L}$ is *hard* for a class $\mathcal{C}$ if any problem from $\mathcal{C}$ can be reduced to it, i.e. for all $\mathcal{L}' \in \mathcal{C}$, $\mathcal{L}' \leqslant_{\text{poly}} \mathcal{L}$ holds. A problem is *complete* for a class $\mathcal{C}$ if it satisfies both *membership* ($\mathcal{L} \in \mathcal{C}$) and *hardness* ($\mathcal{L}$ is hard for $\mathcal{C}$).

[1] It seems that a majority of computer scientists believes that the inclusions are strict [Ros12], but theory has turned out to be rather undemocratic in the past, demonstrated e.g. by the surprising equality of the complexity classes NL and coNL, proven independently by Immerman [Imm88] and Szelepcsényi [Sze87].

To show that a problem $\mathcal{L}$ is hard for a class, it is sufficient to show that $\mathcal{L}' \leqslant_{\text{poly}} \mathcal{L}$, where $\mathcal{L}'$ is a problem for which we already have proven hardness. This is because $\leqslant_{\text{poly}}$ can be easily shown to be transitive (the composition of polynomials is again a polynomial). This observation greatly simplifies proofs of hardness, and motivates the study of *canonical* hard problems for each class.

Alternation versus determinism

In contrast to the relationship of time and space resp. nondeterminism, the relationship between the classes defined by alternating Turing machines and the classes defined by deterministic Turing machines is well understood: The paper that introduced ATMs contains the proof of the following equalities [CKS81]:

$$\text{APTIME} = \text{PSPACE}, \text{APSPACE} = \text{EXP}, \text{AEXPTIME} = \text{EXPSPACE} .$$

For more general statements of the equalities, we refer to the literature.

Fast-growing functions

So far, we have restricted ourselves to classes defined by at most exponential resource consumption. We now define fast-growing functions and their associated complexity classes.

For $m \in \mathbb{N}$, the *m-fold exponential function* $exp_m \colon \mathbb{N} \to \mathbb{N}$ is inductively defined as follows:

$$exp_0(n) = n \qquad\qquad exp_{m+1}(n) = 2^{exp_m(n)} .$$

For example, $exp_2(n) = 2^{2^n}$. For each $m \in \mathbb{N}$, we may define the associated complexity classes, e.g. $\text{mEXPSPACE} = \bigcup_{k\in\mathbb{N}} \text{DSPACE}(exp_m(\mathcal{O}(n^k)))$, similar for mEXP, mNEXP and so on.

The equalities and inclusions among complexity classes mentioned above carry over, for example 1AEXPSPACE = AEXPSPACE = 2EXP.

We define ELEMENTARY to be the class of all problems that can be solved with some m-fold exponential bounded resource consumption, e.g. $\text{ELEMENTARY} = \bigcup_{m\in\mathbb{N}} \text{mEXP}$. Using the results mentioned above, the definition of ELEMENTARY remains the same, no matter whether we define it using time or space. Note that while m is allowed to be arbitrarily high, it is independent of the input size. The class PR includes all languages that can be computed with resources bounded by a *primitive recursive function* (where again it does not matter whether we bound space or time). The precise definition of primitive recursion is irrelevant here; it can be found e.g. in [Koz97]. The class PR includes for example $\text{TOWER} = \text{DTIME}(\mathcal{O}(\text{Tower}))$, where $\text{Tower} \colon \mathbb{N} \to \mathbb{N}$ is the non-elementary function defined by

$$\text{Tower}(n) = exp_n(n) .$$

Note that the height of the tower of exponentials *is* depending on the input size.

Beyond PR, there are the classes defined by non-primitive recursive functions, functions that grow faster than any primitive recursive function. The most well-known example is the (two-parameter version of the) *Ackermann* function [Ack28; Pét35], inductively defined by

$$\begin{aligned} \mathrm{Acker}(0, m) &= m + 1 \\ \mathrm{Acker}(n + 1, 0) &= \mathrm{Acker}(n, 1) \\ \mathrm{Acker}(n + 1, m + 1) &= \mathrm{Acker}(n, \mathrm{Acker}(n + 1, m)) \,. \end{aligned}$$

Roughly spoken, the first parameter of the Ackermann function determines which operation should be executed on the second parameter. We have $\mathrm{Acker}(2, m) \approx 2m$, $\mathrm{Acker}(3, m) \approx 2^m$, and $\mathrm{Acker}(4, m)$ is approximately equal to a tower of exponentials $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ of height m.

To get a one-parameter version, we may define $A(n) = \mathrm{Acker}(n, n)$. The complexity class ACKERMANN can be defined by $\mathrm{DTIME}(\mathcal{O}(A(n)))$.

Remark

a) One can define hierarchies of fast-growing functions (and the associated complexity classes), see e.g. [Sch16]. Within these hierarchies, Tower is a mildly fast-growing non-elementary function, and Acker is a mildly fast-growing non-primitive recursive function. However, for practical usage, such considerations do not matter: An algorithm whose worst-case resource consumption is described by the Ackermann function is useless on inputs that trigger this behavior.

b) Complexity classes defined by fast-growing functions allow for more freedom in the type of reductions: For example, to show that a problem is in PR, it is sufficient to reduce it to a problem already known to be in the class with a reduction that runs in time exp_m for some constant m.

c) There is also interest in complexity classes that are defined by functions with sublinear growth, e.g. L and NL, the problems solvable by deterministic resp. nondeterministic Turing machines within logarithmic space consumption. To give a precise definition of the classes, we would need to modify our definition of ATMs: With our definition, any ATM needs at least $|x|$ space on input x. Polynomial-time reductions are too coarse to handle these classes, instead, one needs to consider logarithmic-space reductions, the definition of which would suffer from the same problem.

   Since the classes L and NL do not play an important role in this thesis – we are mostly concerned with complexity classes that are much larger – we forgo giving the formal definitions. Note that even for larger classes like P, it would actually make sense to define logarithmic-space reductions: Since $\mathrm{L} \subseteq \mathrm{P}$ (and the inclusion is believed to be strict), proving that a problem is hard for a class with respect to logarithmic-space reductions is potentially a stronger

statement than showing that it is hard for the class with respect to polynomial-time reductions. We believe that for our purposes, showing hardness with respect to polytime reductions is fine-grained enough.

A generic hard problem

We present a general way to obtain for each complexity class a complete problem, an approach that has for example been used by Meyer and Stockmeyer [MS72; SM73] and Walukiewicz [Wal01]. For many of the *small* complexity classes, like P, NP, and PSPACE, there is an abundance of well-known problems that are complete for each class. The same is not true for the larger classes.[1] Luckily, each class has a generic problem that is complete for the class, namely the *acceptance problem* for ATMs whose properties coincide with the properties defining the class. The acceptance problem (without any restrictions) is the following.

Acceptance problem
Given: ATM M over Σ, input $x \in \Sigma^*$.
Question: $x \in \mathcal{L}(M)$?

It is easy to show that the acceptance problem is a variant of the undecidable halting problem, which we have discussed briefly in the previous section. If we restrict the machine M to only uses bounded resources, however, it can be solved by simulating the machine. Consider for example the following variant for AEXPSPACE.

Acceptance problem for AEXPSPACE **(Promise version)**
Given: ATM M over Σ that runs in $\mathrm{Space}_M(2^{\mathcal{O}(n^k)})$ for some $k \in \mathbb{N}$, input $x \in \Sigma^*$.
Question: $x \in \mathcal{L}(M)$?

Intuitively, the problem should obviously be AEXPSPACE-complete. However, the problem is not a proper decision problem, but a so-called *promise problem*: We have the guarantee that our input is the encoding of a machine that runs in exponential space. Another minor annoyance is the dependency on the constant k. We solve both issues and consider the following decision problem instead.

Acceptance problem for AEXPSPACE
Given: ATM M over Σ, input $x \in \Sigma^*$.
Question: Is the computation tree for x accepting and uses space at most $2^{|x|}$?

[1] Since the classes are much larger, this is likely not because of an inherent property of these classes, but because humans rarely try to solve such hard problems in practice.

We see that we have replaced the promise that M runs using exponential space for *all inputs* by checking that M uses exponential space for the concrete input. We also replaced $2^{\mathcal{O}(n^k)}$ by 2^n. Now, we can prove that this problem indeed is AEXPSPACE-complete.

3.3.2 Lemma
The acceptance problem for AEXPSPACE is AEXPSPACE-complete.

Proof sketch:
We first show membership. We construct an ATM that simulates the given machine M on the given input x. Should it encounter a configuration with space consumption larger than $2^{|x|}$, it rejects. The overhead caused by the simulation and by checking the space consumption is at most polynomial in $2^{|x|}$, so the resulting machine proves that the acceptance problem is in AEXPSPACE.

We argue that the problem is AEXPSPACE-hard by following the definition: Let $\mathcal{L}$ be any problem from AEXPSPACE. There is an ATM M with $\mathcal{L}(M) = \mathcal{L}$ and constants k, c such that M uses space at most $2^{(c \cdot n)^k}$ on an input of size n. Let Σ be the input alphabet of M, and let $\# \notin \Sigma$ be a fresh symbol. We define a new machine M' over $\Sigma \cup \{\#\}$ that behaves as M does for the symbols in Σ and that treats # as M treats $\sqcup$. In particular, M' will accept input $x\#^\ell$ (with $x \in \Sigma^*$) for any $\ell \in \mathbb{N}$ if and only if M accepts x. We define our reduction to take input (M, x) and yield output $(M', x\#^{(c\cdot|x|)^k - |x|})$. We have that M' accepts $x\#^{(c\cdot|x|)^k-|x|}$ if and only if M accepts x. Furthermore, M' runs on $x\#^{(c\cdot|x|)^k-|x|}$ with space consumption at most $2^{|x\#^{(c\cdot|x|)^k-|x|}|} = 2^{(c\cdot|x|)^k}$, since M runs on x with space consumption at most $2^{(c\cdot|x|)^k}$. ■

Remark
Making the proof sketch above precise requires techniques from complexity theory that are beyond the scope of this thesis, e.g. encoding ATMs as strings, manipulating the encodings, efficiently simulating given ATMs, and the *padding* technique to go from 2^{n^k} to 2^n, see e.g. [Koz06].

We now have a problem that is complete for AEXPSPACE = 2EXP. We will reduce from it in Section 17.5 to show that context-free inclusion games are 2EXP-hard. A similar construction works for all the other complexity classes that are robust against polynomial-time reductions.

3.4 Formulas in propositional logic and Presburger arithmetic

We introduce formulas in propositional logic and in (existential) Presburger arithmetic. Both concepts are used in some complexity proofs throughout the thesis. Positive Boolean formulas additionally play an important role as representations for the semantics of games. Both topics are standard, see e.g. [End72].

Propositional logic

For a set $\mathcal{V}$ of variables, the formulas in *propositional logic* over $\mathcal{V}$, also called *Boolean formulas* $\mathrm{BF}(\mathcal{V})$, are defined by the following grammar:

$$F ::= \mathrm{true} \mid \mathrm{false} \mid x \mid \neg F \mid F \wedge F \mid F \vee F ,$$

where $x \in \mathcal{V}$ is a variable. Given a variable assignment $M \subseteq \mathcal{V}$ (which represents the function $\chi_M \colon \mathcal{V} \to \mathbb{B}$ that evaluates a variable x to true iff $x \in M$), we obtain a map $-(M) \colon \mathrm{BF}(\mathcal{V}) \to \mathbb{B}$ that assigns each formula F the truth value $F(M)$ under evaluation at M. Evaluation is defined inductively in the expected way. Vice versa, each formula defines a map $F(-) \colon \mathcal{P}(\mathcal{V}) \to \mathbb{B}$ that evaluates it at the given variable assignment.

The most important algorithmic problem in this area is satisfiability, given a formula, is there a variable assignment such that the formula evaluates to true. The well-known Cook-Levin theorem, see e.g. [Koz06], states that this problem is NP-complete, even if we restrict the formulas to be in conjunctive normal form.

A *literal* is a variable x or a negated variable $\neg x$, a *clause* is a disjunction of literals $K = L_1 \vee \ldots \vee L_n$, and a formula in *conjunctive normal form (CNF)* is a conjunction of clauses, $F = K_1 \wedge \ldots \wedge K_m$. For any formula $F \in \mathrm{BF}(\mathcal{V})$, there is a satisfiability-equivalent formula F' in CNF that can be computed in polynomial time using the Tseytin transformation [Tse68]. The problem SAT is the task of checking whether a formula in CNF is satisfiable.

Satisfiability in propositional logic (SAT)	
Given:	Formula $F \in \mathrm{BF}(\mathcal{V})$.
Question:	Is F satisfiable?

3.4.1 Theorem: Cook-Levin [Coo71; Lev73]
SAT is NP-complete.

For two formulas $F, G \in \mathrm{BF}(\mathcal{V})$ we write $F \implies G$ and say that F implies G if for any variable assignment $M \subseteq \mathcal{V}$ such that $F(M) = \mathrm{true}$, $G(M) = \mathrm{true}$ holds.

A formula is *positive* if it does not contain negation. The set of *positive Boolean formulas* pBF($\mathcal{V}$) consists of all positive formulas, including the constants true and false.

Sometimes, we may assume that true and false do not occur as operands of $\vee$ and $\wedge$ by using the syntactic elimination rules

$$F \vee \text{false} = \text{false} \vee F = F \qquad F \wedge \text{false} = \text{false} \wedge F = \text{false}$$
$$F \vee \text{true} = \text{true} \vee F = \text{true} \qquad F \wedge \text{true} = \text{true} \wedge F = F\,.$$

This leaves false as the unique unsatisfiable formula in pBF($\mathcal{V}$) and true as the unique tautology, i.e. a formula that is satisfied by all variable assignments. Hence, satisfiability is trivial for positive Boolean formulas. We comment on the complexity of the implication problem (given two formulas, does one imply the other) in Section 17.7.

Presburger arithmetic

Presburger arithmetic [Pre31] is the first order logic over natural numbers with addition and comparison. In contrast to (Peano) arithmetic, which also considers multiplication, Presburger arithmetic enjoys numerous decidability results.

Formally, a *Presburger term* over the set of variables $\mathcal{V}$ is defined by the grammar

$$t ::= n \mid x \mid t + t\,,$$

where n ranges over the natural numbers and $x \in \mathcal{V}$. Note that while multiplication is not available, we may rewrite multiplication with constants as addition, e.g. $5 \cdot t$ can be expressed as $t+t+t+t+t$. Instead of allowing all natural numbers as terms, it would be sufficient to consider 0 and 1 as any other number can be constructed by repeated addition. While this restriction to 0 and 1 would not limit the expressiveness, the complexity results that we are going to state below hold even if the formula contains arbitrary constant numbers (which are assumed to be encoded in binary).

A *quantifier-free Presburger formula* is defined by the grammar

$$\psi ::= t \leqslant t \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi\,,$$

where t ranges over the terms as defined above. A *Presburger formula (in prenex normal form)* is of the shape $Q_1 x_1 \dots Q_k x_k \colon \psi$, where $Q_i \in \{\forall, \exists\}$ for all i, the x_i are variables and ψ is a quantifier-free formula. As usual, assuming prenex normal form does not limit the expressiveness. An *existential Presburger formula* is a Presburger formula in prenex normal form in which all quantifiers are existential, i.e. a formula of the shape $\exists x_1 \dots \exists x_k \colon \psi$ with ψ quantifier-free.

The variables that occur in a Presburger formula φ which are not bound by a preceding quantifier (i.e., referencing the notation introduced above, are not among the $x_1, \dots, x_k$), are called the *free variables* of the formula. We sometimes write $\varphi(y_1, \dots, y_m)$ to highlight that the

variables $y_1, \ldots, y_m$ are free in a formula. Such a formula can be *evaluated* at a vector $\vec{y} \in \mathbb{N}^m$ to obtain the truth value $\varphi(\vec{y})$. The evaluation semantics of Presburger formulas is as expected.

To a formula $\varphi(y_1, \ldots, y_m)$, we can associate its *set of solutions* $\mathrm{Sol}(\varphi) = \{\vec{y} \in \mathbb{N}^m \mid \varphi(\vec{y}) = \text{true}\}$ as the set of all vectors for which the formula evaluates to true. We call a set $M \subseteq \mathbb{N}^k$ that occurs as $\mathrm{Sol}(\varphi)$ for some Presburger formula *semi-linear*. The semi-linear sets enjoy a rich theory and admit several characterizations. For example, they occur as the Parikh images (sets of vectors that count the occurrences of each symbol in a word) of regular and context-free languages [Par66].

Again, satisfiability is the crucial algorithmic problem. We state the *satisfiability problem* for formulas in *existential Presburger arithmetic (EPA)*.

Satisfiability in existential Presburger arithmetic (EPA-SAT)	
Given:	An existential Presburger formula φ.
Question:	Is φ satisfiable?

3.4.2 Theorem (Scarpellini [Sca84])
EPA-SAT is NP-complete.

Surprisingly, satisfiability for the existential fragment of Presburger arithmetic is not harder than satisfiability in propositional logic. When considering expressiveness, the restriction to EPA is not problematic: For any Presburger formula, there is an existential one such that the solution sets and satisfiability coincide. However, this *quantifier elimination* leads to a blowup in the formula size. Indeed, the satisfiability problem for arbitrary Presburger formulas is in 2EXPSPACE and 2NEXP-hard. Even if the number of alternation between universal and existential quantifiers is bounded, the problem remains NEXP-hard, see [Haa14] for detailed information.

4 Labeled transition systems and finite-state automata

Contents

4.1 (Labeled) Transition systems 73
4.2 Finite-state automata 77
4.3 Descriptive complexity 79
4.4 ω-regular languages 82
4.5 The transition monoid 85

We define labeled transition systems as a general type of language-generating mechanism. We discuss the theory of finite-state automata (labeled transition systems with only finitely many configurations) over both finite and infinite words.

4.1 (Labeled) Transition systems

A *transition system* $S = (\Gamma, T)$ consists of a set of *configurations* Γ and a *transition relation* $T \subseteq \Gamma \times \Gamma$. One can say that the transition systems defined by hardware and software systems are the main object of interest in computer science.

Instead of working directly on the configuration space Γ, we want to see transition systems as language-generating devices. This additional level of indirection allow us to compare transition systems with different sets of configurations as long as they define languages over the same alphabet.

Formally, a *labeled transition system (LTS)* over alphabet Σ is of the shape $S = (\Gamma, T)$ where $T \subseteq \Gamma \times \Sigma^* \times \Gamma$ is a set of transitions labeled by finite words over Σ. Most of the time, we are interested in LTSes whose transitions are labeled by words of length exactly one (i.e. $T \subseteq \Gamma \times \Sigma \times \Gamma$), or of length at most one. In the latter case, we get $T \subseteq \Gamma \times \Sigma_\varepsilon \times \Gamma$ where $\Sigma_\varepsilon = \Sigma \uplus \{\varepsilon\}$.

In the case of labeled transitions, we usually write $c \xrightarrow{a} c'$ instead of $(c, a, c') \in T$. We extend the notations towards *computations* (also called *runs*) in the expected way and write $c \xrightarrow{w} c'$ where $w \in \Sigma^*$ is a word if there is a sequence of states $c_0, \ldots, c_{|w|}$ such that $c = c_0 \xrightarrow{w_1} c_1 \xrightarrow{w_2} \ldots \xrightarrow{w_{|w|}} c_{|w|} = c'$.

To see an LTS as a device that generates finite words, we equip it with sets of *initial* and *final* configurations, respectively. This means we consider a tuple $S = (\Gamma, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$, where (Γ, T) is

as before and $\Gamma_{\text{init}}, \Gamma_{\text{final}} \subseteq \Gamma$. A run of such an LTS is *accepting* if it starts in an initial and ends in a final configuration. The language $\mathcal{L}(S) \subseteq \Sigma^*$ of an LTS is the set of all words that occur along accepting runs,

$$\mathcal{L}(S) = \left\{ w \in \Sigma^* \;\middle|\; c_{\text{init}} \xrightarrow{w} c_{\text{final}} \text{ for some } c_{\text{init}} \in \Gamma_{\text{init}}, c_{\text{final}} \in \Gamma_{\text{final}} \right\}.$$

Remark
Seeing an LTS as a device for generating languages of infinite words is more involved, and we will not give a general definition on the level of LTSes. The acceptance condition (ending in a final state) has to be replaced, and there are different ways to do so, each with its own sets of drawbacks and advantages. A detailed discussion is given in Section 4.4.

We define some notation for LTSes with Σ-labeled transitions. For a letter a and a configuration c of some LTS S, we denote by

$$\text{post}_S(a, c) = \left\{ c' \in \Gamma \;\middle|\; c \xrightarrow{a} c' \right\}, \qquad \text{pre}_S(a, c) = \left\{ c' \in \Gamma \;\middle|\; c' \xrightarrow{a} c \right\}$$

the set of *(direct) a-successors* resp. *predecessors*. We extend the notation to sets of configurations by taking the union, e.g. $\text{post}_S(a, C) = \bigcup_{c \in C} \text{post}_S(a, c)$. Additionally, we extend the notation to words by induction,

$$\begin{aligned} \text{post}_S(\varepsilon, C) &= \text{pre}_S(\varepsilon, C) = C\,, \\ \text{post}_S(w.a, C) &= \text{post}_S(a, \text{post}_S(w, C))\,, \qquad \text{pre}_S(w.a, C) = \text{pre}_S(w, \text{pre}_S(a, C))\,. \end{aligned}$$

We will be mostly interested in the configurations reached from the initial configurations and, similarly, in the configurations from which a final configuration can be reached. We define

$$\text{reach}_S(w) = \text{post}_S(w, \Gamma_{\text{init}})\,, \qquad \text{reach}_S^{-1}(w) = \text{pre}_S(w, \Gamma_{\text{final}})\,.$$

We generalize reach, reach^{-1}, post, and pre to set of words by taking the union. A particularly interesting case is the case of all words, which we simply denote by

$$\text{reach}_S = \text{reach}_S(\Sigma^*) = \bigcup_{w \in \Sigma^*} \text{reach}_S(w)\,, \qquad \text{reach}_S^{-1} = \text{reach}_S^{-1}(\Sigma^*) = \bigcup_{w \in \Sigma^*} \text{reach}_S^{-1}(w)\,,$$

the set of all configurations reachable from the initial configurations respectively the set of all configurations from which a final configuration is reachable. It is clear from the definition of the language of an LTS that we have $\mathcal{L}(S) \neq \varnothing$ if and only if one (or all) of the following equivalent conditions is true:

$$\text{reach}_S \cap \Gamma_{\text{final}} \neq \varnothing\,, \qquad \text{reach}_S^{-1} \cap \Gamma_{\text{init}} \neq \varnothing\,, \qquad \text{reach}_S \cap \text{reach}_S^{-1} \neq \varnothing\,.$$

If the system S we are considering is clear from the context, we omit the subscript and simply write e.g. $\mathrm{post}(a, c)$.

The transition relation of an LTS is *finitely branching* if for each configuration c and each symbol $a \in \Sigma$, there are at most finitely many successors c' such that $c \xrightarrow{a} c'$. It is called *unique* if there is at most one successor, *complete* if there is at least one successor, and *deterministic* if there is exactly one successor. If the transition relation T of an LTS is deterministic, we may see it as a function $T\colon \Gamma \times \Sigma \to \Gamma$ such that $T(c, a)$ is the unique c' with $c \xrightarrow{a} c'$.

An LTS is *finitely branching* if its transition relation is and it additionally has only finitely many initial states. It is *deterministic* if its transition relation is and it has a unique initial state.

The synchronized product

We present a standard construction that, given two LTSes, yields an LTS whose language is the intersection of the languages. Intuitively, the product of the LTSes tracks tuples of configurations, simulating a run of one LTS per component. Transitions not labeled by ε need to be applied synchronously to both components. Formally, the construction is as follows: Let $S = (\Gamma, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ and $S' = (\Gamma', T', \Gamma'_{\text{init}}, \Gamma'_{\text{final}})$ be two LTSes with transition labeled by Σ_ε. Their *synchronized product* is $S \times S' = (\Gamma \times \Gamma', T_\times, \Gamma_{\text{init}} \times \Gamma'_{\text{init}}, \Gamma_{\text{final}} \times \Gamma'_{\text{final}})$ where $T_\times$ is defined by $(c, c') \xrightarrow{a} (d, d')$ if (1) $a = \varepsilon$, and $c \xrightarrow{\varepsilon} d$ in T and $c' = d'$, or (2) $a = \varepsilon$, and $c' \xrightarrow{\varepsilon} d'$ in T' and $c = d$, or (3) $a \in \Sigma$, and $c \xrightarrow{a} d$ in T and $c' \xrightarrow{a} d'$ in T'.

It is straightforward to show $\mathcal{L}(S \times S') = \mathcal{L}(S) \cap \mathcal{L}(S')$: For each word w, a run of the product LTS gives rise to a run in each LTS. Similarly, two runs on the same word, one for each system, can be merged into a single run of the product system.

Automata

LTSes are a very general concept since we have not imposed any restriction on the set Γ; in particular, it may be infinite. The drawback is that we cannot treat them algorithmically in this full generality. To solve the problem, we will focus on *automata* in the following. Automata are LTSes of a special shape: Their set of configurations can be written as $\Gamma = Q \times M$ such that Q is a *small* (typically finite) set of *control states*, and M is a potentially large (e.g. infinite) set of *memory values*. The transitions T are allowed to depend arbitrarily on Q, but only *locally* on M. The latter means for example that even if M is infinite (which necessarily means that its elements have unbounded size), the transitions should only depend on a bounded part of the current value $m \in M$. More formally, one can require that there is a finite set of *transition rules* $\to$ that specify whether a transition $(q, m) \xrightarrow{a} (q', m') \in T$ exists. Hence, an automaton can usually be represented by some *finite syntax* (the set Q, a representation of the shape of the elements of M, and $\to$), although it gives rise to a potentially infinite *semantics*, the transition system $(Q \times M, T)$.

For example, sequential imperative programs can be seen as automata: The control states are the lines of code while the memory stores the assignment of the variables. The transitions from one line to another only depend on the memory as specified e.g. by a conditional in the source

code. With this view, the source code is the syntax of the automaton, the transition system that describes its runtime behavior is the semantics. The labeling that we consider for such a system depends on what property we want to analyze. For example, it may be the atomic commands that are executed by the program.

The above definition of automata is imprecise and fuzzy, but it captures the fundamental idea behind automata theory: An infinite semantics is represented by a finite description that can serve as input for algorithms. However, requiring a finite representation of the system is not sufficient to get decidability results. For example, Turing machines also admit a finite description, but by Rice's theorem [Ric53] no non-trivial properties of the languages they define are decidable. At the current state, there is no general theory that explains which types of systems admit which decidability results. The model of valence systems over graph monoids [Zet15b; Cha07] that we present in Chapter 19 can be seen as a step in that direction, but this model is not powerful enough to capture all types of memories, and it has not (yet) gained wide-spread usage.

This leaves us with having to define various classes of automata for which certain properties are decidable. In the following, we define various types of *state-based systems* that are automata in the above sense and for which some properties, e.g. membership in their language, are decidable. Finite automata, Petri nets, and well-structured transition systems fall into this category.

We will also consider *rewriting-based systems* like context-free grammars and higher-order recursion schemes. These systems can also be seen as automata as we will explain in Section 5.2.

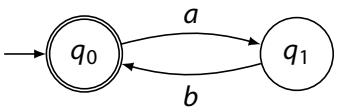


Figure 4.2.a: An NFA with language $(ab)^*$.

4.2 Finite-state automata

We introduce finite-state automata, which are simply LTSes with finitely many configurations. However, one usually uses modified notation in this case. We also follow the convention of having a single initial state. The material is standard and can be found e.g. in [KN01; Koz97].

Formally, a *(nondeterministic) finite(-state) automaton (NFA)* over alphabet Σ is a tuple $A = (Q, \delta, q_{\text{init}}, Q_{\text{final}})$, where Q is a finite set of *control states*, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_{\text{init}} \in Q$ is the unique initial state, and $Q_{\text{final}} \subseteq Q$ is a set of final states.

Apart from the different naming of the components, the notation and definitions for LTSes carry over. For example, we call an NFA a *deterministic finite automaton (DFA)* if it is deterministic.

The size of an automaton A is $|A| = |Q| + |\delta|$. Since $|\delta| \leqslant |Q|^2 \cdot |\Sigma|$, to show that an automaton is e.g. of size polynomial in n, it is sufficient to show that $|Q|$ is polynomial in n (assuming that the alphabet is fixed).

A language $\mathcal{L} \subseteq \Sigma^*$ is called *regular* if there is a finite automaton A over Σ with $\mathcal{L}(A) = \mathcal{L}$. The class of all regular languages is denoted by REG.

To decide the word problem of a regular language, we do not need the capabilities of an unrestricted Turing machine, a finite automaton is sufficient.

4.2.1 Example

The language $(ab)^* \subseteq \{a, b\}^*$ is regular: It is the language of the NFA given by Figure 4.2.a.

The class of regular languages can also be defined by *regular expressions*, finite expressions that may contain the empty language $\varnothing$, the singleton languages $\{\varepsilon\}$, and $\{a\}$ for each letter a of the underlying alphabet and unions, concatenations, and the Kleene star. From this, we immediately obtain that every language consisting of finitely many words is regular, and if $\mathcal{L}, \mathcal{L}' \subseteq \Sigma^*$ are regular, then so are their union $\mathcal{L} \cup \mathcal{L}'$, their concatenation, $\mathcal{L}.\mathcal{L}'$, and the Kleene star $\mathcal{L}^*$. It can also be shown that the intersection $\mathcal{L} \cap \mathcal{L}'$ of two regular languages and that the complement $\overline{\mathcal{L}}$ of a regular language are regular. All these closure properties are effective: Given automata for $\mathcal{L}$ and $\mathcal{L}'$, we can construct an automaton for each of the aforementioned languages. In the case of the intersection, we have explicitly stated the construction in Section 4.1.

In addition to the definitions based on automata or closure properties, the regular languages can also be characterized as the solution to closed formulas in the logic S1S, (weak) monadic second order logic with one successor, see e.g. [KN01]. Discussing the details is beyond the scope of this thesis.

4.3 Descriptive complexity

We introduce some notions regarding the descriptive complexity of regular languages. Unlike computational complexity (see Section 3.3) which studies the amount of resources needed to decide membership in a language, descriptive complexity studies the size of descriptions of languages. However, many interconnections between the two areas exist in the form of results that show that languages for which the membership problem has a certain complexity admit descriptions of a certain shape and size and vice versa [Imm98]. Descriptive complexity theory considers several types of objects to represent languages, including logical formulas and circuits. In this thesis, we are exclusively interested in the descriptive complexity of regular languages, for which we use finite automata as descriptions.

The well-known fact that DFAs and NFAs are equally expressive results in two measures of complexity for regular languages. We will introduce and use both in the following. For a regular language $\mathcal{L}$, its *state complexity* is the minimum number of states that an NFA A with $\mathcal{L}(A) = \mathcal{L}$ has. Its *index* is the minimum number of states that a DFA A with $\mathcal{L}(A) = \mathcal{L}$ has. The latter notion comes from the fact that this number is indeed the index, the number of equivalence classes, of the Nerode right-congruence [Ner58]. It is well-known that every regular language has a unique minimum DFA defining it (up to isomorphism), see e.g. [HU79]; the same is not true for NFAs [JR93].

There are languages for which state complexity and index coincide.

4.3.1 Example

For each $k \in \mathbb{N}$, the language $\{a^k\} \subseteq a^*$ has state complexity and index $k + 1$. Obviously, a DFA with states $q_0, q_1, \ldots, q_k$ and transitions $q_i \xrightarrow{a} q_{i+1}$ for all i accepts the language with the desired number of states. Any smaller automaton would necessarily see a state repetition while processing a^k. This loop could be repeated to accept a word $a^{k+\ell}$ for $\ell > 0$, a contradiction to the assumption that a^k is the only word in the language.

The same is true for the languages

$$\mathcal{L}_{\leqslant k} = \left\{ a^\ell \;\middle|\; \ell \leqslant k \right\}, \qquad \mathcal{L}_{\geqslant k} = \left\{ a^\ell \;\middle|\; \ell \geqslant k \right\}.$$

In general, the index is at most exponentially higher than the state complexity. The *Rabin-Scott powerset construction* [RS59] turns an NFA with states Q into a language-equivalent DFA with states $Q^{\text{det}} \subseteq \mathcal{P}(Q)$, and $|\mathcal{P}(Q)| = 2^{|Q|}$. This is in fact also used when proving that regular languages are closed under complement: Given an NFA for a language, we transform it into a DFA, which can be complemented by making the final states non-final and vice versa. In particular, a regular language and its complement always have the same index. This method of complementation is not sound for NFAs, and we will see in Section 8.2 a language for which the state complexity of its complement is exponentially larger than its state complexity.

In the following, we consider an example for a family of languages in which the worst-case-behavior of an exponential gap between state complexity and index occurs.

4.3.2 Example

For each k, the language $\{w \in \mathbb{B}^{\geq k} \mid \text{the } k\text{-last letter of } w \text{ is } 0\}$ has state complexity $k + 1$ but index 2^k. It is not hard to construct automata with the desired number of states: An NFA can guess when the k-last letter occurs, verify that it is 0 and then verify that it was indeed the k-last letter. A DFA can store the k-last letters so that when the end of the word occurs, it can check that the k-last letter is 0 as desired. Both constructions are optimal.

The example is mentioned without proof in [Koz97]. We give a proof in Section 14.2 when we use the example to establish a lower bound on the descriptive complexity of separators.

Descriptive complexity for families of languages

As in the Examples 4.3.1 and 4.3.2, we are usually not interested in the state complexity of a single language, but rather in the state complexity of a family of languages. In the case where we have a family of languages $(\mathcal{L}_k)_{k\in\mathbb{N}}$ consisting of one language per natural number, the meaning of a statement like "the state complexity is exponential" is obvious. It could be formalized using the notation that we have introduced in Section 3.3: The state complexity of a family is exponential if there is a constant ℓ such that there is a function contained in $2^{\mathcal{O}(n^\ell)}$ that maps each number n to an upper bound for the state complexity of $\mathcal{L}_n$. When we say that the state complexity is exponential and that this is optimal, we mean that there is no such function that can be bounded by a polynomial.

Sometimes, we will consider families of languages that are not indexed by natural numbers, but by more complex objects. For example, when we make a statement like "the languages of NFAs have polynomial state complexity", we are considering the family of language $\{\mathcal{L}(A) \mid A \text{ NFA}\}$. In this family of languages, each language $\mathcal{L}(A)$ is indexed by the NFA A that generates it. By the above statement, we mean that if we measure the size $|A|$ as defined previously, we obtain that the state complexity of each language $\mathcal{L}(A)$ is polynomial in $|A|$. Luckily, we will only need to consider examples where the index set and the size assignment is clear from the context.

The *index* of a family of languages – not to be confused with the indexing of languages mentioned above – can be defined similarly to the state complexity.

ε-Transitions

We also consider NFAs with internal transitions, i.e. NFAs where we allow labels from $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. Intuitively, transitions labeled by ε can be taken without generating a letter. Sometimes it is convenient to allow such transitions to be able to give easier and more intuitive constructions.

It is well-known that given an automaton with ε-transitions, it is possible to compute an NFA without such transitions that has the same set of states. Hence, we may use NFAs and NFAs with internal transitions interchangeably when talking about the state complexity of languages.

Multiple initial states

Similar to the case of ε-transitions, constructions may be simpler if we allow multiple initial states. Such an NFA can be converted to an NFA with a unique initial state as follows: We insert a fresh initial state, and ε-labeled transitions from the fresh state to all former initial states. As mentioned above, this NFA with ε-transitions can be converted to a normal NFA. Hence, the existence of an NFA with multiple initial states and k control states for a language proves that the language has state complexity at most $k + 1$.

In this thesis, we will be exclusively interested in whether the state complexity of a family of languages is polynomial, exponential, and so on. These classes of functions are closed under the operation of adding 1, so we may freely use multiple initial states.

Note that both constructions – ε-transition and multiple initial states – can be seen as a form of nondeterminism. For example, the language $\mathcal{L}_k$ from Example 4.3.2 can easily be expressed using automata with $k + 1$ states that are deterministic but for the existence of ε-transitions or multiple initial states. Hence, the state complexity is only preserved under these constructions in the case of NFAs.

4.4 ω-regular languages

We consider variants of finite-state automata that define ω-languages. Obviously, the acceptance condition of ending in a final state needs to be replaced. Simply requiring that a final state occurs would limit us to so-called reachability conditions and the associated *safety properties* of programs. Reachability conditions are not expressive enough to model so-called *liveness properties* [GTW02] like fair scheduling.

Büchi automata

The easiest way of making an NFA accept an ω-language is to require that final states occur repeatedly. On the syntactic level, *nondeterministic Büchi automata (NBAs)* are defined just like NFAs. Their semantics, however, is defined in terms of infinite words. A run on an infinite word $w \in \Sigma^\omega$ from state $q \in Q$ is an infinite sequence of states and transitions

$$q = q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \cdots.$$

A run of A on w is *accepting* if it starts in the initial state q_{init} and in its sequence of states, final states from Q_{final} occur infinitely often. As Q_{final} is a finite set, this is equivalent to requiring that some state from Q_{final} occurs infinitely often. The language $\mathcal{L}^\omega(A)$ of an NBA A is the set of all infinite words on which A has an accepting run. Note that we use the notation $\mathcal{L}^\omega(A)$ to distinguish the language of A seen as Büchi automaton from $\mathcal{L}(A)$, the language of finite words of A seen as NFA. This should not be confused with the omega-iteration of a language.

An ω-language $\mathcal{L} \subseteq \Sigma^\omega$ is called *ω-regular* if it is the language of an NBA A. The class of ω-regular languages enjoys many of the effective closure properties that also hold for regular languages of finite words: They are closed under union, intersection, and complementation. Furthermore, the following characterization is well-known [Tho90]: A language $\mathcal{L} \subseteq \Sigma^\omega$ is ω-regular if and only if it can be written as

$$\mathcal{L} = \bigcup_{i \in [1,n]} U_i.(V_i)^\omega$$

where $n \in \mathbb{N}$, and for each $i \in [1, n]$, $U_i, V_i \subseteq \Sigma^*$ are regular languages of finite words.

While ω-regular languages are closed under complement, the proof of this fact is much more involved than in the case of NFAs. Recall that in the case of finite words, one uses that an NFA can be determinized and DFAs are easy to complement. The same does not work for Büchi automata: There are languages of NBAs that do not occur as languages of DBAs (deterministic Büchi automata). The language $(a \cup b)^* b^\omega$ of words that contain only finitely many as is one such example. Figure 4.4.a.i) depicts an NBA accepting that language. It nondeterministically guesses the occurrence of the last a.

i) An NBA. ii) A DPA.

Figure 4.4.a: Automata accepting the language $(a \cup b)^* b^\omega$.

Parity automata

To solve the above problem, one possibility is to consider an automaton model with a more involved acceptance condition. There are several choices, including the Rabin [Rab68], Streett [Str81], Muller [Mul63], and parity [Mos84] acceptance conditions. We only consider the parity acceptance condition throughout this thesis as it is widely used in theory and tools.

The idea is to allow the nesting of Büchi and co-Büchi conditions. The latter allow the specification of states that should not be visited infinitely often. We partition the states $Q = Q_1 \uplus Q_2 \uplus \ldots \uplus Q_m$ into several sets, where (1) there is a total order on these sets and (2) each set is marked either as final or as non-final. A run is accepting if the maximal set Q_i (wrt. the total order) such that a state from Q_i occurs infinitely often in the run is final.

We follow the usual convention to formalize the partition by a *priority assignment* $\Omega\colon Q \to \mathbb{N}$ on the states. Each set Q_i is defined as the set of states that have priority i, $Q_i = \{q \in Q \mid \Omega(q) = i\}$. The total order is as indicated by the priority, where we fix the convention that higher priorities are dominating. We consider even priorities as final and odd priorities as non-final.

Altogether, a *(nondeterministic) parity automaton (NPA)* is a tuple $A = (Q, \Sigma, \delta, q_{\text{init}}, \Omega)$, where all components but Ω are as for NFAs and NBAs and $\Omega\colon Q \to \mathbb{N}$ is a priority assignment as introduced above. A run of an NPA A on an infinite word w is accepting if it starts with the initial state q_{init} and the largest priority occurring infinitely often in the run is even. The notion of language is again the set of all infinite words that have an accepting run.

As Q is finite, so is the image of Ω, and we may speak of the *largest priority* that is used by a parity automaton. Similarly, each infinite run has a largest priority that occurs infinitely often. Büchi automata can be seen as special parity automata that only use two priorities, 1 for non-final and 2 for final states.

Remark
For finite-state systems, choosing larger priorities to be dominating and even priorities to be final is arbitrary, and in parts of the literature, other conventions are used.

In contrast to Büchi automata, parity automata are determinizable. *Deterministic parity automata (DPAs)* are defined as expected by requiring the transition relation to be determinis-

tic. For any NPA, there is a language-equivalent DPA. However, there is a potential exponential blowup in the number of states that is even worse than in the case of finite-state automata on finite words.

4.4.1 Theorem (Safra [Saf88])
For any NPA A with $|Q|$ states, there is a language-equivalent DPA with $2^{\mathcal{O}(|Q|\cdot\log|Q|)}$ states. The construction is optimal.

Figure 4.4.a.ii) depicts a deterministic parity automaton (where the names of the states indicate the priorities) that accepts the language $(a \cup b)^* b^\omega$ for which no deterministic Büchi automaton exists.

Both NPAs and DPAs can be used to obtain an alternative but equivalent definition of the ω-regular languages: A language is ω-regular iff it occurs as the language of an NBA/NPA/DPA. To convert an NBA into an NPA is trivial, determinizability is Safra's result, and to convert a DPA to an NBA, we let the NBA guess the highest-occurring priority at the beginning and verify its guess later.

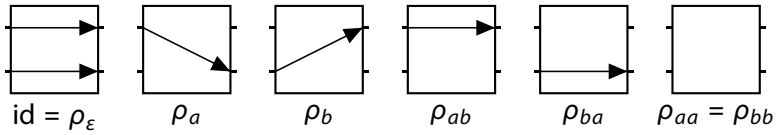


Figure 4.5.a: The boxes representing non-empty equivalence classes for the automaton for $(ab)^*$ from Example 4.2.1 resp. Figure 4.2.a.

4.5 The transition monoid

In Part V. of the thesis, it will be important to have a finite representation of the effect that a finite (but unbounded) word w has on an automaton A. We want that if two finite words w and w' have the same representation, then the behavior they induce on A is the same, independent of the context. More formally, for any word $u.w.v$ that has w as infix, we want that $u.w.v$ is in the language of A if and only if $u.w'.v$ is.

To this end, we introduce the *transition monoid* [Büc62; PP04] of an NFA A. We define an equivalence relation $\equiv_A$ on Σ^* that identifies words that cause the same state changes:

$$w \equiv_A w' \quad \text{iff} \quad \forall q, q' \in Q\colon q \xrightarrow{w} q' \iff q \xrightarrow{w'} q' \,.$$

Using $\equiv_A$, we can assign to each word w its equivalence class $[w]_A$. Let the *transition monoid* $\mathbb{M}_A$ be the set of all equivalence classes, i.e. $\mathbb{M}_A = \Sigma^* / \equiv_A$.

We can define a composition operation on $\mathbb{M}_A$ by $[w]_A \cdot [w']_A = [w.w']_A$. It is not hard to check that this operation is well-defined and associative. To see that the transition monoid is indeed a monoid, note that the equivalence class of ε is the identity with respect to this composition.

To be able to use the transition monoid algorithmically, we need a finite representation for the equivalence classes. Instead of representing equivalence classes by a representative in the classical sense, i.e. by a word that is in the class, we proceed as follows. Each equivalence class $\rho \in \mathbb{M}_A$ is uniquely determined by the state changes that the words in the class induce on A. We may see ρ as a relation on Q, i.e. as an element of $\mathcal{P}(Q \times Q)$ such that a tuple (q, q') is in ρ if and only if we have $q \xrightarrow{w} q'$ for all words (or, equivalently, one word) $w \in \rho$. However, not every element of $\mathcal{P}(Q \times Q)$ represents a (non-empty) equivalence classes: For some elements of $\mathcal{P}(Q \times Q)$, there may be no word that induces the specified state changes.

We can represent the elements of $\mathcal{P}(Q \times Q)$ graphically as shown in Figure 4.5.a. Because of the shape of the graphical representation, we usually call them *boxes*.

As mentioned above, elements $\rho, \tau \in \mathcal{P}(Q \times Q)$ are relations on Q. Hence, we may compose them using the *relational composition*

$$\rho \cdot \tau = \{(q, q'') \in Q \times Q \mid \exists q'\colon (q, q') \in \rho, (q', q'') \in \tau\} \,.$$

One can subsequently check that for elements of $\mathcal{P}(Q\times Q)$ representing non-empty equivalence classes, the composition based on representatives coincides with the relational composition. In particular, we have that $[\varepsilon]_A$ is represented by the identity relation $\mathrm{id} = \{(q,q) \mid q \in Q\}$, which is neutral with respect to relational composition.

Hence, we have that the transition monoid $(\mathbb{M}_A, \cdot, [\varepsilon]_A)$ is isomorphic to a submonoid of $(\mathcal{P}(Q\times Q), \cdot, \mathrm{id})$, namely the submonoid of boxes representing non-empty equivalence classes. This implies $|\mathbb{M}_A| \leqslant |\mathcal{P}(Q\times Q)| = 2^{|Q|^2}$. In particular, $\mathbb{M}_A$ is finite.

In the following, we usually work with boxes as a representation for elements of the transition monoid. Whenever it is important to take into account that not all boxes represent non-empty equivalence classes, we will point this out. We write $\mathcal{L}(\rho)$ to denote the language of a box (i.e. the equivalence class it represents, seen as a set of words). We define the map $\rho_{\_}\colon \Sigma^* \to \mathcal{P}(Q\times Q)$ that assigns each word w the unique box ρ_w with $w \in \mathcal{L}(\rho_w)$.

4.5.1 Lemma
For all $\rho \in \mathcal{P}(Q\times Q)$, $\mathcal{L}(\rho)$ is regular.

Proof:
For a tuple (q,q') of states, we can define a modified version $A_{q,q'}$ of A with q as the initial and q' as the unique final state such that $\mathcal{L}(A_{q,q'})$ is the set of words w with $q \xrightarrow{w} q'$. We obtain

$$\mathcal{L}(\rho) = \bigcap_{(q,q')\in\rho} \mathcal{L}\big(A_{q,q'}\big) \cap \bigcap_{(q,q')\notin\rho} \overline{\mathcal{L}\big(A_{q,q'}\big)}\,,$$

which is clearly regular. ■

Boxes have the desired property of characterizing the behavior of words in arbitrary contexts. Membership in the language of A is invariant under the replacement of infixes by equivalent words: If w, w' such that $\rho_w = \rho_{w'}$, then for any $u, v \in \Sigma^*$, we have $u.w.v \in \mathcal{L}(A)$ iff $u.w'.v \in \mathcal{L}(A)$. In particular, we can read off from ρ_w whether w is a member of the language: We have $w \in \mathcal{L}(A)$ if and only if ρ_w contains a transition from the initial to a final state, i.e. a tuple (q,q') with $q = q_{\mathrm{init}}$ and $q' \in Q_{\mathrm{final}}$. We call boxes ρ for which such a transition exists *accepting*; they satisfy $\mathcal{L}(\rho) \subseteq \mathcal{L}(A)$. Boxes ρ that do not have this property are *rejecting* and satisfy $\mathcal{L}(\rho) \cap \mathcal{L}(A) = \varnothing$. In particular, their language is contained in the complement of $\mathcal{L}(A)$.

Note that it is very simple to obtain the boxes associated to the single letters. For $a \in \Sigma$, ρ_a contains all pairs (q,q') such that there is an a-labeled transition $q \xrightarrow{a} q'$ between the states in the automaton. The set of all boxes with non-empty equivalence classes can be obtained by iteratively composing boxes, it is the least subset of $\mathcal{P}(Q\times Q)$ that contains ρ_ε, ρ_a for all $a \in \Sigma$, and that is closed under composition.

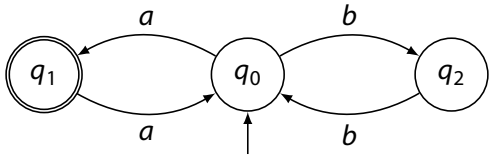


Figure 4.5.b: An NBA with $\rho_{aa} = \rho_{bb}$.

It is also noteworthy that the elements of the transition monoid of an NFA give rise to an alternative construction for determinization. The automaton $(\mathcal{P}(Q \times Q), \delta', \rho_\varepsilon, Q'_{\text{final}})$, where the deterministic transition relation δ' is defined by $\delta(\rho, a) = \rho \cdot \rho_a$ and Q'_{final} is the set of accepting boxes, is a DFA whose language is $\mathcal{L}(A)$. In a sense, computing the boxes of an automaton is an implicit determinization. However, it is not as succinct as the powerset construction: The powerset constructions yields a DFA with at most $2^{|Q|}$ states, but there are at most $2^{|Q|^2}$ boxes.

Nevertheless, boxes are useful. The box ρ_w for some NFA A contains more information than the state reached in the determinization of A when processing w from the initial states: Words that reach the same state behave the same with respect to appending suffixes, but they may show different behavior when prepending prefixes. As mentioned before, boxes allow us to characterize the behavior of a word in all contexts. We will consider an algorithm that makes use of this advantage in Chapter 17.

The transition monoid for Büchi automata

For Büchi and for parity automata, we could use an unchanged definition of the transition monoid. We obtain that if $u.w.v$ is an infinite word where w and w' are finite words that have same box, then $u.w.v$ is contained in the language of the automaton if and only if $u.w'.v$ is. This property can be lifted to finite sequences of replacements in a given word by induction. When we consider infinite sequences of replacements, however, we see that the transition monoid defined as before has undesired properties. Consider for example the NBA depicted in Figure 4.5.b whose language contains the word a^ω. For this automaton, we have $\rho_{aa} = \rho_{bb}$, but the intermediary state after reading the first a is final, while the one after reading the first b is not. The word $b^\omega = (bb)^\omega$ is obtained from $a^\omega = (aa)^\omega$ by an infinite sequence of replacements that substitute a finite infix for another one with the same box. Unlike a^ω, the word b^ω is not in the language of the automaton.

From the example, we see that we need to redefine the transition monoid for Büchi automata such that equivalent words also exhibit the same behavior with respect to visiting final states. Instead of boxes $\rho \in \mathcal{P}(Q \times Q)$,which we can also see as functions with signature $\rho\colon Q \times Q \to 2$, we need to consider boxes of the shape $\rho\colon Q \times Q \to 3$ that specify for each pair of states whether a transition exists, and if it does, whether a final state is seen during the transition.

More formally, for a Büchi automaton $A = (Q, \delta, q_{\text{init}}, Q_{\text{final}})$ we define for each letter $a \in \Sigma$ the box $\rho_a \colon Q \times Q \to 3$ as the function with

$$\rho_a(q, q') = \begin{cases} 0, & \text{if } q \not\to q' \text{ in } \delta\,, \\ 1, & \text{if } q \to q' \text{ in } \delta \text{ and } q, q' \notin Q_{\text{final}}\,, \\ 2, & \text{if } q \to q' \text{ in } \delta \text{ and } q \in Q_{\text{final}} \text{ or } q' \in Q_{\text{final}}\,. \end{cases}$$

The composition of boxes is defined so that it propagates the value 2 if possible. For two boxes $\rho, \tau \colon Q \times Q \to 3$, their composition $\rho \cdot \tau$ is the function with

$$(\rho\cdot\tau)(q, q'') = \begin{cases} 0, & \text{if } \forall q' \in Q \colon \rho(q, q') = 0 \text{ or } \tau(q', q'') = 0\,, \\ 2, & \text{if } \exists q' \in Q \colon \rho(q, q') = 2 \text{ and } \tau(q', q'') > 0, \text{ or } \rho(q, q') > 0 \text{ and } \tau(q', q'') = 2\,, \\ 1, & \text{else, i.e. if } \exists q' \in Q \colon \rho(q, q') = 1 \text{ and } \tau(q', q'') = 1 \\ & \quad \text{and } \forall q' \in Q \colon \rho(q, q') < 2 \text{ and } \tau(q', q'') < 2\,. \end{cases}$$

Note that this operation is associative.

For a non-empty word $w_1 \dots w_n$, we define $\rho_{w_1 \dots w_n} = \rho_{w_1} \cdots \rho_{w_n}$. Informally, we have $\rho_w(q, q') = 0$ if the automaton has no path from q to q' along w, we have $\rho_w(q, q') = 2$ if it has a path from q to q' in which at least one state is final, and we have $\rho_w(q, q') = 1$ if it has a path from q to q', but none in which a final state occurs.

It remains to associate a box to the empty word. It might seem sensible to define $\rho_\varepsilon(q, q') = 1$ if $q = q'$, $\rho_\varepsilon(q, q') = 0$ else. However, we should distinguish the empty word from any other word that might induce this behavior in the automaton: For any non-empty word w, we have $\{w\}^\omega = \{w^\omega\}$, but $\{\varepsilon\}^\omega = \varnothing$. To enable this distinction, we define a new element id and make it the neutral element with respect to composition, $\text{id} \cdot \rho = \rho \cdot \text{id}$ for all $\rho \in (Q \times Q \to 3) \uplus \{\text{id}\}$. We associate this new element to the empty word, $\rho_\varepsilon = \text{id}$, and the empty word is the only word whose box is id.

Altogether, we obtain that the transition monoid of a Büchi automaton B is $(\mathbb{M}_B^{\text{NBA}}, \cdot)$, where $\mathbb{M}_B^{\text{NBA}} = (Q \times Q \to 3) \uplus \{\text{id}\}$ and the operation $\cdot$ is defined by the rules for id and the composition of boxes. The transition monoid of a Büchi automaton satisfies the same properties as the transition monoid of a finite automaton. In particular, $\mathcal{L}(\rho) = \{w \in \Sigma^* \mid \rho_w = \rho\}$ is a regular language of finite words. Additionally, they have properties that allow us to use them as a representation for infinite words. We will discuss these properties in Section 16.3.

The transition monoid for Büchi automata has been used successfully in algorithms, e.g. for universality testing of Büchi automata in [FV10; ACCHH+10; ACCHH+11].

One could extend the concept of boxes to parity automata. Instead of tracking the occurrence of final states, these boxes would need to track the priorities that have been visited. Since we will not need the transition monoid for parity automata, we forgo giving the formal definition.

5 Grammar-based models

Contents

5.1 Context-free grammars 89
5.2 ω-languages of context-free grammars 95
5.3 Higher-ordered recursion schemes 103

We introduce grammars, a type of language-generating mechanisms whose underlying principle is rewriting. We consider languages of finite words generated by context-free grammars (CFGs), languages of infinite words generated by CFGs, and languages of finite words generated by higher-order recursion schemes (HORSes).

5.1 Context-free grammars

We define context-free grammars, one of the simplest types of rewriting systems. The material is standard and can be found e.g. in [Koz97]. In addition to the usual definition of the languages they generate, we present a version that is based on prefix growth.

A *context-free grammar (CFG)* over Σ is a tuple $G = (N, P, S)$ where N is the set of *nonterminal symbols*, $S \in N$ is the initial symbol, and $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of *production rules*. In this context, one calls the alphabet Σ the set of non-rewritable *terminal symbols*. Instead of $(X, \beta) \in P$, we commonly write $X \to_G \beta$, and we omit the subscript G whenever it is clear from the context.

The semantics of a context-free grammar is a transition system whose configurations are *sentential forms* from $\vartheta = (N \cup T)^*$, words consisting of both terminals and nonterminals. The transitions are given by the *derivation relation*, the rewriting of sentential forms conforming to the production rules. We have

$$\beta.X.\gamma \Rightarrow \beta.\eta.\gamma \quad \text{iff} \quad \exists \text{ production rule } X \to \eta \text{ in } P\,.$$

A sequence of derivation steps is called a *derivation process*. The language of the grammar is the set of terminal words that can be obtained by a derivation process from the initial symbol,

$$\mathcal{L}(G) = \left\{ w \in \Sigma^* \;\middle|\; S \Rightarrow^* w \right\}.$$

5.1.1 Example

The non-regular language $\{a^n b^n \mid n \in \mathbb{N}\}$ is the language of the CFG $(\{S\}, \{S \to \varepsilon, S \to aSb\}, S)$.

We additionally introduce a variant of $\Rightarrow$ called the *left-derivation relation* $\Rightarrow_\ell$, which only allows us to replace the leftmost nonterminal. Formally, it is defined by

$$w.X.\beta \Rightarrow_\ell w.\eta.\beta \quad \text{iff} \quad \exists \text{ production rule } X \to \eta$$

where $w \in \Sigma^*$ is a word over the terminals.

It is well-known that exclusively considering left-derivation processes does not restrict the language of a grammar,

$$\mathcal{L}(G) = \left\{ w \in \Sigma^* \mid S \Rightarrow_\ell^* w \right\}.$$

Indeed, derivation steps that work on different nonterminals are independent; they can be reordered so that the leftmost nonterminal is always rewritten first.

With this knowledge at hand, we can present an LTS induced by a context-free grammar that generates terminal words step-by-step (instead of producing the whole word in the final derivation step that eliminates the last nonterminal). This alternative way of defining context-free languages is arguably more complicated, but it can be generalized much easier to the case of infinite words, as we will see in the next section. We start with some preliminary observations: Each sentential form β can be written as $\beta = w.\gamma$ where $w \in \Sigma^*$ is the largest prefix of β exclusively containing terminals. Let us denote by $\text{prefix}(\beta)$ this terminal prefix. During a derivation process $S \Rightarrow_\ell^* v$, the prefix starts with being ε, then keeps growing until it equals v. More precisely, we have that if $\beta \Rightarrow_\ell \beta'$, then $\text{prefix}(\beta)$ is a prefix of $\text{prefix}(\beta')$: We have $\beta = \text{prefix}(\beta).X.\gamma$, and $\beta' = \text{prefix}(\beta).w'.\gamma'$ where $\text{prefix}(\beta') = \text{prefix}(\beta).w'$. We call this w' the *growth* of the terminal prefix for the derivation step $\beta \Rightarrow_\ell \beta'$. In some steps of a derivation process $S \Rightarrow_\ell^* v$, the growth may be the empty word ε. Every letter of v belongs to the growth of exactly one derivation step.

This allows us to define the LTS $(\vartheta, T, \{S\}, \Sigma^*)$ over Σ and conclude that its language coincides with $\mathcal{L}(G)$: The configurations are sentential forms, S is the only initial configurations, and the terminal words are final configurations. If $\beta \Rightarrow_\ell \beta'$ with prefix growth w', then T contains the transition $\beta \xrightarrow{w'} \beta'$.

This view on context-free grammars allows us to see them as automata, similar to state based models. In a sentential form $w.X.\gamma$, the leftmost nonterminal X plays the role of the control state, determining which transitions are applicable. The rest of the sentential form γ can be seen as a memory value.

5.1.2 Example
Consider the grammar for $\{a^n b^n \mid n \in \mathbb{N}\}$ from Example 5.1.1. The (unique) derivation process for the word $a^n b^n$ induces the following sequence of labeled transitions:

$$S \xrightarrow{a} aSb \xrightarrow{a} aaSbb \xrightarrow{a} \ldots \xrightarrow{a} a^n S b^n \xrightarrow{b^n} a^n b^n .$$

As we can see, we need transitions that are labeled by words strictly longer than one.

Closure properties

Straightforward constructions yield that the class of context-free languages contains the regular languages. The class is closed under union, concatenation, and Kleene star. In contrast to the regular languages, it is not closed under intersection and complementation, see e.g. [Koz97].

Alternative models

Context-free languages can also be defined by so-called *pushdown automata (PDA)*, systems that in addition to the finite control have access to an unbounded LIFO (last in, first out) stack as storage. We briefly give the formal definition, starting with the case of an unlabeled *pushdown system (PDS)*. Syntactically, a PDS with stack alphabet Λ is a finite-state LTS over some set of control states Q and transitions labeled by elements from the set $\{\varepsilon, \text{push } A, \text{pop } A \mid A \in \Lambda\}$. Such a PDS induces an infinite transition system with configurations of the shape $(q, m) \in Q \times \Lambda^*$, where q is the control state and the memory value $m \in \Lambda^*$ is the stack content. We follow the convention that the left end of m is the bottom of the stack. Consequently, the transitions of the finite LTS induce transitions on the semantic level as expected: $(q, m) \to (q', m)$ if $q \xrightarrow{\varepsilon} q'$, $(q, m) \to (q', m.A)$ if $q \xrightarrow{\text{push } A} q'$, and $(q, m.A) \to (q', m)$ if $q \xrightarrow{\text{pop } A} q'$. Note that pop-transitions are blocking in the sense that $q \xrightarrow{\text{pop } A} q'$ can only be executed if symbol A is indeed the current top-of-stack. Computations of a PDS are defined as expected. There are multiple possibilities to define the notion of accepting computations. For example, one can equip PDSes with initial and final states, and consider all computations that lead from an initial state with empty stack to a final state with arbitrary stack content as accepting.

Pushdown automata are pushdown systems in which the transitions are additionally labeled by letters from a finite (input) alphabet. These labels carry over from the finite syntax to the transitions on the semantic level. To show the correspondence between CFGs and PDAs, it is not hard to construct for a given grammar a PDA that simulates the left-derivation processes of the grammar, and, vice versa, a grammar that simulates the accepting computations of a PDA.

Recursive programs

It is well-known folklore result that context-free grammars can be used to model recursive programs. This correspondence is twofold: Context-free grammars are sufficient to describe the control flow of a sequential program in classical programming languages (that do not feature

higher-order recursion). If we consider recursive programs with the property that each level of the recursion only uses bounded storage, then context-free grammars are able to precisely model the program behavior. In the rest of this section, we want to make these two correspondences explicit.

We have already given a part of the translation in Section 1.3 of the introduction. Here, we fill in the details. We consider a toy programming language for recursive programs. A *program* is a set of procedures `p()`, including a distinguished procedure `main()`. Each *procedure* is specified by its source code, a finite list of statements. Each *statement* is either an atomic statement, e.g. a variable assignment `x = 5`, a procedure call `f()`, or a conditional. A *conditional* is of the shape `if (cond) then`, consisting of a conditional expression `cond`, followed by some finite number statements, followed by `else`, followed by some finite number of statements, followed by `end`. We assume that the conditionals are well-nested. We do neither formally specify the shape of atomic statements and conditional expressions, nor do we formally define an evaluation semantics.

Formally defining the syntax of programs, e.g. to ensure that conditionals are well-nested and to enable parsing, could be done using a context-free grammar. A modified version of that grammar can be used to model the control flow of programs, which we describe in the following. For each procedure `p()` and each of the n statements defining the source code of the procedure, we introduce a new nonterminal $\mathtt{p}_i$ where $i \in [1, n+1]$, including a special nonterminal $\mathtt{p}_{n+1}$ for the end of the procedure. The initial symbol is the nonterminal $\mathtt{main}_1$ associated to the first statement of the main procedure. The rules for a nonterminal $\mathtt{p}_i$ depend on the statement in the i^{th} line of the source code for `p()`. If the statement is an atomic statement `a`, we get the rule $\mathtt{p}_i \to \mathtt{a}\,\mathtt{p}_j$, using `a` as a terminal symbol. If the statement is a procedure call `f()`, the rule is $\mathtt{p}_i \to \mathtt{f}_1\,\mathtt{p}_j$. In both of these rules, j should be the index of the next line after i, which usually is $i+1$ unless $\mathtt{p}_i$ represents the last statement of the `then` or `else` block of a conditional. In that case, j is the index of the first line after that conditional. Also note that if the body of the procedure ends after $\mathtt{p}_i$, then j will be $n+1$ and $\mathtt{p}_j$ is the special nonterminal introduced for the end of the procedure. If the statement in the i^{th} line of `p()` is a conditional `if (cond) then`, we get two rules $\mathtt{p}_i \to \text{assume}(\mathtt{cond})\,\mathtt{p}_{i+1}$ and $\mathtt{p}_i \to \text{assume}(\mathtt{!cond})\,\mathtt{p}_j$. Here, we have turned the conditional expression and its negation into terminal symbols assume(`cond`) and assume(`!cond`), respectively. Note that $i+1$ is the index of the first line of the `then` block, and we define j to be the first line of the `else` block. For the special nonterminal $\mathtt{p}_{n+1}$, we introduce the rule $\mathtt{p}_{n+1} \to \varepsilon$.

We claim that the control flow of a program is represented by the words in the language of that grammar. To see this, note that the left-derivation processes of words in the language of that grammar correspond to the control flow according to a standard evaluation semantics (which we have not formally specified). Now we can use the fact that the language of the grammar equals the set of finite words that can be obtained using left-derivations. Note that we only model the control flow; we do not take the data values into account. For example, a condi-

tional always yields two rules, one for each branch, even though in a deterministic program, the current memory value uniquely determines which branch will be taken. We simply print which branch we have used as a terminal symbol and leave it to an additional verification step to distinguish words in the language that correspond to valid executions that respects the data values from the words that do not represent valid executions.

Additionally, we want to show that if the program uses only bounded storage at each level of the recursion, we can explicitly model the semantics using a context-free grammar. Consider some finite domain $\mathcal{D}$ from which the data values used at each level of the recursion stem. We associate it to each atomic statement $\mathtt{a}$ a function $F_{\mathtt{a}}\colon \mathcal{D} \to \mathcal{D}$ that specifies the transformation of the current storage that is applied by executing $\mathtt{a}$. Similarly, we associate to each conditional expression `cond` a predicate $P_{\mathtt{cond}}\colon \mathcal{D} \to \mathbb{B}$ that specifies for which data values the expression evaluates to true. To enable communication among the different levels of the recursion, we redefine procedure to be of the shape `x = f(y)`. Each procedure call works on an independent copy of the storage. The variables `y` and `x` allow us to transfer information into and out of that copy, respectively. Formally, we assume that there is a function $F_{\mathtt{f(y)}}\colon \mathcal{D} \to \mathcal{D}$ that transforms the storage used by the current procedure into the storage used by the procedure `f` that we have called. Similarly, we have a function $F_{\mathtt{x = f(-)}}\colon \mathcal{D} \to \mathcal{D}$ that does the opposite.

We modify the above grammar by having one copy $\mathtt{p}_i(d, d')$ of each nonterminal $\mathtt{p}_i$ for each pair of data values $d, d' \in \mathcal{D}$. Intuitively, $\mathtt{p}_i(d, d')$ means that we are in the i^{th} line of procedure $\mathtt{p}$, the current storage is represented by the data value d and we expect the procedure to terminate with data value d'.

The rules are modified as follows: If the i^{th} line of $\mathtt{p}$ is an atomic statement $\mathtt{a}$, we get the rule $\mathtt{p}_i(d, d') \to \mathtt{a}\ \mathtt{p}_j(F_{\mathtt{a}}(d), d')$, where j is the index of the next line as explained before, and $F_{\mathtt{a}}(d)$ is the result of applying the transformation that corresponds to executing $\mathtt{a}$. If the i^{th} line of $\mathtt{p}$ is `if (cond) then`, then the rule associated to $\mathtt{p}_i(d, d')$ is either $\mathtt{p}_i(d, d') \to \text{assert}(\mathtt{cond})\ \mathtt{p}_{i+1}(d, d')$ or $\mathtt{p}_i(d, d') \to \text{assert}(\mathtt{!cond})\ \mathtt{p}_j(d, d')$, depending on whether $P_{\mathtt{cond}}(d)$ evaluates to true. Here, $i + 1$ is the index of the first line of the `then` block and j is index of the first line of the `else` block. The complicated part is modelling procedure calls `x = f(y)`. For each $\mathtt{p}_i(d, d')$, we get one rule for each data value d''. This rule is as follows:

$$\mathtt{p}_i(d, d') \to \mathtt{f}_1(F_{\mathtt{f(y)}}(d), d'')\ \mathtt{p}_j(F_{\mathtt{x = f(-)}}(d''), d')$$

Intuitively, we guess the data value d'' representing the storage at the termination of `f`. We apply the transformation $F_{\mathtt{f(y)}}$ to the current value to obtain the initial storage for `f`, then go into procedure `f` with the obligation to terminate with data value d''. The execution of procedure $\mathtt{p}$ continues from the next line (the one with index j) with the data value that results from applying $F_{\mathtt{x = f(-)}}$ to d''. In order to make sure that a procedure call actually fulfills its obligation, we change the rule for $\mathtt{p}_{n+1}(d, d')$ as follows. If $d = d'$, i.e. the actual data value equals the expected one, there is a rule $\mathtt{p}_{n+1}(d, d') \to \varepsilon$. Otherwise, we have no rule for this nonterminal.

It remains to fix the initial symbol. We introduce a new nonterminal S and use it as initial symbol. For each $d' \in \mathcal{D}$, there is a rule $S \to \texttt{main}_1(d_{\text{init}}, d')$. Intuitively, this means we start in the first line of the procedure `main` with some fixed initial data value d_{init}, but we guess which data value d' we will reach upon termination.

In contrast to the above grammar modeling the control flow of a program, our new grammar features deterministic conditionals. We only have one rule that either jumps to the then-branch or to the else-branch, depending on whether the conditional expression evaluates to true. Instead, we use nondeterminism to model procedure calls. We guess the data value that the procedure that we call will terminate with. Only the correct guess will actually contribute to words in the grammar. If we have guessed some value d' but the procedure call terminates with data value $d \neq d'$, then there will be no rule for the nonterminal $\mathsf{p}_{n+1}(d, d')$. The corresponding derivation gets stuck and does not produce a finite word. In fact, we get that even though the grammar is nondeterministic, its language is a singleton consisting of a unique word corresponding to the unique execution of the deterministic program (assuming that this execution terminates). This property shows the desired statement: A context-free grammar can model a deterministic recursive program that uses a bounded amount of storage at each level of the recursion.

Note that the size of the grammar is polynomial in the product of the total number of statements in the program and the size of $\mathcal{D}$. For each line of code, we have at most $|\mathcal{D}|^2$ many nonterminals and at most $|\mathcal{D}|^3$ many rules. However, the size of $\mathcal{D}$ can get rather large. If the program uses k Boolean variables as storage, then $\mathcal{D}\colon [1, k] \to \mathbb{B}$ is the space of variable assignments, which is of size 2^k. The construction of the grammar is polynomial in the size of $|\mathcal{D}|$, but exponential in k. Verifying Boolean programs is a PSPACE-complete problem even in the absence of recursion [CS99], so we cannot expect to get a deterministic algorithm for it whose running time is better than exponential.

5.2 ω-languages of context-free grammars

We introduce ω-context-free languages, ω-languages defined by context-free grammars. In the literature, ω-context-free languages are an established class of languages that can be defined in several equivalent ways, which we will discuss later. For now, we present a novel definition of this class of languages that we feel is more elegant. This also means that in contrast to the rest of this chapter, this section contains a contribution from the author of this thesis. This research is taken from the publication [MMN17], as detailed in Chapter 20.

Consider an infinite left-derivation process

$$S = \beta_0 \Rightarrow_\ell \beta_1 \Rightarrow_\ell \dots$$

for some CFG $G = (N, P, S)$. As discussed in the previous section, it induces a sequence of terminal words $\mathrm{prefix}(\beta_0), \mathrm{prefix}(\beta_1), \dots$ such that for each $i, k \in \mathbb{N}$, $\mathrm{prefix}(\beta_i)$ is a prefix of $\mathrm{prefix}(\beta_{i+k})$. Intuitively, we want to associate to such a derivation process the infinite word that is the limit of these prefixes. If the length of the prefixes grows unboundedly, we may define the infinite word $\lim \mathrm{prefix}(\beta_i)_{i\in\mathbb{N}} \in \Sigma^\omega$ whose j^{th} letter is the j^{th} letter of the $i_j{}^{\text{th}}$ prefix, where $i_j \in \mathbb{N}$ is an arbitrary index so that the corresponding prefix is sufficiently long. Formally, we define $(\lim \mathrm{prefix}(\beta_i)_{i\in\mathbb{N}})_j = \mathrm{prefix}(\beta_{i_j})_j$, where $i_j \in \mathbb{N}$ is chosen so that $|\mathrm{prefix}(\beta_{i_j})| \geqslant j$. The above observation guarantees that the definition is independent of the choice of each i_j.

With this definition at hand, it is possible to associate to a context-free grammar G a language $\mathcal{L}' = \{\lim \mathrm{prefix}(\beta_i)_{i\in\mathbb{N}} \in \Sigma^\omega \mid (\beta_i)_{i\in\mathbb{N}} \text{ is an infinite left-derivation process}\}$ of all infinite words that occur as the prefix limits along infinite left-derivation processes.

For example, the ω-regular language $(a \cup b)^* b^\omega$ of words that contain only finitely many as occurs as the language of the grammar with the rules $S \to XY, X \to Xa \mid Xb \mid \varepsilon, Y \to bY$. In a left-derivation process, we can first use the rules for X to derive a prefix from $\{a, b\}^*$ of arbitrary, but finite length, and then append b^ω by using $Y \to bY$ infinitely often. We cannot obtain infinitely many as, because to do so, we would need to preserve nonterminal X for infinitely many steps. Such a derivation process consists entirely of sentential forms that start with nonterminal X, so their prefixes are ε and the prefix limit is not an infinite word.

However, the definition is too weak to express some languages that certainly should be ω-context-free according to any reasonable definition. For example, there can be no context-free grammar whose associated language is $\{a^n b^n \mid n \in \mathbb{N}\}^\omega$, which is the ω-iteration of a context-free language. To sketch the proof of this fact, we observe that any candidate grammar needs to contain a nonterminal X that can be reached from the initial symbol and a rule $X \to aXb$ (or a set of rules that after a finite number of replacement steps lead to such a pattern). Such a rule is needed to be able to create an unbounded but equal amount of as and bs on each side: It is well-known that with rules that are exclusively left- or right-linear (rules that only contain one nonterminal that is the leftmost or rightmost symbol), it is impossible to generate non-regular

languages. By infinitely often applying this rule, we obtain a left-derivation process with prefix limit a^ω, a word which is not in the desired language.

We present a definition that solves this problem. Let us call an infinite left-derivation process $S = \beta_0 \Rightarrow_\ell \beta_1 \Rightarrow_\ell \ldots$ *right-infinite* if it contains infinitely many sentential forms of the shape wX where $w \in \Sigma^*$ is a terminal prefix and X is a single nonterminal. In such a derivation process, the rightmost nonterminal is replaced infinitely often. Additionally, every other nonterminal is replaced by a terminal word within finitely many derivation steps.

The name right-infinite comes from the shape of the derivation tree associated to this type of derivation process. A derivation tree (associated to a derivation process) is a tree in which the nodes are labeled by symbol from $N \cup \Sigma$. It is constructed inductively as follows: We start with a tree only consisting of a root node labeled by S. If production rule $X \to \eta$ is used in the derivation process, we consider the node of the derivation tree that corresponds to the occurrence of X that should be replaced and insert children $\eta_1, \ldots, \eta_{|\eta|}$. The limit of this process is a derivation tree in which all inner nodes are labeled by nonterminals, while the leaves are terminals. The yield of the tree (the leaves read from left to right) form the terminal word that is derived by the derivation process. The derivation tree associated to a right-infinite left-derivation process is infinite only in its rightmost branch. Any node in the tree that is not this branch can be reached within finitely many steps.

The proper definition of the omega-language of a context-free grammar associates to a grammar all infinite prefix limits that occur along right-infinite left-derivation processes.

5.2.1 Definition
A CFG G defines the ω-language

$$\mathcal{L}^\omega(G) = \left\{ \lim \operatorname{prefix}(\beta_i)_{i\in\mathbb{N}} \in \Sigma^\omega \;\middle|\; (\beta_i)_{i\in\mathbb{N}} \text{ is a right-infinite left-derivation process} \right\}.$$

In a sense, this definition imposes a Büchi acceptance condition. We may consider the LTS $(\vartheta, T, S, \Sigma^* N)$ with sentential forms as configurations and transitions induced by left-derivations with their prefix growth as label. The final configurations are configurations of the shape $wX \in \Sigma^* N$, i.e. configurations in which the unique nonterminal is the rightmost symbol. The language of the LTS with a Büchi acceptance condition (i.e. requiring that computations contain infinitely many final configurations) is exactly $\mathcal{L}^\omega(G)$ as defined above.

We will later show in Example 5.2.4 and Remark 5.2.5 that this way of defining the ω-languages of CFGs is strictly more expressive than the version without the restriction to right-infinite processes. On the one hand, $\{a^n b^n \mid n \in \mathbb{N}\}^\omega$ is the ω-language of a suitable CFG. On the other hand, we can recover the weaker definition that does not require right-infinity by adding rules to the grammar.

Characterizing ω-context-free languages

Let us call a language of infinite words *ω-context-free* if it occurs as $\mathcal{L}^\omega(G)$ of some CFG G. The class of these languages admits the following characterization.

5.2.2 Theorem
An ω-language $\mathcal{L} \subseteq \Sigma^\omega$ is ω-context-free iff it can be written as

$$\mathcal{L} = \bigcup_{i \in [1,n]} U_i . V_i^\omega$$

where $n \in \mathbb{N}$ and for each $i \in [1, n]$, $U_i, V_i \subseteq \Sigma^*$ are context-free languages.

In words, a language is ω-context-free iff it is the finite union of languages that consist of a finite context-free prefix, and a context-free period that is repeated *ad infinitum*. Note that this is analogous to the characterization of ω-regular languages given in Section 4.4. Once proven, the characterization will in particular show that $\{a^n b^n \mid n \in \mathbb{N}\}^\omega$ is an ω-context-free language.

Before proving the result, we discuss its implications. In particular, we use the result to show that our definition of ω-context-free languages coincides with various definitions from the literature.

Related work

Other works in the past have considered various definitions of ω-context-free languages.

Cohen and Gold [CG77] have studied how ω-context-free language can be defined by (1) pushdown automata with a Büchi acceptance condition on the states, (2) pushdown automata with the condition that the empty stack is reached infinitely often, and (3) context-free grammars that proceed by left-derivations and are equipped with a so-called repetition set. The repetition set specifies which productions need to be used infinitely often (or, equivalently, which nonterminals need to be replaced infinitely often) for a derivation process to be accepting. Hence, it also imposes a Büchi condition. The authors prove that all three definitions are equivalent, since all three are equal to what they call the Kleene closure of the context-free languages of finite words. The Kleene closure is exactly the type of languages as in Theorem 5.2.2. Hence, with the theorem at hand, our definition of ω-context-free languages is also equivalent to the three methods. Arguably, our method is more elegant as the Büchi condition is implicit instead of being explicitly specified in the form of a repetition set.

In the second part of the paper, the authors show that if one allows arbitrary derivations (instead of exclusively considering left-derivations), one obtains a weaker class of languages, even under the presence of a Büchi condition in the form of a repetition set. For example, the language $\{a^n b^n \mid n \in \mathbb{N}\}^\omega$ cannot be obtained by such a grammar.

Independently, Linna [Lin76] has given a definition of ω-context-free languages defined by pushdown automata with a Büchi acceptance condition. He shows that this class is equal to the languages obtained from the ω-regular languages by applying context-free substitutions. Given the characterization of ω-regular languages, this proves a result analogous to Theorem 5.2.2 and shows that his definition coincides with our definition.

He also considers the class of languages defined as limits of context-free languages. The limit of a context-free language is the set of infinite words such that all their finite prefixes are contained in the context-free language. This class of languages is incomparable to the ω-context-free languages. On the one hand, it does not contain the ω-regular language $(a \cup b)^* b^\omega$. On the other hand, the limit of the context-free language $\{w.b.a^k \mid k \text{ equals the number of } b\text{s in } w\}$ is not ω-context-free: It is a language without so-called ultimately periodic words. An *ultimately periodic word* is an infinite word of the shape $w.v^\omega$ for suitable finite words $w, v \in \Sigma^*$. From our characterization result, Theorem 5.2.2, we immediately obtain that every ω-context-free language is guaranteed to contain at least one such word.

Linna also shows that by closing the class of limits of context-free languages under homomorphisms (or, equivalently, context-free substitutions), one obtains a superclass of both that class and the class of ω-context-free languages. The algorithms for languages of infinite words that we will consider later, e.g. in Section 16.3, rely on the presence of ultimately periodic words. Hence, they would not work for the latter two classes of languages defined by Linna.

Proof of Theorem 5.2.2

The rest of this section is dedicated to the proof of Theorem 5.2.2. One direction is easy, as we can give a straightforward construction.

5.2.3 Lemma
If $\mathcal{L} = \bigcup_{i \in [1,n]} U_i.V_i^\omega$ for $n \in \mathbb{N}$ and context-free U_i, V_i, then $\mathcal{L}$ is ω-context-free.

Proof:
Assume that for each i, $U_i = \mathcal{L}(N_i, \Sigma, P_i, S_i)$ and similarly $V_i = \mathcal{L}(N_i', \Sigma, P_i', S_i')$. We assume wlog. that all N_i and N_i' (and hence also all P_i and P_i') are pairwise disjoint, and we assume that none of these sets contain S and X_i for $i \in [1, n]$. We construct a new grammar $G = (N, \Sigma, P, S)$ with $N = \bigcup_i N_i \cup \bigcup_i N_i \uplus \{S\} \uplus \{X_i \mid i \in [1, n]\}$ and $P = \bigcup_i P_i \cup \bigcup_i P_i' \uplus \{S \to S_i X_i, X_i \to S_i' X_i \mid i \in [1, n]\}$.

Grammar G intuitively works as follows: In the first step, one chooses some part $U_i.V_i^\omega$ of the union by picking the corresponding rule $S \to S_i X_i$. From S_i, a word in U_i can be derived (recall that S_i is the initial symbol of the corresponding grammar). This process has to be finite, since S_i is not the rightmost nonterminal. Afterwards, an infinite process starts, in each step of which X_i is replaced using $X_i \to S_i' X_i$. After one such step, the occurrence of S_i' can then be used to derive a finite word from V_i. Hence, the language $\mathcal{L}^\omega(G)$ is indeed $\bigcup_{i \in [1,n]} U_i.V_i^\omega$. ■

5.2.4 Example

To obtain a grammar for $\{a^n b^n \mid n \in \mathbb{N}\}^\omega$, we may apply a simplified version of the construction from the above proof, obtaining $G = (\{S, X\}, \{a, b\}, \{S \to XS, X \to aXb \mid \varepsilon\}, S)$. This grammar indeed generates the desired language: Intuitively, it produces an infinite number of occurrences of S, each of which is then replaced by a finite word from $\{a^n b^n \mid n \in \mathbb{N}\}$.

5.2.5 Remark

We briefly discuss how to recover the weaker definition of the ω-language of a CFG in which we do not restrict ourselves to right-infinite derivation processes. Observe that because a derivation process is linear, at most one branch of the derivation tree can be infinite. Since we only consider left-derivations, any node in the tree that is to the right of an infinite branch will not be replaced: The infinite branch will always contain a nonterminal that is more to the left and has to be replaced first. Similarly, terminal symbols that are to the right of an infinite branch will not contribute to the word generated by the derivation process. Since the infinite branch will always contain some nonterminal that occurs earlier, these terminals are not contained in any prefix.

This observation yields a method to turn a grammar G into a grammar G' so that $\mathcal{L}^\omega(G')$ is equal to $\{\lim \text{prefix}(\beta_i)_{i\in\mathbb{N}} \in \Sigma^\omega \mid (\beta_i)_{i\in\mathbb{N}} \text{ is a right-infinite left-derivation process of } G\}$. The idea is to allow any branch of the derivation tree to be infinite by making it the rightmost branch. To this end, we simply drop the part of the derivation tree that is to the right of this infinite branch. As argued before, dropping this part will not influence the words that can be produced.

To implement this idea, we let G' consist of the nonterminals of G as well as a fresh version X' for every nonterminal X. Intuitively, we will make sure that the infinite branch of the derivation tree will consist of nonterminals of the shape X'. The initial symbol is S', the copy of S. The old nonterminals retain their rules. For each rule $X \to \eta$ of G and each prefix $\beta.Y$ of η that ends in a nonterminal, we add a rule $X' \to \beta.Y'$ for X'. Note that we use the old versions of the nonterminals in β, but we replace Y by its copy Y'.

All sentential forms that can be reached from S' contain exactly one nonterminal of the shape X', which is the rightmost symbol. Whenever we replace such a terminal, we intuitively pick a rule $X \to \eta$ of G, designate a nonterminal Y in η to be on the infinite branch of the derivation tree and drop the rest of η.

We apply this construction to the grammar from Example 5.2.4. We obtain the grammar $G' = (\{S', X', X\}, \{a, b\}, \{S' \to XS' \mid X', X \to aXb \mid \varepsilon, X' \to aX'\}, S')$. The rules $S' \to XS'$ and $S \to X'$ result from the rule $S \to XS$ of G by choosing X resp. S as the nonterminal on the infinite branch. Similarly, $X' \to aX'$ results from $X \to aXb$. Note that we have omitted the nonterminal S because it does not occur in any sentential form that is reachable from S'.

A right-infinite left-derivation process of this grammar either uses the rule $S' \to XS'$ infinite often. In this case, it behaves as the derivation processes of G from Example 5.2.4

and generates a word of the shape $(a^{n_i}b^{n_i})^\omega$. Alternatively, the derivation process uses the rule $S' \to X'$ at some point, followed by an infinite sequence of applications of the rule $X' \to aX'$. Hence, the word we obtain is a member of the language $\{a^n b^n \mid n \in \mathbb{N}\}^*.a^\omega$. Altogether, $\mathcal{L}^\omega(G) = \{a^n b^n \mid n \in \mathbb{N}\}^\omega \cup \{a^n b^n \mid n \in \mathbb{N}\}^*.a^\omega$.

The other direction of the proof requires more work. The first step is to present a different view of the language $\mathcal{L}^\omega(G)$ that separates the infinite rightmost branch of the derivation tree from the other branches. We start by noting that if $w.X$ is a sentential form in a right-infinite left-derivation process, then the next step will not apply a production rule whose rightmost symbol is a terminal: Otherwise, we would end up in a sentential form with a terminal as the rightmost symbol, and it becomes impossible to reach another sentential form that ends in a nonterminal.

We use this observation to define the *spinal graph*, a finite LTS with nonterminals as configurations that is labeled by sentential forms. Formally we define $SG = (N, T, S)$ where $X \xrightarrow{\beta} Y$ if grammar G contains a production rule $X \to \beta.Y$. Since the number of nonterminals and productions in the grammar is finite, so is SG. Note that we have not equipped SG with an acceptance condition. We define $\mathcal{L}^\omega(SG) \subseteq \vartheta^\omega$ to be all sequences of labels that occur along infinite paths in SG starting in S. We may flatten $\vartheta^\omega = ((\Sigma \cup N)^*)^\omega$ to see such a sequence as an element of $(\Sigma \cup N)^\omega$, and hence $\mathcal{L}^\omega(SG)$ as an ω-language over $\Sigma \cup N$. A problem arises in the case that in an infinite path ε is the only label occurring infinitely often. In this case, the resulting sequence over $\Sigma \cup N$ is finite. We will take care of this problem later by excluding such paths.

It remains to relate the language of SG to $\mathcal{L}^\omega(G)$. To this end, we define $\mathcal{L}_G(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}_G(a) = a$ for terminals $a \in \Sigma$ and $\mathcal{L}_G(X) = \mathcal{L}(N, P, X)$, i.e. the context-free language of finite words obtained by seeing X as the initial symbol of grammar G. For finite and infinite sequences over $N \cup T$, we define their language to be $\mathcal{L}_G(\beta_1\beta_2\ldots) = \mathcal{L}_G(\beta_1).\mathcal{L}_G(\beta_2)\ldots$, the (finite or infinite) concatenation of the respective languages. For sets of such sequences, we define the language by applying the operator $\mathcal{L}_G$ element-wise and then taking the union.

5.2.6 Lemma
$\mathcal{L}^\omega(G) = \mathcal{L}_G(\mathcal{L}^\omega(SG)) \cap \Sigma^\omega$.

Note that the intersection with Σ^ω removes paths from $\mathcal{L}^\omega(SG)$ in which ε is the only label occurring infinitely often.

Proof:
Consider a word w in $\mathcal{L}^\omega(G)$ and a right-infinite left-derivation process for that word. Extracting the infinite sequence of derivation steps that replace nonterminal yields a path π in SG. The infinite sequence of labels over $(N \cup \Sigma)^*$ along π can be flattened to obtain an infinite word $\beta \in \mathcal{L}^\omega(SG)$ over $N \cup \Sigma$. The rest of the derivation steps in the right-infinite left-derivation

process for w can be used to replace each nonterminal X in β by a finite word in $\mathcal{L}_G(X)$. The concatenation of these finite words with the nonterminals that may occur in β yields the infinite word w. Altogether, we obtain that w is an infinite word in $\mathcal{L}_G(\beta) \subseteq \mathcal{L}_G(\mathcal{L}^\omega(SG))$ as desired.

For the other direction, consider an infinite word w in $\mathcal{L}_G(\mathcal{L}^\omega(SG))$. By definition, it is an element of $\mathcal{L}_G(\beta)$ for some $\beta \in \mathcal{L}^\omega(SG)$, where β corresponds to an infinite path π in SG. We construct a right-infinite left-derivation process for w as follows. We start from the initial symbol of the grammar. Whenever we have to replace the rightmost nonterminal in the current sentential form, say X, we consider the earliest transition in the path π that has not been processed yet. It is of the shape $X \xrightarrow{\beta^{(i)}} Y$, corresponding to a rule $X \to \beta^{(i)}Y$ of the grammar. We apply this rule, producing a finite infix $\beta^{(i)}$ of the infinite sequence β and a new rightmost nonterminal. In order to replace the nonterminals in $\beta^{(i)}$, we use $w \in \mathcal{L}_G(\beta)$ to obtain a finite sequence of left-derivations that replace such a nonterminal by an infix of w. For each such nonterminal, we use the corresponding sequence of left-derivations until Y is the only nonterminal remaining and we proceed by processing the next transition from π. The infinite word produced by this derivation process is w, as it is the concatenation of the terminals in β and the finite infixes of w derived from the nonterminals. Hence, $w \in \mathcal{L}^\omega(G)$. ■

Finally, we can finish the proof of Theorem 5.2.2 by showing the missing implication.

5.2.7 Proposition
Each ω-context-free language $\mathcal{L} \subseteq \Sigma^\omega$ can be written as

$$\mathcal{L} = \bigcup_{i \in [1,n]} U_i.V_i^\omega$$

for some $n \in \mathbb{N}$ with each $U_i, V_i \subseteq \Sigma^*$ context-free.

Proof:
Consider an ω-context-free language $\mathcal{L}^\omega(G)$ for some CFG G. We may construct the associated spinal graph SG. For nonterminals X, Y, we define $\mathcal{L}_{X,Y}$ to be set of sequences over $N \cup \Sigma$ that occur in SG as the labels along finite paths from node X to Y.

We claim that

$$\mathcal{L}^\omega(G) = \bigcup_{X \in N} \mathcal{L}_G(\mathcal{L}_{S,X})(\mathcal{L}_G(\mathcal{L}_{X,X}))^\omega .$$

Firstly, we argue that the expression on the right-hand side is of the required shape. Because there are only finitely many nonterminals, the union is finite. Each language $\mathcal{L}_{X,Y}$ is a regular language over $N \cup \Sigma$. To this end, observe that we may see the spinal graph as a finite automaton (after inserting suitable intermediary states to make sure that each transition generates at most

one letter) with X as the initial and Y as the unique final state. Finally, observe that $\mathcal{L}_G(\mathcal{L}_{X,Y})$ is hence context-free. The regular languages are a subclass of the context-free ones, so there is a CFG over $N \cup \Sigma$ for $\mathcal{L}_{X,Y}$. By adding the rules for the nonterminals from G to this grammar, we obtain a CFG over Σ that generates $\mathcal{L}_G(\mathcal{L}_{X,Y})$. In particular, $\mathcal{L}_G(\mathcal{L}_{S,X})$ and $\mathcal{L}_G(\mathcal{L}_{X,X})$ are context-free languages and the expression is as required.

Secondly, we argue that $\mathcal{L}^\omega(G) \subseteq \bigcup_{X \in N} \mathcal{L}_G(\mathcal{L}_{S,X})(\mathcal{L}_G(\mathcal{L}_{X,X}))^\omega$. By Lemma 5.2.6, $\mathcal{L}^\omega(G) = \mathcal{L}_G(\mathcal{L}^\omega(SG)) \cap \Sigma^\omega$. Consider a word $w \in \mathcal{L}^\omega(G)$. It is a member of $\mathcal{L}_G(\beta)$ for some $\beta \in \mathcal{L}^\omega(SG)$, i.e. β is the sequence of labels along an infinite path π in SG that starts in S. Because the spinal graph has only finite many nodes, there is some nonterminal X that is visited infinitely often by π. By summarizing transitions in the graph, we may write π as

$$S \xrightarrow{\eta} X \xrightarrow{\gamma_1} X \xrightarrow{\gamma_2} X \xrightarrow{\gamma_3} \dots,$$

where η and the γ_i are words over $N \cup \Sigma$ so that $\beta = \eta.\gamma_1.\gamma_2 \dots$ By definition, $\eta \in \mathcal{L}_{S,X}$ and each $\gamma_i \in \mathcal{L}_{X,X}$ for all i. Hence, $\beta \in \mathcal{L}_{S,X}.(\mathcal{L}_{X,X})^\omega$ and $w \in \mathcal{L}_G(\beta)$ is an element of $\mathcal{L}_G(\mathcal{L}_{S,X}).(\mathcal{L}_G(\mathcal{L}_{X,X}))^\omega$ as desired.

Finally, we consider the other direction. Let $w \in \mathcal{L}_G(\mathcal{L}_{S,X}).(\mathcal{L}_G(\mathcal{L}_{X,X}))^\omega$ for some $X \in N$. Note that w is an infinite word because the omega-iteration of a language is defined to only produce words in Σ^ω. By definition, we may write $w = w^{(0)}.w^{(1)}.w^{(2)} \dots$ with $w^{(0)} \in \mathcal{L}_G(\mathcal{L}_{S,X})$ and $w^{(i)} \in \mathcal{L}_G(\mathcal{L}_{X,X})$ for all $i > 0$. This in turns means there is $\eta \in \mathcal{L}_{S,X}$ with $w^{(0)} \in \mathcal{L}_G(\eta)$ and $\gamma_i \in \mathcal{L}_{X,X}$ with $w^{(i)} \in \mathcal{L}_G(\gamma_i)$ for all $i > 0$. By the definition of the languages $\mathcal{L}_{S,X}$ and $\mathcal{L}_{X,X}$, $\beta = \eta.\gamma_1.\gamma_2 \dots$ is a sequence of labels along an infinite path in the spinal graph starting in S, hence $\beta \in \mathcal{L}^\omega(SG)$. We have that $w \in \Sigma^\omega$ is an infinite word in $\mathcal{L}_G(\beta)$ with $\beta \in \mathcal{L}^\omega(SG)$. By Lemma 5.2.6, this is sufficient so show $w \in \mathcal{L}^\omega(G)$, which completes the proof. ■

5.3 Higher-ordered recursion schemes

We introduce *higher-order recursion schemes (HORSes)* [Niv72; CN78], a rewriting-based type of language generating mechanism that can be seen as a generalization of context-free grammars. As briefly mentioned, context-free grammars can be used to model recursive programs; they can be seen as a HORSes of order one. Modern programming languages like Haskell [Pey03] support higher-order functions, i.e. functions whose parameters themselves are functions. Context-free grammars (and HORSes of order one) are insufficient to model this concept, while HORSes of order greater than one provide a model for such programs. Our presentation follows [Had12].

We first introduce a typing discipline. The typing fixes for each term a type, which in particular determines its order, i.e. whether it is a value (a term of order zero), a function of order one that operates on values, or a function of higher order that has functions as parameters. Representing both functions and values by terms corresponds to the concept of treating functions as *first-class citizens* in modern programming languages. Instead of having a clear distinction between values and functions, functions are simply seen as values of the corresponding function type.

We will actually not use the name type in the following. We refer to the concept that was explained above as *kinds*, to avoid confusion with type-based approaches to the verification of HORSes as used e.g. in [Kob09].

We define o to be the unique kind *ground* of data values. Functions kinds are derived by composition. Formally, the kinds κ are defined by the following grammar:

$$\kappa ::= o \mid (\kappa \to \kappa) .$$

Intuitively, a term of kind $\kappa_1 \to \kappa_2$ is a function that takes a value of kind κ_1 and returns a value of kind κ_2 (where both the parameter and the return value may be functions). We define K to be the set of all kinds.

We usually omit unnecessary brackets by assuming right-associativity. For instance, this means that $o \to o \to o$ denotes $o \to (o \to o)$, i.e. the kind of functions that take a ground value and return a function that takes and returns a ground value.

Our definition of kinds does not allow for multi-parameter functions. Instead, we use the concept of currying. A function that takes two values and returns one, which intuitively should have kind $o \times o \to o$ is seen as a function of kind $o \to (o \to o)$: It takes the first parameter and returns a function that takes the second parameter and then returns the final value. Formally, we consider $f\colon o \times o \to o$ as a function $f'\colon o \to (o \to o)$ by defining $f'(x)\colon o \to o$ with $(f'(x))(y) = f(x, y)$. This concept is commonly implemented in functional programming languages like Haskell to simplify the usage of partially evaluated functions.

We define the notion of *arity* that describes the number of parameters that a function takes (in its uncurried form), and its *order* as explained above. We formally define

$$\begin{aligned} \operatorname{arity}(o) &= 0, & \operatorname{order}(o) &= 0, \\ \operatorname{arity}(\kappa_1 \to \kappa_2) &= \operatorname{arity}(\kappa_2) + 1, & \operatorname{order}(\kappa_1 \to \kappa_2) &= \max(\operatorname{order}(\kappa_1) + 1, \operatorname{order}(\kappa_2)) . \end{aligned}$$

For example, we have $\operatorname{arity}(o \to (o \to o)) = \operatorname{arity}(o \to o) + 1 = \operatorname{arity}(o) + 2 = 2$ and $\operatorname{order}(o \to (o \to o)) = 1$. Indeed, functions of type $o \to (o \to o)$ in their uncurried form take two parameters and are of order one, since their parameters and their return value are ground values. In contrast, we have $\operatorname{arity}((o \to o) \to o) = 1$ and $\operatorname{order}((o \to o) \to o) = 2$. A function of kind $(o \to o) \to o$ takes a function as parameter and returns a ground value.

Just as a context-free grammar, a HORS consists of terminals and nonterminals. In contrast to CFGs, we see the nonterminals of HORSes as functions. Syntactically, this means that we also need a set of variables. To get a consisting typing, each of the three types of symbols has an associated kind. More formally, a *kinded symbol* is a symbol a together with a kind κ, which we write as $a\colon\kappa$. Let Λ be a set of kinded symbols. When convenient, we see Λ as a set of (distinct) symbols and assume that the kind assignment is implicitly given. By Λ_κ, we denote the restriction of Λ to symbols with kind κ.

For each kind κ, $\mathcal{T}(\Lambda)_\kappa$ is the set of terms of kind κ. These sets are defined by simultaneous inductions over all kinds. They are the smallest sets so that

(1) All symbols from Λ are terms of the appropriate kind,

$$\forall \kappa\colon \Lambda_\kappa \subseteq \mathcal{T}(\Lambda)_\kappa$$

(2) If $f \in \mathcal{T}(\Lambda)_{\kappa_1 \to \kappa_2}$ is a (function) term of kind $\kappa_1 \to \kappa_2$, and $v \in \mathcal{T}(\Lambda)_{\kappa_1}$ is a term of kind κ_1, then the *application* $f\ v$ is a term in $\mathcal{T}(\Lambda)_{\kappa_2}$ of kind κ_2,

$$\forall \kappa_1, \kappa_2\colon \left\{ f\ t \;\middle|\; f \in \mathcal{T}(\Lambda)_{\kappa_1 \to \kappa_2}, v \in \mathcal{T}(\Lambda)_{\kappa_1} \right\} \subseteq \mathcal{T}(\Lambda)_{\kappa_2} .$$

(3) If $t \in \mathcal{T}(\Lambda)_{\kappa_2}$ is a term of kind κ_2, and $x \in \mathcal{T}(\Lambda)_{\kappa_1}$ is a term of kind κ_1, then the *λ-abstraction* $\lambda x.t$ is a term of kind $\kappa_1 \to \kappa_2$,

$$\left\{ \lambda x.t \;\middle|\; t \in \mathcal{T}(\Lambda)_{\kappa_2}, x \in \mathcal{T}(\Lambda)_{\kappa_1} \right\} \subseteq \mathcal{T}(\Lambda)_{\kappa_1 \to \kappa_2} .$$

The set of all terms $\mathcal{T}(\Lambda)$ is defined as the union of the $\mathcal{T}(\Lambda)_\kappa$ for all kinds κ.

As we have seen in the second rule, we usually omit the brackets for function application, i.e. we write $f\ t$ instead of $f(t)$. The last rule is interesting as it allows us to construct infinitely many terms of arbitrarily large order assuming there is at least one symbol of kind ground. Instead

of writing $t \in \mathcal{T}(\Lambda)_\kappa$, we simply write $t\colon\kappa$ (where we assume that Λ is clear from the context). Terms that do not contain λ-abstractions, i.e. they are build only using the first two cases of the definition of terms, are called λ-free.

When defining HORSes, we will assume that Λ contains a set of variables. To distinguish these variables form other types of variables that we will consider in Part V. of the thesis, we speak of *HORS variables*. We assume that only the HORS variables are allowed to take the place of x in the λ-abstraction $\lambda x.t$. This allows us to speak of the free variables of a term, i.e. variables that occur in the term and that are not bound by a preceding λ-abstraction. A *variable-closed* term is a term that does not contain any free variables.

We have now gathered the prerequisites to formally defines HORSes.

A *higher-order recursion scheme (HORS)* is a tuple $G = (V, N, T, P, S)$, where V are the HORS variables, T are the terminals, N are the nonterminals. All three sets are finite sets of kinded symbols that are pairwise disjoint. The symbol $S \in N$ is the initial symbol. The set P contains a finite set of *production rules* of the shape $F \to t$, where t is a term over $\Lambda = V \uplus N \uplus T$, the union of variables, nonterminals and terminals. We require that P is well-typed in that for each production $X \to t$, the kinds of X and t coincide. We require that each right-hand side t is a variable-closed term $\lambda x_1 \ldots \lambda x_n.e$ where the x_i are variables and e is a λ-free term of kind o. (This implies that $x_1, \ldots, x_n$ is a superset of the variables that occur in e.)

We generalize the notions of order and arity from kinds to terms of that kind. The order of a HORS is the maximum order of any of its nonterminals.

The semantics of a HORS G is defined in the form of a term rewriting system, whose terms are the terms (over the union of variables, nonterminals, and terminals), and the rewriting rules of which are induced by the productions of G. To make this definition formal, we define a *context* $C[\bullet]$ to be a term over $\Lambda \uplus \{\bullet\colon o\}$ in which the placeholder $\bullet$ of kind ground occurs exactly once. Given $C[\bullet]$ and a term $t\colon o$ of kind ground, we define $C[t]$ as the term over Λ obtained by replacing the unique occurrence of $\bullet$ by t. Note that the kinds of $C[\bullet]$ and $C[t]$ coincide. We introduce a derivation relation on terms over Λ by defining $t \Rightarrow t'$ to hold iff if there is a context $C[\bullet]$, a production rule $X \to \lambda x_1 \ldots \lambda x_n.e$, and a term $X\ t_1\ \ldots\ t_n\colon o$ such that $t = C[X\ t_1\ \ldots\ t_n]$ and $t' = C[e[x_1 \mapsto t_1, \ldots, x_n \mapsto e_n]]$. In words, the derivation relation allows us to choose a subterm $X\ t_1\ \ldots\ t_n$ of kind ground (which intuitively means that all parameters of X are present) and replace it by the right-hand side of a production for X, with the occurrences of the variables replaced by the appropriate parameters. We call such a subterm a *reducible expression (redex)*. Note that since we required productions to be well-typed, the derivation relation is kind-preserving: If $t \Rightarrow t'$, then t and t' are of the same kind.

To associate a language to a HORS, one usually imposes several restrictions. The first one is that we require the initial symbol S to be of kind ground. Since the rewriting relation is kind-preserving, this implies that any term derivable from S is of kind ground. The second one is that all terminals have order at most one. Together, this means that any term over the terminals that

can be derived from the initial word can be seen as a tree: The term $a\ t_1\ t_2\ \ldots t_n$, where a is a terminal of arity n, is a tree consisting of an a-labeled root and n subtrees that correspond to the parameters. In particular, a terminal of arity 0 is seen as a tree consisting of a single leaf.

Word-generating schemes

We are interested in schemes that define a language of finite words. To this end, we impose the stronger restriction that the set of terminals is of the shape $T = \Sigma \uplus \{\$: o\}$, where \$ is the *word-end marker* of kind ground, and the other terminals in Σ are of kind $o \to o$, i.e. they are first-order and have arity one.

We say that the word $w = a_1 \ldots a_n \in \Sigma^*$ (with the symbols a_i from Σ seen as normal letters) can be derived by HORS G if $S \Rightarrow^* a_1(a_2(a_3 \ldots a_n(\$)))$. In the following, we will identify terms of this shape with the corresponding words. The language $\mathcal{L}(G)$ of G is defined to be the set of all words that can be derived by G.

A derivation step is *outermost to innermost (OI)* if there is no redex that contains the one that was replaced as a proper subterm. Haddad [Had12] has shown that we do not lose expressiveness by restricting ourselves to OI-derivations: Any word that can be derived from the initial symbol can be derived by a sequence of OI-derivation steps. In a sense OI-derivation steps for HORSes correspond to left-derivations in the case of CFGs.

There is the analogue notion of innermost to outermost (IO) derivations. However, there are words that can be derived by OI derivations, but not by IO derivations. Also, we will later make use of the fact that each derivable term in a word-generating scheme has a unique outermost redex. The same is not true for innermost redexes.

5.3.1 Example

Let $G = (N, P, S)$ be a context-free grammar over Σ. We define a word-generating scheme $G' = (V, N', T, P, S')$ of order one with the same language. The set of variables consists of the unique variable $x : o$. The set N' consists of the nonterminals N of G, each seen $X \in N$ as a symbol of kind $o \to o$. Additionally, we introduce a fresh nonterminal $S' : o$ that is the initial symbol of the HORS. The terminals T of the HORS consist of the word end marker $\$: o$ and of the terminals Σ of the grammar, each terminal a seen as a symbol of kind $o \to o$.

For the nonterminals in N, each rule $X \to \eta$ of the grammar with $\eta = \eta_1\eta_2 \ldots \eta_m \in (N \cup \Sigma)^*$ induces a rule $X \to \lambda x.\eta_1(\eta_2(\ldots \eta_m(x)\ldots))$. Intuitively, a nonterminal takes the suffix of the terminal word that has already been generated as the parameter x of kind o. It then prepends the sentential form η. Technically, the concatenations in η are seen as a sequence of function applications. Additionally, there is a rule $S' \to S\ \$$ for the fresh initial symbol S'. Intuitively, it starts the derivation process with the empty suffix.

We obtain a one-to-one correspondence between derivation steps in the grammar and derivation steps of the HORS, and between OI-replacement steps and left-derivations. To

see that this is true, note that any term that can be derived by the HORS from S is of the shape $\beta_1(\beta_2(\dots\beta_k(\$)\dots))$, where each β_i is a terminal or a nonterminal from N. Each subterm $\beta_i(\beta_{i+1}(\dots\beta_k(\$)\dots))$ where β_i is a nonterminal is a redex. Altogether, we obtain $\mathcal{L}(G) = \mathcal{L}(G')$ as desired.

We instantiate this construction for the grammar from Example 5.1.1 and obtain the HORS $G' = (\{x\colon o\}, \{S'\colon o, S\colon o \to o\}, \{a\colon o \to o, b\colon o \to o, \$\colon o\}, P, S')$ with the rules $S' \to S\ \$$, $S \to \lambda x.x$, and $S \to \lambda x.a\ S\ b\ x$. Note that $a\ S\ b\ x$ stands for $a(S(b(x)))$. This HORS generates the language $\{a^n b^n \mid n \in \mathbb{N}\}$ as expected.

5.3.2 Example

We give an example showing that HORSes are strictly more expressive than CFGs. Consider the HORS $G = (\{f\colon o \to o, x\colon o\}, N, T, P, S)$ where $N = \{S\colon o, X, Y\colon (o \to o) \to o \to o, Z\colon o \to o\}$, $T = \{a, b, c\colon o \to o, \$\colon o\}$, and the rules are $S \to X\ Z\ \$$, $X \to \lambda f.\lambda x.f\ x$, $X \to \lambda f.\lambda x.X\ (Y\ f)\ (c\,x)$, $Y \to \lambda f.\lambda x.a\ f\ b\ x$, and $Z \to \lambda x.x$. It is of order two and generates the non-context-free language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$.

Determinism

A HORS is deterministic if each nonterminal X has a unique rule with X as its left-hand side. If we restrict ourselves to OI-derivation steps, this makes the transition relation on the semantic level deterministic for word-generating schemes.

Many works in the literature present word-generating schemes in a different way: They consider deterministic schemes without restricting the terminals to be of arity one. These schemes have a unique derivation process, the limit of which is a tree. The language of finite words of the scheme occurs as the set of all labels along finite branches of that tree. Both views have advantages and drawbacks, and it is easy to convert between them. We will formally introduce the construction for determinization in Section 18.3. The definition of word-generating schemes here makes it easier to see the correspondence between CFGs and HORSes of order one.

Alternative models

In the same way that context-free languages can also be defined via pushdown automata, there is a state-based model for languages of word-generating higher-order schemes. A higher-order pushdown automaton of order n is an automaton that uses an order-n stack as storage. A stack of order one is simply a LIFO stack as defined in Section 5.1. A stack of order $i + 1$ for $i \geqslant 1$ is a sequence of order i stacks. In addition to the order-one operations push A and pop A for every symbol of the stack alphabet, the transitions can be labeled by push_i and pop_i for $i \in [2, n]$. An order-i push duplicates the topmost order-i stack (inside the topmost order-$(i + 1)$ stack inside the topmost-$(i + 2)$ stack and so on, if $i \neq n$). Similarly, an order-i pop operation removes the topmost order-i stack.

Unfortunately, order-*n* pushdown automata do not generate all languages of HORSes of order *n*. Rather, they generate a subclass, the languages of *safe* HORSes, where safety is a syntactic restriction on the production rules of HORSes [DG86; KNU02].

To get the desired equivalence, one has to enhance the model to obtain collapsible higher-order pushdown systems [HMOS17]. In this model, each (order-1) stack symbol is equipped with a link, a pointer into the stack that represents the context in which the symbol was created. Additionally, there is a collapse operation that removes a part of the stack so that the target of the link becomes the top-of-stack. Giving the formal definitions and a more in-depth explanation is not needed for this thesis.

6 Petri nets and well-structured transition systems

Contents

6.1 Unlabeled Petri nets 109
6.2 Algorithmic problems for Petri nets 113
6.3 Petri net coverability languages 118
6.4 BPP nets 123
6.5 Well-structured transition systems 132

We present (labeled) Petri nets, an automaton model that is particularly suitable for modeling concurrent processes. We study the languages defined by Petri nets and algorithmic techniques for checking properties of these languages. Finally, we discuss well-structured transition systems, a generalization of Petri nets.

Most of the material can be found in standard textbooks on the topic, e.g. [Rei85], and is not a contribution of this thesis. Section 6.4 contains a minor contribution, it studies the complexity of the word problem for BPP nets.

6.1 Unlabeled Petri nets

We start by defining the syntax and semantics of unlabeled Petri nets.

A *Petri net* N is a tuple $N = (P, T, \text{in}, \text{out})$.[1] Here, P is a finite set *places*, T is a finite set of *transitions* with $T \cap P = \varnothing$, and $\text{in}, \text{out}: T \times P \to \mathbb{N}$ assign each place and transition the incoming resp. outgoing multiplicity.

We may see in and out as a function taking two arguments (e.g. $\text{in}(p, t) \in \mathbb{N}$) or as a function that assigns to each transition a vector of multiplicities (e.g. $\text{in}(t) \in \mathbb{N}^P$). Here, we will usually write $\mathbb{N}^P$ instead of $\mathbb{N}^{|P|}$ to denote the set of vectors with one tuple per place.

The *effect* $\text{e}(t) \in \mathbb{Z}^P$ of a transition t is defined to be $\text{e}(t) = \text{out}(t) - \text{in}(t)$. We lift operations like addition, subtraction, and comparison from numbers to such vectors by applying them component-wise, e.g. $\text{e}(t)$ is the vector that has entry $\text{out}(t, p) - \text{in}(t, p)$ for each place p.

[1] In the literature, in and out are commonly combined into a single *flow matrix* $F: (P \uplus T) \times (T \uplus P) \to \mathbb{N}$, where $F(x, y)$ can only be non-zero if x is a transition and y is a place or vice versa. The functions in and out can be recovered by defining $\text{in}(t) = F(-, t)$ and $\text{out}(t) = F(t, -)$ for each transition t. Sometimes, $\text{in}(t)$ and $\text{out}(t)$ are called the *pre-set* and *post-set* of transition t, respectively (often denoted by ${}^\bullet t$ and $t^\bullet$).

With the syntax at hand, we can define the semantics of Petri nets. A Petri net $N = (P, T, \text{in}, \text{out})$ induces an infinite transition system: The configurations are *markings* $M \in \mathbb{N}^P$, vectors that associate to each place p a number $M(p)$ of *tokens* carried by that place.

In a marking, a transition can be executed, consuming tokens depending on the incoming multiplicities and producing tokens depending on the outgoing multiplicities. Taking a transition is only possible if there are enough tokens that can be consumed. Formally, in marking M, transition t is enabled if $M \geqslant \text{in}(t)$. In this case, it can be *fired*, leading to the new marking M' with $M' = M + \text{e}(t)$. We write $M \boxed{t} M'$.[1] This defines the transition relation of the transition system. If we want to express that t can be fired from M, but we do not care about the marking that is reached, we write $M \boxed{t}$.

We extend the notion of firing to sequences of transitions $\sigma \in T^*$. We say that σ is a valid *firing sequence* from marking M, reaching marking M' and write $M \boxed{\sigma} M'$ if $\sigma = t_1 \dots t_n$ and there are intermediary markings $M_1, \dots, M_{n-1}$ such that

$$M \boxed{t_1} M_1 \boxed{t_2} \dots \boxed{t_{n-1}} M_{n-1} \boxed{t_n} M' .$$

The combination of the markings and transitions is called a *computation* of M.

We can extend the definition of the effect to transition sequences by $e(\sigma) = \sum_{i=1}^{n} \text{e}(t_i)$. Note that $M \boxed{\sigma} M'$ implies $M' = M + \text{e}(\sigma)$.

Petri nets as graphs

Sometimes it will be useful to see a Petri net as a directed graph in which the set of nodes is the union of the set of transitions and the set of places. This graph has an edge from a place p to a transition t if $\text{in}(t, p) > 0$, and an edge t to p if $\text{out}(t, p) > 0$. Note that the graph is bipartite: There are no edges among places or transitions.

Alternative models

Petri nets are equally expressive to *vector addition systems (VASes)* and *vector addition systems with states (VASSes)*: A Petri net can be converted in polynomial time into a VAS or VASS that essentially has the same algorithmic properties. While Petri nets are particularly useful to model concurrent systems, VASSes can be understood as a restricted version of *counter machines*, a Turing-complete model.

Counter machines

We start by giving the definition of counter machines. The syntax of a counter machine with counters $x_1, \dots, x_k$ is a finite LTS with transitions labels from $\{x_i{+}{+}, x_i{-}{-}, x_i{=}0, x_i{\neq}0 \mid i \in [1, k]\}$. The induced semantics is an infinite transition system with configurations from $Q \times \mathbb{N}^k$ consisting of a control state of the finite LTS and a marking that stores one value for each counter. The

[1] In the literature, the notation $M[t\rangle M'$ is commonly used. The author of this thesis is convinced that $[t\rangle$ is a stylized version of $\boxed{t}$ – the technology just was not there yet.

effect of the transitions is as expected: Transitions labeled by $x_i{+}{+}$ increment the i^{th} counter by one while leaving the other counter values unchanged, transitions labeled by $x_i{-}{-}$ decrement it. Transitions labeled by $x_i{=}0$ or $x_i{\neq}0$ do not change the counter values, but they can only be executed if the value of the i^{th} counter is currently zero or non-zero, respectively.

The decrements $x_i{-}{-}$ are usually defined to be *blocking*: Such a transition can only be executed if the value of counter i is currently strictly positive. With this assumption, one could actually eliminate the occurrences of non-zero tests by replacing them with the sequence $\xrightarrow{x_i{-}{-}}\xrightarrow{x_i{+}{+}}$ of a blocking decrement and of an increment that reverts the effect of the decrement

One can show that counter machines are a Turing-powerful model: A (nondeterministic or deterministic) Turing machine can be converted in polynomial time into a counter machine that essentially has the same algorithmic properties. To be precise, the existence of two counters is sufficient for the conversion. It is not too hard to see that a counter machine with three counters can simulate a Turing-machine: We assume that the tape uses tape alphabet $\{0, 1, \text{␣}\}$ and use one counter to encode the tape content from the start to the current head position (seen as binary number from left-to-right) and a second counter to encode the rest of the tape (seen as binary number from right-to-left). To simulate the tape-manipulating transitions of the Turing machine, the counter values are modified. For example, checking whether the tape content at the current head position is 1 amounts to checking whether the value of the first counter is odd. To preserve the original counter values while conducting such a check, the third counter is used.

To get from three counters down to two counters, one can use Minsky's famous *prime encoding trick* [Min67] which allows us to store the three counter values x_1, x_2, x_3 as the single number $2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3}$. Operations on the counters x_1, x_2, x_3 translate into a sequence of operations on this number, for which the second counter of the two-counter machine is needed as intermediary storage. Counter machines with just one counter are not Turing-powerful: They can be simulated by a pushdown system with a binary stack alphabet: one symbol to store the unary encoding of the current counter value, another symbol that represents the bottom of the stack; the latter is needed for the zero tests.

The above discussion in fact shows a stronger statement: The acceptance problem for Turing-machines with space consumption bounded by n translates into the control-state reachability problem for counter machines with three counters and counter values bounded by 2^n. This complexity-theoretic result does not extend to two-counter machines, since the prime encoding introduces an exponential blowup of the counter values.

Vector addition systems

With the notation at hand, we can define a VASS as a counter machine in which the operations $x_i{=}0$ and $x_i{\neq}0$ do not occur. Unlike a counter machine, a VASS has only limited access to its counter values during runtime. While it is still possible to assert non-zeroness of a counter value by using a blocking decrement, we cannot assert that a counter is zero. We say that the counters of a VASS are *partially blind*.

Similarly, a Petri net can check for the presence of tokens, but it cannot check the absence of tokens on a place. It is not hard to translate a VASS into a Petri net and vice versa. With our definition of VASSes, a transition t of a Petri net would be translated into a sequence of $||\mathrm{in}(t)||_1 + ||\mathrm{out}(t)||_1$ transitions of the VASS, one transition per token that is consumed or produced by t. To obtain a more efficient translation, we could consider VASSes in which transitions can increment or decrement a counter by an arbitrary number in a single step. Since we will not need this construction, we forgo giving the formal definition.

A VAS is a VASS with a unique state. A VASS can be encoded as a VAS with two additional counters that store the current control state. To be precise, to store control state q_i with $i \in [0, k]$, we would have one counter with value i and another *complement counter* with value $k - i$. Just storing i would be insufficient, because then a transition that should be enabled only in control state q_i would also be enabled in any q_j with $j > i$.

6.2 Algorithmic problems for Petri nets

We mention some algorithmic problems for Petri nets that are important in the rest of this thesis.

As usual, automata should have an initial and a final state. In the case of Petri nets, this means we consider *Petri net instances* $(N, M_{\text{init}}, M_{\text{final}})$ consisting of a Petri net N, an initial marking M_{init} for N from which the computation should start, and a final marking M_{final} for N that specifies where the computation should end.

To be able to talk about the computational complexity of these algorithms, we need to define the size of (the encoding) of a Petri net instance. Firstly, we define the size $|M|$ of a marking M as $|M| = \sum_{p \in P}(\lceil \log M(p) \rceil + 1)$. The size $|N|$ of a Petri net $N = (P, T, \text{in}, \text{out})$ is $|N| = \sum_{t \in T} |\text{in}(t)| + |\text{out}(t)|$, where we see $\text{in}(t)$ and $\text{out}(t)$ as markings. Finally, the size $|(N, M_{\text{init}}, M_{\text{final}})|$ of a Petri net instance is defined to be the sum of the size of its components, $|(N, M_{\text{init}}, M_{\text{final}})| = |N| + |M_{\text{init}}| + |M_{\text{final}}|$. Note that we have $|M| \in \mathcal{O}(|P| \cdot (\lceil \log ||M||_\infty \rceil + 1))$ and $|N| \in \mathcal{O}(|P| \cdot |T| \cdot (\lceil \log ||\text{in}||_\infty \rceil + \lceil \log ||\text{out}||_\infty \rceil + 2))$

In the definition of the size, we have measured the size of all numbers via their *binary encoding* which is logarithmic in the absolute value of the number. Altogether, to guarantee that the size of a Petri net instance is polynomial in some number $k \in \mathbb{N}$, it is sufficient to guarantee that its number of places and transitions are polynomial in k and the occurring multiplicities and numbers of tokens are exponential in k.

Sometimes, we will explicitly refer to the unary encoding of a Petri net instance. Formally, the size of the unary encoding of a marking is $|M|_{\text{unary}} = \sum_{p \in P} |M(p)| + 1 = ||M||_1 + |P|$. The definitions of the size of the unary encoding of a net $|N|_{\text{unary}}$ resp. of a net instance are similar to the binary case, with occurrences of $|-|$ replaced by $|-|_{\text{unary}}$.

Reachability and coverability

The most natural algorithmic problem for Petri nets is the *reachability* problem.

Petri net reachability (PNREACH)	
Given:	Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$.
Question:	Is M_{final} reachable from M_{init} in N, i.e. $\exists \sigma \in T^* \colon M_{\text{init}} \xrightarrow{\sigma} M_{\text{final}}$?

Unfortunately, the Petri net reachability problem turned out to be very hard, see the discussion below. To circumvent this problem, one often considers the *coverability problem*.

Petri net coverability (PNCOV)	
Given:	Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$.
Question:	Is there $\sigma \in T^*$ such that $M_{\text{init}} \xrightarrow{\sigma} M'$ with $M' \geqslant M_{\text{final}}$?

We say that a marking M' with $M' \geqslant M_{\text{final}}$ *covers* M_{final}. A computation $M \xrightarrow{\sigma} M'$ with $M' \geqslant M_{\text{final}}$ is called *covering computation*. For the ease of notation, we contract the expression and simply write $M \xrightarrow{\sigma} M' \geqslant M_{\text{final}}$.

The reason for considering coverability instead of reachability is not only that coverability is much simpler to solve. In many applications, it is in fact sufficient to decide coverability. We provide two examples to justify this claim.

For the first example, consider a Petri net N in which the places are states of a concurrent system. The number of tokens that a place carries in a marking is the number of threads that are in a specific state at a certain point in time. (Note that threads that are in the same state are indistinguishable in this model.) A typical verification problem considers one of the states $p_{\text{bad}} \in P$ to be a bad state, and one wants to ensure that from some initial configuration M_{init}, it is not possible for any component to reach state p_{bad}. This is modeled by the coverability problem for $(N, M_{\text{init}}, M_{\text{final}})$, where M_{final} is the unit vector that is zero but for $M_{\text{final}}(p_{\text{bad}}) = 1$. This marking M_{final} is coverable from M_{init} if and only if the system is incorrect (with respect to the specification that p_{bad} is not reachable by any component). Here, we use that it does not matter whether exactly one or more than one component is able to enter the bad state.

The second example is similar. We consider a Petri net modeling a *mutex (mutual exclusion) protocol*. It has a state p_{cs} modeling the critical section. The protocol should guarantee that no two components ever enter the critical section at the same time. We model this by considering coverability with respect to the marking M_{final} that is zero but for $M_{\text{final}}(p_{\text{cs}}) = 2$. Modeling this problem using coverability is suitable because the system is incorrect as soon as any number of components greater than 1 is able to enter the critical section and the same time.

6.2.1 Example

We make the second example explicit. Consider a program that continuously spawns a nondeterministic amount of worker threads that first compute on their own for some time. At some point, each worker thread enters a critical section to transfer the result of its computations to a data structure in shared memory. To ensure mutual exclusion, i.e. only one worker accesses the critical section at the same time, we protect the critical section by a lock which the threads have to acquire. After the threads have transferred their data, they leave the critical section, release the lock and die.

We model this behavior by the Petri net depicted in Figure 6.2.a. Tokens on place p represents worker threads that are not in the critical section; transition t_{spawn} creates such threads. The transition that enters the critical section by moving a token from p to p_{cs} also acquires the lock by moving a token from ℓ_{free} to ℓ_{held}. The transition that kills the thread by removing a token from p_{cs} returns the token from ℓ_{held} to ℓ_{free}. The initial marking puts one token on ℓ_{free}, i.e. the lock is initially not acquired, and no token elsewhere. It is easy to verify that it is impossible to create more than one token in p_{cs} at the same time.

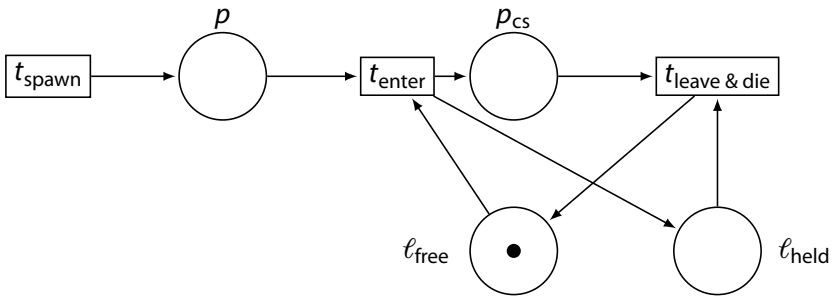


Figure 6.2.a: A Petri net modeling a simple concurrent system.

In the following, we recall classical results on the complexity of coverability and reachability.

Lipton's result

Both coverability and reachability were shown to be EXPSPACE-hard by Lipton [Lip76] (cf. a presentation of the proof by Esparza [Esp98]).

6.2.2 Theorem (Lipton [Lip76])
PNREACH and PNCOV are EXPSPACE-hard.

The hardness of reachability follows directly from the hardness of coverability: Given an instance of the coverability problem, we can construct an equivalent instance of the reachability problem by adding transitions to the net that consume superfluous tokens.

The proof uses a polytime reduction from the acceptance problem for EXPSPACE, which is EXPSPACE-complete. Formally, this problem can be defined similarly to the AEXPSPACE acceptance problem that we have considered in Section 3.3. Firstly, one observes that a Turing machine with exponential space consumption can be simulated by a counter machine with counter values bounded by a number doubly exponential in n. Lipton shows how to simulate such a counter machine with counters bounded by 2^{2^n} with a Petri net of size polynomial in n. The crucial step is the simulation of zero tests that are present in counter machines, but not in Petri nets. To this end, each counter x_i of the machine is represented by two places x_i and $\overline{x_i}$ such that every marking M that is reachable from the initial one satisfies $M(x_i) + M(\overline{x_i}) = 2^{2^n}$. To check that counter x_i is currently zero, we need to check that place $\overline{x_i}$ contains at least 2^{2^n} tokens. Furthermore, the places need to be initialized to carry the correct number of tokens. This is made possible by polynomially sized gadgets that increment or decrement the number of tokens at a place by precisely 2^{2^n}. Since we will need these gadgets in the Sections 8.1 and 14.2, we give a formal specification of their properties in the form of the following proposition.

6.2.3 Proposition
Let $n \in \mathbb{N}$.

a) There is a Petri net instance $(N_{\text{inc}}, M_{\text{initinc}}, \vec{0})$ polynomial in n with two special places p_{haltinc}, p_{outinc} such that any computation $M_{\text{initinc}} \boxed{\sigma} M$ of N_{inc} with $M(p_{\text{haltinc}}) \geqslant 1$ satisfies $M(p_{\text{outinc}}) = 2^{2^n}$.

b) There is a Petri net instance $(N_{\text{dec}}, \vec{0}, M_{\text{finaldec}})$ polynomial in n with a special place p_{indec} such that for any marking M, there is a covering computation $M \boxed{\sigma} M' \geqslant M_{\text{finaldec}}$ if and only if $M(p_{\text{indec}}) \geqslant 2^{2^n}$.

Rackoff's result

When Lipton proved the EXPSPACE-hardness of reachability and coverability, reachability was not known to be decidable. Coverability, however, was already proven to be decidable using a technique by Karp and Miller [KM69] which we will discuss in detail in Section 9.1. The precise complexity was determined by Rackoff [Rac78] who provided an algorithm whose complexity that matches Lipton lower bound.

6.2.4 Theorem (Rackoff [Rac78])
PNCOV can be solved using exponential space.

6.2.5 Corollary
PNCOV is EXPSPACE-complete.

Rackoff's proved this result by showing that if there is a covering computation, then there is one of doubly exponential length. With this bound, one can enumerate and simulate all candidate executions using only exponential space. We will discuss Rackoff's proof in detail in Section 8.1.

Petri net reachability

After an incomplete proof by Sacerdote and Tenney [ST77], Petri net reachability was finally proven to be decidable in 1981 by Mayr [May81].

6.2.6 Theorem (Mayr [May81])
PNREACH is decidable.

Simplified versions of the proof were later published by Kosaraju [Kos82] and Lambert [Lam92]. The algorithm that can be extracted from the proof of decidability is known to be non-primitive recursive. A more precise complexity analysis was presented 2015 by Leroux and Schmitz [LS15b], showing that the running time of the algorithm in the worst case is at least

Ackermann and at most cubic Ackermann, i.e. roughly spoken the Ackermann function applied to itself applied to itself applied to the size of the input.

The gap between the EXPSPACE-hardness proven by Lipton and the non-primitive recursive running time of the only known algorithm remained unclosed for more than 30 years, becoming one of the most important open problems in theoretical computer science. From 2009 to 2012, Leroux published a series of papers, e.g. [Ler11], that contributed new insights about the structure of the set of reachable markings and a new algorithm, but not a better upper bound. In 2019 Leroux and Schmitz [LS19] have shown that the reachability problem can be solved in ACKERMANN time for arbitrary Petri nets (improving the previous analysis of the algorithm), and in primitive recursive time for Petri nets when the number of places is fixed.

Also in 2019, the lower bound was improved by Czerwiński, Lasota, Lazic, Leroux, and Mazowiecki [CLLLM19]. They have shown that Petri net reachability is TOWER-hard. In 2021, the problem has finally been solved. Independently, Leroux [Ler21] and Czerwiński and Orlikowski [CO21] have proven that Petri net reachability is ACKERMANN-complete by providing an ACKERMANN lower bound that matches the earlier upper bound from [LS19].

6.2.7 Theorem (Leroux [Ler21], Czerwiński and Orlikowski [CO21], Leroux and Schmitz [LS19])
PNREACH is ACKERMANN-complete.

6.3 Petri net coverability languages

In the following, we want to see Petri nets as language-generating devices. To this end, we equip the transitions with a labeling.

Formally, a *labeled Petri net* $N = (P, T, \text{in}, \text{out}, \lambda)$ over Σ consists of a Petri net $(P, T, \text{in}, \text{out})$ together with a labeling function $\lambda\colon T \to \Sigma_\varepsilon$. A *computation* $M \boxed{\sigma} M'$ generates the word $\lambda(\sigma)$. Here, we see λ as a function of type $T^* \to \Sigma^*$, namely as the unique homomorphism obtained by extending $\lambda\colon T \to \Sigma_\varepsilon$.

To obtain a language, we equip a net with an initial marking and a set of final markings. Here, we will only consider so-called *coverability languages*, where the set of final markings is the set of markings greater or equal to a specified marking. Formally, the *coverability language* of a labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$ is

$$\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = \{\lambda(\sigma) \mid M_{\text{init}} \boxed{\sigma} M \geqslant M_{\text{final}}\}\,.$$

Petri net coverability languages are strictly less expressive than Petri net reachability languages (which are defined similarly, but the acceptance condition is reaching precisely the specified marking). For example, $\{a^n b^n \mid n \in \mathbb{N}\}$ is a reachability language, but not a coverability language, see Section 6.5. The reason for studying coverability languages is that coverability is sufficient to express the desired behavior in many cases (see above), and that any algorithmic problem for Petri net reachability languages inherits the intractability of PNREACH.

6.3.1 Example

Note that any NFA can be seen as a labeled Petri net instance: The places are the states of the automaton, the initial marking puts a single token on the initial state and no token elsewhere. The idea is that at each point in time, there is a unique place that carries a single token. The transitions are essentially the transitions of the automaton, which specifies their incoming and outgoing multiplicities and their labels. We add a special final place and for each final state of the NFA an ε-labeled transition that consumes a token on that state and produces one on the final place. The final marking requires a token on the final place.

Hence, the Petri net coverability languages are a superset of the regular languages. To see that the inclusion is strict, one can show that the non-regular language $\{a^n b^m \mid n, m \in \mathbb{N}, m \leqslant n\}$ can be generated by a Petri net.

Algorithmic problems for coverability languages

The emptiness problem for Petri net coverability languages is the following problem.

Emptiness problem for Petri net coverability languages (PNCOV-EMPTY)	
Given:	Labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$ over Σ.
Question:	$\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = \varnothing$?

Obviously, a labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$ has non-empty language if and only if M_{final} is coverable from M_{init} in N. Hence, the emptiness problem is essentially equivalent to PNCOV and inherits its complexity, i.e. it is EXPSPACE-complete.

In the following, we want to derive that the *word problem* for Petri net coverability languages is also EXPSPACE-complete.

Word problem for Petri net coverability languages (PNCOV-WORD)	
Given:	Labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$ over Σ, word $w \in \Sigma^*$.
Question:	$w \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$?

Hardness can be easily derived from the hardness of coverability: A normal Petri net can be equipped with the labeling that maps all transitions to ε. The language of this labeled Petri net contains the word ε if and only if the final marking is coverable from the initial one.

For membership, we first explain how to intersect two Petri net coverability languages. Similar to the construction for LTSes, we do this via some sort of product. Unlike in the case of general LTSes, Petri nets have inherent support for concurrency, so we can form the product by taking the disjoint union of places.

For the formal definition, let $N = (P, T, \text{in}, \text{out}, \lambda)$, $N' = (P, T', \text{in}', \text{out}', \lambda')$ be labeled Petri nets over the same alphabet Σ. Assume $P \cap P' = \varnothing$, $T \cap T' = \varnothing$. The *synchronized product* $N \times N'$ of N and N' is the Petri net

$$N \times N' = (P \uplus P', T_\times, \text{in}_\times, \text{out}_\times, \lambda_\times)$$

with $T_\times = \{(t, t') \mid t \in T, t' \in T, \lambda(t) = \lambda'(t') \in \Sigma\} \cup \{t \mid t \in T, \lambda(t) = \varepsilon\} \cup \{t' \mid t' \in T, \lambda'(t') = \varepsilon\}$. The transitions of shape (t, t') behave as t on the P-components of markings and as t' on the P'-components. Formally, $(\text{in}_\times(t, t'))_{\restriction_P} = \text{in}(t)$, similar for P' and similar for out. The ε-labeled transitions behave on the net they come from as before and have no effect on the places of the other net, e.g. for $t \in T$ with $\lambda(t) = \varepsilon$, we have $\text{in}_\times(t, p) = \text{in}(t, p)$ for $p \in P$, and $\text{in}_\times(t, p') = 0$ for $p' \in P'$. The labeling of the transitions is as expected: $\lambda_\times(t, t') = \lambda(t) = \lambda'(t)$, $\lambda_\times(t) = \varepsilon$, $\lambda_\times(t') = \varepsilon$.

For two labeled Petri net instances $(N, M_{\text{init}}, M_{\text{final}})$, $(N', M'_{\text{init}}, M'_{\text{final}})$ over the same alphabet, their synchronized product is $(N \times N', M_{\text{init}\times}, M_{\text{final}\times})$ where $M_{\text{init}\times}$ is equal to M_{init} on P and equal to M'_{init} on P', similar for $M_{\text{final}\times}$.

Intuitively, non-ε-labeled transitions of N and N' have to synchronize in $N \times N'$, while ε-labeled transitions can be fired freely. This corresponds to the definition of the synchronized product for LTSes in Section 4.1. Indeed, the LTS associated to the synchronized product of two Petri nets is exactly the synchronized product of the LTSes associated to the nets. In particular, we obtain that the language of the product is the intersection of the languages as desired,

$$\mathcal{L}\big(N \times N', M_{\text{init}\times}, M_{\text{final}\times}\big) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) \cap \mathcal{L}\big(N', M'_{\text{init}}, M'_{\text{final}}\big)\,.$$

To obtain an EXPSPACE algorithm for PNCOV-WORD, we see the input word w as an NFA A_w with $\mathcal{L}(A_w) = \{w\}$. As in Example 6.3.1, we can see this automaton in turn as a labeled Petri net instance $(N_w, M_{\text{init}w}, M_{\text{final}w})$. Using the previous lemma, we have that $w \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ iff the language of the product of $(N, M_{\text{init}}, M_{\text{final}})$ and $(N_w, M_{\text{init}w}, M_{\text{final}w})$ is non-empty. Since the size of this product is polynomial in the sum of the input sizes (that is, the size of the net and the size of w), applying the EXPSPACE algorithm for coverability yields the desired complexity.

Two useful approximations

We have seen that a Petri net coverability language is not regular in general. In the following, we define both a regular underapproximation and a regular overapproximation of such a language. The approximations are parametric in a number k that determines their precision. Later in the thesis, we will see that by choosing k appropriately, certain structural properties will carry over from the Petri net language to its approximation.

The length-approximation

For a Petri net language $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$, we define its *length-$k$ approximation* by only considering firing sequences of length at most k:

$$\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}}) = \{\lambda(\sigma) \mid M_{\text{init}} \xrightarrow{\sigma} M \geqslant M_{\text{final}}, |\sigma| \leqslant k\}\,.$$

This language is an underapproximation, since $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}}) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ holds for any k. It is also not hard to observe that $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})$ is regular, and its state complexity is exponential in n, the size of $(N, M_{\text{init}}, M_{\text{final}})$.

6.3.2 Proposition
For each k, $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})$ is a regular language of state complexity at most $\mathcal{O}(k^n \cdot 2^{n^2})$, where n is the size of the binary encoding of $(N, M_{\text{init}}, M_{\text{final}})$.

Proof:
We construct an NFA that simulates N for k steps. The NFA is $A = (Q, \to, q_{\text{init}}, Q_F)$, where

$Q = [0,k] \times (P \to [0,(k+1)\cdot 2^n])$, i.e. a state is of the shape (j,M), where j is the number of steps that have been taken and M is a marking that assigns at most $(k+1)\cdot 2^n$ tokens to each place. The initial state is $(0, M_{\text{init}})$, a state (j,M) is final if $M \geqslant M_{\text{final}}$. There is a transition from (j,M) to (j',M'), labeled by a, if $j' = j+1 \leqslant k$ and in N, there is a transition t such that $M \boxed{t} M'$ with $\lambda(t) = a$.

It is obvious that A indeed simulates precisely the covering computations induced by firing sequences of length at most k. The maximum number that we can encode with n bits is 2^n, so a Petri net instance of size n can add at most 2^n tokens to each place per transition. Since we perform at most k transitions, each reachable state (j,M) has $M(p) \leqslant M_{\text{init}}(p) + k\cdot 2^n \leqslant (k+1)\cdot 2^n$, i.e. we stay within the bounded state space.

To prove the statement on the state complexity, we count the number of states of A:

$$|Q| = (k+1)\cdot(((k+1)\cdot 2^n)+1)^{|P|} \leqslant (k+1)\cdot(2\cdot(k+1)\cdot 2^n)^n \leqslant 2k\cdot 2^n\cdot 2^{2n}\cdot k^n\cdot(2^n)^n \in \mathcal{O}\left(k^n\cdot 2^{n^2}\right).$$

■

The ω-approximation

In addition to the above underapproximation, we also define a regular overapproximation of a Petri net language. It only tracks the token count on each place up to a bound k that is the parameter of the approximation. If the token count ever exceeds k, the value of the place is set to ω (read as: unbounded) and remains so for the rest of the computations. All transitions are enabled with respect to places that have become unbounded.

More formally, a *generalized marking* is a vector $M \in \mathbb{N}_\omega^P$ with entries in $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$. We extend the order to generalized markings by setting $x \leqslant \omega$ for all $x \in \mathbb{N}_\omega$. Furthermore, we define $\omega + n = \omega - n = \omega$ for all $n \in \mathbb{N}$. The firing relation extends to generalized markings in the natural way: Transition t is enabled in M if $M \geqslant \text{in}(t)$, and firing it yields $M + e(t)$. Note that $\text{in}(t)$, $\text{out}(t)$ and $e(t)$ are vectors over the integers, so there is no need to define the values of $\omega + \omega$ and $\omega - \omega$.

For a number $k \in \mathbb{N}$, we define the $[0,k]$-ω-*approximation* of a Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$ as the finite automaton $A_{>k}$ with state space $([0,k] \cup \{\omega\})^P$. Its initial state is the marking M'_{init} with $M'_{\text{init}}(p) = M_{\text{init}}(p)$ if $M_{\text{init}}(p) \leqslant k$ and $M'_{\text{init}}(p) = \omega$ else, and the final states are $\{M \in ([0,k] \cup \{\omega\})^P \mid M \geqslant M_{\text{final}}\}$. Its transition relation is induced by the firing relation of N: We have $M \xrightarrow{a} M'$ if there is an a-labeled transition of N such that $M \boxed{a} M''$ with $M'(p) = M''(p)$ if $M''(p) \leqslant k$ and $M'(p) = \omega$ else. The $[0,k]$-ω-*approximation* $\mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$ of a Petri net language $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ is defined to be the language of $A_{>k}$.

6.3.3 Lemma

For each k, $\mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$ is a regular overapproximation of $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ with state complexity at most $2^n k^n$, where n is the size of $(N, M_{\text{init}}, M_{\text{final}})$.

Proof:

Language $\mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$ is trivially regular since it is defined by a finite automaton. Observe that any covering computation of N induces an accepting run of $A_{>k}$ generating the same word, so $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) \subseteq \mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$ holds. For the state complexity, note that the number of generalized markings in which the natural entries are bounded by k (for $k \geqslant 2$) is

$$\left|([0, k] \cup \{\omega\})^P\right| = (k + 2)^P \leqslant 2^n k^n \ .$$

■

6.4 BPP nets

The hardness of most computational problems for Petri nets motives studying subclasses of nets with better algorithmic properties. There are several classes of nets that are studied in this context, including *safe*, *communication-free*, *conservative* Petri nets and many others.

In a *k-safe* net instance, we assume that each marking M reachable from the initial marking satisfies $M(p) \leqslant k$ for all places p. The case of 1-safety (often simply called *safety*) is of particular interest. The state space of a k-safe net instance is finite; it is a transition system with at most $k^{|P|}$ configurations. In particular, the computational problems for k-safe Petri nets are as hard as the corresponding problems for finite state systems of exponential size. For example, the reachability problem for (1-)safe Petri nets is PSPACE-complete, which corresponds to the reachability problems in directed graphs. The latter can be solved in NL (nondeterministic logarithmic space) for polynomially sized graphs, and hence in polynomial space for exponentially sized graphs. Altogether, this class is of high practical relevance, but rather boring from a theoretical point of view.

In this thesis, we will, in addition to general Petri nets, exclusively focus on the following subclass. A Petri net N is a *BPP net* or *communication-free* if each transition consumes at most one token, i.e. for each t, we have $\sum_{p \in P} \mathrm{in}(t, p) \leqslant 1$.

BPP nets are interesting for various reasons. On the practical side, a BPP net can model an unbounded number of instances of a finite state-automaton running in parallel. Each token in a place corresponds to an instance of the automaton that is in said place. Instances can *die* (when the corresponding token is consumed), and new instances can be spawned at runtime (by transitions that produce more than one token). However, the instances cannot communicate or synchronize during runtime. This setting is modeled by the *calculus of basic parallel processes* [Chr93], from which BPP nets get their name.

On the theoretical side, BPP nets are special in that they have some structural properties that do not hold for general Petri nets: The state equation is an equivalence, and the reachability set is semi-linear. After giving an example, we discuss each of these properties in detail.

6.4.1 Example

Consider the labeled BPP net instance $(N, \vec{0}, \vec{0})$ over the alphabet $\{a, b, c\}$, where N is given in Figure 6.4.a. Its language is the non-context-free language of all words $w \in \{a, b, c\}^*$ such that in every prefix of w, the number of occurrences of letter a is greater than or equal to the number of occurrences of b, and the number of bs is greater than or equal to the number of cs. The fact that this language is not context-free can be shown similar to the well-known proof for $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ not being context-free.

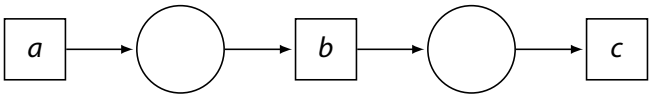


Figure 6.4.a: A labeled BPP net with a non-context-free language.

The state equation

Recall that $M \boxed{\sigma} M'$ implies $M' = M + e(\sigma)$. In turn, this means that a marking M' can only be reachable from M if there is a sequence of transitions with effect $M' - M$. Since the effect of a transition sequence does not depend on the order of the transitions, this can be described by a system of linear equations. If M' is reachable from M, then the system of equations given by

$$M' - M = \sum_{t \in T} c_t \cdot e(t)$$

has a non-negative solution $c \in \mathbb{N}^T$. This equation is called the *state equation* (or *marking equation*). It can be used as an easy preliminary check to rule out reachability: If the system has no solution, reachability cannot hold. Checking whether the state equation has a non-negative integer solution is an instance of the satisfiability problem for integer programming, which is NP-complete [Kar72; BT76]. However, the state equation does not provide a characterization of Petri net reachability: Firstly, it is not hard to construct an example where the system of equations has a solution, but it is impossible to arrange the transitions such that a valid firing sequence is obtained. Secondly, even if reachability holds, the shortest valid firing sequence may be vastly longer, i.e. it contains much more transitions, than the transition counts provided by the least solution to the state equation. In fact, the decision procedure for Petri net reachability in its presentation by Lambert [Lam92] can be understood as a procedure that expands the given Petri net until the satisfiability of the marking equation is equivalent to the reachability Problem. Note that this expansion process can lead to a blow-up that is non-primitive recursive.

For BPP nets, however, the characterization of reachability by the state equation is precise in the following sense.

6.4.2 Theorem (Esparza [Esp97])
For a BPP net N, an initial marking M and a vector $c \in \mathbb{N}^T$ of transition counts, there is a valid firing sequence σ with $M \boxed{\sigma}$ containing each transition t exactly c_t times if and only if

(1) the state equation holds, i.e. $M_{\text{init}} + \sum_{t \in T} c_t \cdot e(t) \geqslant 0$ and

(2) the net is connected in the sense that for each place p such that there is a transition t with $c_t > 0$ and $\text{in}(t, p) \neq 0$ or $\text{out}(t, p) \neq 0$, there is a path from a place p' with $M_{\text{init}}(p') > 0$ to p that only uses transitions t' with $c_{t'} > 0$ (in the Petri net seen as graph).

This characterization has two important consequences that are stated and proven in [Esp97]. Firstly, one obtains that, given a BPP net N and a marking M_{init}, the set of reachable markings $\{M \mid \exists \sigma\colon M_{\text{init}} \boxed{\sigma} M\}$ is effectively semi-linear. Secondly, it yields an NP-algorithm for reachability. The first step of this algorithm is to get rid of the dependency on c in the theorem.

6.4.3 Corollary (Esparza [Esp97])
For a BPP net N, and markings $M_{\text{init}}, M_{\text{final}}$, there is a computation $M_{\text{init}} \boxed{\sigma} M_{\text{final}}$ if and only if there is a subset of transitions $U \subseteq T$ such that

(1) the system of equations formed by the state equation, i.e. $M_{\text{init}} - M_{\text{final}} = \sum_{t \in T} c_t \cdot e(t)$, the equations $c_t = 0$ for $t \notin U$, and the inequalities $c_t > 0$ for $t \in U$ has an integer solution c, and

(2) the net is connected in the sense that for each place p such that there is a transition $t \in U$ that is adjacent to p (in the net seen as graph), there is a path from a place p' with $M_{\text{init}}(p') > 0$ to p that only uses transitions from U.

The corollary gives rise to an algorithm for reachability in a straightforward manner. One first guesses U, then checks the second property by a linear number of reachability checks in a directed graph, and finally solves the extended state equation using integer linear programming. As mentioned before, linear programming is NP-complete, so the whole algorithm runs in NP.

By inserting transitions that consume superfluous tokens, an instance of the coverability problem for BPP nets can be easily reduced to an instance of reachability. In fact, both coverability and reachability are NP-complete [Esp97]. Since we will later present some hardness proofs that rely on the NP-hardness of coverability, we give the proof.

6.4.4 Lemma
Coverability in BPP nets is NP-hard.

Proof:
We reduce from SAT, the satisfiability problem for propositional formulas in conjunctive normal form. Let $F = K_1 \wedge \ldots \wedge K_n$ be the given formula with $K_i = L_{i1} \vee \ldots \vee L_{im_i}$ for each clause K_i. Each literal is of the shape $L_{ij} = x_k$ or $L_{ij} = \neg x_k$ for one of the variables $x_1, \ldots, x_\ell$. We construct a net that has three places x_i, x_i^+, x_i^- for each variable. The initial marking M_{init} puts a token on each place x_i. For each variable, there is one transition moving that token to x_i^+, and one that moves it to x_i^-. Intuitively, one assigns a truth value to variable x_i using these transitions.

To encode the formula, we introduce places for all of its parts. For each positive literal $L_{ij} = x_k$, there is a place L_{ij} and a transition that checks for a token x_k^+ and produces a token on L_{ij}. By checking for a token, we mean that the transition both consumes and produces a token on this place; the transition does not change the number of tokens on that places, but it requires a

token to be able to be fired. Similarly, there is a place for negative literals $L_{ij} = \neg x_k$ that check for a token on x_k^-. For each clause K_i, there is a place K_i. Every literal L_{ij} from that clause induces a transition that moves one token from L_{ij} to K_i.

The final marking M_{final} requires one token on each place K_i. Each covering computation induces a satisfying truth assignment and vice versa: If $M_{\text{init}} \xrightarrow{\sigma} M \geqslant M_{\text{final}}$, define the assignment φ by $\varphi(x_k) = \text{true}$ if σ contains the transition moving a token from x_k to x_k^+, $\varphi(x_k) = \text{false}$ else. The structure of the net ensures that (1) no variable is set to true and false at the same time, i.e. the places x_k^+ and x_k^- are mutually exclusive, (2) each clause is satisfied, since (3) each clause contains at least one literal that is satisfied. For the other direction, assume that φ is a satisfying truth assignment, and consider the computation φ that first moves tokens to the places x_k^- or x_k^+, depending on the truth value of $\varphi(x_k)$, then creates tokens on all literals L_{ij} that are satisfied, and then creates tokens on all clauses that are satisfied. Since $\varphi(F) = \text{true}$ by assumption, this computation will create a token for each clause and hence is covering. ■

6.4.5 Corollary
Coverability and reachability in BPP nets are NP-complete.

The word problem for BPP nets

Let us now study the class of BPP net coverability languages and its algorithmic properties, starting with the word problem. The class of *BPP (coverability) languages* is not well-studied yet. To the best of our knowledge, this is one of the first works that considers this class. BPP nets are commonly used to model the languages of commutative context-free grammars. We comment on this class of languages at the end of this section. In particular, we will argue that the theory of the languages of commutative context-free grammars is less rich than the theory of the BPP net languages.

Word problem for BPP net coverability languages (BPPCOV-WORD)	
Given:	Labeled BPP net instance $(N, M_{\text{init}}, M_{\text{final}})$ over Σ, word $w \in \Sigma^*$.
Question:	$w \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$?

In the case of Petri nets, the complexity results for unlabeled Petri nets directly translate into complexity results for the corresponding problems for languages. Unfortunately, some constructions do not carry over to BPP nets. For example, to show that the word problem for general Petri nets is EXPSPACE-complete, we have constructed the product of the given net and an NFA for the given word. This construction cannot be used for BPP nets, since the product of a BPP net with an NFA is not a BPP net: Each transition will consume two tokens, one from the BPP net and one from the automaton. Nevertheless, the word problem is NP-complete, just as the coverability problem.

6.4.6 Proposition
BPPCOV-WORD is NP-complete.

The hardness of the word problem is easy to obtain: Consider an unlabeled BPP net as a labeled BPP net in which every transition is labeled by ε, then the final marking is coverable in the original net if and only if ε is an element of the coverability language of the labeled version. Hence, the NP-hardness result for coverability implies the NP-hardness of BPPCOV-WORD.

The proof for membership in NP is harder. To get the desired result, we use the correspondence of BPP nets to *Presburger arithmetic* and the concept of semi-linearity that we have introduced in Section 3.4.

6.4.7 Theorem (Verma, Seidl, and Schwentick [VSS05]; Esparza [Esp97])
The reachability set of a BPP is effectively semi-linear: Given a BPP net N and an initial marking M_{init}, one can compute in polynomial time an existential Presburger formula $\Psi(P)$ so that for all markings M: $\Psi(M)$ is true iff $M_{\text{init}} \xrightarrow{\sigma} M$ for some $\sigma \in T^*$.

The statement is not true for general Petri nets with at least 6 places [HP79]. We can now use Theorem 6.4.7 to prove that the word problem for BPP net coverability languages is in NP.

6.4.8 Proposition
BPPCOV-WORD is in NP.

Proof:
Assume that $(N, M_{\text{init}}, M_{\text{final}})$ is the given labeled BPP net instance and $w = w_1 \dots w_m \in \Sigma^*$ is the word for which membership should be checked. We could simply guess for each letter w_i of w the transition t_i of N with $\lambda(t_i) = w_i$ that will be used to produce w_i. The problem is that a computation $M_{\text{init}} \xrightarrow{\sigma} M \geqslant M_{\text{final}}$ with $\lambda(\sigma) = w$ can use an unbounded number of ε-transitions in between the occurrences of the t_i. Hence, we have to find a computation

$$M_{\text{init}} \xrightarrow{\sigma_0} M_1 \xrightarrow{t_1} M_1' \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_m} M_m \xrightarrow{t_m} M_m' \xrightarrow{\sigma_{m+1}} M_{m+1} \geqslant M_{\text{final}}$$

where each σ_i for $i \in [1, m+1]$ only consists of ε-labeled transitions.

However, we cannot simply guess the markings M_i in polynomial time, since we have no a priori bound on their number of tokens. To overcome this issue, the characterization of reachability in terms of existential Presburger formulas is crucial. Our goal is to design an existential Presburger formula $\varphi_{(N,M_{\text{init}},M_{\text{final}}),w}$ that is satisfiable if and only if $w \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$,

On a high level of abstraction, the formula guesses the transitions t_i that are used for the letters of the word. It also guesses the markings M_i and M_i' and verifies that each M_i' results from M_i

by applying the effect of transition t_i, and that each M_{i+1} is reachable from M'_i by a sequence of ε-transitions. To implement this idea, we let N_ε be a version of N in which all transitions that are not labeled by ε have been removed. We create $m+2$ copies of N_ε, say $N_\varepsilon^0, N_\varepsilon^i, \ldots, N_\varepsilon^m, N_\varepsilon^{m+1}$. Intuitively, the sequence σ_i of ε-transitions will be executed in copy N_ε^i. This is needed since the Presburger formula can only talk about the marking that is reached in the end, so we will need to preserve the intermediary markings in a separate copy to be able to access them.

However, we need to ensure that each copy N_ε^i starts from the correct initial marking. To this end, we add transitions that allow us to populate N_ε^i for $i > 0$ with an arbitrary initial marking. Our formula will check that the marking reached by firing σ_i plus the effect of transition t_{i+1} is indeed the initial marking of N_ε^{i+1} from which σ_{i+1} is fired. To implement this, we let $N_\varepsilon^{1,\text{init}}, \ldots, N_\varepsilon^{m,\text{init}}, N_\varepsilon^{m+1,\text{init}}$ be copies of N that do not contain any transition. We let N' be the disjoint union of all N_ε^i for $i \in [0, m+1]$ and all $N_\varepsilon^{i,\text{init}}$ for $i \in [1, m+1]$. For each $i \in [1, m+1]$, and each place p of N, we add a transition that simultaneously adds a token in place p of copy N_ε^i, and place p in copy $N_\varepsilon^{i,\text{init}}$. Furthermore, we let M' be the initial marking of N' that assigns $M_{\text{init}}(p)$ tokens to the places p of the first copy N_ε^0 and no tokens elsewhere.

It remains to construct the Presburger formula $\varphi_{(N,M_{\text{init}},M_{\text{final}}),w}$ that is satisfiable if and only if $w \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$. The free variables of $\varphi_{(N,M_{\text{init}},M_{\text{final}}),w}$ will correspond to the markings M_i and M'_i in the desired computation. The formula

(1) checks that the markings correspond to a valid computation of net N',

(2) it guesses the transitions t_i used for each of the letters of w, checks that the marking reached in copy N_ε^{i-1} enables transition t_i, and that for each copy $i > 0$, the initial marking of N_ε^i, which is stored in copy $N_\varepsilon^{i,\text{init}}$, is equal to the marking reached in the copy N_ε^{i-1} plus the effect of t_i, and

(3) it checks that the final marking reached in the last copy N_ε^{m+1} covers M_{final}.

For the formal construction, we let p^i and $p^{i,\text{init}}$ denote the copy of place p of N in copy N_ε^i and $N_\varepsilon^{i,\text{init}}$, respectively. In the formula, we will use the name of each place as the variable describing the number of tokens on that place. For vectors of variables $\vec{y}, \vec{z}$ of equal dimension k, we will write $\vec{y} \leqslant \vec{z}$ for the conjunction $y_1 \leqslant y'_1 \wedge \ldots \wedge y_k \leqslant y'_k$. For two variables y_i, z_i, we write $y_i = z_i$ for $y_i \leqslant z_i \wedge z_i \leqslant y_i$. We extend this notation to vectors and write $\vec{y} = \vec{z}$ for $\vec{y} \leqslant \vec{z} \wedge \vec{z} \leqslant \vec{y}$.

We define the formula $\varphi_{(N,M_{\text{init}},M_{\text{final}}),w}$ to be

$$\varphi_{(N,M_{\text{init}},M_{\text{final}}),w} = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \,,$$

where each φ_i expresses Property (i) from above. (To be precise, φ should be in prenex normal form; however, it will be easy to achieve this without a blowup in size.)

We define φ_1 to be the formula characterizing reachability in net N' from marking M'. It exists, is of polynomial size, and can be computed in polynomial time by Theorem 6.4.7.

Formula φ_2 is defined as follows:

$$\varphi_2 = \bigwedge_{i\in[1,m]} \bigvee_{\substack{t_i \\ \lambda(t_i)=w_i}} \bigwedge_{p\in P} \left(p^{i-1} \geqslant \mathrm{in}(t_i,p)\right) \wedge \left(p^{i,\mathrm{init}} = p^{i-1} + e(t_i)\right).$$

For each of the letters w_i, it guesses a transition t_i with the correct label using a disjunction, and then checks Property (2). Note that $e(t_i)$ can be negative, but we have not allowed subtraction resp. negative values in Presburger terms. However, we may bring it to the other side of the equality, which allows us to replace subtraction by addition.

The construction of formula φ_3 is straightforward

$$\varphi_3 = \bigwedge_{p\in P} p^{m+1} \geqslant M_{\mathrm{final}}(p)$$

Obviously, $\varphi_{(N,M_{\mathrm{init}},M_{\mathrm{final}}),w}$ is of size polynomial in N and w. It remains to argue that the formula is indeed satisfiable iff $w \in \mathcal{L}(N, M_{\mathrm{init}}, M_{\mathrm{final}})$. For one direction, assume that $M_{\mathrm{init}} \xrightarrow{\sigma_0} M_1 \xrightarrow{t_1} M'_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_m} M_m \xrightarrow{t_m} M'_m \xrightarrow{\sigma_{m+1}} M_{m+1} \geqslant M_{\mathrm{final}}$ is a computation generating word $\lambda(\sigma)$. We obtain a satisfying assignment for formula φ by setting p^i to the value M_{i+1} reached by firing σ_i, and by setting $p^{i,\mathrm{init}}$ to the value $M'_i(p)$.

For the other direction, note that a satisfying assignment of the formula is a witness for the existence of a computation σ' of N' from M' such that the markings reached in each of the copies satisfy the properties postulated by the formula. Since the copies are independent, we can wlog. assume that σ' has been rearranged such that

$$\sigma' = \sigma'_0.\tau_1.\sigma'_1\tau_2 \ldots \tau_{m+1}.\sigma_{m+1}$$

where each σ'_i exclusively contains transitions from copy N^i_ε, and each τ_i contains transitions that initially populate N^i_ε and $N^{i,\mathrm{init}}_\varepsilon$. We define σ_i to be obtained from σ'_i by projecting the transitions in N^i_ε back to the transitions in N. Let us denote by t_i a transition with label w_i such that the corresponding disjunct of the formula φ_2 is satisfied. The computation $M_{\mathrm{init}} \xrightarrow{\sigma_0} M_1 \xrightarrow{t_1} M'_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_m} M_m \xrightarrow{t_m} M'_m \xrightarrow{\sigma_{m+1}} M_{m+1} \geqslant M_{\mathrm{final}}$ of net N produces word w as desired. ■

Commutative context-free grammars

We discuss how BPP nets and their languages relate to the class of languages of commutative context-free grammars. A *commutative context-free grammar* is a CFG as defined in Section 5.1 for which we identify the terminal words that it produces that are equal up to commutativity. This can be formalized in various ways. One possibility is to use the *Parikh image* $\Psi(w)$ of a

word $w \in \Sigma^*$, the vector in $\mathbb{N}^\Sigma$ that counts for each letter its number of occurrences in w. By generalizing the definition to languages, we may define the language of a commutative context-free grammar G as $\Psi(\mathcal{L}(G)) = \{\Psi(w) \in \mathbb{N}^\Sigma \mid w \in \mathcal{L}(G)\}$, where $\mathcal{L}(G)$ is the language of G seen as CFG. Alternatively, we may define the language via the *commutative closure* $\mathrm{commcl}(\mathcal{L}(G))$, where $\mathrm{commcl}(\mathcal{L}) = \{w \in \Sigma^* \mid \exists v \in \mathcal{L} \colon \Psi(w) = \Psi(v)\}$ adds all words to a language that are equal to a word from the language up to commutativity. Here, we have used that equality up to commutativity means that the number of occurrences of each letter is equal.

The correspondence between commutative context-free grammars and BPP nets has first been observed by Esparza [Esp97]. To see a commutative context-free grammar as a BPP net, we introduce one place for each nonterminal and each terminal. A production $X \to \eta$ of the grammar induces a transition in the net that consumes one token from the place for X and for each occurrence of a symbol in η produces one token on the corresponding place. The initial marking M_{init} puts one token on the place for the initial symbol and no tokens elsewhere. With this construction at hand, we recover the language of the context-free grammar as

$$\left\{ M_{\restriction_\Sigma} \;\middle|\; \exists \sigma \in T^* \colon M_{\text{init}} \xrightarrow{\sigma} M, M \in \mathbb{N}^\Sigma \times \{\vec{0}\} \right\},$$

the set of all markings reachable from M_{init} in which there are not tokens on the places for the nonterminals, restricted to the places for the terminal symbols. The fact that each production in a context-free grammar replaces exactly one nonterminal means the net is indeed a BPP net. In particular, the results on the structure of BPP nets apply which means that the set as defined before is effectively semi-linear.

On the practical side, this correspondence has been used to develop an algorithm that in linear time constructs a formula in existential Presburger arithmetic that describes the Parikh image of a context-free grammar [VSS05]. On the theoretical side, Parikh's theorem [Par66] states that the classes of Parikh images of context-free languages and regular languages coincide; both are equal to the class of semi-linear sets of vectors. However, context-free grammars may give a representation that is more succinct: For each $n \in \mathbb{N}$, there is a context-free grammar over the alphabet $\{a\}$ of size polynomial in n that generates the Parikh image $\{2^n\}$; the same is not true for finite automata or regular expressions. It is noteworthy that some classes of languages that we have introduced are not closed under taking the commutative closure: The commutative closure of the regular language $(abc)^*$ is the non-context-free language $\mathrm{commcl}((abc)^*)$ of words in which all three letters appear equally often. Similarly, the commutative closure of the non-context-free language $\{w.w \mid w \in \{a,b\}^*\}$ is the context-free (and even regular) language in which the number of occurrences of each of the two letters is even.

We show how we can represent commutative context-free languages with our definition of the languages of BPPs that is based on labeled transitions. If we consider *BPP reachability languages* instead of coverability languages, it is easy to design a BPP net whose language is the commutative closure of the language of a given context-free grammar. We use the aforementioned construction by Esparza and make all transitions ε-labeled. Then, we insert for each terminal

symbol $a \in \Sigma$ an a-labeled transition that consumes a token from the corresponding place. The final marking of that net requires no tokens to be present. The language of this BPP net is exactly the commutative closure of the language of the given CFG: The final marking enforces that the derivation process has been completed, i.e. no nonterminals remain, as well as that all terminal symbols have been taken into account. The fact that the transitions that are not ε-labeled are independent of each other ensures that the language is commutatively closed.

It seems difficult to obtain a similar construction with coverability as the acceptance condition. While coverability and reachability in BPP nets are both NP-complete, which means that there is a polytime reduction from reachability to coverability, to the best of the author's knowledge, there is no known reduction that preserves the language of the net.

Nevertheless, we can still represent all languages of commutative context-free grammars in the following sense: For every language of a context-free grammar there is a BPP net whose coverability language has the same Parikh image. To see this, we use the aforementioned result that shows that for every CFG, there is an NFA with the same Parikh image. In Example 6.3.1, we have shown how to see an NFA as a Petri net with the same language. This net is a BPP net. Hence, there is a BPP net with the same Parikh image.

For the other direction, we observe that the Parikh images of all BPP net languages are semi-linear (and hence the Parikh image of a context-free or regular language). To see this, we add places to a given net that keep track of the letters that have been produced. Similar to Esparza's construction, we let each a-labeled transition produce an additional token on the place for letter a. As mentioned before, the set of reachable markings in a BPP net is effectively semi-linear. It is easy to extend the Presburger formula by conjunctions that make sure that the final marking has been reached or covered. Finally, the semi-linear sets can be shown to be closed under projection. We apply this to project to the components that count the occurrences of letters. Altogether, we obtain that the Parikh image of a BPP net language (with either coverability or reachability as acceptance condition) is semi-linear.

In summary, BPP net coverability languages can represent semi-linear sets, and their Parikh images are always semi-linear. The same holds true for both the context-free and the regular languages. It makes sense to consider the class of languages of BPP nets without applying the Parikh image abstraction (or, equivalently, the commutative closure): With Example 6.4.1 at hand, one can show that it is incomparable to both aforementioned classes.

6.5 Well-structured transition systems

We present *well-structured transition systems (WSTSes)*, a generalization of Petri nets with coverability as the acceptance condition. In Section 4.1, we have explained that decidability results for automata models are usually based on the fact that the transitions are only allowed to depend on the memory in a way that is local. In the case of WSTSes, this locality is formalized by requiring an order on the states that is respected by the transition relation. Our presentation loosely follows [FS01].

Upward and downward closures

We start by introducing some notation. Let $(X, \leqslant)$ be a quasi-ordered set. For a subset $Y \subseteq X$, its *downward closure*

$$Y{\downarrow_{\leqslant}} = \{x \in X \mid \exists y \in Y \colon x \leqslant y\}$$

is the set of all elements smaller than some element of Y. The set Y is called *downward closed* if it contains all elements that are smaller than some element. We may formalize this by requiring that Y equals its downward closure, $Y = Y{\downarrow_{\leqslant}}$.

The notions of the *upward closure* $Y{\uparrow_{\leqslant}} = \{x \in X \mid \exists y \in Y \colon y \leqslant x\}$ and of being *upward closed* are defined similarly. If the order is clear from the context, we omit the corresponding subscript.

The complement of an upward-closed set is a downward-closed set and vice versa. Taking the upward closure commutes with unions, $(Y \cup Y'){\uparrow} = Y{\uparrow} \cup Y'{\uparrow}$, similarly for the downward closure.

A *set of minimal elements* for a set Y is a subset $B \subseteq Y$ such that B is an antichain, and for each $y \in Y$, there is some $b \in B$ with $b \leqslant y$. The second condition may be rephrased as $B{\uparrow} = Y{\uparrow}$. We are particularly interested in the case where Y is upward closed and B is finite, $B = \{b_1, \ldots, b_k\}$. In this case, we call B a *basis* of Y and we have $Y = b_1{\uparrow} \cup \ldots \cup b_k{\uparrow}$. Here and in the following, we omit the set-brackets when taking the upward closure or downward closure of single elements.

We define the operator min that takes a set Y and returns $\min Y$, a set of minimal elements of Y of minimum size. In particular, if a finite set of minimal elements exists, it will return one. Since we have not required the order to be antisymmetric, the set of minimal elements does not have to be unique. For us, it will suffice that min returns some arbitrary set of minimal elements.

Ordered LTS

An *ordered LTS* over Σ is of the shape $W = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$, where $(\Gamma, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ is an LTS with labels from Σ as defined in Section 4.1, $\leqslant \subseteq \Gamma \times \Gamma$ is a quasi-order on configurations (i.e. it is reflexive and transitive), the set of final configurations is upward-closed, i.e. $\Gamma_{\text{final}} = \Gamma_{\text{final}}{\uparrow}$, and the transition relation is compatible with respect to $\leqslant$ in the following sense: If $c \xrightarrow{a} d$, and $c \leqslant c'$, then $c' \xrightarrow{a} d'$ for some d' with $d \leqslant d'$. We say that T is *upward-compatible* with $\leqslant$, or that $\leqslant$

is a *simulation relation*, since intuitively, the condition states that larger states can simulate the behavior of smaller states.

Requiring an LTS to be ordered is actually not a real restriction: Any LTS is an ordered LTS with respect to equality as order. In this case, upward-compatibility is trivially satisfied since $c \leqslant c'$ means $c = c'$ in this case, and we have $S = S{\uparrow}$ for any set $S \subseteq \Gamma$ of configurations. To get a model with good properties, we have to restrict the order.

Well-quasi orders

The crucial restriction is requiring the order to be a well-quasi-order. Before finally giving the definition of WSTSes, we define such orders and list some of their properties.

Consider a quasi-ordered set $(X, \leqslant)$. We call it a *well-quasi-order* (WQO) if any infinite sequence $x_0, x_1, x_2, \ldots$ of elements of X contains an *increasing pair*, i.e. there are indices $i < j$ with $x_i \leqslant x_j$.

6.5.1 Example

a) Equality on a set, i.e. the order $(X, =)$, is a WQO if and only if X is finite. Indeed, if X is finite, any infinite sequence has to contain a repetition of elements. If X is infinite, then any infinite sequence without repetitions is a *bad sequence*, a sequence without increasing pair.

b) The usual order $(\mathbb{N}, \leqslant)$ is a WQO: Any sequence $n_0, n_1, n_2, \ldots$ of numbers contains an increasing pair. In fact, the increasing pair has to occur within the first $n_0 + 2$ indices: After starting with n_0, there are only $|[0, n_0 - 1]| = n_0$ distinct numbers that are smaller than n_0. Either the first $n_0 + 2$ entries of the sequence already contain the repetition of a number, which constitutes an increasing pair, or n_{n_0+2} or an earlier entry is larger than n_0.

The generalized version of Dickson's Lemma [Dic13] states that the product of WQOs is a WQO.

6.5.2 Lemma (Dickson [Dic13])

If $(X_1, \leqslant_1), (X_2, \leqslant_2)$ are WQOs, then the product order $(X_1 \times X_2, \leqslant_\times)$ is a WQO.

Consequently, $(\mathbb{N}^k, \leqslant)$ is a WQO for each $k \in \mathbb{N}$.

WQOs admit many different characterizations. We list some of them in the following lemma.

6.5.3 Lemma
Let $(X, \leqslant)$ be a quasi-order. The following are equivalent:

(1) $(X, \leqslant)$ is a WQO.

(2) Every infinite sequence in X contains an infinite ascending subsequence.

(3) Every subset of X has a finite set of minimal elements.

(4) All antichains and strictly descending sequences in X are finite.

(5) Every infinite ascending chain of upward-closed sets $U_0\uparrow \subseteq U_1\uparrow \subseteq U_2\uparrow \subseteq \dots$ gets stationary, i.e. there is $i \in \mathbb{N}$ such that $U_i\uparrow = U_{i+1}\uparrow$.

(6) Every strictly ascending chain of upward-closed sets $U_0\uparrow \subsetneq U_1\uparrow \subsetneq U_2\uparrow \subsetneq \dots$ is finite.

For the proof we refer to [FS01]. The equivalence of the Properties (1) and (2) is sometimes credited to an unpublished draft by Erdős and Rado. The last two properties can also be phrased in terms of (strictly) descending chains of downward closed sets.

With Property (3), we have that $\min Y$ is finite for all $Y \subseteq X$. If Y is upward closed, $\min Y = \{b_1, \dots, b_k\}$ is a basis with $Y = b_1\uparrow \cup \dots b_k\uparrow$. Note that this property cannot be dualized: It is not true that in a WQO, every downward-closed set can be represented by a finite set of maximal elements. For example, the downward-closed set $\mathbb{N}$ in the WQO $(\mathbb{N}, \leqslant)$ does not occur as the downward closure of any finite set of numbers. In many cases, this problem can be overcome by considering ideals, which we will do in Section 13.3.

Well-structured transition systems

With all preliminaries at hand, we can finally give the crucial definition. We call an ordered LTS $W = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ a *well-structured transition system (WSTS)* if $(\Gamma, \leqslant)$ is a WQO.

The notions that we have introduced for general LTSes, e.g. the notion of being finitely branching, carry over to ordered LTSes and WSTSes.

In the definition of WSTSes, the WQO restriction and upward-compatibility come together to create a model that has nice structural properties. Upward compatibility states that large states can simulate the behavior of smaller states; the underlying order being a WQOs means that this will eventually be possible in a computation that is long enough. To obtain algorithms, e.g. for coverability, it is sufficient to impose very mild restrictions: In [FS01], the authors show that it is sufficient that the order is decidable (i.e. one can check, given two configurations, whether one is smaller than the other), and that for each configuration c, one can compute $\min \operatorname{pre}(\Sigma, c\uparrow)$, a finite basis for the predecessors of the upward closure of c.

WSTSes have their origins in a collection of papers [Fin87; Fin90; AJ93; ACJT96; FS01]. They provide a framework that subsumes several widely studied models. This includes Petri nets [Esp98] (with coverability as the acceptance condition, see the example below) and their extensions, e.g. Petri nets with transfer or reset transitions [DFS98], as well as lossy channel systems (LCSes) [AJ93]. The importance of WSTSes also comes from a collection of general decidability results that have been proven on the level of WSTSes with mild assumptions. These results include the decidability of termination and boundedness [Fin87; Fin90], coverability [AJ93], as well as several simulation and equivalence problems [FS01]. Going into the details is beyond the scope of the thesis.

6.5.4 Example

Consider a labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$ where $N = (P, T, \text{in}, \text{out}, \lambda)$ is a net in which no transition is labeled by ε. We consider coverability as the acceptance condition. We may see it as an ordered LTS $(\mathbb{N}^P, \leqslant, T', \{M_{\text{init}}\}, M_{\text{final}}{\uparrow})$ that has markings as configurations, the initial marking as initial configuration, and the markings that cover M_{final} as final configurations. There is a transition $M \xrightarrow{a} M'$ if the Petri net contains some a-labeled transition t such that $M \boxed{t} M'$. The order is the product order $\leqslant$ on $\mathbb{N}^P$. Compatibility holds since greater markings potentially enable more transitions, and firing a transition in a greater marking leads to a greater marking. By Lemma 6.5.2, the product order on $\mathbb{N}^P$ is a WQO, so this ordered LTS is in fact a WSTS.

6.5.5 Remark

The previously mentioned papers consider WSTS without transition labels. We will discuss the related work on WSTS languages below. Here, we explain why we have not allowed ε-labeled transitions for WSTS.

The paper [FS01] studies various versions of upward-compatibility. The version that we have introduced for ordered LTS is called *strong compatibility* in [FS01]: A single transition originating in a small state can be simulated by a single transition from a larger state. In the presence of ε-labeled transitions, it would be consequential to consider (a variant of) *transitive compatibility*. In this version of compatibility, an a-labeled transition from a small state could be simulated by a non-empty sequence of transitions from a larger state in which all transitions are labeled with ε except for one a-labeled transition.

To avoid this complicated definition, we have simply disallowed ε-labeled transitions. Instead, we will proceed as follows when we want to apply a result that we have proven for WSTSes (where we do not allow ε-transitions) to Petri net coverability languages (where we allow ε-transitions) in Section 14.1: We argue that we can preprocess the nets under consideration so that all ε-transitions get eliminated, and then apply the result.

WSTS languages

The literature predominantly considers the unlabeled variant of well-structured transitions systems. The class of languages of WSTSes, which we will sometimes simply denote by WSTS, has received attention in [GRV07]. We summarize some results from that paper that show why this class is worth studying.

The class of WSTS languages satisfies some nice closure properties. In particular, it is closed under both union and intersection. Indeed, with the help of Lemma 6.5.2, it is easy to show that the synchronized product of two WSTSes is a WSTS. The aforementioned decidability of coverability proves that the class is a strict subclass of RE, the semi-decidable languages. For this property, it is important that we require the set of final configurations to be upward-closed. If we allow arbitrary finite sets or the set of deadlocked configurations (configurations with no successor) as the final configurations, we would obtain in both cases the class RE and hence lose all decidability results.

The paper [GRV07] also formulates the following *pumping lemma* for WSTS languages. It intuitively states that in an infinite sequence of words from a WSTS language, one can recombine the suffix of some word with a prefix of a later word from the sequence.

6.5.6 Lemma (Geeraerts, Raskin, and Van Begin [GRV07])
Let $\mathcal{L}(W)$ be the language of a WSTS and let $(w_n.v_n)_{n\in\mathbb{N}}$ be an infinite sequence of words in $\mathcal{L}(W)$. There are $i, j \in \mathbb{N}$ with $i < j$ and $w_j.v_i \in \mathcal{L}(W)$.

With this lemma, it is not difficult to show that the context-free language $\{a^n b^n \mid n \in \mathbb{N}\}$ is not a WSTS language: For each $n \in \mathbb{N}$, define $w_n = a^n$, $v_n = b^n$. If the language were the language of a WSTS W, the sequence $(w_n.v_n)_{n\in\mathbb{N}}$ would satisfy the assumptions of the lemma. Hence, there are $i \neq j$ such that $w_j.v_i = a^j.b^i \in \mathcal{L}(W)$, a contradiction. Since the language $\{a^n b^n \mid n \in \mathbb{N}\}$ is the language of a Petri net with reachability as the acceptance condition, we can also conclude that the class of Petri net reachability languages is incomparable to the WSTS languages. Note that $\{a^j.b^i \mid i, j \in \mathbb{N}, j > i\}$, a modified version of this language, does not violate the lemma and is indeed a Petri net coverability language.

In [GRV07], the authors also show that there is a non-context-free language that occurs as a Petri net coverability language. Hence, both the Petri net coverability languages and the WSTS languages are incomparable to the class of context-free languages. Furthermore, the paper considers extensions of Petri nets, like Petri nets with transfer arcs [DFS98], that can be modelled as WSTSes. One can prove that these models can generate languages that are not Petri net coverability languages, while still being WSTS languages. We obtain that the class of WSTS languages is a strict superclass of the Petri net coverability languages.

Downward-compatible WSTSes

To conclude this section, we define downward-compatible ordered LTS and WSTS similar to their upward-compatible variants.

A *downward-compatible ordered LTS* $M = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ is an LTS $(\Gamma, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ together with a quasi-order $(\Gamma, \leqslant)$ on the configurations such that Γ_{final} is downward closed and T is *downward compatible*: If $c \xrightarrow{a} d$, and $c' \leqslant$, then $c' \xrightarrow{a} d'$ for some d' with $d' \leqslant d$.

We call such a system a *downward-compatible well-structured transition system (DWSTS)* if $(\Gamma, \leqslant)$ is a WQO.

One might expect that each result that holds for WSTSes can be dualized to obtain the dual result for DWSTSes and vice versa. This would be true if we would dualize the definition of WQO and require the existence of *decreasing pairs*. With the definition above, however, DWSTSes form a separate category of systems, the results for which differ from the results for (upward-compatible) WSTSes.

DWSTSes are less common than WSTSes. A natural source of examples are *gainy* models, like gainy counter system machines or gainy communicating finite state machines. For an overview, see page 31 of [FS01].

Part III.

Closures of Petri net languages

Contents

7 Closures of Petri net languages 143
- 7.1 Basic definitions . . . 143
- 7.2 Related work . . . 146
- 7.3 Results . . . 148

8 Upward closures 151
- 8.1 Upward closures for Petri nets . . . 151
- 8.2 Upward closures for BPP nets . . . 161
- 8.3 SRE inclusion in the upward closures for Petri nets . . . 170
- 8.4 SRE inclusion in the upward closures for BPP nets . . . 173

9 Downward closures 175
- 9.1 Downward closures for Petri nets . . . 175
- 9.2 Downward closures for BPP nets . . . 183
- 9.3 SRE inclusion in the downward closures for Petri nets . . . 191
- 9.4 SRE inclusion in the downward closures for BPP nets . . . 198

10 Being upward/downward closed and regular containment 205
- 10.1 Regular containment . . . 207

Part III.
Closures of Petri net languages

This part of the thesis provides the theoretical development corresponding to Section 1.2 of the introduction. We study the closures of Petri net languages with respect to the subword order. We do so for both general Petri nets and for restrictions thereof.

Outline

In Chapter 7, we give some basic definitions. In particular, we formally introduce the upward and downward closures of languages and mention some of their properties. We also discuss related work from the literature and we summarize the results contained in this part of the thesis.

Chapter 8 is concerned with the upward closures of Petri nets, while Chapter 9 contains our work on downward closures.

Finally, Chapter 10 contains our study on the problem of deciding whether a given Petri net language is upward resp. downward closed. In this context, we also explore the related problem of deciding regular containment.

Publication

Most of the content of this section has been published in the form the paper [AMMS17] (resp. its full version [AMMS17a]). The author's contributions to that publication are discussed in detail in Chapter 20. In comparison to the publication, this thesis adds some details, e.g. a study of the state complexity of the language closures in terms of the size of the Petri net encoded in unary.

7 Closures of Petri net languages

Contents

7.1 Basic definitions 143
7.2 Related work 146
7.3 Results 148

This chapter serves as an introduction to this part of the thesis. In Section 1.2 of the introduction, we have outlined the importance of the set of all superwords and subwords of a language for several verification tasks that deal with systems containing unreliable communication. In the following, we will give some definitions to make this formal. Then, we summarize some results from the literature, presenting classes of languages for which these sets have already been shown to be computable. Finally, we outline the contributions contained in this part of the thesis.

7.1 Basic definitions

We start by making formal the notions of superwords and subwords and the sets of such. To be precise, we show that they can be incorporated into the framework of WQOs, see Section 6.5.

Let $(X, \leqslant)$ be a quasi-order. The *subsequence ordering* $(X^*, \leqslant^*)$ on the set of finite-length sequences over X is defined as follows. A sequence v is smaller than sequence w, $v \leqslant^* w$, if it can be obtained from v by iteratively deleting elements and replacing elements by smaller elements with respect to the underlying order $(X, \leqslant)$. If $v = a_1 a_2 \dots a_n$ and $v \leqslant^* w$, this means that we can write $w = c^{(0)} b_1 c^{(1)} b_2 \dots c^{(n-1)} b_n c^{(n)}$ for sequences $c^{(i)} \in X^*$ and elements $b_i \in X$ with $a_i \leqslant b_i$ for all i. Note that here, we use X^* to denote the finite-length sequences over X following the notation for words, even if X is not finite.

Higman's lemma [Hig52] guarantees that the subsequence ordering induced by a WQO is a WQO again.

7.1.1 Lemma (Higman [Hig52])
If $(X, \leqslant)$ is a well-quasi-order, then so is $(X^*, \leqslant^*)$.

Throughout this thesis, we will be exclusively interested in the case where the underlying order is $(\Sigma, =)$, a finite alphabet ordered by equality. Indeed, such a set is a WQO, see Part a) of Example 6.5.1, so Higman's Lemma guarantees that $(\Sigma^*, =^*)$ is a WQO. In this special case, we

call the order the *subword order* and denote it by $(\Sigma^*, \preceq)$. Since the underlying order is equality, the definition of the subsequence order is restricted: It is impossible to replace an element by a distinct smaller element. Hence, the definition is as expected: A word w is a subword of another word v, $w \preceq v$, if it can be obtained by deleting letters. In that case, we also say that v is a *superword*, which means that it can be obtained from w by inserting letters. Note that when we say *subword*, we are talking about *scattered subwords*, i.e. we do not require subwords to occur as infixes. It is also noteworthy that the subword ordering inherits the property of being partial order from $(\Sigma, =)$, meaning that unlike WQOs in general, it is antisymmetric.

In Section 6.5, we have introduced the notion of the upward and the downward closure of a set. Here, we apply these operations to a language with respect to the subword ordering. Given a language $\mathcal{L}$, its downward closure $\mathcal{L}{\downarrow} = \{w \in \Sigma^* \mid \exists v \in \mathcal{L} \colon w \preceq v\}$ is the set of all words that are a subword of a word in the language. Similarly, the upward closure $\mathcal{L}{\uparrow} = \{w \in \Sigma^* \mid \exists v \in \mathcal{L} \colon v \preceq w\}$ is the set of all superwords of a word in the language, or equivalently, all words that have a subword in the language. Unless stated otherwise, all closure operators will refer to the closures with respect to the subword ordering in this part of the thesis.

Regularity and simple regular expressions

With Higman's Lemma, we also get that the various properties of WQOs stated in Lemma 6.5.3 apply to the subword ordering. A surprising consequence of this is the classical result states that both the upward closures and the downward closures of arbitrary languages are guaranteed to be regular. We give the proof to show how the properties of well-quasi-orders come into play.

7.1.2 Theorem (Haines [Hai69])
For any language $\mathcal{L} \subseteq \Sigma^*$, $\mathcal{L}{\downarrow}$ and $\mathcal{L}{\uparrow}$ are regular.

Proof:
By Higman's Lemma $(\Sigma^*, \preceq)$ is a WQO, so every language over Σ has a finite set of minimal elements by Item (3) of Lemma 6.5.3. Applying this property to $\mathcal{L}{\uparrow}$ yields a finite set of words $w^{(1)}, \ldots, w^{(k)}$ such that $\mathcal{L}{\uparrow} = \{w^{(1)}, \ldots, w^{(k)}\}{\uparrow} = w^{(1)}{\uparrow} \cup \ldots \cup w^{(k)}{\uparrow}$. To show that $\mathcal{L}{\uparrow}$ is regular, we observe that the upward closure of a single word $w^{(i)} = a_1 \ldots a_n$ can be represented by the regular expression $\Sigma^* a_1 \Sigma^* \ldots \Sigma^* a_n \Sigma^*$, and that the class of regular languages is closed under finite unions.

To see that the downward closures are regular as well, we use that the complement of a downward-closed language is upward-closed, hence regular, and complements of regular languages are again regular. ■

A more recent result [ACBJ04] shows that a certain type of restricted regular expressions is sufficient to describe downward and upward closures. A *simple regular expression (SRE)* is a choice among products,

$$sre ::= p \mid sre \cup sre ,$$

where a product is a concatenation of letters, optional letters, and iterations over subsets $\Gamma \subseteq \Sigma$ of the alphabet,

$$p ::= a \mid (a \cup \varepsilon) \mid \Gamma^* \mid p.p .$$

SREs are sufficient to describe all upward and downward-closed languages.

7.1.3 Theorem (Abdulla, Collomb-Annichini, Bouajjani, and Jonsson [ACBJ04])
For any language $\mathcal{L} \subseteq \Sigma^*$, there are simple regular expressions describing its downward and upward closures.

For upward closures, the proof follows directly from the proof of Theorem 7.1.2. For downward closures, one needs to carefully look at the shape of the language obtained by complementing an upward-closed language.

Actually, the original definition of SREs in [ACBJ04] is simpler than ours. In [ACBJ04], a product is defined to be a concatenation $p.p$ of products, an optional letter $(a \cup \varepsilon)$, or the Kleene iteration of a subalphabet Γ^*, omitting the case of a single non-optional letter. With this version, one obtains a tight correspondence with the downward-closed languages: The language of an SRE is downward closed, and any downward-closed language can be described by an SRE. We have modified the definition to ensure that also upward-closed languages are covered, losing the property that every language of an SRE is downward closed.

Effectivity

As we have seen, closures of languages are guaranteed to be regular. This in turn means that they cannot be *effectively regular* in general. We know that they can be represented by e.g. an NFA, but it may be impossible to compute such a representation. We have given a sketch of the corresponding proof in Section 1.2, which we repeat here for the sake of completeness. By the well-known results of Turing and others, all non-trivial decision problems for languages of Turing machines are undecidable. This in particular means that the emptiness problem for Turing machine languages is undecidable. However, the upward and downward closure of a language are empty if and only if the language itself is empty. Hence, the computability of an effective representation of the closure of a Turing machine language (e.g. an NFA or a regular expression) would allow us to decide the emptiness problem, a contradiction.

7.2 Related work

The fact that closures of languages are regular, but their representations cannot always be computed, has motivated research to study for which classes of languages the closures are effectively regular. For some classes that have this property, studies have determined the optimal size of NFAs representing the closures. This size is important: If we use an NFA representing a closure as the input to a decision problem, then the running time of solving that problem will scale with the size of the automaton.

We present some simple examples before citing more involved results from the literature.

For a regular language $\mathcal{L} = \mathcal{L}(A)$, represented by an NFA A, it is easy to compute NFAs $A{\downarrow}$ and $A{\uparrow}$ with $\mathcal{L}(A{\downarrow}) = \mathcal{L}{\downarrow}$ and $\mathcal{L}(A{\uparrow}) = \mathcal{L}{\uparrow}$, respectively. Note that the arrow in $A{\downarrow}$ is part of the notation and does not denote a closure, similar for $A{\uparrow}$. To construct $A{\downarrow}$, we add for each transition $q \xrightarrow{a} p$ of A that is labeled by a symbol $a \in \Sigma$ an ε-labeled version of this transition, $q \xrightarrow{\varepsilon} p$. By using these transitions, it is easy to show that any subword of a word in the language of A has an accepting run in $A{\downarrow}$. For the construction of $A{\uparrow}$, we mimic the proof of Theorem 7.1.2 by allowing the insertion of arbitrary words. To be more precise, for each state q and each letter $a \in \Sigma$, we add a loop $q \xrightarrow{a} q$ to automaton A.

Alternatively, one could construct a representation of $\mathcal{L}{\uparrow}$ by complementing $\mathcal{L}$, computing the downward closure, and complementing again. However, this approach does not work for classes of languages for which the complement cannot be effectively constructed. In general, the computability of the upward closure and the computability of the downward closure are distinct problems that have to be studied separately.

The construction that we have used for NFAs translates into similar constructions for Petri nets and context-free grammars. Given a Petri net N, we can construct a Petri net $N{\downarrow}$ by adding for each transition an ε-labeled version. Similarly, we construct $N{\uparrow}$ by adding for each letter a an a-labeled spontaneous transition that does not produce any tokens, i.e. a transition t with $\mathrm{in}(t) = \mathrm{out}(t) = \vec{0}$. We obtain that $\mathcal{L}(N{\downarrow}, M_{\mathrm{init}}, M_{\mathrm{final}}) = \mathcal{L}(N, M_{\mathrm{init}}, M_{\mathrm{final}}){\downarrow}$ for all markings $M_{\mathrm{init}}, M_{\mathrm{final}}$, similar for the upward closure. The constructions for context-free grammars are similar and realize the downward closure by being able to omit letters and the upward closure by allowing the insertion of arbitrary words.

While these constructions are simple and sometimes even useful – we will use the Petri net $N{\downarrow}$ in several proofs later in this part of the thesis – they do not yield the desired result. For example, $N{\downarrow}$ is a Petri net that represents the downward closure of N, but it is not a regular representation, i.e. an NFA or a regular expression. It may be hard or even impossible to construct an algorithm that transform a non-regular representation for some input language $\mathcal{L}$ into a regular representation, even when knowing that the input language is regular.

For the languages generated by context-free grammars, the computability of the downward and upward closure has been studied in [Lee78; Cou91; GHK07; BLS15]. The effective regularity has been first proven in [Lee78]. The paper [GHK07] shows that for a context-free grammar of size n, the state complexity of the upward closure is at most $2^{\mathcal{O}(n)}$, i.e. one can compute an NFA with at most $2^{\mathcal{O}(n)}$ states representing the upward closure. The same upper bound on the state complexity was proven [BLS15] for the downward closure. Additionally, [BLS15] present a family of finite languages that show that the aforementioned upper bounds are in fact optimal.

For the languages of one-counter automata, a strict subclass of the class of context-free languages, finite automata representing the downward and upward closure of the language can be computed in polynomial time [ACHKSZ16].

In [Zet15a], Zetzsche has shown that for any class of languages that is a *full trio*, i.e. it satisfies certain closure properties, the computability of the downward closure is equivalent to the decidability of a *simultaneous unbounded problem*. In his paper, he applies this result to show that the downward closures of *indexed languages*, the languages of higher-order pushdown automata of order 2, are computable. Later, the result has been used to prove the effective computability of the downward closures of higher-order pushdown automata and higher-order recursion schemes of arbitrary order [HKO16; CPSW16].

The case of Petri net languages has been considered in [HMW10]. In addition to the general case of reachability as the acceptance condition, the paper contains a construction specifically for coverability languages. In contrast to the general case, this construction does not rely on details from Lambert's proof of decidability of Petri net reachability [Lam92], but only on the well-known Karp-Miller tree [KM69]. However, the size of the Karp-Miller tree is known to be non-primitive recursive in the worst case, and the paper leaves the question of whether a smaller construction exists unanswered.

We conclude this section with a negative example: The downward closures of the languages of lossy channel systems are not computable. A *lossy channel system (LCS)* consists of several finite-state components that communicate by writing to and reading from FIFO (first in, first out) channels. To avoid the undecidability of the Turing-complete model of *perfect channel systems*, the channels are lossy. This means that in each step of the computation, the content w of some channel can be replaced by some v with $v \preceq w$. LCSes are captured by the framework of WSTSes, which in particular means that reachability is decidable. However, many other problems that are decidable for other WSTSes like Petri nets are undecidable, as proven in [May03]. In addition to repeated reachability, this also includes the unboundedness problem, i.e. the problem of deciding whether there is a constant bounding the length of the content of all channels. It is easy to design an LCS that at some point stops its computation, selects one channel, and generates (a subword of) the current content of this channel as a word. Assuming that one could compute an NFA representing the downward closure of the language of this LCS, one could check whether the channel content of the original LCS is bounded. Hence, the undecidability of the boundedness problem means that downward closures are not effectively regular for LCSes.

7.3 Results

The goal of this part of this thesis is to study the computability and size of representations of the upward and downward closures of Petri net coverability languages. We recall the result for coverability languages from [HMW10] and provide a lower bound that matches the non-primitive recursive complexity of the algorithm. We extend the result by studying various restrictions of the problem that yield lower complexities, providing matching upper and lower bounds in each case. Furthermore, we study upward closures which have not been considered in [HMW10]. Because the class of coverability languages is not closed under complement, we cannot simply use the computability of the downward closures to deduce the computability of the upward closures. It turns out that the upward closures are computable, and the complexity of doing so is much lower than in the case of downward closures.

Let us briefly comment on the structure of this part of the thesis and summarize the results that will be proven in each of the chapters. Chapter 8 considers the upward closures of Petri net coverability languages. We first consider PN-UC, the problem of computing an NFA representing the upward closure. We prove in Theorem 8.1.1 that this problem can be solved in doubly exponential time, which is optimal. This bad complexity leads us to considering three restricted versions of the problem: If we restrict the input to be a BPP net, we obtain the problem BPP-UC which can be solved in singly exponential time which is again optimal, see Theorem 8.2.1. Instead of computing the upward closure, we then check whether the language of a given SRE is contained in it. This problem, PN-SREUC, is EXPSPACE-complete as proven in Theorem 8.3.1. Combining both restriction leads to the problem BPP-SREUC which is NP-complete, Theorem 8.4.1. In the corresponding sections of Chapter 8, we provide more motivation for considering each of these restrictions.

In Chapter 9, we conduct a similar study for the downward closures of Petri net languages. The computability of the downward closure in the general case, problem PN-DC, was already settled in [HMW10]. We briefly recall the procedure that results in an NFA of potentially non-primitive recursive size and complement it by a matching lower bound, showing that the construction is optimal, Theorem 9.1.1. We again consider the two restrictions and their combination in the following sections, which yields the following results: Problem BPP-DC can be solved in exponential time, which is optimal (Theorem 8.2.1); problem PN-SREDC is EXPSPACE-complete (Theorem 9.3.1), and BPP-SREDC is NP-complete (Theorem 9.4.1). While these complexities match the corresponding problems for the upward closure, the proofs are vastly different.

In both Chapter 8 and Chapter 9, we will also consider the computational resp. state complexity of the various problems for Petri nets encoded in unary. For all problems, we obtain the same complexity class as in the case of the usual binary encoding. This means that the difficulty in computing the languages closures comes from the concurrent nature of Petri nets, and not from their capability of encoding exponential transition multiplicities using polynomial space.

	Petri nets	BPP nets
PN-UC / BPP-UC	Doubly exponential (Theorem 8.1.1)	Exponential (Theorem 8.2.1)
PN-SREUC / BPP-SREUC	EXPSPACE-complete (Theorem 8.3.1)	NP-complete (Theorem 8.4.1)
PN-DC / BPP-DC	Non-primitive recursive (Theorem 9.1.1,[HMW10])	Exponential (Theorem 8.2.1)
PN-SREDC / BPP-SREDC	EXPSPACE-complete (Theorem 9.3.1)	NP-complete (Theorem 9.4.1)
PN-BEINGUC, PN-BEINGDC	Decidable (Theorem 10.0.1)	
PN-REGCONT	Decidable (Theorem 10.1.1)	

Figure 7.3.a: A summary of our results regarding the upward and downward closures of Petri net coverability languages.

We conclude this part of the thesis by studying whether the upward resp. downward closure of a Petri net coverability language – representations of which we can compute as demonstrated in the Chapters 8 and 9 – are equal to the language itself. It turns out that this problem is decidable, as we will show in the form of Theorem 10.0.1. The proof of this theorem uses another new decidability result that is of independent interest. We show that the regular containment problem PN-REGCONT of checking whether a given regular language is contained in the coverability language of a given Petri net is decidable, Theorem 10.1.1.

The table in Figure 7.3.a summarizes the results that we will present in detail in the rest of this part of the thesis.

8 Upward closures

Contents

8.1 Upward closures for Petri nets 151
8.2 Upward closures for BPP nets 161
8.3 SRE inclusion in the upward closures for Petri nets 170
8.4 SRE inclusion in the upward closures for BPP nets 173

This chapter of the thesis is dedicated to studying the computation of the upward closures of Petri net languages. The task is to compute an NFA whose language is the upward closure of the language of a given Petri net. We first study this problem in the case of Petri net coverability languages, showing that the state complexity of the upward closure is doubly exponential, and that a corresponding NFA can be computed in doubly exponential time. The proof techniques for the upper and lower bound use and extend the well-known results on coverability by Rackoff [Rac78] and Lipton [Lip76], respectively.

The doubly exponential complexity motivates studying restricted versions of the problem. We do so in the second section by considering the upward closures of BPP net languages. We show that in this case, both the state complexity and the time to compute a corresponding NFA are singly exponential.

Finally, we consider a different kind of restriction. Instead of actually computing the upward closure, we consider the problem of checking whether a candidate SRE is included in it. We will give more motivation for studying this problem in Section 8.3. This restriction results in even better complexity: The problem is EXPSPACE-complete for general Petri nets and NP-complete for BPP nets. In both cases, the proofs make use of coverability having the same complexity.

8.1 Upward closures for Petri nets

We start by considering the problem of computing the upward closure of a given Petri net coverability language. This problem is formalized as follows.

Computing the upward closure for Petri net languages (PN-UC)	
Given:	A labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$.
Compute:	An NFA A with $\mathcal{L}(A) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$.

The result that we will show in this section is the following.

8.1.1 Theorem
Upward closures of Petri net coverability languages have doubly exponential state complexity, and the corresponding automata can be constructed in doubly exponential time. These bounds are tight.

The proof consists of two steps. Firstly, we show the upper bound: Given a Petri net instance, one can compute in doubly exponential time an NFA representing the upward closure. Obviously, the time needed for the construction also limits the size of the NFA. Secondly, we present a family of Petri nets that provides a lower bound. We conclude the section by considering the case of Petri nets that are encoded in unary.

Upper bound

We prove one direction of Theorem 8.1.1 in the form of the following theorem.

8.1.2 Theorem
Given a labeled Petri net instances $(N, M_{\text{init}}, M_{\text{final}})$, we can compute in doubly exponential time an NFA A with $\mathcal{L}(A) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$. The state complexity of $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ is at most doubly exponential, as witnessed by automaton A.

Our proof strongly relies on an extension of the algorithm presented by Rackoff to show that Petri net coverability is in EXPSPACE [Rac78]. Let us recapitulate Rackoff's approach. The key result is that if there is a covering computation, then there is one of doubly exponential length. With this property, it is not hard to enumerate and check all candidates of doubly exponential length using only exponential space. To prove the result, Rackoff considers *pseudo-computations* that are valid only on a subset of the places, say on the first i places wrt. some total order on the places. Assuming that the maximum length of the shortest covering pseudo-computation that is valid on the first i places is known, one obtains a bound for $i+1$ places. This yields a recursive formula which one can evaluate for i equal to the number of places to get a bound on the length of ordinary computations (instead of pseudo-computations). A closed form for this formula provides the desired doubly exponential bound. Establishing the recursive formula itself is quite involved: One has to show that any covering computation can be pruned and partially replaced to obtain one that fits the bound. This requires quantifying over all possible initial markings.

We want to use and extend Rackoff's ideas to prove that any minimal word results from a computation of length at most doubly exponential. With this result, it is easy to find all minimal words and construct the automaton for the upward closure in doubly exponential time. Establishing the bound requires two major modifications in Rackoff's proofs: Firstly, the definition of

the bound has to be adjusted such that it guarantees that it is larger than (the length of) all covering computations producing minimal words, instead of just one single covering computation. Secondly, the technique that is applied to long covering computations to establish the bound has to be handled with care to ensure that the resulting shortened computation generates a subword of the original one.

Let us assume that $(N, M_{\text{init}}, M_{\text{final}})$ is the labeled Petri net instance of interest, and that its places are ordered i.e. $P = [1, \ell]$. We fix $n = |(N, M_{\text{init}}, M_{\text{final}})|$ to be the size of this instance. A *pseudo-marking* of N is a function $M\colon P \to \mathbb{Z}$ that assigns an integer number of tokens to each place.

Let $i \in [0, \ell]$ be a number. Following Rackoff [Rac78], we define versions of most notions that we have introduced for markings that work on pseudo-markings, but require the first i places to be treated properly. We call pseudo-marking M *i-non-negative* if $M(p) \geqslant 0$ for $p \in [1, i]$. An i-non-negative marking M *i-enables* a transition t if $M(p) - \text{in}(t, p) \geqslant 0$ for $p \in [1, i]$. If we fire it, we obtain the i-non-negative marking $M' = M + e(t)$. We write $M \xrightarrow{t}_i M'$. A sequence of pseudo-markings and transitions

$$M_0 \xrightarrow{t_1}_i M_1 \dots \xrightarrow{t_m}_i M_m$$

is an *i-valid computation* if all markings are i-non-negative and all transitions are i-enabled in the marking from which they are fired. A pseudo-marking M is *i-covering* if $M(p) \geqslant M_{\text{final}}(p)$ for $p \in [1, i]$. A computation on pseudo-markings is *i-covering* if it is i-valid and it reaches a marking that is i-covering.

There are two special cases that are worth mentioning. The case of $i = 0$ imposes no restrictions: Any pseudo-marking is 0-non-negative, any transition is 0-enabled in such a marking and any sequence of pseudo-markings and transitions that respects the effect of the transitions is i-covering. In the case of $i = \ell$, the definitions coincide with the usual definitions for markings, since we require all places to be treated properly. For instance, an ℓ-non-negative pseudo-marking is a marking and an ℓ-covering computation is a covering computation.

In the proof, we will need to consider markings other than M_{init} as the initial marking. Consequentially, the following definitions are parametric in the initial pseudo-marking M. Later, we maximize over all possible initial markings, which on the one hand removes the dependency on M, and on the other hand ensures that the bound holds for M_{init}, the initial marking of interest.

We proceed by giving the formal definitions. Let M be some pseudo-marking. We define $\text{Paths}(M, i)$ to be the set of all firing sequences inducing i-covering computations from M,

$$\text{Paths}(M, i) = \left\{\sigma \in T^* \;\middle|\; M \xrightarrow{\sigma}_i M' \text{ is } i\text{-covering}\right\}.$$

Let $\text{Words}(M, i) = \{\lambda(\sigma) \mid \sigma \in \text{Paths}(M, i)\}$ be the corresponding set of words, and let $\min \text{Words}(M, i)$ be its minimal elements with respect to the subword relation. Since we allow ε-labeled transitions, each element of $\min \text{Words}(M, i)$ is potentially generated by an infinite

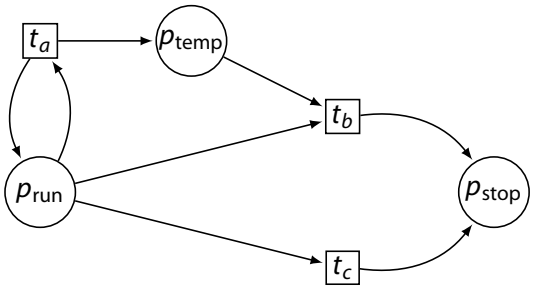


Figure 8.1.a: The Petri net N_{ce}.

number of computations. The definition of $\text{SPath}(M, i)$ takes care of this by considering for each such element only the shortest computation(s),

$$\text{SPath}(M,i) = \left\{ \sigma \in \text{Paths}(M,i) \;\middle|\; \begin{array}{l} \lambda(\sigma) \in \min \text{Words}(M,i), \\ \nexists\, \sigma' \in \text{Paths}(M,i)\colon \quad |\sigma'| < |\sigma|,\ \lambda(\sigma') = \lambda(\sigma) \end{array} \right\}.$$

Finally, we define $m(M,i) = \max\{|\sigma| + 1 \mid \sigma \in \text{SPath}(M,i)\}$ to be the maximum number of markings (which is the number of transitions plus one) associated to any firing sequence from $\text{SPath}(M,i)$. If $\text{SPath}(M,i)$ is empty, we set $m(M,i) = 0$. Note that $\min \text{Words}(M,i)$ is finite since the subword relation is a WQO, see Section 6.5. Hence, also $\text{SPath}(M,i)$ is finite and m is well-defined.

By definition, we have that any word that occurs along an i-covering computation from M has a subword that labels an i-covering computation from M with length at most $m(M,i)$. To get rid of the parameter M, we maximize over all pseudo-markings and define

$$f(i) = \max\{m(M,i) \mid M\colon P \to \mathbb{Z}\}\,.$$

Obviously, we have $f(i) \geqslant m(M_{\text{init}}, i)$. The well-definedness of f is not clear from its definition. Instead, it will be implied by the bound that we will prove below.

The above definitions incorporate the first major change to Rackoff's original proof in [Rac78]. Rackoff simply defines $m(i,M)$ to be the minimum length of an i-covering computation from M plus one. Before continuing with the theory, we present our definitions on an example.

8.1.3 Example

Consider the Petri net $N_{ce} = (\{p_{\text{run}}, p_{\text{temp}}, p_{\text{stop}}\}, \{t_a, t_b, t_c\}, \text{in}, \text{out}, \lambda)$ over $\{a,b,c\}$, where the multiplicities are given by Figure 8.1.a and we have $\lambda(t_a) = a, \lambda(t_b) = b$, and $\lambda(t_c) = c$. Let us use the total order $p_{\text{run}} < p_{\text{temp}} < p_{\text{stop}}$ on the places. Consider the initial marking $M_{\text{init}} = (1,0,0)$ with one token on p_{run} and no token elsewhere, and the final marking $M_{\text{final}} = (0,0,1)$ that requires one token on p_{stop}. We see that $\text{SPath}(M_{\text{init}}, 1)$ contains the firing sequence t_b, which is 1-valid, but not 2-valid. When choosing $i = 3$, we obtain that $\text{Words}(M_{\text{init}}, 3) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = a^+b \cup a^*c$, $\min \text{Words}(M_{\text{init}}, 3) = \{ab, c\}$ and thus $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow = \Sigma^* a \Sigma^* b \Sigma^* \cup \Sigma^* c \Sigma^*$. We have $\text{SPath}(M_{\text{init}}, 3) = \{t_a t_b, t_c\}$ and consequently

$m(M_{\text{init}}, 3) = 3$. Indeed, to generate the minimal word ab, a computation consisting of three markings and two transitions is needed. It is not difficult to see that in fact, $f(3) = 3$ holds, since removing the token from p_{run} in the initial marking leads to M_{final} becoming non-coverable, while adding more tokens only makes covering easier.

In the following, we will first obtain a recursive formula that establishes an upper bound for $f(i+1)$ depending on $f(i)$. The proof proceeds by Rackoff's famous case distinction from the proof of Lemma 3.4 in [Rac78]. The second case of the proof incorporates the changes necessary to maintain the subword relationship.

8.1.4 Proposition
$f(0) = 1$ and $f(i+1) \leqslant (2^n f(i))^{i+1} + f(i)$ for all $i \in [1, \ell - 1]$.

Proof:
To prove $f(0) = 1$, we use that $\varepsilon \in \min \text{Words}(M_0, 0)$ for any $M_0 \in \mathbb{Z}^\ell$, and the empty firing sequence induces a 0-covering computation just consisting of the initial marking M_0 that generates the word ε.

For the second part, we prove that $m(M_0, i) \leqslant (2^n f(i))^{i+1} + f(i)$ for any marking $M_0 \in \mathbb{Z}^\ell$. If M_{final} is not $(i+1)$-coverable from M_0, we have $m(M_0, i) = 0$ by definition and the bound holds. Hence, we may assume that there is an $(i+1)$-covering computation, say $M_0 \boxed{\sigma}_{i+1} M'$. We show that there is a firing sequence σ' such that $M_0 \boxed{\sigma'}_{i+1} M''$ is also an $(i+1)$-covering computation, $\lambda(\sigma')$ is a subword of $\lambda(\sigma)$, and σ' satisfies the bound $|\sigma'| < 2^n (f(i))^{i+1} + f(i)$. To see that the desired statement is then indeed proven, note that by picking $M_0 \boxed{\sigma}_{i+1} M'$ as a computation generating a word from $\min \text{Words}(M_0, i+1)$ and applying the argument, we obtain a computation for the same word (since we assumed the word to be minimal) whose length satisfies the bound. Hence, also a firing sequence for that word from $\text{SPath}(M_0, i+1)$, i.e. one with minimum length, satisfies the bound. By the definition of $m(M_0, i+1)$ as the maximum over the length of all computations associated to firing sequences from $\text{SPath}(M_0, i+1)$, we obtain that $m(M_0, i+1)$ itself satisfies the bound as required.

In order to prove the bound, we distinguish two cases.

1st Case: Suppose that for each marking M occurring in the computation $M \boxed{\sigma}_{i+1} M'$ and for each place $p \in [1, i+1]$, $M(p) < 2^n \cdot f(i)$ holds. Our goal is to remove infixes of the computation, resulting in an $(i+1)$-covering computation in which no two markings agree on their restriction to the first $i+1$ places. Whenever a repetition occurs, i.e. we have markings M_j and $M_{j'}$ with $j < j'$ and $M_j(p) = M_{j'}(p)$ for all $p \in [1, i+1]$, we remove the infix $\boxed{\sigma_{j+1}}_{i+1} M_{j+1} \cdots \boxed{\sigma_{j'}}_{i+1} M_{j'}$. Note that this removal changes the markings occurring in the rest of the computation only on places p with $p > i+1$. Hence, the resulting computation is still a valid $(i+1)$-covering

computation. After iterating this process until all repetitions have been eliminated, we have obtained a computation $M_0 \xrightarrow{\sigma'}_{i+1} M''$ of length strictly less than $(2^n f(i))^{i+1}$ since each of the relevant $i+1$ places can only attain values from $[0, 2^n \cdot f(i) - 1]$. The strictness comes from the fact that a computation consisting of h markings has $(h-1)$ transitions. Since σ' was obtained from σ by deleting infixes, it is a subword of σ, and $\lambda(\sigma') \preceq \lambda(\sigma)$ holds.

2nd Case: Otherwise, the computation contains a marking that assigns $2^n \cdot f(i)$ or more tokens to one of the first $i+1$ places. Let M_j be the first such marking. Without loss of generality, we assume that for place $i+1$, $M_j(i+1) \geqslant 2^n \cdot f(i)$ holds. Otherwise, swap the order of the places. We decompose the computation into

$$M_0 \xrightarrow{\sigma_1}_{i+1} M_{j-1} \xrightarrow{t}_{i+1} M_j \xrightarrow{\sigma_2}_{i+1} M'$$

where, $\sigma_1, \sigma_2 \in T^*$ are a firing sequences.

Note that the computation $M_0 \xrightarrow{\sigma_1}_{i+1} M_{j-1}$ satisfies the requirement that we made in the first case of the proof. Hence, we can assume that $|\sigma_1| < (2^n f(i))^{i+1}$ holds. More formally, we could replace the computation by a repetition-free computation $M_0 \xrightarrow{\sigma_1'}_{i+1} M'_{j-1}$, where M_{j-1} and M'_{j-1} coincide on the first $i+1$ places. In particular, the computation obtained by now firing $t.\sigma_2$ is an $(i+1)$-covering computation. Since $\sigma_1' \preceq \sigma_1$, we also have $\lambda(\sigma_1'.t.\sigma_2) \preceq \lambda(\sigma_1.t.\sigma_2)$.

Let us consider the third part of the computation, $M_j \xrightarrow{\sigma_2}_{i+1} M'$. By the definition of $f(i)$, we have $m(M_j, i) \leqslant f(i)$. Hence, for any word $w \in \text{Words}(M_j, i)$, there is an associated subword $w' \in \min \text{Words}(M_j, i)$ with $w \preceq w'$ and an i-covering computation $M_j \xrightarrow{\sigma_2'}_i M''$ of minimum length with $\lambda(\sigma_2') = w'$ and $|\sigma_2'| < m(M_j, i) \leqslant f(i)$. In the following, we consider the word $w = \lambda(\sigma_2) \in \text{Words}(M_j, i)$ and consider the firing sequence σ_2' with the aforementioned properties. Here, we have used that any $(i+1)$-covering computation is also i-covering. We claim that $M_0 \xrightarrow{\sigma_1}_{i+1} M_{j-1} \xrightarrow{t}_{i+1} M_j \xrightarrow{\sigma_2'}_{i+1} M''$ is a valid $(i+1)$-covering computation. Since $M_j \xrightarrow{\sigma_2'}_i M''$ was an i-covering computation, it only remains to argue that the transitions in the computation are enabled with respect to the place $i+1$, and that $M''(i+1) \geqslant M_{\text{final}}(i+1)$. To this end, note that we had $M_j(i+1) \geqslant 2^n \cdot f(i)$. Each transition in σ_2' can only consume at most 2^n tokens from this place, since this is the largest number that can be encoded in binary using $n \geqslant |N|$ bits. Hence, in the worst case, all markings M occurring in the computation $M_j \xrightarrow{\sigma_2'}_{i+1} M''$ satisfy

$$M(i+1) \geqslant M_j(i+1) - \left|\sigma_2'\right| \cdot 2^n \geqslant \left(2^n \cdot f(i)\right) - \left((f(i)-1) \cdot 2^n\right) = 2^n \,.$$

Here, we have used $|\sigma_2'| < f(i)$. By the definition of n as the size of the Petri net instance, 2^n is larger than any incoming transition multiplicity and larger than $M_{\text{final}}(i+1)$. Thus, the transitions occurring in the computation are enabled, and M'' is covering with respect to the first $i+1$ places.

In total, we have obtained that $M_0 \xrightarrow{\sigma_1}_{i+1} M_{j-1} \xrightarrow{t}_{i+1} M_j \xrightarrow{\sigma_2'}_{i+1} M''$ is an $(i+1)$-covering computation with

$$\left|\sigma_1.t.\sigma_2'\right| = |\sigma_1| + 1 + \left|\sigma_2'\right| \leqslant \left(\left(2^n f(i)\right)^{i+1} - 1\right) + 1 + (f(i) - 1) < \left(2^n f(i)\right)^{i+1} + f(i)\,,$$

and the desired bound holds. ■

Using Proposition 8.1.4, it is easy to prove a non-recursive bound on the length of computations that generate minimal words.

8.1.5 Proposition
For each computation $M_{\text{init}} \xrightarrow{\sigma} M \geqslant M_{\text{final}}$, there is σ' with $M_{\text{init}} \xrightarrow{\sigma'} M' \geqslant M_{\text{final}}$, $\lambda(\sigma') \preceq \lambda(\sigma)$ and $|\sigma'| \leqslant 2^{2^{cn \log n}}$ for some constant c.

The proof uses the same line of argumentation as Theorem 3.5 in [Rac78]. We give it for the sake of completeness.

Proof:
We define the function g inductively by $g(0) = 2^{3n}$ and $g(i+1) = (g(i))^{3n}$. It is easy to see that $g(i) = 2^{((3n)^{(i+1)})}$. Using Proposition 8.1.4 we can conclude $f(i) \leqslant g(i)$ for all $i \in [0, \ell]$, where we again assume that $P = [1, \ell]$. Furthermore,

$$f(\ell) \leqslant g(\ell) \leqslant 2^{((3n)^{(\ell+1)})} \leqslant 2^{((3n)^{n+1})} \leqslant 2^{2^{(\ell+1)\cdot\log(3n)}}$$

which, using $\ell \leqslant n$ and the laws for logarithms, is of the required shape.

Now assume that $M_{\text{init}} \xrightarrow{\sigma} M \geqslant M_{\text{final}}$, i.e. $\lambda(\sigma) \in \text{Words}(M_{\text{init}}, \ell)$. This means that there is some $\sigma' \in \text{SPath}(M_{\text{init}}, \ell)$ with $M_{\text{init}} \xrightarrow{\sigma'} M' \geqslant M_{\text{final}}$ and $\lambda(\sigma') \preceq \lambda(\sigma)$. By the definition of function f, $|\sigma'| \leqslant m(M_{\text{init}}, \ell) \leqslant f(\ell)$ holds. If we combine this with the above estimation, we obtain the desired result. ■

8.1.6 Remark
Lemma 5.3 in [LPS13] essentially states Proposition 8.1.5. For the proof, it is claimed that Rackoff's original proof already implies the statement. While it is indeed true that the same (recursive as well as non-recursive) bounds hold as in Rackoff's paper, this claim is misleading. This can be demonstrated using the Petri net instance from Example 8.1.3. When we compute Rackoff's bound according to his definitions, we obtain that if there is a computation covering M_{final} from M_{init}, then there is one consisting of at most one transition. Indeed, the computation $M_{\text{init}} \xrightarrow{t_c} M_{\text{final}}$ is covering.

However, computations whose length is within Rackoff's upper bound do not necessarily generate all minimal words: We have that ab is a minimal word in $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ and the shortest covering computation that generates ab is $M_{\text{init}} \xrightarrow{t_a} (1,1,0) \xrightarrow{t_b} M_{\text{final}}$, consisting of 3 markings and 2 transitions.

Recall that in order to prove Theorem 8.1.2, we need to construct an automaton for the upward closure of doubly exponential size. To this end, we employ the length-k approximation that we have introduced in Section 6.3. It represents all words that can be generated by computations of length at most k. By choosing k equal to the bound for $f(\ell)$, Proposition 8.1.5 guarantees that the upward closure of the approximation is equal to the upward closure of the language of the Petri net. Since the length-k approximation is regular and we can construct an automaton for it, we can construct an automaton for the upward closure of the Petri net language with the desired size.

Proof of Theorem 8.1.2:
Let $(N, M_{\text{init}}, M_{\text{final}})$ be the Petri net instance of interest, and let $n = |(N, M_{\text{init}}, M_{\text{final}})|$ be its size. We define $k = 2^{2^{cn \log n}}$ to be the doubly exponential bound from Proposition 8.1.5. We consider the length-k approximation $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})$ of the given Petri net, see Proposition 6.3.2, or rather, its upward closure $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})\uparrow$.

Since the length-k approximation is an underapproximation, we have $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}}) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ and hence $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})\uparrow \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$. It remains to argue that equality holds. Since $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ is upward closed, it is sufficient to show that the minimal words in $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ are contained in $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})$. Using Proposition 8.1.5 for some minimal word w, we obtain the existence of a covering computation for that word of length at most k. Thus, $w \in \mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})$ as desired.

To finish the proof, we need to consider the automaton for $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})\uparrow$.

Proposition 6.3.2 gives us an automaton A for $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})$. Furthermore, we get that the number of states of A is at most $\mathcal{O}(k^n \cdot 2^{n^2}) = \mathcal{O}(2^{2^{c \cdot n^2 \cdot \log n}} \cdot 2^{n^2})$, which is doubly exponential as required. We have argued in Section 7.2 that we can transform an automaton into an automaton for the upward closure by inserting self-loops $q \xrightarrow{a} q$ for every letter of the alphabet a and every state q. In particular, this construction does not add any new states. Applying this construct to A yields an automaton $A\uparrow$ for $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})\uparrow = \mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ with a doubly exponential number of states. ■

Lower bound

We complete the proof of Theorem 8.1.1 by providing a lower bound that matches the doubly exponential upper that we have just proven, showing that the above construction is optimal. To achieve this goal, we present for each $n \in \mathbb{N}$ a Petri net instance of size polynomial in n

such that any NFA describing its upward closure is at least of doubly exponential size. To be precise, we show that we can generate the language $\{a^k \mid k \geqslant 2^{2^n}\}$ with a Petri net of size polynomial in n. Note that this language is already upward closed, and its state complexity is doubly exponential as we have discussed in Example 4.3.1. To ensure that a doubly exponential number of a-labeled transitions is fired, we rely on Lipton's construction, Proposition 6.2.3, from his proof of the EXPSPACE-hardness of coverability [Lip76].

8.1.7 Proposition

For each $n \in \mathbb{N}$, there is a labeled Petri net instance $(N(n), M_{\text{init}}(n), M_{\text{final}}(n))$ of size polynomial in n such that $\mathcal{L}(N(n), M_{\text{init}}(n), M_{\text{final}}(n))\uparrow$ has state complexity at least 2^{2^n}.

Proof:

Let $n \in \mathbb{N}$. Using Proposition 6.2.3, there is a Petri net instance $(N_{\text{dec}}, \vec{0}, M_{\text{finaldec}})$ polynomial in n with a place p_{indec} such that for any marking M, there is a covering computation $M \boxed{\sigma} M' \geqslant M_{\text{finaldec}}$ if and only if $M(p_{\text{indec}}) \geqslant 2^{2^n}$. Let us see N_{dec} as a labeled Petri net in which all transitions are labeled by ε. We construct $N(n)$ by adding places and transitions to N_{dec}. A schematic representation of the net $N(n)$ is given in Figure 8.1.b. Firstly, we add two new places p_{init} and p_{run}. All existing transitions t from N_{dec} are modified so that they check for the existence of a token on p_{run}, i.e. we set $\text{in}(t, p_{\text{run}}) = \text{out}(t, p_{\text{run}}) = 1$. Other than that, the effect of the transitions remain unchanged. In particular, they do not consume or produce tokens on p_{init}. We add a fresh transition t_{a} that is labeled by a, checks for the existence of a token on p_{init} and produces a token on the place p_{indec} of N_{dec}. We add another ε-labeled transition t_{start} that moves a token from p_{init} to p_{run}. The initial marking M_{init} assigns a single token to p_{init} and no tokens elsewhere. The final marking M_{final} coincides with M_{finaldec} on the places of N_{dec} and requires no token on the new places.

A computation of $N(n)$ from M_{init} first fires t_a an arbitrary number of times, then fires p_{run} to enable the transition in N_{dec} and ends with a computation of N_{dec}. This final part of the computation starts from a marking in which the number of tokens on p_{indec} is equal to the number of times that t_a has been fired. By Proposition 6.2.3, any covering computation of $N(n)$ has to fire t_a at least 2^{2^n} times. Firing t_a more often is also possible. Since all transitions but t_a are labeled by ε, we obtain that the language of the Petri net instance is

$$\mathcal{L}(N(n), M_{\text{init}}, M_{\text{final}}) = \left\{ a^k \;\middle|\; k \geqslant 2^{2^n} \right\}.$$

This language is upward closed and has state complexity $2^{2^n} + 1$ states, see Example 4.3.1. To conclude the proof, observe that the size of $N(n)$ is polynomial in the size of N_{dec}, which in turn is polynomial in n. ■

This completes the proof of Theorem 8.1.1.

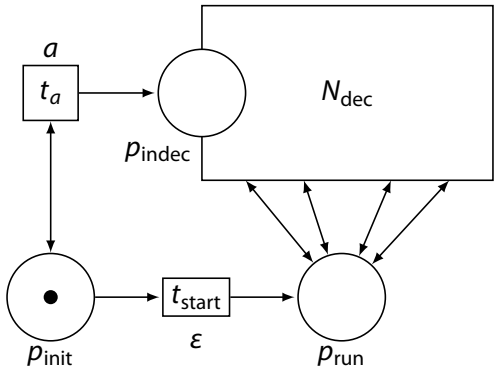


Figure 8.1.b: A schematic representation of the Petri net $N(n)$ used to prove the lower bound.

The unary case

To complete our study of the state complexity of the upward closures of Petri net languages, we consider the case in which the Petri net is encoded in unary. Let us consider the lower bound first. The transitions that are occurring in the nets whose properties we have specified in Proposition 6.2.3 use in- and outgoing multiplicities only from the set $\{0, 1\}$. This means that the size of the unary encoding of these nets is the same as the size of their binary encoding. Hence, the net $N(n)$ in the proof of Proposition 8.1.7 is still polynomial in n. We obtain the same lower bound as before.

With this result at hand, it is already clear that we cannot obtain a better upward bound. Nevertheless, it is interesting to inspect the proof of the upper bound. Assume that n is the size of the unary encoding of the given Petri net instance. We could now prove $f(i+1) \leqslant (n \cdot f(i))^{i+1} + f(i)$ for all $i \in [1, \ell - 1]$, improving the bound $f(i+1) \leqslant (2^n \cdot f(i))^{i+1} + f(i)$ from Proposition 8.1.4: If we consider an instance whose unary-encoded size is n, any transition can consume at most n tokens and the final marking can require at most n tokens on any place. However, this better bound for $f(i+1)$ does not lead to a better bound for $f(\ell)$. When computing $f(\ell)$ by evaluating the recursive formula, applying the exponent $(i+1)$ repeatedly turns out to be the dominating part. Altogether, we obtain that Theorem 8.1.1 holds for Petri nets, independently of whether we measure their size using the unary or binary encoding.

8.2 Upward closures for BPP nets

The doubly exponential state complexity of upward closures in the case of Petri net coverability languages motivates studying a restricted version. In Section 6.4, we have already introduced BPP nets as a version of Petri nets in which transitions may consume at most one token. This restricts the expressiveness of the model compared to general Petri nets, but still leads to an interesting theory, e.g. the possibility to express non-context-free languages with such nets. With respect to their practical applicability, note that BPP nets correspond to concurrent systems in which threads cannot communicate after they have been spawned, as modeled by the calculus of basic parallel processes [Chr93]. Regarding the algorithmics of BPP nets, it is well known that both the coverability and the reachability problem of BPP nets are NP-complete, compared to EXPSPACE-complete and Ackermann-complete for general Petri nets. Altogether, computing the language closures for BPP net languages is an interesting problem for which it is reasonable to hope for a better computational and state complexity than in the case of general Petri nets. We formalize this problem in the case of computing the upward closure as follows.

Computing the upward closure for BPP net languages (BPP-UC)	
Given:	A labeled BPP net instance $(N, M_{\text{init}}, M_{\text{final}})$.
Compute:	An NFA A with $\mathcal{L}(A) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$.

We show that this restriction improves the state complexity as well as the time needed to construct the NFA to be singly exponential. The following theorem formalizes this.

8.2.1 Theorem
Upward closures of BPP net coverability languages have exponential state complexity, and the corresponding automata can be constructed in exponential time. These bounds are tight.

We proceed as we did when proving the result for general Petri nets: Firstly, we show the upper bound, then the lower bound, and finally, we discuss the case of using a unary encoding.

Upper bound

The goal of this part of the section is to prove the upper bound.

8.2.2 Theorem
Given a labeled BPP net instances $(N, M_{\text{init}}, M_{\text{final}})$, we can compute in exponential time an NFA A with $\mathcal{L}(A) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$. The state complexity of $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ is at most exponential, as witnessed by automaton A.

We proceed similar as in Section 8.1: We show that the minimal words in the upward closure are generated by computations of a bounded length. However, to establish the bound, we cannot reuse the argumentation from before. Simply using the fact that the given net is a BPP net does not lead to a modification of the proof of Proposition 8.1.4 that yields an improved bound. Instead, we have to come up with a new proof approach.

We will establish that the bound is a number that depends on the transitions of the net and the amount of tokens assigned by the final marking. Since we are considering a binary encoding, the latter may be exponential in the size of the instance. Hence, k is exponential in the size of the given net. Once this result has been proven, we can proceed as in the proof of Theorem 8.1.2: We obtain the upward closure as the upward closure of the regular length-k underapproximation of the BPP net.

The bound is formalized in the form of the following proposition.

8.2.3 Proposition
For every covering computation $M_{\text{init}} \boxed{\sigma} M \geqslant M_{\text{final}}$ there is a firing sequence σ' such that $M_{\text{init}} \boxed{\sigma'} M \geqslant M_{\text{final}}$, $\lambda(\sigma') \leq \lambda(\sigma)$ and $|\sigma'| \leqslant k = (||M_{\text{final}}||_1)^2 \cdot (|T| + 1)$.

Before giving the proof, we introduce the unfolding of a BPP net, a construction that we will need in the following. The correctness of the construction will rely on an additional restriction: We require that each transition in the given BPP net consumes exactly one token. If the BPP net contains a *spontaneous* transition that consumes no token, we apply the following preprocessing. We introduce a new place p and let the initial marking put one token on this place. We also introduce a new ε-labeled transition t_{gen} with $\text{in}(t, p) = 1$, $\text{out}(t, p) = 2$ that allows us to generate an arbitrary number of tokens on this place. Every spontaneous transition t with $\text{in}(t) = \vec{0}$ is modified so that it consumes one token from place p. The final marking does not require any tokens on p.

The net that results from the processing is a BPP net in which every transition consumes exactly one token. It is only polynomially larger than the original net. Furthermore, it has the same language: A computation of the original net can be turned into a computation of the new net that generates the same word by inserting a suitable number of firings of transition t_{gen} at the beginning, and vice versa.

Hence, we may assume in the following that the preprocessing has already been applied to the net N; it contains no spontaneous transitions. We consider its unfolding, the infinite BPP net obtained by unfolding the transitions so that each layer of transitions produces tokens on a fresh layer of places. Any computation of N corresponds to a computation in a finite prefix of the unfolding. Combining the fact that the unfolding moves the tokens to fresh places after each transition and the defining properties of BPPs, this corresponding computation has a forest-like structure. By backtracking the tokens from the marking that has to been covered,

we identify subtrees in the unfolding that induce a computation in the BPP net consisting of at most $(||M_{\text{final}}||_1)^2 \cdot |T|$ transitions. Note that this bound is subtly different from the bound $(||M_{\text{final}}||_1)^2 \cdot (|T| + 1)$ stated in Proposition 8.2.3. This is to take the new transition into account which is added by the preprocessing.

Unfoldings of Petri nets are a standard notion in research on Petri nets [EH08] and not a contribution of this thesis. Unfoldings are a true concurrency semantics for Petri nets, meaning they are the true concurrency analogue of the computation tree, the latter being used for sequential models of computation. On the one hand, the unfolding represents all possible computations, just as a computation tree does. On the other hand, it reflects the concurrent nature of Petri nets by keeping track of the flow of each token individually.

To make this formal, we consider occurrence nets $O = (P', T', \text{in}', \text{out}')$, which are defined like (unlabeled) BPP nets with the following modifications: (1) O may be infinite. (2) Each place has at most one incoming transition. (3) Each transition creates at most one token per place. (4) O seen as directed graph is acyclic.

We call two elements $x, y \in P' \cup T'$ *causally related* and write $x \trianglelefteq y$ if there is a path from x to y in O seen as graph. Note that this is a partial order on the nodes of the graph: Reflexivity and transitivity obviously hold true, while antisymmetry is guaranteed by O being acyclic. We use $x{\downarrow_{\trianglelefteq}} = \{y \in P' \cup T' \mid y \trianglelefteq x\}$ to denote the ancestors of $x \in P' \cup T'$. As suggested by the notation, this is indeed the downward closure of the set $\{x\}$ wrt. the relation $\trianglelefteq$. The $\trianglelefteq$-minimal places are denoted by $\min O$. Because the relation $\trianglelefteq$ is antisymmetric, this set is uniquely determined, The initial marking of O is fixed to put one token in each place of $\min O$ and no tokens elsewhere.

Occurrence nets (together with their initial marking) are 1-safe, i.e. each place can carry at most one token in any reachable marking. This allows us to identify markings with subsets $P'' \subseteq P'$ by assuming that P'' contains exactly the places that carry a token. Similarly, we may write $P_1' \boxed{t'} P_2'$ for subsets $P_1', P_2' \subseteq P'$.

To associate an occurrence net O to a BPP net N together with an initial marking M_{init}, we use the notion of *folding homomorphisms*. Such a homomorphism is a function $h : P' \cup T' \to P \cup T$. For a place p of the original net, we think of all places p' of O with $h(p')$ as *copies* of p, similar for the transitions.

A folding homomorphism should satisfy three properties, but in order to state them, we will need some more notation. We extend h to a function that, given a subset $P'' \subseteq P'$ of places of O, returns the marking $h(P'')$ defined by $h(P'')(p) = |\{p' \in P_1' \mid h(p') = p\}|$, i.e. we count all tokens assigned to copies of place p. Furthermore, we may use that transitions in O consume and produce at most token per place and see $\text{in}'(t)$ and $\text{out}'(t)$ as subsets of P' (for some transition t). This allows us to consider $h(\text{in}'(t))$ and $h(\text{out}'(t))$ as markings.

The three properties of folding homomorphisms are the following:

(1) Initiation: The initial marking of O maps to the initial marking of N, $h(Min(O)) = M_{\text{init}}$.

(2) Consecution: in' and out' map to in and out: For all $t' \in T'$: $h(\text{in}'(t')) = \text{in}(h(t'))$, $h(\text{out}'(t')) = \text{out}(h(t'))$.

(3) Irredundancy: For all $t'_1, t'_2 \in T'$, with $\text{in}(t'_1) = \text{in}(t'_2)$ and $h(t'_1) = h(t'_2)$, we have $t'_1 = t'_2$.

A pair (O, h) consisting of an occurrence net O and a folding homomorphism h is called a *branching process* of (N, M_{init}). Branching processes are partially ordered by the prefix relation which, intuitively, states how far they unwind the BPP. The limit of the unwinding process is the *unfolding* $\text{Unf}(N, M_{\text{init}})$, the unique (up to isomorphism) maximum branching process. Note that $\text{Unf}(N, M_{\text{init}})$ is infinite in general.

We forgo making this definition more formal. Instead, we specify its properties on which we will rely in the following. There is a correspondence between computations from M_{init} in N and computations from the initial marking in $\text{Unf}(N, M_{\text{init}})$: Each computation in $\text{Unf}(N, M_{\text{init}})$ induces a unique computation of N, and each computation of N induces a set of computations that only differ by a permutation of the tokens.

Note that even if $\text{Unf}(N, M_{\text{init}})$ is infinite, each finite computation – and in particular each computation induced by a finite computation of N – only uses places and transitions whose distance from $\min O$ is bounded by the length of the computation times two.

Recall that Proposition 8.2.3 requires us to prove that ever covering computation has a covering subcomputation consisting of at most $(\|M_{\text{final}}\|_1)^2 \cdot |T|$ transitions. (Note that this is the modified bound that takes the aforementioned preprocessing step into account.) Using the unfolding, we can now prove this statement.

Proof of Proposition 8.2.3:
Consider the covering computation $M_{\text{init}} \xrightarrow{\sigma} M \geqslant M_{\text{final}}$ of the BPP net N. Let (O, h) with $O = (P', T', \text{in}', \text{out}')$ be the unfolding $\text{Unf}(N, M_0)$. We pick $\min O \xrightarrow{\tau} P''$ as an arbitrary computation of O corresponding to the computation of N, i.e. $h(\tau) = \sigma$ and $h(P'') = M$. Using $M \geqslant M_{\text{final}}$, for each place $p \in P$, the set P'' contains at least $M_{\text{final}}(p)$ places p' that get mapped to p under h. For each place p, let $X_p \subseteq P''$ be a set of size $M_{\text{final}}(p)$ of places p' with $h(p') = p$. Let $X = \bigcup_{p \in P} X_p$ be their union, and note that $h(X) = M_{\text{final}}$ holds by construction.

Since we assume that every transition in the original net consumes exactly one token, so do the transitions in the unfolding. Furthermore, they produce at most one token per place, each place has at most one incoming transition and the unfolding is acyclic. Thus, each computation in the unfolding induces a forest in O seen as graph. For each place in $\min O$, the forest contains a tree which is rooted in that place. The leaves of the forest correspond to places that carry a token in

the marking reached by the computation. The inner nodes consist of places that carry a token at some point during the computation, and the transitions that are used in the computation.

Let us consider this forest for the computation defined by the firing sequence τ. Note that X is a set of leaves by definition. We select the subforest induced by the set $X{\downarrow_{\trianglelefteq}}$, i.e. the places in X, their ancestors in the graph and the edges connecting them. We project the firing sequence τ to the transitions occurring in this subforest, obtaining the sequence τ_1. Since $X{\downarrow_{\trianglelefteq}}$ is downward closed wrt. the relation $\trianglelefteq$, τ_1 is a valid firing sequence from $\min O$ in the sense that all transitions are enabled whenever they are fired. Furthermore, τ_1 still generates all tokens in X and we have that $\min O \xrightarrow{\tau_1} P_1' \supseteq X$ is a covering computation in the sense that $h(P_1') \geqslant M_{\text{final}}$.

In the following, our goal is to delete transitions from τ_1 so that the resulting sequence τ_2 matches the bound $|\tau_2| \leqslant (||M_{\text{final}}||_1)^2 \cdot |T|$. This will allow us to show that $h(\tau_2)$, its image under the folding homomorphism, is the sequence whose existence is required by the proposition.

We first need to introduce a notion for the $\trianglelefteq$-maximal transitions in τ that lead to at least two different places in X. Formally, a transition t' is a *join transition* if there are $x, y \in X$ with $t' \in x{\downarrow_{\trianglelefteq}} \cap y{\downarrow_{\trianglelefteq}}$ and there is no $t'' \in x{\downarrow_{\trianglelefteq}} \cap y{\downarrow_{\trianglelefteq}}$ with $t' \trianglelefteq t''$.

Assume that $t' \neq t''$ are two adjacent join transitions that occur on the same branch of the subforest with $t' \trianglelefteq t''$. This means that $t' t^{(1)} \dots t^{(m)} t''$ is an infix of the sequence of transitions along that branch, where none of the $t^{(i)}$ is a join transition. Note that for two places in X, there is either no join transition or a unique one leading to these two places, since join transitions are required to be $\trianglelefteq$-maximal. Consequently, t' and t'' have to lead to different places of X.

Since t', t'' occur in τ_1, all $t^{(i)}$ also have to occur in τ_1. If there are indices $j < k$ such that $h(t^j) = h(t^k)$, we may delete $t^{(j+1)} \dots t^{(k)}$ from τ_1. To ensure validity of the computation, we need to modify $t^{(k+1)} \dots t^{(m)} t''$: Transition $t^{(k+1)}$ should consume the token produced by $t^{(j)}$ (instead of the token produced by the deleted transition $t^{(k)}$) and so on. This is possible since $h(t_j) = h(t_k)$ implies $h(\text{in}'(t_j)) = h(\text{in}'(t_k))$ by Consecution, Property 2 of folding homomorphisms. Furthermore, we exhaustively need to delete from τ all transitions that rely on tokens that are not produced anymore, starting with transitions relying on tokens produced by the transitions $t^{(j+1)} \dots t^{(k)}$. To see that the resulting computation still covers, note that we have not deleted any join transitions. Hence, the leaves of the subforest that are removed by deleting the transitions are not contained in X.

We repeat this deletion process until we obtain that between each two join transitions of the same branch of the subforest, there are no repetitions, i.e. transitions whose image under h is the same. Let the resulting transition sequence be τ_2, inducing the computation $\min O \xrightarrow{\tau_2} P_2' \supseteq X$. Firstly, note that for any $x \in X$, there are at most $||M_{\text{final}}||_1$ join transitions on the branch from the corresponding minimal place to x: In the worst case, for each place in $X \setminus \{x\}$, there is a join transition on the branch, and $|X| = ||M_{\text{final}}||_1$. Between any two adjacent join transitions along such a path, there are at most $|T|$ transitions in τ_2 Hence, the number of

transitions in such a path is bounded by $||M_{\text{final}}||_1 \cdot |T|$. Since we have $||M_{\text{final}}||_1$ places in X, the total number of transitions in τ_2 is bounded by $(||M_{\text{final}}||_1)^2 \cdot |T|$.

To conclude the proof, consider the firing sequence $\sigma' = h(\tau_2)$ obtained by applying the folding homomorphism to τ_2 and the computation $M_{\text{init}} \boxed{\sigma'} M_2$ of N that is induced by it. We indeed have $M_2 \geqslant M_{\text{final}}$ since $M_2 = h(P_2) \geqslant h(X) = M_{\text{final}}$ and $|\sigma'| = |h(\tau_2)| = |\tau_2| \leqslant (||M_{\text{final}}||_1)^2 \cdot |T|$. ■

With Proposition 8.2.3 at hand, we can prove Theorem 8.2.2. We consider the length-k approximation for $k = (||M_{\text{final}}||_1)^2 \cdot |T|$, and obtain an automaton for $\mathcal{L}_{\leqslant k}(N, M_{\text{init}}, M_{\text{final}})$ of size at most $\mathcal{O}(k^n \cdot 2^{n^2})$, see Proposition 6.3.2. We compute

$$k^n \cdot 2^{n^2} = \left((||M_{\text{final}}||_1)^2 \cdot (|T|+1)\right)^n \cdot 2^{n^2} \leqslant \left(2^n \cdot |T|\right)^n \cdot 2^{n^2} = 2^{n^2} \cdot |T| + 1^n \cdot 2^{n^2} = 2^{2n^2} \cdot (|T|+1)^n,$$

which is exponential as required. The upward closure of the language of this automaton can be constructed without adding new states, see Section 7.2. To show correctness, i.e. this upward closure coinciding with the upward closure of the original BPP net language, we proceed exactly as in the proof of Theorem 8.1.2.

Lower Bound

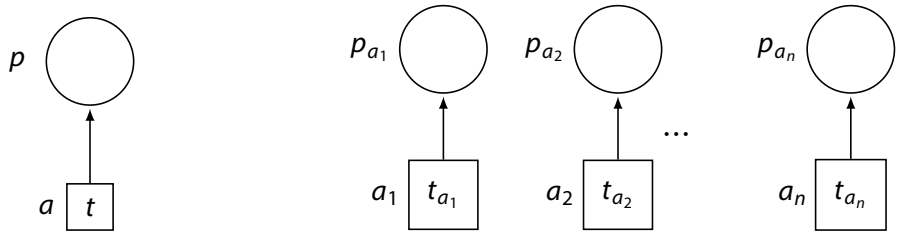
To prove that the above construction is optimal, we present a family of BPP languages for which the state complexity of the upward closure is exponential in the size of the nets. The proof uses the fact that we can encode the exponential number 2^n occurring in the final marking using only $\log 2^n = n$ bits.

8.2.4 Proposition
For each $n \in \mathbb{N}$, there is a labeled BPP net instance $(N, M_{\text{init}}, M_{\text{final}}(n))$ of size polynomial in n such that $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}(n))\uparrow$ has state complexity at least 2^n.

Proof:
The net N required to prove the proposition is depicted in Figure 8.2.a.i). It consists of a single place p, and a single a-labeled transition t that produces a token on this place. The initial marking M_{init} is the zero vector, the final marking M_{final} requires 2^n tokens on p.

Obviously, to cover the final marking, transition t needs to be fired at least 2^n times. Hence, $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}(n)) = \{a^k \mid k \geqslant 2^n\}$. This language is already upward closed and has state-complexity $2^n + 1$, see Example 4.3.1. Both N and M_{init} are of constant size. As mentioned above, $|M_{\text{final}}|$ is of size polynomial in n since the size of a marking is defined based on the length of the binary encoding of the numbers. ■

With the lower bound at hand, the proof of Theorem 8.2.1 is completed.



i) In the binary case. ii) In the unary case.

Figure 8.2.a: BPP nets to prove the exponential lower bound for the upward closure.

The unary case

We complete our study by considering the case in which the BPP net is encoded in unary. We start by inspecting the upper bound, noting that the bound $k = (||M_{\text{final}}||_1)^2 \cdot |T|$ provided by Proposition 8.2.3 is polynomial in the size of the net if we consider the unary encoding. However, we then use this bound to construct an automaton with state space $[1, k] \times (P \to [0, (k+1) \cdot n])$. This is a modification of Proposition 6.3.2 that takes into account that with the net of size n being encoded in unary, each transition can add at most n tokens. This set of states is of size

$$(k+1) \cdot (((k+1) \cdot n) + 1)^{|P|},$$

i.e. it is exponential in the number of places, even if k is a number that is polynomial in the size of the net. In summary, we still obtain an exponential upper bound.

Let us see whether the lower bound matches. Obviously, the net that we have constructed in Proposition 8.2.4 to prove the lower bound in the binary case does not do the job. The net uses a final marking that requires 2^n tokens on a place, so its unary encoding is of exponential size. Can a net with the language $\{a^k \mid k \geqslant 2^n\}$ whose unary encoding is polynomial in n be constructed in a different way? To the best of the author's knowledge, there is no such construction. However, there is a different language that will provide a lower bound that matches the upper bound, i.e. it is exponential in the number of places.

Let $n \in \mathbb{N}$, and consider an alphabet $\Sigma = \{a_1, \ldots, a_n\}$ of size n. We define the language $\mathcal{L}_{\text{all}}$ as the language of all words over Σ that contain each letter at least once,

$$\mathcal{L}_{\text{all}} = \left\{w \in \Sigma^* \;\middle|\; \forall i \in [1, n] \exists j \in [1, |w|] \colon w_j = a_i\right\} = \Sigma^* a_1 \Sigma^* \cap \ldots \cap \Sigma^* a_n \Sigma^* .$$

This language and its complement $\overline{\mathcal{L}_{\text{all}}}$, the set of words in which at least one letter does not occur, are well-known examples in the context of the state complexity of regular languages. Both languages can be easily represented by a DFA of size 2^n: The DFA has one state for each subset of Σ and tracks the set of letters that has occurred. One can find a small NFA of size $n+1$ that accepts $\overline{\mathcal{L}_{\text{all}}}$ by initially guessing a letter that will not occur and verifying that guess later. The same is not true for $\mathcal{L}_{\text{all}}$: Any NFA for this language has at least 2^n states, just as the DFA. Hence, the

pair of languages shows that complementing a regular language can increase its state complexity exponentially. This behavior has been first observed by Sakoda and Sipser [SS78], although on a pair of languages that is more complex to define. Later, Birget [Bir93] provided a simpler proof for this fact (but still based on the languages from [SS78]). This proof uses an extension of the fooling set technique for DFAs to NFAs. For the sake of completeness, we give a proof for the fact that $\mathcal{L}_{\text{all}}$ cannot be represented by a small NFA that uses the argumentation by Birget.

8.2.5 Lemma
For each positive $n \in \mathbb{N}$, $\mathcal{L}_{\text{all}}$ has state complexity 2^n.

Proof:
As mentioned before, it is easy to construct a DFA – which can be seen as NFA – with the desired number of states. It remains to show that there is no NFA with a smaller number of states.

Let us assume that A is an NFA with $\mathcal{L}(A) = \mathcal{L}_{\text{all}}$. For each subset $\Gamma \subseteq \Sigma$ of the alphabet, we let w_Γ be an arbitrary word that contains exactly the letters from Γ. For example, we may assume $\Gamma = \{a_{i_1}, \ldots, a_{i_{|\Gamma|}}\}$ with $1 \leqslant i_1 < i_2 < \ldots < i_{|\Gamma|} \leqslant n$ and define $w_\Gamma = a_{i_1} \ldots a_{i_{|\Gamma|}}$.

Note that for each $\Gamma \subseteq \Sigma$, the word $w_\Gamma.w_{\Sigma\setminus\Gamma}$ is an element of $\mathcal{L}_{\text{all}}$ because it contains each letter. Hence, we can define the state q_Γ to be the state that occurs after w_Γ has been processed in an accepting run on $w_\Gamma.w_{\Sigma\setminus\Gamma}$, i.e.

$$q_{\text{init}} \xrightarrow{w_\Gamma} q_\Gamma \xrightarrow{w_{\Sigma\setminus\Gamma}} q_{\text{final}} \in Q_{\text{final}} .$$

If multiple accepting runs exist, we pick an arbitrary one.

We claim that if $\Gamma \neq \Gamma'$ are two distinct subsets of Σ, then $q_\Gamma \neq q'_\Gamma$. Towards a contradiction, assume equality, i.e. we have runs

$$q_{\text{init}} \xrightarrow{w_\Gamma} q_\Gamma \xrightarrow{w_{\Sigma\setminus\Gamma}} q_{\text{final}} \in Q_{\text{final}} ,$$

$$q_{\text{init}} \xrightarrow{w_{\Gamma'}} q_\Gamma \xrightarrow{w_{\Sigma\setminus\Gamma'}} q'_{\text{final}} \in Q_{\text{final}} .$$

Since $\Gamma \neq \Gamma'$, there is some letter that occurs only in one of the subalphabets. Wlog., we assume that $a_i \in \Gamma$, but $a_i \notin \Gamma'$. Consider the word $w_{\Gamma'}.w_{\Sigma\setminus\Gamma}$. This word is not contained in $\mathcal{L}_{\text{all}}$, cause neither $w_{\Gamma'}$ nor $w_{\Sigma\setminus\Gamma}$ contain letter a_i. However, we can construct an accepting run $q_{\text{init}} \xrightarrow{w_{\Gamma'}} q_\Gamma \xrightarrow{w_{\Sigma\setminus\Gamma}} q_{\text{final}} \in Q_{\text{final}}$ on that word, proving that it is contained in $\mathcal{L}(A)$. We obtain a contradiction to $\mathcal{L}(A) = \mathcal{L}_{\text{all}}$.

Hence, automaton A needs to contain a distinct state q_Γ for each subalphabet $\Gamma \subseteq \Sigma$. The number of subalphabets is $2^{|\Sigma|} = 2^n$, which means A needs to have at least 2^n states. This completes the proof. ■

With the lemma at hand, it remains to see that $\mathcal{L}_{\text{all}}$ occurs as the language of a BPP net whose unary encoding is of size polynomial in n. The required net is rather simple: It counts how often each letter has been generated. We depict it in Figure 8.2.a.ii). For each $i \in [1, n]$, there is a place p_{a_i} and an a_i-labeled transition t_{a_i} that consumes no tokens and produces a token on p_{a_i}. The initial marking puts no tokens anywhere, while the final marking is $\vec{1}$, i.e. it is covered once at least one token has been produced on every place.

Altogether, we obtain matching lower and upper bounds: The state complexity of the upward closure of a BPP net is a number that is exponential in the number of places of the net.

8.3 SRE inclusion in the upward closures for Petri nets

In the first two sections, we have established that the upward closure can be computed in doubly exponential time for Petri nets and in exponential time for BPP nets. These intractable complexities motivate studying further restrictions of the problem. We have argued before that the upward and downward closure of any language can be represented by simple regular expressions (SREs). This result yields a new approach to the closures of Petri net languages: Instead of computing the closure, we check for a given SRE whether the closure is equal to the language of that SRE. Let us focus on the case of the upward closure for now. Formally, we will check $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow = \mathcal{L}(sre)$, where $(N, M_{\text{init}}, M_{\text{final}})$ is the given Petri net and sre is an SRE. This SRE may come from an oracle that iteratively creates candidate SREs based on membership queries and on whether the checks for the previously generated candidates have been successful.

Checking the equality decomposes into checking both inclusion. Checking one of the inclusions is conceptually easy, namely checking $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow \subseteq \mathcal{L}(sre)$. We proceed as follows: We use that the inclusion holds if and only if the intersection $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow \cap \overline{\mathcal{L}(sre)}$ is empty. To represent $\overline{\mathcal{L}(sre)}$, we construct an NFA for the language of the SRE, determinize it and construct a DFA for the complement which we see as a Petri net in the following. To represent $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$, we may use the Petri net $N\uparrow$ whose language is $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$. Finally, we construct the synchronized product of the two nets and check language emptiness, which amounts to a coverability query.

The complexity of this procedure depends on the size of the representation of $\overline{\mathcal{L}(sre)}$. In principle, an NFA for the complement of a regular language could be exponentially larger than an NFA for the language itself. We leave it as future work to see whether this worst-case behavior can actually occur in the case of languages defined by SREs. Instead, we focus on checking the other inclusion.

The reason for focusing on the inclusion $\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ is the following: We envision a refinement procedure in which an oracle iteratively outputs candidate SREs $sre_1, sre_2, \ldots$.For each candidate, we check whether the inclusion $\mathcal{L}(sre_i) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ holds. The answer to each check is sent to the oracle, which uses it for refinement. Once we are sufficiently sure that some sre_i represents the actual upward closure of the language (e.g. because the inclusion $\mathcal{L}(sre_i) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ holds, but the inclusion for SREs with a larger language fails), we check $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow \subseteq \mathcal{L}(sre_i)$. This procedure uses a high number of inclusion checks with the SRE as the left-hand side, compared to a low number of inclusion checks with the SRE as the right-hand side.

Formally, our goal in this section is to study the complexity of the following decision problem.

SRE inclusion in the upward closure (PN-SREUC)	
Given:	A labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$, an SRE *sre*.
Question:	$\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$?

We will show that this problem is EXPSPACE-complete. This is an improvement over the doubly exponential time needed to construct a representation of the upward closure, as proven in the first section of this chapter. Furthermore, it matches the complexity of the coverability problem. Indeed, the lower bound is straightforward reduction from coverability, while the upper bound reduces PN-SREUC to a polynomial number of membership queries.

8.3.1 Theorem
PN-SREUC is EXPSPACE-complete.

We proceed by proving the upper bound, i.e. EXPSPACE membership, and the lower bound, EXPSPACE-hardness, separately.

Upper bound

We show one direction of Theorem 8.3.1 by proving EXPSPACE membership.

8.3.2 Proposition
The problem PN-SREUC can be solved in exponential space.

To prove the proposition, we use the following observation. A language $\mathcal{L}$ is contained in an upward-closed language $\mathcal{L}'$ if every minimal word (wrt. the subword relation) of $\mathcal{L}$ is a member of $\mathcal{L}'$, i.e. $\min \mathcal{L} \subseteq \mathcal{L}'$. Indeed, $\mathcal{L} = \min \mathcal{L}\uparrow \subseteq \mathcal{L}'\uparrow = \mathcal{L}'$ holds in this case. We even know that the set of minimal words of a language is finite, by Property 3 from Lemma 6.5.3 and the fact that the subword relation is a WQO. The problem with using this observation to decide inclusions is that it can be difficult or even impossible to compute the set of minimal words.

Here, we use that for an SRE, it is easy to construct the set of minimal words in polynomial time. To be precise, we can construct for each product p in the SRE of interest a word $\min p$ that is the unique minimal word in $\mathcal{L}(p)$, i.e. $(\min p)\uparrow = \mathcal{L}(p)\uparrow$, in linear time. It then only remains to check that for each product p of the given SRE, $\min p$ is contained in the upward closure of $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$. To conduct this check, we do not need to construct an NFA representation of the upward closure. It is sufficient to use the net $N\uparrow$ whose size is polynomial in the size of N.

We show how to extract from a product its minimal word, then give the formal proof.

8.3.3 Definition
Let p be the product of an SRE. We define $\min p$ by induction as follows:

$$\min a = a \qquad \min p.p' = \min p.\min p'$$
$$\min a \cup \varepsilon = \varepsilon \qquad \min \Gamma^* = \varepsilon\,.$$

Using the definitions, it is easy to see that for each product p, we have $\mathcal{L}(p)\uparrow = \min p\uparrow$. To obtain $\min \mathcal{L}(sre)$, the minimal words of the language of an SRE, we can construct the minimal words for each of the products. Some of them might be comparable, so we obtain $\min \mathcal{L}(sre)$ by removing the non-minimal ones. The proof of Proposition 8.3.2 follows immediately.

Proof of Proposition 8.3.2:
Let $(N, M_{\text{init}}, M_{\text{final}})$ be the given net and let $sre = p_1 \cup \ldots \cup p_k$ be the given SRE, consisting of the products p_i. As argued above, we have that $\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ if and only if for each i, $\min p_i \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$. We consider the net $N\uparrow$ with $\mathcal{L}(N\uparrow, M_{\text{init}}, M_{\text{final}}) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$, and check $\min p_i \in \mathcal{L}(N\uparrow, M_{\text{init}}, M_{\text{final}})$. Note that both the number of products and the length of each $\min p_i$ is polynomial in the size of the SRE. Hence, we have reduced the problem PN-SREUC to a polynomial amount of membership queries for instances of polynomial size. The desired result follows as PNCOV-WORD is in EXPSPACE, see Section 6.3. ■

Lower bound

The lower bound for PN-SREUC follows directly from Theorem 6.2.2, the EXPSPACE-hardness of coverability.

8.3.4 Lemma
PN-SREUC is EXPSPACE-hard.

Proof:
We reduce from the coverability problem for unlabeled Petri nets, which is EXPSPACE-hard by Theorem 6.2.2. Let $(N, M_{\text{init}}, M_{\text{final}})$ be a Petri net instance. We see N as a labeled Petri net in which all transitions are labeled by ε. Consequently, we have $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = \{\varepsilon\}$ if M_{final} is coverable from M_{init} in N, else $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = \varnothing$. Hence, $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ is either Σ^* or $\varnothing$. We obtain that the instance $\varnothing^* \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ of PN-SREUC is equivalent to the coverability problem, since the SRE $\varnothing^*$ expresses the language $\{\varepsilon\}$. ■

This completes the proof of Theorem 8.3.1. Note that the same result holds in the case of the unary encoding, since coverability remains EXPSPACE-complete for nets encoded in unary: Lipton's construction only uses transition multiplicities from the set $\{0, 1\}$.

8.4 SRE inclusion in the upward closures for BPP nets

We complete the chapter by combining the two restrictions: We consider the problem of deciding whether a given SRE defines a language that is included in the upward closure of the language of a given BPP net.

SRE inclusion in the upward closure for BPP nets (BPP-SREUC)	
Given:	A labeled BPP net instance $(N, M_{\text{init}}, M_{\text{final}})$, an SRE *sre*.
Question:	$\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$?

We show that this decision problem is complete for the class NP. As in the case of general Petri nets, this means that the complexity matches the complexity of the coverability problem.

8.4.1 Theorem
BPP-SREUC is NP-complete.

The proofs of membership and hardness are extremely similar to the proofs in the case of general Petri nets. To show the upper bound, we proceed as in Proposition 8.3.2. We compute in polynomial time the minimal word of each product and check membership in the upward closure of the language of the given BPP net. The latter is represented by the net $N\uparrow$ which is a BPP net if N is. Since the word problem for BPP nets can be solved in NP, Proposition 6.4.8, this completes the proof.

The lower bound can be proven analogous to Lemma 8.3.4. The coverability problem for BPP nets is NP-hard, Lemma 6.4.4, and $\varnothing^* \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ holds if and only if $(N, M_{\text{init}}, M_{\text{final}})$ is a yes-instance of coverability.

The result holds even if we consider the net to be encoded in unary. The NP-hardness of coverability constructs a net with transition multiplicities in $\{0, 1\}$.

Altogether, this finishes our study of the upward closure of Petri net coverability languages. We have proven all results regarding upward closures that we mentioned in Section 7.3.

9 Downward closures

Contents

9.1 Downward closures for Petri nets 175
9.2 Downward closures for BPP nets 183
9.3 SRE inclusion in the downward closures for Petri nets 191
9.4 SRE inclusion in the downward closures for BPP nets 198

In this chapter, we study the downward closures of Petri net coverability languages. Our goal is to obtain a finite automaton representing it with a minimal number of states. The structure of the chapter is similar to the structure of Chapter 8. Firstly, we consider the problem of computing a representation of the upward closure of a Petri net. This problem has already been solved in [HMW10] with a procedure that is based on the Karp-Miller tree. Hence, the resulting automaton can be of non-primitive recursive size. Here, we complement the result from [HMW10] by a matching lower bound, using the well-known fact that a Petri net can weakly compute a variant of the Ackermann function.

This high complexity motivates studying the same restrictions that we have considered in Chapter 8: We consider the problems of computing the downward closure of BPP net languages, of checking whether a given SRE is included in the downward closure of a given Petri net, and finally whether an SRE is included in the downward closure of a BPP net. The results that we obtain are similar to the results in the previous chapter. The upward closure of a BPP net can be computed in exponential time, which we will show by proving some sort of pumping lemma for BPP nets. The SRE-inclusion problems are EXPSPACE and NP-complete in the case of Petri nets and BPP nets, respectively. While the complexities match those for the problems PN-SREUC and BPP-SREUC, the proofs are much more involved. Also note that in contrast to the results from the previous chapter, the complexities of the three restricted versions are drastic improvements over the non-primitive complexity of the problem in the general case.

9.1 Downward closures for Petri nets

We first consider the problem of computing the downward closure of a given Petri net coverability language, formalized as follows.

Computing the downward closure (PN-DC)	
Given:	A labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$.
Compute:	An NFA A with $\mathcal{L}(A) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$.

Our goal is to prove the following result.

9.1.1 Theorem
Downward closures of Petri net coverability languages have non-primitive-recursive state complexity, and the corresponding automata can be constructed in non-primitive recursive time. These bounds are tight.

The computability has already been shown by Habermehl, Meyer, and Wimmel [HMW10]. We will start by recalling the proof. Later, we give a matching lower bound.

Computability / Upper bound

It has been shown in [HMW10] that the downward closure of Petri net languages is computable. In addition to the main result, which shows the computability for Petri net reachability languages, the paper also provides an algorithm specifically for coverability languages. The advantage of the latter is that the construction does not rely on techniques from the proof of the decidability of Petri net reachability [Lam92], but only on the coverability graph [KM69].

9.1.2 Theorem (Habermehl, Meyer, and Wimmel [HMW10])
The downward closures of Petri net languages are computable.

We briefly recall how, given a Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$, an NFA A representing the downward closure, i.e. $\mathcal{L}(A) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$, can be constructed. For the proof of correctness, we refer to [HMW10]. The construction relies on the *Karp-Miller tree* [KM69] associated to the Petri net, a standard construction to represent all markings that can be covered from the initial one.

To explain the construction, we will need the notion of *generalized markings* that we have presented in Section 6.3. We briefly recall the definition. We define $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$, the natural numbers plus a new top element. The order and addition are extended from $\mathbb{N}$ to $\mathbb{N}_\omega$ by defining $n < \omega$ and $\omega + n = \omega - n = \omega$ for all numbers $n \in \mathbb{N}$. (In the following, we will never need to add ω to or subtract ω from itself.) A *generalized marking* is a function $M\colon P \to \mathbb{N}_\omega$ that assigns to each place a finite number of tokens, or the special value ω. Intuitively, $M(p) = \omega$ means that p carries unboundedly many tokens. A transition t is enabled in generalized marking M if $M \geqslant \text{in}(t)$ (which means that places p with $M(p) = \omega$ are essentially ignored), and firing it then yields $M \boxed{t}\!\!> M + e(t)$.

With these preliminaries at hand, we can specify the construction of the Karp-Miller tree. It is a finite tree in which the nodes are labeled by generalized markings, and in which the edges are labeled by transitions. During the construction, we store for each node of the graph whether it has been processed or not. The construction is set up so that all unprocessed nodes are leaves of the tree.

Initially, we consider a tree that solely consists of the root node that is labeled by the initial marking M_{init} and not yet processed. While the tree contains an unprocessed leaf, we proceed as follows: Pick an unprocessed leaf v, say labeled by generalized marking M, and mark it as processed. For each transition t that is enabled in M, compute M' with $M \boxed{t}\!\!> M'$. For each place p, check whether on the unique path from the root of the tree to v, there is a marking $\hat{M}$ with $\hat{M} \leqslant M'$ and $\hat{M}(p) < M'(p)$. If so, we modify M' by setting $M'(p) = \omega$. After iterating over all places, we insert a new leaf v' into the tree, labeled by M', and add a t-labeled edge from v to v'. If the tree already contains a node labeled by M', mark v' as processed. Else, it is unprocessed.

Intuitively, the construction explores the behavior of the Petri net while tracking *accelerations* or *pumps*, i.e. infixes of the computation that lead from a marking $\hat{M}$ to a marking M' that is strictly greater than $\hat{M}$. If such a pump has been discovered, it could be inserted as often as desired to obtain arbitrarily high token counts on the places p with $\hat{M}(p) < M'(p)$.

The termination of the algorithm relies on WQO arguments: $(\mathbb{N}_\omega, \leqslant)$ is a WQO, similar to $(\mathbb{N}, \leqslant)$, see Part b) of Example 6.5.1. Hence, also the product order on $\mathbb{N}_\omega^P$ is a WQO. The WQO properties guarantee that each path in the tree is finite: After finitely many steps, for each place we have either discovered a pump that sets this place to ω in the successor, or we have explored all the finitely many values that $M(p)$ can obtain in reachable markings. Now observe that the out-degree of each node of the tree is bounded by the number of transitions. Both facts combined yield the finiteness of the Karp-Miller tree.

One should mention that the Karp-Miller tree is not unique, as its construction depends on the order in which unprocessed leafs and transitions are considered. However, the interesting properties of the tree are independent of this order. In the literature, the Karp-Miller tree is often transformed into the so-called *coverability graph* by merging all nodes that are labeled by the same generalized marking. For our purpose, we will keep the tree structure.

The Karp-Miller tree is commonly used to represent the set *coverability set* of a net N together with an initial marking M_{init}. A marking M_{final} is coverable in N from M_{init} (i.e. it is contained in the coverability set) if and only if the Karp-Miller tree contains some node labeled by a generalized marking M with $M_{\text{final}} \leqslant M$. Since the tree is finite, this yields a simple procedure to decide coverability. Unlike Rackoff's procedure [Rac78], its worst-case complexity is not optimal, as we will discuss below.

Habermehl, Meyer, and Wimmel [HMW10] provide a procedure that constructs the downward closure of a Petri net language based on the Karp-Miller tree. Consider the NFA $A = (Q, \delta, v_{\text{init}}, Q_F)$, where Q is the set of nodes of the Karp-Miller tree, v_{init} is the root node, and Q_F is the set of nodes labeled by generalized markings M' with $M_{\text{final}} \leqslant M'$. The transitions decompose into the transitions induced by the Karp-Miller tree and back edges. Each edge $v \to v'$ in the Karp-Miller tree, say labeled by transition t, induces the transition $v \xrightarrow{\lambda(t)} v'$ in A. If node v is labeled by the generalized marking M, and $\hat{v}$ is a node on the unique path from the root to v, labeled by a marking $\hat{M}$ with $\hat{M} \leqslant M$, we add the transition $v \xrightarrow{\varepsilon} \hat{v}$.

NFA A has the property that its downward closure coincides with the downward closure of the coverability language.

9.1.3 Lemma (Habermehl, Meyer, and Wimmel [HMW10])
$\mathcal{L}(A)\downarrow = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$.

A representation for $\mathcal{L}(A)\downarrow$ is easy to compute by inserting an ε-labeled variant of each transition in A downward Section 7.2. Hence, this proves Theorem 9.1.2.

The result is not accompanied by a statement on the complexity of the algorithm and the state complexity of the downward closure. An upper bound for the latter is the size of the Karp-Miller, which is well-known to be non-primitive recursive in the worst case. This fact can be shown similar to the proof of our lower bound below.

Lower bound

Our lower bound will be based on the well-known fact that Petri nets can weakly compute fast-growing functions. We say that a Petri net weakly computes a function f if for each number m of tokens on a special input place in the initial marking, there is a computation that produces $f(m)$ tokens on a special output place, and no computation produces more than $f(m)$ tokens on the output place. More formally, consider a net N with a designed input place p_{in} and a designed output place p_{out} and an initial marking M_{init}. We define by $M_{\text{init}}(m)$ the marking that coincides with M_{init} but for $M(p_{\text{in}}) = m$. The net N *weakly computes* function $f\colon \mathbb{N} \to \mathbb{N}$ if for any $m \in \mathbb{N}$ and any marking M reachable from $M_{\text{init}}(m)$, $M(p_{\text{out}}) \leqslant m$ holds, and there is at least one reachable marking M with $M(p_{\text{out}}) = m$.

It has been observed by Mayr and Meyer [MM81] that a function closely related to the non-primitive recursive Ackermann function can be computed by a Petri net. We adapt their construction to show that for each number $n \in \mathbb{N}$ there is a Petri net of size linear in $n \in \mathbb{N}$ that weakly computes the function Acker$(n, -)$, i.e. a version of the Ackermann function in which the first parameter is fixed. With this net at hand, we will be able to prove the following result.

9.1.4 Proposition
For all $n, m \in \mathbb{N}$, there is a Petri net instance $(N(n), M_{\text{init}}(n, m), M_{\text{final}}(n))$ of size polynomial in $n + m$ such that $\mathcal{L}(N(n), M_{\text{init}}(n, m), M_{\text{final}}(n)) = \{a^k \mid k \leqslant \text{Acker}(n, m)\}$.

This language is already downward closed and any NFA for it needs at least Acker(n, m) states, see Example 4.3.1. Hence, it proves that the downward closure of a Petri net coverability language can have non-primitive recursive state complexity. This also implies that the time needed to construct the corresponding automaton may be non-primitive recursive. Proposition 9.1.4 together with Theorem 9.1.2 prove the main result of this section, Theorem 9.1.1.

The rest of this section is dedicated to the proof of Proposition 9.1.4. We first present an unlabeled family of Petri nets $(\mathrm{AN}(n))_{n\in\mathbb{N}}$ such that $\mathrm{AN}(n)$ weakly computes $\mathrm{Acker}(n,-)$. Later, we show how to modify it to obtain the family of language required by the proposition.

The definition of $\mathrm{AN}(n)$ is inductive in n and imitates the definition of the Ackermann function.

9.1.5 Definition

We define the Petri net $\mathrm{AN}(0)$ to be

$$\begin{aligned}\mathrm{AN}(0) &= (P^0, T^0, \mathrm{in}^0, \mathrm{out}^0) \quad \text{with}\\ P^0 &= \left\{\mathrm{in}^0, \mathrm{out}^0, \mathrm{start}^0, \mathrm{stop}^0, \mathrm{move}^0\right\}, \text{and}\\ T^0 &= \left\{t^0_{\mathrm{start}}, t^0_{\mathrm{stop}}, t^0_{\mathrm{move}}\right\}.\end{aligned}$$

The transition multiplicities are given by Figure 9.1.a, where each edge has a weight of 1.

For $n \in \mathbb{N}$, we define $\mathrm{AN}(n+1)$ inductively by

$$\begin{aligned}\mathrm{AN}(n+1) &= (P^{n+1}, T^{n+1}, \mathrm{in}^{n+1}, \mathrm{out}^{n+1}) \quad \text{with}\\ P^{n+1} &= P^n \cup \left\{\mathrm{in}^{n+1}, \mathrm{start}^{n+1}, \mathrm{move}^{n+1}, \mathrm{out}^{n+1}, \mathrm{stop}^{n+1}, \mathrm{swap}^{n+1}, \mathrm{temp}^{n+1}\right\}, \text{and}\\ T^{n+1} &= T^n \cup \left\{t^{n+1}_{\mathrm{start}}, t^{n+1}_{\mathrm{move}}, t^{n+1}_{\mathrm{stop}}, t^{n+1}_{\mathrm{restart}}, t^{n+1}_{\mathrm{in}}, t^{n+1}_{\mathrm{swap}}, t^{n+1}_{\mathrm{temp}}\right\}.\end{aligned}$$

The transition multiplicities are given by Figure 9.1.b, where again each edge has a weight of 1.

Let us furthermore define for each $m \in \mathbb{N}$ the marking $M_{\mathrm{init}}(n,m)$ of $\mathrm{AN}(n)$ that places one token on start^n, m tokens on in^n and no token elsewhere.

We show that the family of nets $(\mathrm{AN}(n))_{n\in\mathbb{N}}$ has the desired property.

9.1.6 Lemma

For all $n \in \mathbb{N}$, $\mathrm{AN}(n)$ weakly computes $\mathrm{Acker}(n,-)$: For each $m \in \mathbb{N}$, there is a computation $M_{\mathrm{init}}(n,m) \xrightarrow{\sigma} M$ of $\mathrm{AN}(n)$ such that $M(\mathrm{out}^n) = \mathrm{Acker}(n,m)$, $M(\mathrm{stop}^n) = 1$, and there is no computation $M_{\mathrm{init}}(n,m) \xrightarrow{\sigma} M$ with $M(\mathrm{out}^n) > \mathrm{Acker}(n,m)$.

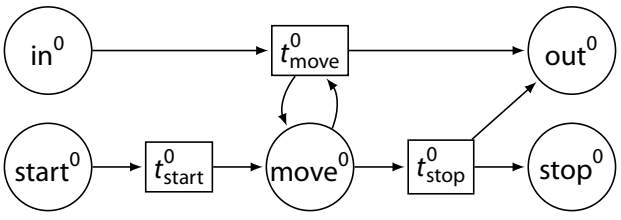


Figure 9.1.a: The Petri net AN(0).

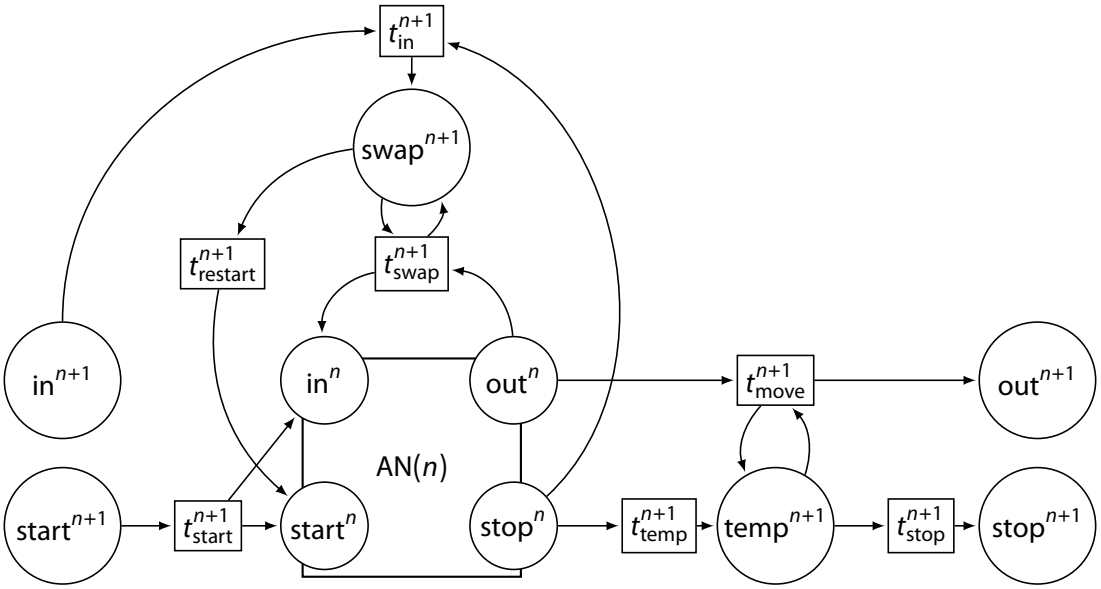


Figure 9.1.b: The Petri net AN($n+1$).

Before starting the proof, recall the recursive definition of the Ackermann function that we have already given in Section 3.3: $\text{Acker}(0,m) = m+1$, $\text{Acker}(n+1,0) = \text{Acker}(n,1)$, and $\text{Acker}(n+1,m+1) = \text{Acker}(n,\text{Acker}(n+1,m))$. We will need the following two properties of the Ackermann function, which are easy to show using induction. For all $n, m, m' \in \mathbb{N}$, we have

$$\text{Acker}(n+1,m) = \underbrace{\text{Acker}(n,\text{Acker}(n,\ldots\text{Acker}(n,1)\ldots))}_{(m+1)\text{ times}} \tag{A1}$$

$$\text{Acker}(n,m) + m' \leqslant \text{Acker}\big(n,m+m'\big) \tag{A2}$$

$$\text{Acker}(n,m) \leqslant \text{Acker}(n,\text{Acker}(n,m))\,. \tag{A3}$$

Proof of Lemma 9.1.6:
We proceed by an induction on n that proves both statements simultaneously.

Base case, $n = 0$: All firing sequences that are enabled from $M_{\text{init}}(n,m)$ and produce a token on stop^0 are of the shape $t_{\text{start}}^0.(t_{\text{move}}^0)^k.t_{\text{stop}}^0$, where $k \in [0,m]$. Such a transition sequence creates between 1 and $m+1 = \text{Acker}(0,m)$ tokens on out^0. It is impossible to create more than $m+1$ tokens on out^0.

Inductive step, $n \to n+1$: We first demonstrate how to create a token on stop^n and $\text{Acker}(n,1)$ tokens on out^n. The only transition that is enabled in $M_{\text{init}}(n,m)$ is t_{start}^{n+1}. Firing it creates one token on in^n and one token on start^n. We can now execute the computation of AN(n) that creates $\text{Acker}(n,1)$ tokens on out^n and one token on stop^n, which exists by induction.

Next, we show how to create Acker(n, Acker(n, 1)) tokens on out^n. Using t_{in}^{n+1}, we consume one token from in^{n+1} and the token on stop^n to create a token on swap^{n+1}. This token allows us to swap all Acker(n, 1) tokens from out^n to in^n using t_{swap}^{n+1}. After doing this, we move the token from swap^{n+1} to t_{start}^{n} using the restart transition t_{restart}^{n+1}. We are now able to execute the computation of AN(n) that creates Acker(n, Acker(n, 1)) tokens on out^n and one token on stop^n, which exists by induction.

The process described in the previous paragraph can be repeated for each of the m tokens on place in^{n+1}. After doing so, we end up with one token on stop^n and $\underbrace{\text{Acker}(n, \text{Acker}(n, \ldots \text{Acker}(n,}_{(m+1)\text{ times}} 1)\ldots)) = \text{Acker}(n+1, m)$ tokens on out^n using Equation (A1).

It remains to use t_{temp}^{n+1} once to get a token on temp^{n+1}. Then we can transfer all Acker($n+1, m$) tokens from out^n to out^{n+1} with t_{move}^{n+1}. Finally, we use t_{stop}^{n+1} to create a token on t_{stop}^{n+1}.

To prove the second part of the statement, we argue that it is not possible to create more than Acker($n+1, m$) tokens on out^n. Obviously, the number of tokens on out^n during the computation also limits the number of tokens on out^{n+1} at the end. Having ℓ tokens on in^n, we cannot create more than Acker(n, ℓ) tokens on out^n by induction. This in particular means that initially, with only one token on in^n, we cannot create more than Acker(n, 1) tokens on out^n.

Afterwards, if we do not execute t_{swap}^{n+1} as often as possible, say we leave ℓ' out of ℓ tokens on out^n, we end up with Acker($n, \ell - \ell'$) + $\ell' \leqslant$ Acker(n, ℓ) tokens, using Equation (A2). This means that in each iteration, we maximize the number of tokens on out^n by using t_{swap}^{n+1} as often as possible, as in the previously described computation.

Finally, we use that Acker(n, ℓ) $\leqslant$ Acker(n, Acker(n, ℓ)), Equation (A3). We obtain the maximum number of tokens on out^n by conducting the maximum number of iterations consuming all tokens on in^{n+1}.

Altogether, the computation of AN($n+1$) as previously described maximizes the number of tokens on out^{n+1}; no computation can create more than Acker($n+1, m$) tokens. ■

It remains to modify the unlabeled Petri net AN(n) to obtain the Petri net instance $\big(N(n), M_{\text{init}}(n,m), M_{\text{final}}(n)\big)$ with $\mathcal{L}(N(n), M_{\text{init}}(n,m), M_{\text{final}}(n)) = \{a^k \mid k \leqslant \text{Acker}(n,m)\}$.

Proof of Proposition 9.1.4:
Let $n, m \in \mathbb{N}$. Consider AN(n), the net from Definition 9.1.5, as a labeled Petri net with all transitions labeled by ε. We define $N(n)$ to be the Petri net that is obtained from it by adding an a-labeled transition t_a that consumes one token from out^n. The initial marking is the marking $M_{\text{init}}(n,m)$ of AN(n), the final marking is the zero vector.

By Lemma 9.1.6, there is a computation creating Acker(n,m) tokens on out^n, and no computation creates more than Acker(n,m) tokens. Hence, a computation of $N(n)$ can fire t_a

up to Acker(n, m) times, producing the same number of as in the process. This proves $\mathcal{L}(N(n), M_{\text{init}}(n, m), M_{\text{final}}(n)) = \{a^k \mid k \leqslant \text{Acker}(n, m)\}$.

Note that the size of $N(n)$ is polynomial in AN(n), which has a number of places and transitions linear in n. The binary encoding of the initial marking is linear in $n + \lceil \log m \rceil$. ■

With the proof of Proposition 9.1.4 completed, we have provided the lower bound matching the construction from Theorem 9.1.2, showing Theorem 9.1.1.

The unary case

Theorem 9.1.1 also holds if we define the size of Petri nets based on their unary encoding. The upper bound obviously continues to hold. For the lower bound, observe that AN(n) only uses transition multiplicities from the set $\{0, 1\}$, and that the unary encoding of the initial marking is linear in $n + m$.

9.2 Downward closures for BPP nets

The non-primitive recursive complexity of the downward closure in the case of general Petri nets is a dire insight. On the practical side, one should take into account that we are talking about the worst-case complexity. It has been shown in experiments that the worst-case behavior occurs rarely. On many instances that are of practical interest (e.g. for the purpose of verifying systems), the Karp-Miller tree, which can be used to represent the downward closure, can be computed within acceptable time. On the theoretic side, this unexpectedly tame behavior of these instances can partially be explained by the fact that they may stem from restricted subclasses of Petri nets. Hence, we study the problem of computing downward closures for one such subclass. As in Chapter 8, we focus on the class of BPP nets. This means we consider the following problem.

Computing the downward closure for BPP net languages (BPP-DC)	
Given:	A labeled BPP net instance $(N, M_{\text{init}}, M_{\text{final}})$.
Compute:	An NFA A with $\mathcal{L}(A) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$.

In this section, we will prove that this problem can be solved in exponential time. This also implies that the state complexity is at most exponential.

9.2.1 Theorem
Downward closures of BPP net coverability languages have exponential state complexity, and the corresponding automata can be constructed in exponential time. These bounds are tight.

The result is similar to Theorem 8.2.1 in the case of upward closures of BPP nets. However, in the case of the upward closure, the exponential complexity of the problem for BPP nets was only a moderate improvement over the doubly exponential complexity in the general case. Here, the exponential complexity is a vast improvement over the non-primitive recursive complexity in the case of unrestricted Petri nets.

As usual, we prove upper and lower bound separately, starting with the upper bound. We comment on the proof approach for each of the bounds after formally stating the result.

Upper bound

We prove one direction of Theorem 9.2.1: Within exponential time, we can construct an NFA of exponential size accepting the downward closure of the language of a given BPP net instance.

9.2.2 Theorem
Given a labeled BPP net instances $(N, M_{\text{init}}, M_{\text{final}})$, we can compute in exponential time an NFA A with $\mathcal{L}(A) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$. The state complexity of $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$ is at most exponential, as witnessed by automaton A.

To show the theorem, we prove a result that can be seen as some sort of *pumping lemma* for BPP nets: If it is possible to create more than an exponential number of tokens on a place (where the precise number will be given later), then it is possible to create arbitrarily many tokens on that place. To be precise, to get the desired number of tokens one can insert a suitable number of occurrences of a pump, a sequence of transitions, into the computation that creates these tokens. The insertion of these pumps contributes to the word generated by the computation, but the downward closure in which we are interested will also contain a version of the word in which the additional letters are deleted.

Let us formally state this pumping lemma. We assume that the labeled BPP net instance $(N, M_{\text{init}}, M_{\text{final}})$ of size n is fixed. We use $||N||_\infty$ to denote $\max_{p\in P}\{||\text{in}(p)||_\infty, ||\text{out}(p)||_\infty\}$, the maximum multiplicity of any transition.

9.2.3 Lemma
If $M_{\text{init}} \boxed{\sigma} M$ and there is a place p with $M(p) > k$, where

$$k = (||M_{\text{init}}||_1 + 2) \cdot ((|P| + 1) \cdot ||N||_\infty)^{(|T|+2)},$$

then for each $m \in \mathbb{N}$, there is $M_{\text{init}} \boxed{\hat{\sigma}} \hat{M}$ such that (1) $\sigma \preceq \hat{\sigma}$, (2) $M \leqslant \hat{M}$, and (3) $\hat{M}(p) > m$.

We defer the proof of the lemma, and first show that under its assumption, Theorem 8.2.2 is easy to prove. We consider the regular $[0, k]$-ω-overapproximation from Lemma 6.3.3. It provides an automaton $A_{>k}$ that simulates the BPP net, but only tracks the number of tokens up to a bound k. We instantiate it with k as in the above lemma. It is not hard to see that the downward closures of the language of $(N, M_{\text{init}}, M_{\text{final}})$ and the downward close of $\mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$, the language of the automaton $A_{>k}$, coincide.

Proof of Theorem 9.2.2:
Let $k = (||M_{\text{init}}||_1 + 2) \cdot ((|P| + 1) \cdot ||N||_\infty)^{(|T|+2)}$ be as in Lemma 9.2.3. Consider the automaton $A_{>k}$ for the $[0, k]$-ω-approximation and its language $\mathcal{L}(A_{>k}) = \mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$ from Section 6.3. As argued in Lemma 6.3.3, $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) \subseteq \mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$ and hence $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow \subseteq \mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})\downarrow$ hold. For the converse inclusion, consider $w \in \mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})\downarrow$. We consider an accepting run of $A_{>k}$ on a word $v \in \mathcal{L}(A_{>k})$ with $w \preceq v$ that has to exist by the definition of the downward closure. The transition of $A_{>k}$ used in this run induce a firing sequence σ that is formed by the corresponding transitions of net N.

If the accepting run of $A_{>k}$ only uses markings from $\mathbb{N}^P$, i.e. no component is ever ω, this firing sequence σ is valid. This means it induces a covering computation of N, proving $v \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ and thus $w \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$.

Otherwise, the firing sequence decomposes into $\sigma = \sigma'.\sigma''$ such that $M_{\text{init}} \boxed{\sigma'} M'$ is a valid computation of N, and the state M' of $A_{>k}$ reached by the transitions of the automaton corresponding to σ' is the first state in which some place, say place p, is set to ω.

In the following, we demonstrate how to obtain a new sequence of transitions $\hat{\sigma}'.\sigma''$ such that $\sigma' \preceq \hat{\sigma}'$ and all transitions used in the sequence are enabled with respect to place p. More formally, we associate to a sequence of transitions σ a pseudo-computation $M_{\text{init}} = M_0 \boxed{\sigma_1} M_1 \ldots \boxed{\sigma_{|\sigma|}} M_{|\sigma|}$, where the markings are allowed to have negative entries, as in Section 8.1. In all markings M_i occurring in the pseudo-computation associated to $\hat{\sigma}'.\sigma''$, we will have $M_i(p) > 0$, $M_i(p) \geqslant \text{in}(t, p)$ if t is the i^{th} transition, and the last marking $M_{|\sigma|}$ satisfies $M_{|\sigma|}(p) \geqslant M_{\text{final}}(p)$.

Let $m = \sum_{j=1}^{|\sigma''|} \text{in}(\sigma''_j, p) + M_{\text{final}}(p)$. It is an upper bound for the number of tokens required on place p so that all transitions in σ'' are enabled and so that the final marking is covered with respect to place p. Since after firing σ', we had obtained value ω for place p in the run of the automaton, we have $M'(p) > k$. We instantiate Lemma 9.2.3 for m. There is a firing sequence $\hat{\sigma}'$ with $M_{\text{init}} \boxed{\hat{\sigma}'} \hat{M}'$, $\sigma' \preceq \hat{\sigma}'$, $M' \leqslant \hat{M}'$, and $\hat{M}'(p) > m$. The desired modification of σ' is $\hat{\sigma}'$: The number m was chosen so that in the pseudo-computation associated to $\hat{\sigma}'.\sigma''$, all markings are enabled with respect to place p and the final marking is larger than M_{final} for place p.

After repeating this process for each of the places, i.e. at most $|P|$ times, we obtain a pseudo-computation associated to some transition sequence τ in which all markings are non-negative, all transitions are enabled whenever they are fired, and the final marking covers M_{final}. Hence, $M_{\text{init}} \boxed{\tau} M \geqslant M_{\text{final}}$ is a covering computation of N, proving that $\lambda(\tau) \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$, and thus $w \preceq \lambda(\sigma'.\sigma'') \preceq \lambda(\tau)$ is in the downward-closure $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$.

We have established that the downward closures of $\mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$ and $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ coincide. To obtain a representation of $\mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})\downarrow$, we take the automaton $A_{>0}$ and add for each transition an ε-labeled variant as described in Section 7.2. This does not change the number of states, so to finish the proof, we argue that the state complexity of $A_{>k}$ is as desired. It is as most

$$2^n \cdot k^n \in \mathcal{O}\left(2^n \cdot (||M_{\text{init}}||_1 + 2)^n \cdot ((|P| + 1) \cdot ||N||_\infty)^{n^2}\right)$$

using Lemma 6.3.3 and the definition of k. By the power laws, this number is singly exponential (even if $||M_{\text{init}}||_1$ and $||N||_\infty$ are exponential). ■

It remains to show the lemma on which the proof of the theorem relies. For the proof, we will again need the unfolding of a BPP net and its properties, see Section 8.2. This in particular

means that we have to eliminate spontaneous transitions from the net. To this end, we assume that have applied the preprocessing described in Section 8.2. In the following, we will consider the bound $k = ||M_{\text{init}}||_1 \cdot (|P| \cdot ||N||_\infty)^{(|T|+1)}$. This bound is subtly different from the bound $(||M_{\text{init}}||_1 + 2) \cdot ((|P|+1) \cdot ||N||_\infty)^{(|T|+2)}$ stated in Lemma 9.2.3. The modifications take into account that the preprocessing adds a new place, a new transition, and a new token in the initial marking. We also assume that the original net satisfies $||N||_\infty \geqslant 2$, which means that the preprocessing does not increase $||N||_\infty$.

Proof of Lemma 9.2.3:
Let (O, h) be the unfolding $\text{Unf}(N, M_{\text{init}})$ of the given net with respect to the initial marking, as defined in Section 8.2. Assume that $M_{\text{init}} \boxed{\sigma} M$ is a computation of N reaching a marking M such that $M(p) > k = ||M_{\text{init}}||_1 \cdot (|P| \cdot ||N||_\infty)^{(|T|+1)}$ for some place p.

We consider a computation $\min O \boxed{\sigma'} M'$ of O induced by σ, i.e. we have $h(\sigma') = \sigma$. We also have $h(M') = M$, meaning that for each place p_1 of N, the total number of tokens in places p_1' with $h(p_1') = p_1$ is equal to $M(p)$. In particular, we have

$$\sum_{\substack{p' \in P', \\ h(p')=p}} M'(p') = M(p) > k$$

for the place p fixed above.

Recall that O can be seen as a forest, each of its tree rooted in a place that carries a token in $\min O$. Considering only the part of O that is used by the computation associated to σ', i.e. the transitions that are used in σ' and the places that carry a token at some point during the computation, yields a finite prefix of O that is again a forest. The places that carry a token in the final marking M' form the leaves of the trees in the forest. We will call the places p' with $M'(p') = 1$ and $h(p') = p$ the *p-leaves* of the forest. The name is justified by the above discussion.

We identify in the forest associated to σ' the tree with the largest number of p leaves. Let us denote this tree by $\mathcal{T}$ and its root node by r'. The number of p-leaves in $\mathcal{T}$ is at least

$$k_1 = \frac{k}{||M_{\text{init}}||_1} = (|P| \cdot ||N||_\infty)^{(|T|+1)}.$$

The forest consists of exactly $||M_{\text{init}}||_1$ trees, since $h(\min O) = M_{\text{init}}$ and each token in the initial marking uniquely identifies the root of a tree. If every tree had less than $\frac{k}{||M_{\text{init}}||_1}$ many p-leaves, then it could not be true that the forest has at least k many p-leaves in total.

In the following, we will show that the minimal subtree of $\mathcal{T}$ that contains the root and reaches all its p-leaves contains a *pump*, i.e. a substructure that can be replicated to obtain arbitrarily many tokens in place p. The idea is similar to the one behind the proof of the pumping lemma for context-free languages executed on parse trees. However, the structure of the unfolding causes some technical difficulties that we have to overcome.

Let us denote by $\mathcal{T}'$ the subtree of $\mathcal{T}$ that consists of the p-leaves and their ancestors. Its leaves are exactly the p-leaves, and every other place has out-degree exactly one. That the out-degree cannot be larger than one is clear by the definition of O. A place with no outgoing edge has to be a p-leaf, as we have removed all nodes that are not ancestors of p-leaves. The out-degree of the transitions occurring in $\mathcal{T}'$ is at most $||N||_\infty|P|$, since each transition of N can create at most $||N||_\infty$ tokens in each of the $|P|$ places.

Similar to the proof of Proposition 8.2.3, we will call a transition of out-degree at least two a *join transition*, as it leads to at least two different p-leaves. The latter property is guaranteed by the acyclicity of O. The definition of join transitions is simpler here (compared to the proof of Proposition 8.2.3), because we only consider the part of the computation that leads to p-leaves.

We claim $\mathcal{T}'$ has a branch with at least $|T| + 1$ join transitions on it. To this end, we state and prove the following claim.

Claim: Let T be a tree with ℓ leaves in which all nodes have out-degree at most c. Then T has a branch with at least $\log_c \ell$ nodes of out-degree at least 2.

To prove the claim, assume that in any branch of such a tree T, the number of nodes of out-degree greater than two in any branch of the tree is $m < \log_c \ell$. Such a tree has at most $c^m < c^{\log_c \ell} = \ell$ nodes, which is a contradiction to the assumption that T has ℓ leaves.

We now use the claim for the tree $\mathcal{T}'$, $\ell = k_1$, $c = ||N||_\infty \cdot |P|$, and obtain that $\mathcal{T}'$ has at least

$$k_2 = \log_{||N||_\infty \cdot |P|} k_1 = |T| + 1$$

many join transitions. This means that $\mathcal{T}'$ has a branch with transitions t', t'' of O such that $h(t') = h(t'') = t$ for some transition t of N, since N has only $|T|$ different transitions.

Since t' is a join transition, t' is contained in (at least) two branches: The first is a branch that also contains t'' and ends in a p-leaf; let $\sigma'_a.t'.\sigma'_b.t''.\sigma'_c$ be the transitions along that branch. The other branch does not contain t'' and ends in a different p-leaf; let $\sigma'_a.t'.\sigma'_d$ be the associated transitions. Note that $\sigma'_a, \sigma'_b, \sigma'_c, \sigma'_d \in T'^*$ denote sequences of transitions here.

We now argue that we can first use $t'.\sigma'_b$ to create an arbitrary number of tokens on the places from which transition t of N consumes a token, and later use $t'.\sigma'_d$ to create an arbitrary number of tokens on place p as desired. More formally, we have that $\sigma'_a.t'.\sigma'_b.t''.\sigma'_c$ is a subword of the computation σ' of O, and hence that $h(\sigma'_a.t'.\sigma'_b.t''.\sigma'_c)$ is a subword of σ. Hence, we may write $\sigma = \sigma_A.t.\sigma_B$, where t corresponds to the occurrence of t' in $\sigma'_a.t'.\sigma'_b.t''.\sigma'_c$.

Let $m \in \mathbb{N}$ be an arbitrary number. We have to construct $\hat{\sigma}$ such that $M_{\text{init}} \xrightarrow{\hat{\sigma}} \hat{M}$ satisfies (1) $\sigma \preceq \hat{\sigma}$, (2) $M \leqslant \hat{M}$, and (3) $\hat{M}(p) > m$. We define $\text{pump}_1 = h(\sigma'_b.t'')$, and $\text{pump}_2 = h(\sigma'_d)$. We claim that $\hat{\sigma} = \sigma_A.t.\text{pump}_1{}^m.\sigma_B.\text{pump}_2{}^m$ is as required. Firstly, note that indeed transition t produces the token required by the first transition of pump_1 and the token required by σ_B.

Secondly, the last transition of pump_1 is again $h(t'') = t$. Altogether, $\sigma_A.t.\text{pump}_1{}^m.\sigma_B$ is a valid firing sequence. Furthermore, the m iterations of pump_1 produce at least m tokens on the place from which pump_2 consumes a token. Hence, $M_{\text{init}} \boxed{\hat{\sigma}} \hat{M}$ is a valid firing sequence. The properties (1) – (3) are easy to justify: (1) $\sigma = \sigma_A.t.\sigma_B$ is a subword of $\hat{\sigma}$ by definition, (2) the pumps only create additional tokens, and (3) in particular, each iteration of pump_2 creates a token on place p, since $\sigma'_a.t'.\sigma'_d$ corresponds to a branch in O leading to a p-leaf. ■

With the proof of Lemma 9.2.3 completed, the upper bound, Theorem 9.2.2 is proven.

Remark

In the case of the upward closure, the proofs of the upper bounds were based on the length-k approximation, both in the general case and in the case of BPP nets. In the case of the downward closure, the constructions seem to be fundamentally different: In the general case, the construction was based on the Karp-Miller tree, while in the case of BPP nets, we have used the $[0, k]$-ω-approximation. We claim that these differences can be overcome: The proof concept from this section can be generalized so that it also applies to the case of general nets. We claim that to obtain the downward closure of a Petri net, we can take the downward closure of the $[0, k]$-ω-approximation, where k is the greatest non-ω number occurring in a general marking labeling a node in the Karp-Miller tree. Proving this fact is not difficult, but relies on some properties of the Karp-Miller tree that we have not stated here. Hence, we forgo giving the proof.

Lower Bound

To prove that the above construction is optimal, we present a family of BPP languages for which the state complexity of the downward closure is exponential in the size of the nets. The proof is similar to the proof of Proposition 8.2.4 and uses that exponential numbers can be stored in polynomial space using their binary encoding. The difference is that this time, we use an exponential number in the initial marking instead of in the final marking.

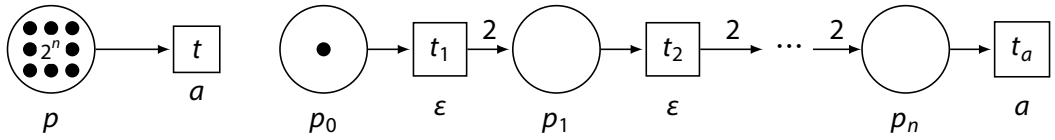
9.2.4 Proposition

For each $n \in \mathbb{N}$, there is a labeled BPP net instance $(N, M_{\text{init}}(n), M_{\text{final}})$ of size polynomial in n such that $\mathcal{L}(N, M_{\text{init}}(n), M_{\text{final}})\!\downarrow$ has state complexity at least 2^n.

Proof:

We define N to be the net with a single place p and a single a-labeled transition t that consumes a token from this place. The final marking M_{final} is the zero vector, the initial marking M_{init} places 2^n tokens on p. This net is depicted in Figure 9.2.a.i).

Any computation of the net is covering, and each sequence t^k for $k \leqslant M_{\text{init}}(p) = 2^n$ is a valid firing sequence. Hence, $\mathcal{L}(N, M_{\text{init}}(n), M_{\text{final}}) = \{a^k \mid k \leqslant 2^n\}$. This language is already downward



i) In the binary case. ii) In the unary case.

Figure 9.2.a: BPP nets to prove the exponential lower bound for the downward closure.

closed and has state-complexity $2^n + 1$, see Example 4.3.1. To conclude the proof, note that the size of N and M_{init} is constant, while the size of M_{final} is linear in n. ■

The proof of the proposition also completes the proof of Theorem 9.2.1.

The unary case

We complete our study by considering the case in which the BPP net is encoded in unary. Consider the upper bound. The size of the automaton that we construct is based on the number $k = (\|M_{\text{init}}\|_1 + 2) \cdot ((|P| + 1) \cdot \|N\|_\infty)^{(|T|+2)}$ from Lemma 9.2.3. This number is exponential, even if we assume that the net is encoded in unary. Hence, we do not get an improved upper bound.

The proof of the lower bound, Proposition 9.2.4, obviously becomes invalid if we consider nets to be encoded in unary. With a unary encoding, it is not possible to assign 2^n tokens to a place in the initial marking of an instance whose size is polynomial in n. However, it turns out that we can construct an instance $(N, M_{\text{init}}, \vec{0})$ of size polynomial in n whose language coincides with the language of the net from Proposition 9.2.4.

The construction is more involved than the one from the proof of Proposition 9.2.4. It uses the well-known fact that a context-free grammar of size polynomial in n can generate the word a^{2^n}. Here, we adapt this construction to BPP nets.

The net that we construct is depicted schematically in Figure 9.2.a.ii). It has $n + 1$ places $p_0, p_1, \ldots, p_n$. The initial marking assigns one token to place p_0 and zero tokens elsewhere. For each $i \in [1, n]$, there is a transition t_i that consumes a token from place p_{i-1} and produces two tokens on p_i. All these transitions are labeled by ε. Additionally, there is an a-labeled transition t_a that consumes a token from p_n. The final marking is the zero vector. Note that this net is indeed a BPP net, every transition consumes exactly one token.

Let us prove that the language of the instance is indeed $\{a^k \mid k \leqslant 2^n\}$. By firing the transitions t_i exhaustively, one can generate 2^n tokens on place 2^n: Assuming that 2^i tokens are present on place p_i, one can fire transition t_{i+1} exactly 2^i times to generate $2 \cdot 2^i = 2^{i+1}$ tokens on p_i. Now, we may fire transition t_a up to 2^n times to generate an arbitrary number of as between zero and 2^n.

To conclude the proof, it remains to observe that the size of the instance is indeed polynomial in n. Hence, we have re-proven the lower bound, Proposition 9.2.4. We obtain a matching exponential lower and upper bound in the case of BPP nets encoded in unary.

9.3 SRE inclusion in the downward closures for Petri nets

Similar to Section 8.3, we study the problem of checking whether a given SRE is contained in the downward closure of a given Petri net coverability language.

SRE inclusion in the downward closure (PN-SREDC)	
Given:	A labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$, an SRE sre.
Question:	$\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$?

The goal of this section is to show that the problem is complete for EXPSPACE, the class of problems solvable with exponential space. This matches the EXPSPACE-completeness of PN-SREUC, and is a huge improvement over the non-primitive recursive complexity of computing the downward closure.

9.3.1 Theorem
PN-SREDC is EXPSPACE-complete.

We first briefly comment on the lower bound, before we turn to developing some techniques needed for proving EXPSPACE-membership.

Lower bound

Similar to the EXPSPACE-hardness of the problem PN-SREUC, Lemma 8.3.4, the hardness of PN-SREDC directly follows from the hardness of coverability. We briefly recall the idea of the proof, but refer to Lemma 8.3.4 for the technical details.

If we modify an unlabeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$ by labeling all transitions by ε, we have $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = \{\varepsilon\}$ if M_{final} is coverable from M_{init}. If this is not the case, we have $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = \varnothing$. Both these languages are already downward closed. Hence, we have that $\varnothing^* = \{\varepsilon\} \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ holds if and only if $(N, M_{\text{init}}, M_{\text{final}})$ is a yes-instance of coverability.

Upper bound

It remains to prove EXPSPACE membership.

9.3.2 Theorem
PN-SREDC can be solved using exponential space.

Let $(N, M_{\text{init}}, M_{\text{final}})$ be the Petri net instance of interest, and let $sre = p_1 \cup \ldots \cup p_k$ be the given SRE. Our goal is to check whether the inclusion $\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$ holds.

In Section 8.3, we did proceed as follows to solve the problem in the case of the upward closure: Firstly, we observed that the inclusion holds if and only if $\mathcal{L}(p_i) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ holds for each product p_i. Secondly, we specified an operation that computes for each product p_i the unique minimal word of $\mathcal{L}(p_i)\uparrow$. Checking the inclusion then amounts to checking whether this minimal word is contained in the downward closure of the net. Altogether, we reduced the problem PN-SREUC to a polynomial number of membership queries.

The first observation still holds true in the case of PN-SREDC: $\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$ holds if and only if $\mathcal{L}(p_i) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$ for all products p_i. Since the number of products is polynomial in the size of the SRE *sre*, this will allow us to focus on the simpler problem of checking whether the language of a single product p is contained in the downward closure.

Unfortunately, the second observation does not carry over. Unlike upward-closed languages, downward-closed languages cannot be represented by a single word in general. Consider for example the product Σ^* (whose language is already downward closed). A priori, there is no reason why there should be a single word such that inclusion of the product and the membership problem for that word are equivalent. This means that checking whether a product is contained in the downward closure is much more involved.

Nevertheless, we can provide a syntactic transformation of the product that will later simplify solving the problem. Intuitively, our goal is to replace each occurrence of the iteration of a subalphabet by the iteration of a single word. To this end, we fix some total order on the alphabet, e.g. $a_1 < a_2 < \ldots < a_m$, where $\Sigma = \{a_1, \ldots, a_m\}$. We obtain a word $w_\Sigma = a_1 a_2 \ldots a_m$ by concatenating the letters in ascending order. To each subalphabet $\Gamma \subseteq \Sigma$, we assign the word $w_\Gamma = \pi_\Gamma(w)$ that is obtained from w_Σ by removing all letters not present in Σ.

We can now define a linearization operation that turns a product into a regular expression.

9.3.3 Definition
The *linearization* of a product is inductively defined as follows.

$$\text{lin}(a \cup \varepsilon) = a \qquad \text{lin}(a) = a$$
$$\text{lin}(\Gamma^*) = w_\Gamma{}^* \qquad \text{lin}(p_1.p_2) = \text{lin}(p_1).\text{lin}(p_2)\,.$$

9.3.4 Example
For example, if $\Sigma = \{a, b, c\}$ and we take $w_\Sigma = abc$, then $p = (a \cup c)^*(a \cup \varepsilon)(b \cup c)^*$ is turned into $\text{lin}(p) = (ac)^* a (bc)^*$.

The following lemma states that the linearization indeed allows us to simplify the problem: The language of a product is contained in the downward closure if and only if the language of its linearization is.

9.3.5 Lemma
For a product p, we have $\mathcal{L}(p) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$ if and only if $\mathcal{L}(\text{lin}(p)) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$.

Proof:
We claim that $\mathcal{L}(p)\downarrow = \mathcal{L}(\text{lin}(p))\downarrow$, which implies the desired statement. We proceed by induction on the structure of the product. We have $\mathcal{L}(a \cup \varepsilon)\downarrow = \{a, \varepsilon\} = \mathcal{L}(a)\downarrow = \mathcal{L}(\text{lin}(a))\downarrow = \mathcal{L}(\text{lin}(a \cup \varepsilon))\downarrow$, which proves two of the base cases.

Consider Γ^*. We have $\mathcal{L}(\Gamma^*)\downarrow = \Gamma^*$. It remains to show that $\mathcal{L}({w_\Gamma}^*)\downarrow = \Gamma^*$. One direction, $\mathcal{L}({w_\Gamma}^*)\downarrow \subseteq \Gamma^*$ is clear since w_Γ only contains letters from Γ. For the other direction, consider $v \in \Gamma^*$. We claim that v is a subword of $(w_\Gamma)^{|v|}$: Each letter of v is from Γ, so it occurs in w_Γ. Dropping all other letters in each iteration of $(w_\Gamma)^{|v|}$ proves the statement. Hence, $v \in \mathcal{L}({w_\Gamma}^*)\downarrow$ as desired.

Finally, for the induction step, assume that $\mathcal{L}(p_i)\downarrow = \mathcal{L}(\text{lin}(p_i))\downarrow$ for $i \in \{1, 2\}$. We have

$$\begin{aligned}\mathcal{L}(p_1.p_2)\downarrow &= (\mathcal{L}(p_1).\,\mathcal{L}(p_2))\downarrow = \mathcal{L}(p_1)\downarrow\,.\,\mathcal{L}(p_2)\downarrow \\ &= \mathcal{L}(\text{lin}(p_1))\downarrow\,.\mathcal{L}(\text{lin}(p_2))\downarrow = (\mathcal{L}(\text{lin}(p_1)).\mathcal{L}(\text{lin}(p_2)))\downarrow \\ &= \mathcal{L}(\text{lin}(p_1).\text{lin}(p_2))\downarrow = \mathcal{L}(\text{lin}(p_1.p_2))\downarrow\ .\end{aligned}$$

Here, we have used induction, the fact that the downward closure distributes over concatenation, and the definition of the linearization. ■

With the lemma at hand, the remaining task is the following: For each infix ${w_\Gamma}^*$ occurring in the linearization of the product, we need to check whether arbitrarily long instantiations lead to a word contained in the downward closure. Intuitively, this means that we need to count the number of instantiations of each ${w_\Gamma}^*$. Formally, we reduce checking the inclusion to an unboundedness problem.

The specific type of unboundedness that we require is *simultaneous unboundedness*, formally defined in the following.

Simultaneous unboundedness for Petri nets (PNSU)	
Given:	Petri net N, initial marking M_{init}, set of places X of N.
Question:	Are the places in X *simultaneously unbounded*, i.e. is there for each $m \in \mathbb{N}$ a computation $M_{\text{init}} \boxed{\sigma}\!\!> M$ with $M(p) \geqslant m$ for all $p \in X$?

Simultaneous unboundedness has been introduced by Demri [Dem13] as a generalization of several classical notions of unboundedness, i.e. the questions whether a given net Petri is unbounded in some arbitrary resp. a specified component. Indeed, we need this simultaneous unboundedness to encode our problem: It is not sufficient to separately check that for each infix ${w_\Gamma}^*$, arbitrarily long instantiations lead to a word in the downward closure. Rather, we need

that even if we combine arbitrarily long instantiations of each of the infixes, we still obtain a word in the downward closure.

All aforementioned unboundedness problems could be solved using the Karp-Miller tree. For example, the places in X are simultaneously unbounded if the Karp-Miller tree contains a node labeled by a generalized marking in which all places $p \in X$ attain value ω. However, this procedure would inherit the non-primitive recursive complexity of constructing the Karp-Miller tree.

Luckily, Demri [Dem13] has shown that the complexity of PNSU is in fact EXPSPACE.

9.3.6 Theorem (Demri [Dem13])
PNSU is EXPSPACE-complete.

The proof of this result is an extension of Rackoff's technique [Rac78]. Demri shows that if a set of places is simultaneously unbounded, then there are specific computations that witness this fact. With a Rackoff-like argumentation, one obtains a doubly exponential bound on the length of these computations. This finally allows for checking all such computations using only exponential space by using counters encoded in binary and Savitch's theorem [Sav70].

With the EXPSPACE membership of PNSU at hand, it remains to reduce the inclusion $\mathcal{L}(\mathrm{lin}(p)) \subseteq \mathcal{L}(N, M_{\mathrm{init}}, M_{\mathrm{final}})\downarrow$ to an instance of PNSU in polynomial time. The first step is to consider the net $N\downarrow$ with $\mathcal{L}(N, M_{\mathrm{init}}, M_{\mathrm{final}})\downarrow = \mathcal{L}(N\downarrow, M_{\mathrm{init}}, M_{\mathrm{final}})$, see Section 7.2.

As explained above, we need to check whether for each infix ${w_\Gamma}^*$, arbitrarily long instantiations still lead to words in the language of $N\downarrow$ We encode this by introducing for each infix ${w_\Gamma}^*$ a place that tracks that w_Γ has been seen. The instance of the unboundedness problem that we construct requires all such places to be simultaneously unbounded. We will make this formal in the following. Afterwards, the minor problem of also encoding the final marking of the net into the unboundedness problem remains to be solved.

To make the construction formal, we see $\mathrm{lin}(p)$ as a Petri net. To be precise, we construct an NFA that is language-equivalent to $\mathrm{lin}(p)$, e.g. using the construction by McNaughton and Yamada [MY60]. Then, we see this NFA as a Petri net. We have given a construction in Example 6.3.1, but here, it will be important that we obtain a net with no ε-labeled transitions. To this end, we assume that the NFA has a unique final state, a condition that is easy to enforce, especially for the NFAs with language $\mathrm{lin}(p)$. Then, we construct a Petri net instance $(N_p, M_{\mathrm{init},p}, M_{\mathrm{final},p})$. Net N_p has one place per state of the NFA and one transition per transition of the NFA. The labeling and effect of the transitions is as expected. The initial marking $M_{\mathrm{init},p}$ puts one token on the place corresponding to the initial state of the NFA and no tokens elsewhere. The final marking $M_{\mathrm{final},p}$ just requires one token on the place corresponding to the unique final state of the NFA. We obtain $\mathcal{L}(\mathrm{lin}(p)) = \mathcal{L}(N_p, M_{\mathrm{init},p}, M_{\mathrm{final},p})$ as desired.

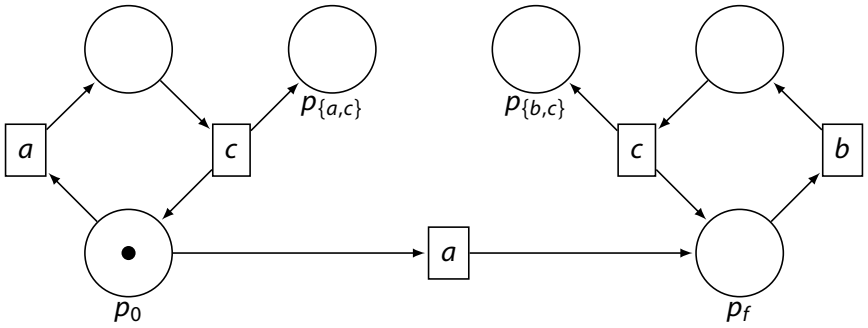


Figure 9.3.a: A net N_p with language $(ac)^*a(bc)^*$ that also counts the number of iterations.

It remains to modify the net so that for each infix ${w_\Gamma}^*$ of $\mathrm{lin}(p)$, it counts how many iterations of ${w_\Gamma}^*$ have occurred. For such an infix, net N_p has a transition that correspond to the last letter of w_Γ. We add a new place p_Γ and modify the transition so that it additionally creates a token on p_Γ. The desired result is that if a computation produces m tokens on p_Γ, then ${w_\Gamma}^m$ was the instantiations of ${w_\Gamma}^*$ that has been read.

Note that the notation is intentionally a bit sloppy here: The linearization may contain several occurrences of ${w_\Gamma}^*$ for the same subalphabet Γ. In this case, each occurrence gets its own place p_Γ. To improve readability, we forgo introducing additional indices

9.3.7 Example

Consider $\mathrm{lin}((a \cup c)^*(a \cup \varepsilon)(b \cup c)^*) = (ac)^*a(bc)^*$ from Example 9.3.4. The net N_p for this product is depicted in Figure 9.3.a, including the two places $p_{\{a,c\}}$ and $p_{\{b,c\}}$ that count the iterations. To improve readability, some names of places and transitions have been omitted.

We now construct the synchronized product of $N{\downarrow}$ and N_p, which we will denote by $(N', M'_{\text{init}}, M'_{\text{final}})$. Recall that its set of places is the disjoint union of the places of $N{\downarrow}$ and N_p. Transitions labeled by ε can be fired freely, while transitions labeled by $a \in \Sigma$ are synchronized in that we can only fire an a-labeled transition of $N{\downarrow}$ and an a-labeled transition of N_p together. For the correctness of the construction, it will be crucial that the synchronized product acts as a one-sided synchronized product in the special case at hand: (1) N_p has no ε-labeled transitions, so all transitions of N_p have to be synchronized with an appropriate transition of $N{\downarrow}$, while (2) $N{\downarrow}$ has an ε-labeled version of each transition, so each transition can be fired freely.

We would like to have that the desired inclusion holds if and only if the set of places $\{p_\Gamma \mid {w_\Gamma}^*$ infix of $\mathrm{lin}(p)\}$ is simultaneously unbounded in N' from M'_{init} However, we still have to take the final marking into account. To achieve this, we add a new place to N' that can only become unbounded as soon as the final marking has been covered. Formally, we add a new place p_{final} to N'. We add a transition t_{final} with $\mathrm{in}(t_{\text{final}}) = M'_{\text{final}}$ that consumes tokens from all places as specified by the final marking and that produces one token on p_{final} (and no token elsewhere). Finally, we add a transition t_{pump} that consumes one token on p_{final}, but produces

two tokens on p_{final}. Let N'' denote the resulting net, and let M''_{init} be the initial marking for N'' that coincides with M'_{init} on the places of N' and assigns no token to p_{final}. The problem PNSU works on unlabeled nets, but for the proof of correctness later, it will be helpful to consider both t_{final} and t_{pump} to be labeled by ε.

As desired, we obtain that the inclusion holds if and only if the set of places $\{p_\Gamma \mid w_\Gamma{}^* \text{ infix of } \text{lin}(p)\} \cup \{p_{\text{final}}\}$ is simultaneously unbounded in N''. To formally prove the statement, let us assume that $\text{lin}(p) = w_{(0)}w_{\Gamma_1}{}^*w_{(1)}\dots w_{k-1}w_{\Gamma_k}{}^*w_{(k)}$ for some words $w_{(i)} \in \Sigma^*$ and subalphabets $\Gamma_i \subseteq \Sigma$ for all i.

9.3.8 Proposition
The inclusion $\mathcal{L}(\text{lin}(p)) \subseteq \mathcal{L}(N{\downarrow}, M_{\text{init}}, M_{\text{final}})$ holds iff the places in $X = \{p_{\text{final}}\} \cup \{p_{\Gamma_i} \mid i \in [1,k]\}$ are simultaneously unbounded in N'' from M''_{init}.

Proof:
Assume that the inclusion holds, and let $m \in \mathbb{N}$ be arbitrary. We have to show that there is a marking M with $M_{\text{init}} \xrightarrow{\sigma} M$ in N'' and $M(p_{\Gamma_i}) \geqslant m$ for all i and $M(p_{\text{final}}) \geqslant m$. Consider the word $w = w_{(0)}w_{\Gamma_1}{}^m w_{(1)}\dots w_{k-1}w_{\Gamma_k}{}^m w_{(k)}$ obtained by taking m iterations of each w_Γ. We have $w \in \mathcal{L}(\text{lin}(p))$ by construction, and since we assume that the inclusion holds also $w \in \mathcal{L}(N{\downarrow}, M_{\text{init}}, M_{\text{final}})$. Recall that the language of the synchronized product is the intersection of the languages of the original nets. Hence, we have $w \in \mathcal{L}(N'', M''_{\text{init}}, M''_{\text{final}})$, where M''_{final} coincides with M'_{final} on the places of N' and requires no token on p_{final}. Let us consider a corresponding computation $M''_{\text{init}} \xrightarrow{\sigma} M \geqslant M''_{\text{final}}$ of N'' with $\lambda(\sigma) = w$. Since the non-ε-labeled transitions in N' can only be fired in a synchronized fashion, for each infix $w_{\Gamma_i}{}^m$ the transition that corresponds to the last letter of w_{Γ_i} is fired m times. Hence, we have m tokens on each place p_{Γ_i} that will not be consumed by other transitions, and for each i, $M(p_{\Gamma_i}) \geqslant m$ as desired. To also obtain m tokens on p_{final}, we extend the run and consider the firing sequence $\sigma.t_{\text{final}}.t_{\text{pump}}{}^m$. Note that t_{final} is indeed enabled in M since $M \geqslant M''_{\text{final}}$, and that firing $t_{\text{pump}}{}^m$ creates m tokens on p_{final}. The marking reached by firing this sequence has the desired properties.

For the other direction, assume that the places in X are simultaneously unbounded. Consider a word $w \in \mathcal{L}(\text{lin}(p))$, say $w = w_{(0)}w_{\Gamma_1}{}^{m_1}w_{(1)}\dots w_{\Gamma_k}{}^{m_k}w_{(k)}$. To prove that $w \in \mathcal{L}(N{\downarrow}, M_{\text{init}}, M_{\text{final}})$, we first show that w is in the language of the product net N'.

To this end, we use the assumption that the places in X are simultaneously unbounded. We define $m = \max\{1, \max_{i\in[1,k]} m_i\}$, and obtain a computation $M_{\text{init}} \xrightarrow{\sigma} M$ of N'' such that $M(p) \geqslant m$ for all $p \in X$. We may assume wlog. that σ is of the shape $\sigma = \sigma'.t_{\text{final}}.t_{\text{pump}}{}^{m_{\text{pump}}}$, where σ' does not contain t_{final} and t_{pump}. Indeed, any sequence that is not of this shape can be reordered while preserving its validity (since executing t_{final} later yields greater intermediary markings) and the word that is generated (since t_{final} and t_{pump} are labeled by ε). Note that σ needs to contain occurrences of t_{final} and t_{pump} because we had $M(p_{\text{final}}) \geqslant m$. If t_{final} occurs multiple times in σ, we simply drop all occurrences but one.

Consider the computation $M_{\text{init}} \boxed{\sigma'} M'$, which we may see as a computation of the product net N'. It produces the same word, and we have $M' \geqslant M'_{\text{final}}$ since t_{final} was enabled in M'. Hence,

$$\lambda(\sigma) = \lambda(\sigma') \in \mathcal{L}\big(N', M_{\text{init}}, M'_{\text{final}}\big) = \mathcal{L}(\text{lin}(p)) \cap \mathcal{L}(N{\downarrow}, M_{\text{init}}, M_{\text{final}}) \,,$$

using that the language of the synchronized product is the intersection of the languages.

Since we have $M'(p_{\Gamma_i}) \geqslant m$ for all i, each infix w_{Γ_i} occurs at least m times in $\lambda(\sigma)$. We have that $\lambda(\sigma) = w_{(0)} {w_{\Gamma_1}}^{m'_1} w_{(1)} \dots {w_{\Gamma_k}}^{m'_k} w_{(k)}$, where $m'_i \geqslant m \geqslant m_i$ for each i. Hence, the word $w \in \mathcal{L}(\text{lin}(p))$ that we started with is a subword of $\lambda(\sigma)$. We obtain that $\lambda(\sigma) \in \mathcal{L}(N{\downarrow}, M_{\text{init}}, M_{\text{final}})$ implies $w \in \mathcal{L}(N{\downarrow}, M_{\text{init}}, M_{\text{final}})$, which we wanted to show. ■

With the proposition at hand, it only remains to combine all ingredients.

Proof of Theorem 9.3.2:
The inclusion $\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}}){\downarrow}$ holds if and only if $\mathcal{L}(p) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}}){\downarrow}$ holds for each product p of the SRE *sre*. The number of products is polynomial in the size of the SRE.

For each product p, it is easy to construct $\text{lin}(p)$ in polynomial time. In general, an automaton representing a regular expression can be substantially larger than the expression [GH15]. For the expression $\text{lin}(p)$ which has nesting-depth at most 2, however, it is easy to construct first an automaton and then a Petri net instance $(N_p, M_{\text{init},p}, M_{\text{final},p})$. This net is of size polynomial in the size of p, and it has the same language as $\text{lin}(p)$. By definition, the net $N{\downarrow}$ is of size polynomial in the size of N. Combining the two facts yields that the size of the product net N' and also the size of the net N'' are polynomial in the input size.

Altogether, we obtain that to decide the inclusion, we need to invoke a polynomial number of queries for instances of the problem PNSU of polynomial size. We use that PNSU can be solved in EXPSPACE, Theorem 9.3.6, and that the calling polynomially many exponential space algorithms still results in an exponential space algorithm to deduce that the overall procedure runs using exponential space.

The correctness results directly from Lemma 9.3.5 and Proposition 9.3.8. ■

With the proof of Theorem 9.3.2 completed and the matching lower bound, we have shown that PN-SREDC is EXPSPACE-complete as stated in Theorem 9.3.1. Note that since the lower bound is based on the lower bound for coverability, the result also holds if we consider Petri nets to be encoded in unary.

9.4 SRE inclusion in the downward closures for BPP nets

It should come as no surprise that the final section of this chapter combines the two restricted versions of the downward closure computation into a single problem. We study the complexity of checking whether the language of a given SRE is contained in the downward closure of the language of a given BPP net.

SRE inclusion in the downward closure for BPP nets (BPP-SREDC)	
Given:	A labeled BPP net instance $(N, M_{\text{init}}, M_{\text{final}})$, an SRE sre.
Question:	$\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$?

We prove that the complexity of the problem is the same as the problem BPP-SREUC in the case of the downward closure: BPP-SREDC is complete for the class of problems solvable in polynomial time by a nondeterministic algorithm.

9.4.1 Theorem
BPP-SREDC is NP-complete.

While the statement is similar, the proof is vastly more complex than the proof of Theorem 8.4.1. The same reasoning as in Section 9.3 applies: A downward-closed language cannot be represented by finitely many maximal words. Hence, to solve the problem, it is not sufficient to conduct a finite number of membership queries. Before we go into detail about the proof approach, let us briefly comment on the lower bound.

Lower bound

The NP-hardness of BPP-SREDC is a direct consequence of the NP-hardness of coverability in BPP nets, Lemma 6.4.4. The formal reasoning is similar to the one for the EXPSPACE-hardness of PN-SREUC (see Lemma 8.3.4) and PN-SREDC, and the NP-hardness of BPP-SREUC.

Upper bound

With the lower bound out of the way, it remains to show that BPP-SREDC can be solved by a nondeterministic algorithm in polynomial time.

9.4.2 Theorem
BPP-SREDC is in NP.

The proof of this theorem is the most complex proof in this part of the thesis. It combines various insights that we have gathered in the previous chapters.

Let us start by considering the proof of the EXPSPACE-membership of PN-SREDC, Theorem 9.3.2, and understand which parts of it can be reused. The proof proceeds in four steps: (1) Observe that instead of checking $\mathcal{L}(sre) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$, we can check $\mathcal{L}(p) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$ for each of the polynomially many products of the SRE. (2) Instead of checking this inclusion, we can check $\mathcal{L}(\text{lin}(p)) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$ where $\text{lin}(p)$ is the linearization of the product from Definition 9.3.3. (3) Construct the product of $N\downarrow$, the downward closure of the given net, and N_p, a net with language $\mathcal{L}(\text{lin}(p))$. (4) Construct an instance of the simultaneous unboundedness problem PNSU based on this product.

The first two steps carry over. The fact that the inclusion can be checked for each product as well as Lemma 9.3.5 remain unchanged if we consider an input net that is a BPP net. Unfortunately, there is a problem with the third step. While $N\downarrow$ and N_p both are BPP nets, their product will not be a BPP net. Even if it would be a BPP net, there is also a problem with the reduction to PNSU. The problem PNSU is EXPSPACE-complete, so we would not get the desired complexity.

The question if PNSU becomes NP-complete if we restrict the input net to be a BPP net arises naturally. One could approach this problem by first using Theorem 6.4.7 which yields that the reachability set of a BPP net is effectively semi-linear, and then trying to encode simultaneous unboundedness as a Presburger formula. While the latter seems to be easy, one will obtain a formula that contains a universal quantification, so we cannot use Theorem 3.4.2, the NP-completeness of existential Presburger arithmetic. We conjecture that the proof that we will present in the rest of this section could be adapted to show that PNSU is NP-complete for BPP nets. However, since this result would not help us settle the main question, we refrain from investigating this further.

We come back to the problem under consideration. The following proposition is the result that we need. Together with the first to steps from the proof of Theorem 9.3.2 as outlined above, it proves Theorem 9.4.2.

9.4.3 Proposition
Given a product p and a BPP net N, one can decide whether the inclusion $\mathcal{L}(\text{lin}(p)) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$ holds in NP.

The proof of the proposition combines various techniques: In addition to the linearization from Section 9.3, it also uses the tricks that we have used to show that the word problem for BPP nets is in NP, Proposition 6.4.6, and the pumping lemma for BPP nets, Lemma 9.2.3.

Let $(N, M_{\text{init}}, M_{\text{final}})$ be the given BPP net, and let p be the given product. We assume that the linearization of p is

$$w_{(0)}\left(w_{\Gamma_1}\right)^* w_{(1)} \dots a_{m-1}\left(w_{\Gamma_m}\right)^* w_{(m)}$$

with words $w_{(i)} \in \Sigma^*$ and subalphabets $\Gamma_i \subseteq \Sigma$ for all i. The idea is the following: We check that there is a computation for words from the linearization in which the parts corresponding to $(w_{\Gamma_i})^*$ can be repeated. This means that we require that after seeing (w_{Γ_i}) once, we are in a larger marking than before.

We want to encode this property in Presburger arithmetic. To avoid having to express unbounded markings, we use the exponential constant

$$k = (||M_{\text{init}}||_1 + 2) \cdot ((|P| + 1) \cdot ||N||_\infty)^{(|T|+2)}$$

from Lemma 9.2.3. For two markings M, M', we write $M \leqslant^k M'$ if for all places $p \in P$ we have that $M'(p) < k$ implies $M(p) \leqslant M'(p)$.

Using this notation, we can formalize the above explanation.

9.4.4 Definition
A *witness* for $\text{lin}(p)$ is a covering computation

$$M_{\text{init}} = M_0 \xrightarrow{\sigma_0} M'_0 \xrightarrow{\tau_1} M_1 \xrightarrow{\sigma_1} M'_1 \xrightarrow{\tau_2} \dots M'_{m-1} \xrightarrow{\tau_m} M_m \xrightarrow{\sigma_m} M'_m \geqslant M_{\text{final}} ,$$

of $N{\downarrow}$ where $\sigma_i, \tau_i \in T^*$ are firing sequences so that (1) $\lambda(\sigma_i) = w_{(i)}$ for all $i \in [0, m]$, (2) $\lambda(\tau_i) = w_{\Gamma_i}$ for all $i \in [1, m]$, and (3) $M'_i \leqslant^k M_{i+1}$ for all $i \in [0, m-1]$.

Note that unlike in the rest of this thesis, σ_i and τ_i denote firing sequences instead of single transitions here. The first two properties express that the word generated by the computation is $w_{(0)} w_{\Gamma_1} \dots w_{\Gamma_m} w_{(m)}$. The last property means that each infix τ_i that correspond to the occurrence of some w_{Γ_i} can be repeated. Because we only require $M'_i \leqslant^k M_{i+1}$ (instead of $M'_i \leqslant M_{i+1}$) we might need to insert pumps, as in the proof of Theorem 8.2.2.

With the definition of a witness at hand, the proof of Proposition 9.4.3 decomposes into two steps. We first show that checking the inclusion between the linearization of a product and the language of $N{\downarrow}$ amounts to checking the existence of a witness. Then, we show how the latter can be checked in NP by encoding it into an existential Presburger formula.

9.4.5 Lemma
The inclusion $\mathcal{L}(\text{lin}(p)) \subseteq \mathcal{L}(N{\downarrow}, M_{\text{init}}, M_{\text{final}})$ holds if and only if $\text{lin}(p)$ has a witness.

Proof:
Let us assume that the computation

$$M_{\text{init}} = M_0 \xrightarrow{\sigma_0} M'_0 \xrightarrow{\tau_1} M_1 \xrightarrow{\sigma_1} M'_1 \xrightarrow{\tau_2} \dots M'_{m-1} \xrightarrow{\tau_m} M_m \xrightarrow{\sigma_m} M'_m \geqslant^k M_{\text{final}} ,$$

of $N{\downarrow}$ is a witness for $\text{lin}(p)$. Consider the word $w = w_{(0)}(w_{\Gamma_1})^{j_1} \dots (w_{\Gamma_m})^{j_m} w_{(m)} \in \mathcal{L}(\text{lin}(p))$. Our goal is to show that $w \in \mathcal{L}(N{\downarrow}, M_{\text{init}}, M_{\text{final}})$. The firing sequence $\sigma_0.{\tau_1}^{j_1}.\sigma_1 \dots {\tau_m}^{j_m}.\sigma_m$ has the correct label and intuitively should induce a covering computation of $N{\downarrow}$. However, we have not required $M'_i \leqslant M_{i+1}$ in the definition of a witness, but only $M'_i \leqslant^k M_{i+1}$.

Instead of using Lemma 9.2.3 to insert pumps as in the proof of Theorem 8.2.2, we simply use Theorem 8.2.2: We have shown that the downward closures of the language of N and of $\mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$, the $[0,k]$-ω-overapproximation coincide. We first project the transition sequence back to the net N, meaning that we use the non-ε-labeled variant of each transition wherever needed. This sequence then induces an accepting run of the automaton $A_{>k}$ with $\mathcal{L}(A_{>k}) = \mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}})$ on a superword of w. Indeed, the automaton is defined to ignore the precise value of markings on places where bound k has been exceeded. Hence, requiring $M'_i \leqslant^k M_{i+1}$ is sufficient to show that the transitions can be repeated. Altogether, we have $w \in \mathcal{L}_{>k}(N, M_{\text{init}}, M_{\text{final}}){\downarrow} = \mathcal{L}(N{\downarrow}, M_{\text{init}}, M_{\text{final}})$.

For the other direction, define $\ell = (k+2)^{|P|}$, and consider the word

$$w_{(0)}\big(w_{\Gamma_1}\big)^{\ell} w_{(1)} \dots w_{(m-1)}\big(w_{\Gamma_m}\big)^{\ell} w_{(m)} \in \mathcal{L}(\text{lin}(p)) .$$

Since we assume that the inclusion holds, $N{\downarrow}$ has a computation

$$M_{\text{init}} = M_0 \xrightarrow{\sigma_0} M'_0 \xrightarrow{\tau_1} M_1 \xrightarrow{\sigma_1} M'_1 \xrightarrow{\tau_2} \dots M'_{m-1} \xrightarrow{\tau_m} M_m \xrightarrow{\sigma_m} M'_m \geqslant M_{\text{final}} ,$$

where $\lambda(\sigma_i) = w_{(i)}$ for all i and $\lambda(\tau_i) = (w_{\Gamma_i})^{\ell}$ for all i.

This computation already covers the final marking and satisfies the first property from Definition 9.4.4. For the other two properties, we essentially need to identify for each τ_i an infix τ'_i such that the marking before firing τ''_i is $\leqslant^k$-smaller than the marking after firing τ'_i. The number ℓ has been chosen so that this is possible.

Let us focus on one infix $M'_{i-1} \xrightarrow{\tau_i} M_i$. Since $\lambda(\tau_i) = (w_{\Gamma_i})^{\ell}$, we may split it into

$$M'_{i-1} = M^{(0)} \xrightarrow{\tau^{(1)}} M^{(1)} \xrightarrow{\tau^{(2)}} \dots \xrightarrow{\tau^{(\ell)}} M^{(\ell)} = M_i$$

where $\lambda(\tau^{(j)}) = w_{\Gamma_i}$ for each j and $\tau_i = \tau^{(1)} \dots \tau^{(\ell)}$. Now observe that the set $([0,k] \cup \{\omega\})^P$ has cardinality $(k+2)^{|P|} = \ell$. This means that if we identify all numbers strictly greater than k with each other, there are only ℓ different markings. The sequence $M^{(0)}, M^{(1)}, \dots, M^{(\ell)}$ of markings from the above computation contains $\ell + 1$ markings. Hence, there need to be indices $j < j'$

such that $M^{(j)}$ and $M^{(j')}$ assign more than k tokens to the same places, and coincide on the rest of the places. This in particular means that $M^{(j)} \leqslant^k M^{(j')}$.

We may rewrite $M'_{i-1} \xrightarrow{\tau_i} M_i$ as

$$M'_{i-1} \xrightarrow{\tau^{(1)} \dots \tau^{(j)}} M^{(j)} \xrightarrow{\tau^{(j+1)} \dots \tau^{(j')}} M^{(j')} \xrightarrow{\tau^{(j'+1)} \dots \tau^{(\ell)}} M_i \,.$$

To obtain a witness, it remains to get rid of superfluous letters. Recall that in $N\!\downarrow$, each transition has an ε-labeled variant. We thus may define τ'_i to be obtained from $\tau^{(1)} \dots \tau^{(j)}$ by replacing all non-ε-labeled transitions by their ε-labeled variant, τ'''_i by applying the same to $\tau^{(j'+1)} \dots \tau^{(\ell)}$, and finally τ''_i to be obtained from $\tau_{(j+1)} \dots \tau_{(j')}$ by keeping the non-ε-labeled transitions in $\tau_{(j')}$, but applying the replacement to all other transitions. We obtain $\lambda(\tau'_i) = \lambda(\tau'''_i) = \varepsilon$, $\lambda(\tau''_i) = w_{\Gamma_i}$.

To finish the construction, we replace τ_i by $\tau'_i.\tau''_i.\tau'''_i$, then merge τ'_i with the preceding sequence σ_{i-1} and τ'''_i with σ_i. This means we replace

$$M_{i-1} \xrightarrow{\sigma_{i-1}} M'_{i-1} \xrightarrow{\tau_i} M_i \xrightarrow{\sigma_i} M'_i$$

by

$$M_{i-1} \xrightarrow{\sigma_{i-1}.\tau_i'} M^{(j)} \xrightarrow{\tau_i''} M^{(j')} \xrightarrow{\tau_i'''.\sigma_i} M'_i \,,$$

where we have $\lambda(\sigma_{i-1}.\tau'_i) = a_{i-1}$, $\lambda(\tau''_i) = w_{\Gamma_i}$, $\lambda(\tau'''_i.\sigma_i) = a_i$, and $M^{(j)} \leqslant^k M^{(j')}$.

Applying this replacement to each τ_i yields a witness. ■

With the lemma proven, it remains to check the existence of witnesses in nondeterministic polynomial time. To solve this problem, we construct a formula in existential Presburger arithmetic φ that is satisfiable if and only if a witness exists. We proceed similar to the proof of NP membership of the word problem for BPP nets, Proposition 6.4.6. The formula is a conjunction of the formula characterizing reachability in a modified net N' and several formulas that express the conditions that we impose on a witness.

Consider the word

$$w = w_{(0)} w_{\Gamma_1} w_{(1)} \dots w_{(m-1)} w_{\Gamma_1} w_{(m)} \in \mathcal{L}(\mathrm{lin}(p))$$

in which we take one iteration of each w_{Γ_i}. The proof of Proposition 6.4.8 yields a formula $\varphi = \varphi_{(N\downarrow, M_{\mathrm{init}}, M_{\mathrm{final}}), w}$ in existential Presburger arithmetic of size polynomial in the input size that is satisfiable if and only if $w \in \mathcal{L}(N\!\downarrow, M_{\mathrm{init}}, M_{\mathrm{final}})$.

We recall the construction on a high level of abstraction: For each of the letters w_i of w, the formula guesses the transition t_i of $N\!\downarrow$ with $\lambda(t_i) = w_i$ that will generate it. To take care of the sequences of ε-labeled transitions between the letters, we use the characterization of reachability in BPP nets by an existential Presburger formula. Intuitively, for each letter w_i, the formula

existentially quantifies over the marking that will be reached before resp. after transition t_i has been fired. To implement this, we consider the net N_ε obtained from $N{\downarrow}$ by removing all non-ε-labeled transitions. For each letter w_i of w, we create two copies of $N{\downarrow}$. In one copy, the sequence of ε-labeled transitions before t_i is executed. The initial marking for this copy should be the marking reached after firing the transition corresponding to the letter before w_i. Since we do not have this marking at hand, we add transitions that allow a computation to populate the copy with an arbitrary initial marking. These transitions generate the same markings in both copies. The other copy does not have any transitions; it is simply used to keep the initial marking available for the formula.

The formula φ that we construct then checks that there is a marking that is reachable in the modified net (which in particular implies that in each copy, the transition relation of net $N{\downarrow}$ has been respected), the marking reached in the copy for letter w_i enables transition t_i, the initial marking of the copy for the next letter after w_i is populated with the marking that results from firing transition t_i, and the final marking reached in the last copy covers M_{final}.

To be able to use the construction from the proof of Proposition 6.4.8, we need to incorporate a small modification. The formula φ makes available to us the markings that occur before and after each of the letters of the word w in the form of its free variables. However, the markings M_i and M'_i that we need to consider when checking for the existence of a witness may occur not directly before or after a proper letter, but in the midst of a sequence of ε-labeled transitions. To overcome this problem, we change the construction so that for the last letter of each infix w_{Σ_i}, the formula actually makes available to us two markings: The marking that occurs after the transition corresponding to the letter, and another marking that is seen after a (potentially empty) sequence of ε-transitions. Similarly, for the first letter of each infix w_{Σ_i}, we assume that in addition to the marking directly before the transition corresponding to the letter, we also have a marking from which the first can be reached by ε-transitions available. Formally implementing this is an extension of the proof of Proposition 6.4.8 that is straightforward. We simply insert additional copies of the nets at the appropriate locations and extend the formulas to ensure that these copies are populated correctly.

With the modification in place, we can assume that the modified formula φ makes available to us the following markings in the form of free variables: M_i, the freshly introduced marking in the corresponding copy of N_ε before (a sequence of ε-transitions before) the first letter of w_{Γ_i}, and M_{i+1}, the marking reached in the copy for the last letter of w_{Γ_i} (and a sequence of ε-transitions).

Finally, we construct a new formula $\varphi \wedge \psi$ that checks for the existence of a witness, where ψ checks Condition (3), $M'_i \leqslant^k M_{i+1}$: The marking before seeing w_{Γ_i} should be $\leqslant^k$-smaller than the marking after seeing w_{Γ_i}. Formally, we have

$$\psi = \bigwedge_{i \in [1,m]} \bigwedge_{p \in \mathbb{P}} (M'_i(p) < k) \implies M_i(p) \leqslant M_{i+1}(p)\,.$$

We claim that this formula can be constructed in polynomial time. The only critical part is the encoding of k. Number $k = (||M_{\text{init}}||_1 + 2) \cdot ((|P| + 1) \cdot ||N||_\infty)^{(|T|+2)}$ is exponential, but its binary encoding is of size polynomial in the size of the input.

Altogether, the construction proves the following lemma.

9.4.6 Lemma
We can construct in polynomial time a formula in existential Presburger arithmetic that is satisfiability if and only if there is a witness for $\text{lin}(p)$.

With that lemma at hand, we simply need to collect all our results to get the proof of Proposition 9.4.3, which implies Theorem 9.4.2.

Proof of Proposition 9.4.3:
Consider a product p and a BPP net $(N, M_{\text{init}}, M_{\text{final}})$ for which we want to check the inclusion $\mathcal{L}(\text{lin}(p)) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$. By Lemma 9.4.5, it holds if and only if there is a witness for $\text{lin}(p)$. With Lemma 9.4.6, we can construct a formula in existential Presburger arithmetic of polynomial size that is satisfiable if and only if a witness exists. By the NP-completeness of EPA-SAT, Theorem 3.4.2, this can be checked in nondeterministic polynomial time. ■

This completes this section and the proof of Theorem 9.4.1. Note that the problem remains NP-complete in the unary case since the lower bound continues to hold, as in Section 8.4.

Altogether, this finishes our study of the downward closure of Petri net coverability languages. We have proven all results regarding upward closures that we mentioned in Section 7.3.

10 Being upward/downward closed and regular containment

We complement our study of computing the upward/downward closure of Petri net coverability languages by considering the problems of checking whether such a language is upward resp. downward closed. If this is the case, the regular downward resp. upward closure, a representation of which we are able to compute, is actually the language of the Petri net.

In Section 1.2, we have motivated our interest in the upward and downward closures of languages by situations in which we want to take the lossy or gainy behavior of a system into account. Let us assume a situation in which we have incorporated this lossiness or gaininess into the model. For example, we may have constructed a Petri net that models a lossy network, where the fact that messages can be dropped is already integrated into the model. In this case, one part of verifying the model could consist of checking that the language of the Petri net is indeed downward closed, which means that it is equal to its downward closure.

We start by formally defining the problems PN-BEINGUC and PN-BEINGUC.

Being upward closed (PN-BEINGUC)
Given: A labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$.
Question: $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$?

Being downward closed (PN-BEINGDC)
Given: A labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$.
Question: $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\downarrow$?

The goal of this chapter is to prove that both problems are decidable. For the proof of decidability, we consider *regular containment*, the problem of checking whether a given regular language is contained in given Petri net language. This problem is of independent interest: One the practical side, it allows us to solve verification tasks in which the goal is to show that a given set of behaviors can actually occur. On the theoretical side, other interesting problems like universality can be reduced to regular containment. Proving its decidability will imply the decidability of PN-BEINGUC and PN-BEINGDC.

10.0.1 Theorem
PN-BEINGUC and PN-BEINGDC are decidable.

Assume that we want to check $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ for some given Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$. Note that the inclusion $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}}) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ always holds. It remains to check whether $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ is true. Here, the language $\mathcal{L}(N, M_{\text{init}}, M_{\text{final}})\uparrow$ is effectively regular, i.e. it is regular and Theorem 8.1.2 allows us to compute an NFA representing it. The situation is similar in the case of PN-BEINGDC: Theorem 9.1.2 yields computability of the downward closure.

Therefore, once we prove that checking whether $\mathcal{L}(A) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ holds for some arbitrary NFA A is decidable, the decidability of PN-BEINGUC immediately follows.

10.1 Regular containment

As mentioned before, the problem of checking whether a given regular language is contained in a Petri net coverability language is of independent interest. We formalize it as follows.

Regular containment (PN-REGCONT)
Given: A labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$, an NFA A.
Question: $\mathcal{L}(A) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$?

In the rest of the section, our goal is to prove that PN-REGCONT is decidable.

10.1.1 Theorem
PN-REGCONT is decidable.

Before giving the proof, we discuss the consequences of this result. We have already mentioned potential practical applications in the previous sections.

Firstly, note that the decidability of PN-REGCONT implies the decidability of other interesting computational problems. In particular, we get the decidability of the universality problem for the coverability languages of Petri nets, i.e. checking whether $\Sigma^* = \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$, as an immediate consequence. This is, to the best of the author's knowledge, a new result. In fact, Wimmel [Wim08] conjectures that this problem is undecidable even for Petri nets that do not use ε as transition label. The results extend the known decidability of universality for freely labeled Petri nets, i.e. Petri nets in which each transition has a distinct non-ε label. The latter holds for several acceptance conditions, including coverability and reachability. It contrasts with the undecidability of universality in the case of non-freely labeled Petri nets with reachability as the acceptance condition [Wim08].

Secondly, the decidability is surprising in that the proof approach that is commonly used to show the decidability of the regular containment problem does not apply in the case of Petri net languages. Usually, one uses that $\mathcal{L}_1 \subseteq \mathcal{L}_2$ holds if and only if there is no counterexample to inclusion, i.e. $\mathcal{L}_1 \cap \overline{\mathcal{L}_2}$ is empty. This approach requires $\mathcal{L}_2$ to stem from a class of languages that can be effectively complemented. It is well-known that the class of Petri net coverability languages is not closed under complement [MKRS98b; MKRS98a]. For some other classes that are not closed under complement, e.g. the context-free languages, the regular containment problem is undecidable. Also, several other problems that could be solved if an effective complementation of Petri net languages would be possible, are known to be undecidable. These problems include the inclusion problem, i.e. the question of whether $\mathcal{L}(N_1) \subseteq \mathcal{L}(N_2)$ holds for two given Petri nets N_1, N_2, even in the simple case that the nets do not use ε as label and that the

acceptance condition is trivial, i.e. it is coverability with respect to the zero vector. The undecidability of the inclusion problem can be deduced using a technique introduced by Jančar [Jan95], see e.g. [Wim08] for a proof.

Trace containment

Let us turn to the proof of Theorem 10.1.1. It strongly relies on a result of Jančar, Esparza, and Moller [JEM99] that shows that the containment problem is decidable for *trace languages* of finite-state automata resp. Petri nets. A trace of an automaton is a finite word that occurs along a computation from the initial state, regardless of whether it reaches a final state. Formally, we define the *trace language* of an NFA A as

$$\mathcal{T}(A) = \left\{ w \in \Sigma^* \;\middle|\; q_{\text{init}} \xrightarrow{w} q \text{ for some } q \in Q \right\}.$$

Similarly, we define the trace language of a Petri net N together with an initial marking M_{init} by

$$\mathcal{T}(N, M_{\text{init}}) = \left\{ \lambda(\sigma) \in \Sigma^* \;\middle|\; M_{\text{init}} \boxed{\sigma}\!\!\rangle \right\}.$$

Observe that $\mathcal{T}(A)$ is the language of A if we assume that all states are final. Similarly, the trace language of the Petri net is the coverability language with the zero vector as the final marking, $\mathcal{T}(N, M_{\text{init}}) = \mathcal{L}(N, M_{\text{init}}, \vec{0})$. An important property of trace languages that limits their expressiveness is that they are necessarily *prefix closed*: If $w = w_1 \dots w_n \in \mathcal{T}(A)$, then also $w_1 \dots w_k \in \mathcal{T}(A)$ for all $k \leqslant n$.

The *trace containment* problem is a variant of regular containment in which we restrict both languages to be trace languages.

Trace containment (PN-TRACECONT)	
Given:	A labeled Petri net with initial marking (N, M_{init}), an NFA A.
Question:	$\mathcal{T}(A) \subseteq \mathcal{T}(N, M_{\text{init}})$?

The result that we rely on is the decidability of this problem.

10.1.2 Theorem (Jančar, Esparza, and Moller [JEM99])
PN-TRACECONT is decidable.

We have argued above that an algorithm for containment cannot simply complement a Petri net coverability language. The same reasoning holds in the case of trace containment. We briefly discuss the algorithm proving Theorem 10.1.2. Let us assume that A is a DFA, which in particular means that it has no ε-labeled transitions. The algorithm simultaneously explores the behavior of the automaton and the net, in a fashion that is similar to the construction of the Karp-Miller tree, see Section 9.3. It constructs a tree whose nodes are labeled by pairs $(q, \mathcal{M})$, where q is a

state of the automaton and $\mathcal{M}$ is a set of incomparable generalized markings. This means that $\mathcal{M} \subseteq \mathcal{P}(\mathbb{N}_\omega^P)$, and for $M, M' \in \mathcal{M}$, neither $M \leqslant M'$ nor $M' \leqslant M$ holds.

The root node is labeled by the initial state and the singleton set consisting of the initial markings. Successors of a node $(q, \mathcal{M})$ are constructed by picking an applicable transition of the automaton, say $q \xrightarrow{a} q'$, and then constructing the finite set of maximal markings from the set $\mathcal{M}' = \{M' \in \mathbb{N}_\omega^P \mid \exists M \in \mathcal{M}, \exists \sigma\colon \lambda(\sigma) = a, M \boxed{\sigma} M'\}$. Note that there are potentially infinitely many candidates for σ, since we allow ε-labeled transitions in the net. Altogether, we obtain the successor $(q', \mathcal{M}')$.

If we have constructed a successor of the shape $(q', \varnothing)$, trace containment is violated. Indeed, the path from the root node to $(q', \varnothing)$ can be pumped to obtain a witness for a word that is contained in $\mathcal{T}(A)$, but not in $\mathcal{T}(N, M_{\text{init}})$. If we have constructed a successor $(q', \mathcal{M}')$ such that on the path from the root node to $(q', \mathcal{M}')$, there is a node $(q', \mathcal{M})$ such that the control states are equal and for every $M' \in \mathcal{M}'$, there is $M \in \mathcal{M}$ with $M' \leqslant M$, $(q', \mathcal{M}')$ becomes a leaf for which we do not have constructed further successors.

The soundness of the algorithm is not hard to prove. The difficult part is showing that the succors can be constructed in finite time, and that the tree is necessarily finite. This part of the proof relies on well-quasi ordering arguments. Hence, a naive complexity analysis of the algorithm yields a non-primitive recursive complexity. A more detailed analysis using the techniques developed by Schmitz and others, see e.g. [LS15a], is beyond the scope of this thesis.

Reducing regular containment to trace containment

To prove Theorem 10.0.1, we show how to reduce an instance of PN-REGCONT to an instance of PN-TRACECONT. Let $(N, M_{\text{init}}, M_{\text{final}})$ be the labeled Petri net instance of interest, and let $A = (Q, \to, q_{\text{init}}, Q_F)$ be the NFA for which we want to check language containment. As argued above, we have $\mathcal{T}(N, M_{\text{init}}) = \mathcal{L}(N, M_{\text{init}}, \vec{0})$. Hence, we consider the zero vector as the new final marking. To take the original final marking M_{final} into account, we will use a new transition with a fresh label that can only be fired once M_{final} has been covered. Similarly, will we add transitions with this label to the automaton that witness that a final state has been reached.

More formally, let $a \notin \Sigma$ be a fresh letter and define $\Sigma_a = \Sigma \uplus \{a\}$. We define $N.a$ to be the labeled Petri net over Σ_a that is obtained from N by adding an a-labeled transition t_a that produces no token and consumes M_{final}, i.e. $\text{in}(t_a) = M_{\text{final}}$, $\text{out}(t_a) = \vec{0}$. Similarly, we construct an automaton $A.a$ over Σ_a for the language $\mathcal{L}(A).a$. For the proof, it will be important that $A.a$ is reduced in these sense that it has a unique final state that is reachable from all states. This property guarantees that each word $w \in \mathcal{T}(A.a)$ is the prefix of a word from $\mathcal{L}(A.a)$. To ensure it, we construct $A.a$ from A as follows: (1) We add a fresh final state q_{final}, then (2) add a-labeled transitions from all other final states to this state, (3) make all other states non-final, and finally (4) remove all states from which q_{final} is not reachable.

The net $N.a$ and the automaton $A.a$ have been constructed so that we can use them to reduce regular containment to trace containment, as stated by the following lemma.

10.1.3 Lemma
$\mathcal{L}(A) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ holds iff $\mathcal{T}(A.a) \subseteq \mathcal{T}(N.a, M_{\text{init}})$ holds.

Proof:
Assume $\mathcal{L}(A) \subseteq \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$ to hold. We prove that trace containment holds. Let $v \in \mathcal{T}(A.a)$, and let q be some state of $A.a$ such that $q_{\text{init}} \xrightarrow{v} q$. Since the unique final state is reachable from every state of $A.a$ by construction, there is some word w with $q \xrightarrow{w} q_{\text{final}}$. Hence, we have $v.w \in \mathcal{L}(A.a) = \mathcal{L}(A).a$, and we may write $v.w = x.a$ for some $x \in \mathcal{L}(A)$. By assumption, $x \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$, so there is a computation $M_{\text{init}} \boxed{\sigma} M \geqslant M_{\text{final}}$ of N with $\lambda(\sigma) = x$. This computation is also a computation of $N.a$, and it reaches a marking greater than M_{final}. Hence, we have that $\sigma.t_a$ is a valid firing sequence from M_{init}. We obtain that $\lambda(\sigma.t_a) = x.a \in \mathcal{T}(N.a, M_{\text{init}})$. Recall that v is a prefix of $v.w = x.a$, and that trace languages are prefix-closed. Hence, we obtain $v \in \mathcal{T}(N.a, M_{\text{init}})$ as desired.

For the other direction, assume $\mathcal{T}(A.a) \subseteq \mathcal{T}(N.a, M_{\text{init}})$ to hold. Let $w \in \mathcal{L}(A)$, and consider $w.a \in \mathcal{L}(A.a) \subseteq \mathcal{T}(A.a)$. By assumption, we have $w.a \in \mathcal{T}(N.a, M_{\text{init}})$, so we may choose a computation $M_{\text{init}} \boxed{\sigma} M$ of $N.a$ with $\lambda(\sigma) = w.a$. Note that w is a word over Σ and does not contain a. Hence, σ is of the shape $\sigma = \sigma'.t_a$, where σ' only uses transitions from the original net N. We get that $M_{\text{init}} \boxed{\sigma'} M$ is a computation of N reaching a marking in which transition t_a is enabled, which implies $M \geqslant M_{\text{final}}$. Hence, $\lambda(\sigma') = w \in \mathcal{L}(N, M_{\text{init}}, M_{\text{final}})$, as we wanted to prove. ■

Combining the reduction from regular containment to trace containment, Lemma 10.1.3, with the decidability of trace containment, Theorem 10.1.2, yields the desired decidability of regular containment, Theorem 10.1.1.

This also finishes the proof of Theorem 10.0.1. With this proof, all results mentioned in Section 7.3 have been shown and our study of the computability and complexity of the upward and downward closures of Petri net coverability languages is complete.

Part IV.

Separability

Contents

11 Separability and regular separability 215
- 11.1 Basic definitions . . . 215
- 11.2 Related work . . . 218
- 11.3 Results . . . 221

12 WSTS expressiveness 223
- 12.1 ω^2-WSTSes . . . 223
- 12.2 Results on WSTS expressiveness . . . 226

13 Regular separability for WSTSes 235
- 13.1 The results and their consequences . . . 235
- 13.2 Technical core . . . 239
- 13.3 Ideals and the ideal completion . . . 247

14 Bounds on the separator size for Petri nets 255
- 14.1 Upper bound . . . 255
- 14.2 Lower bound . . . 265

Part IV.
Separability

In Section 1.1 of the introduction, we have explained that the (regular) separability problem is related to the compositional verification of concurrent systems. This part of the thesis is concerned with studying regular separability for well-structured transition systems (WSTSes).

Outline

We start by providing the basic definitions regarding separability and regular separability in Chapter 11. We also discuss related work from the literature and summarize the results that we will show in this part of the thesis.

Before actually considering the separability problem for the class of languages of WSTSes, we study the relations among various subclasses thereof in Chapter 12. These results widen the applicability of our main result, which we will present in Chapter 13. We will prove that under mild preconditions, any two disjoint WSTS languages are regularly separable. We first prove a technical core result that relates a certain type of invariant of the state space of a WSTS to the existence of a regular separator. Then, we use the ideal completion of a WSTS to prove that such an invariant always has to exist.

In Chapter 14, we demonstrate our result on regular separability by applying it to Petri net coverability languages, a subclass of the class of WSTS languages. We explicitly provide the construction of a regular separator, which yields a triply exponential state complexity. Furthermore, we prove a doubly exponential lower bound for that space complexity.

Publication

The content we present in this section is mostly taken from the publication [CLMMKS18] (resp. its full version [CLMMKS18a]). We will discuss the author's contributions to this paper in Chapter 20.

11 Separability and regular separability

Contents

11.1 Basic definitions 215
11.2 Related work 218
11.3 Results 221

In Section 1.1, we have motivated our interest in separability by its potential applications in compositional verification. In this chapter, which serves as an introduction to this part of the thesis, we will elaborate on the theoretical background of the separability problem. This includes giving the formal definition. Afterwards, we discuss some results from the literature that settle the status of decidability of the separability problem for several language classes. Finally, we outline the results that we will prove in this part of the thesis.

11.1 Basic definitions

The separability problem can be seen as an extended version of the intersection-emptiness problem. The intersection-emptiness problem asks whether two given languages are disjoint. Usually, we fix the classes from which the languages should come beforehand and expect that the languages are given in the form of an effective representation, e.g. we expect context-free languages to be given as context-free grammars.

Intersection-emptiness for $\mathcal{F}, \mathcal{F}'$

Given: Language $L \subseteq \Sigma^*$ from $\mathcal{F}$, Language $K \subseteq \Sigma^*$ from $\mathcal{F}'$.

Question: Are L and K disjoint, i.e. $L \cap K = \varnothing$?

For their usage in verification, two variants of this problem are considered to be particularly interesting. The first is the case in which one of the classes is the class of regular languages and the other class is a class beyond the regular languages. This models the verification problem in which we have a system, represented by the potentially non-regular language consisting of its possible executions, and a specification, represented by the language of executions that violate it. The latter can be assumed to be regular because the expressive power of specification languages (e.g. the logics LTL [Pnu77] and S1S [Büc62] on finite words) is typically at most equal to the regular languages. The system violates the specification if it has an execution violating the specification, i.e. the intersection of the languages is non-empty. This variant of the intersection-emptiness problem is decidable for many classes of languages. Assuming that the

class that we are considering is effectively closed under regular intersection, we can modify the given system. We obtain a system whose language is the intersection of the original language with the regular language. Then, we invoke a decider for the emptiness problem for that system. The assumptions that we have used here are met for example in the case of context-free languages and in the case of Petri net languages with both coverability and reachability as acceptance conditions.

In the second interesting variant of the intersection-emptiness problem, we consider two languages coming from the same class of languages $\mathcal{F}$; we speak of the intersection-emptiness problem for $\mathcal{F}$. This problem can be used to model a setting in which we consider a concurrent system. Let us for the sake of simplicity assume that the system only has two components. We model each of the components by a language that contains all executions that (1) respect the behavior of the corresponding component, (2) are arbitrary with respect to the other component, and (3) violate the specification. See Section 1.1 for a more in-depth explanation of the latter two aspects. We obtain two languages whose intersection is non-empty if and only if there is an execution that respects the behavior of both components of the concurrent system but violates the specification.

This version of the intersection-emptiness problem is typically much more complicated than the intersection-emptiness problem where one of the classes is the class of regular languages. As in the rest of this thesis, we are interested in an algorithm that does not only decide the problem, but also provides a certificate. In the case of the two languages not being disjoint, a natural candidate for this certificate is a word in the intersection of the languages. Consider for example the intersection-emptiness problem for context-free languages, which is well-known to be undecidable (see e.g. [HU79] for a proof). If we manage to identify a word that is in the intersection, it is easy to verify the output by one membership query for each of the languages.

The problem of providing a certificate for the emptiness of the intersection of two languages is much more involved. To this end, the notion of a separator can be used. A *separator* for two languages $\mathcal{L}, \mathcal{K} \subseteq \Sigma^*$ is a third language $\mathcal{R} \subseteq \Sigma^*$ that fully contains $\mathcal{L}$ and is disjoint from $\mathcal{K}$, i.e. $\mathcal{L} \subseteq \mathcal{R}$, $\mathcal{K} \cap \mathcal{R} = \varnothing$. Obviously, the existence of a separator is equivalent to intersection-emptiness: If the languages are not disjoint, a separator cannot exist. Otherwise, $\mathcal{L}$ is a separator. Hence, it only makes sense to study separators if we restrict them to come from a simpler class – one with less expressive power – than $\mathcal{L}$ and $\mathcal{K}$ do. We call a separator that comes from language class $\mathcal{S}$ an *$\mathcal{S}$-separator*, and we say that $\mathcal{L}$ and $\mathcal{K}$ are *$\mathcal{S}$-separable* if an $\mathcal{S}$-separator exists. The decision problem of checking the existence of such a separator is the following.

$\mathcal{S}$-separability for $\mathcal{F}, \mathcal{F}'$	
Given:	Languages $\mathcal{L} \subseteq \Sigma^*$ from $\mathcal{F}$ and $\mathcal{K} \subseteq \Sigma^*$ from $\mathcal{F}'$.
Question:	Are $\mathcal{L}, \mathcal{K}$ $\mathcal{S}$-separable, i.e. is there $\mathcal{R} \subseteq \Sigma^*$ from $\mathcal{S}$ with $\mathcal{L} \subseteq \mathcal{R}$ and $\mathcal{K} \cap \mathcal{R} = \varnothing$?

We are mostly interested in the case that both given languages come from the same class $\mathcal{F}$, which we call the $\mathcal{S}$-separability problem for $\mathcal{F}$. A positive answer to intersection emptiness is necessary for a positive answer to $\mathcal{S}$-separability, independent of the choice of $\mathcal{S}$. If we chose $\mathcal{S}$ to be $\mathcal{F}$, the two problems are equivalent as mentioned before. If we pick $\mathcal{S}$ to be a simpler class, the problem becomes interesting. Assume that we have an algorithm solving the $\mathcal{S}$-separability problem for some class $\mathcal{F}$ that is constructive in the sense that it outputs a representation of the separator if one exists. This algorithm can be seen as an algorithm for intersection-emptiness that produces a certificate in the case that the intersection is indeed empty. Here, we use that checking that a given language $\mathcal{R}$ is a separator is typically – depending on the choice of class $\mathcal{S}$ – much simpler than checking intersection-emptiness. However, one has to be careful since intersection-emptiness does not imply the existence of a separator if the class $\mathcal{S}$ is too restrictive. Consider for example the context-free languages $\mathcal{L} = \{a^n b^n \mid n \in \mathbb{N}\}$ and $\mathcal{K} = \overline{\mathcal{L}}$ that are complements of each other. The languages are obviously disjoint, but not $\mathcal{S}$-separable if we choose $\mathcal{S}$ to be a subclass of the context-free languages that does not contain $\mathcal{L}$; the regular languages for example.

From the perspective of verification, the most important variant of the separability problem is *regular separability*, i.e. checking the existence of a separator from the class of regular languages. In this case, we call the separators *regular separators* and we call two languages that have such a separator *regularly separable*. We formally state regular separability as a decision problem.

Regular separability of $\mathcal{F}$

Given:	Languages $\mathcal{L}, \mathcal{K} \subseteq \Sigma^*$ from $\mathcal{F}$.
Question:	Are $\mathcal{L}, \mathcal{K}$ regularly separable,
	i.e. is there $\mathcal{R} \subseteq \Sigma^*$ regular with $\mathcal{L} \subseteq \mathcal{R}$ and $\mathcal{K} \cap \mathcal{R} = \varnothing$?

Its importance stems from the fact that regular languages admit a plethora of representations and algorithms that work on these. On the theoretical side, this means that given a regular candidate language $\mathcal{R}$, the problem of checking whether $\mathcal{R}$ is indeed a regular separator is decidable for many language classes. We have argued above that the intersection-emptiness problem is decidable in many cases when one of the classes is the class of regular languages. This allows us to check $\mathcal{K} \cap \mathcal{R} = \varnothing$. We can then use that the class of regular languages is closed under complement and check $\mathcal{L} \subseteq \mathcal{R}$ by checking $\mathcal{L} \cap \overline{\mathcal{R}} = \varnothing$. Another nice property of the regular separability problem is that it is symmetric in the input: If $\mathcal{R}$ is a regular separator for $\mathcal{L}$ and $\mathcal{K}$, then its complement $\overline{\mathcal{R}}$ is a regular separator for $\mathcal{K}$ and $\mathcal{L}$. Most language classes beyond the regular languages do not satisfy these properties, a statement that we will make formal in the form of Corollary 13.1.5.

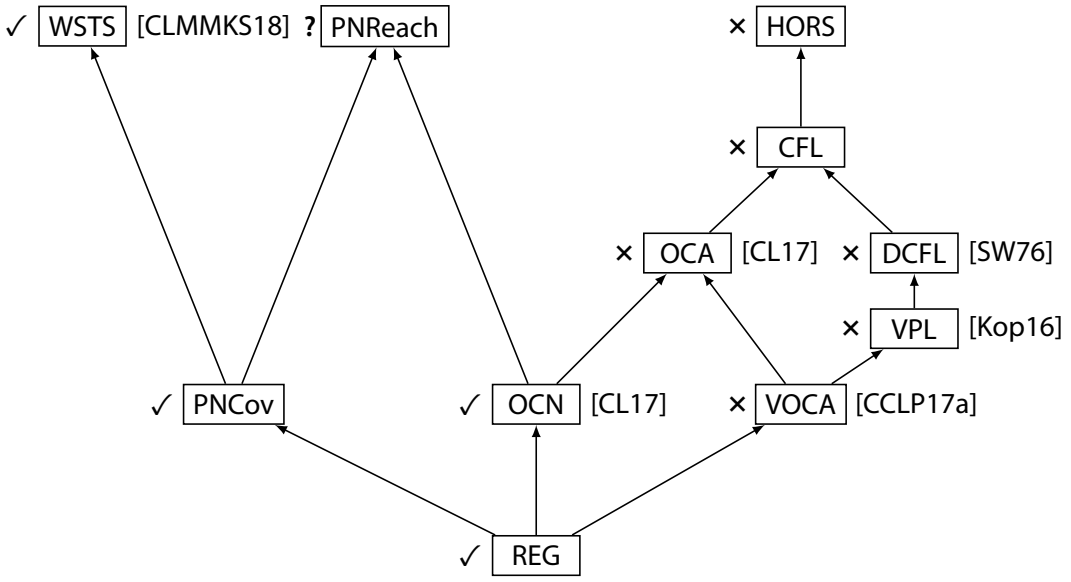


Figure 11.2.a: An overview of the decidability status of the regular separability problem for various languages classes. Boxes represent classes, arrows represent inclusions among classes. The symbol on the left-hand side of each node shows whether regular separability is decidable (✓), undecidable (×), or whether the decidability status is unknown (**?**).

11.2 Related work

We summarize results from the literature regarding separability and regular separability.

The separability *of* regular languages *by* subclasses of the regular languages is a widely studied problem, but beyond the scope of this thesis. Let us briefly mention that the decision problem of $\mathcal{S}$-separability of regular languages is decidable if $\mathcal{S}$ is the class of piecewise-testable languages [CMM13; PRZ13b], or the class of locally testable or locally threshold-testable languages [PRZ13a], or the class of languages definable in first-order logic [PZ16], or one of certain classes in the higher levels of the first-order hierarchy [PZ14].

Figure 11.2.a contains an overview of the results that have been established regarding regular separability. We will discuss these results in detail in the following.

Undecidable cases

A classical result by Szymanski and Williams [SW76] shows that regular separability is undecidable for the class DCFL, the *deterministic context-free languages*. This class is a strict subclass of CFL, the class of context-free languages, that is defined by deterministic pushdown automata. The undecidability for DCFL implies undecidability for all superclasses, including CFL, HORS (the languages of higher-order recursion schemes), and the class of Turing machine languages RE.

Kopczynski [Kop16] has shown that the problem is also undecidable for VPL, the class of languages of *visibly pushdown automata (VPAs)*. A VPA over an input alphabet that is partitioned

into push-letters, pop-letters, and internal letters, acts like a pushdown automaton, but in each step, the type of letter that is read determines the type of stack operation that should be performed. The undecidability for VPL is surprising because VPL shares many closures properties and the resulting algorithmic properties with the regular languages. In particular, VPL is closed under complementation and intersection-emptiness for VPL can be decided (assuming that both languages use the same partitioning of the alphabet) [AM04].

Czerwiński and Lasota [CL17] have shown that regular separability is also undecidable for OCA, the class of languages of one-counter automata. A one-counter automaton is an automaton that uses a single non-negative counter as storage in addition to its finitely many control states. The class OCA is a subclass of CFL – one-counter automata can be seen as pushdown automata over two stack symbols, where one is exclusively used to mark the bottom-of-stack – that is incomparable to VPL.

Finally, the paper [CCLP17a] strengthens the aforementioned results by showing that regular separability is undecidable even for languages of *visibly one-counter automata (VOCA)*, one-counter automata for which each letter that is read determines the counter operation that is performed. This type of automata defines a class of languages that is a strict subclass of both VPL and OCA. This shows that regular separability becomes undecidable for classes that extend the regular languages towards the context-free ones, even if the extension is very restricted.

Decidable cases

The paper by Czerwiński and Lasota [CL17] exhibits a case in which regular separability is decidable. It is decidable for OCN, the class of *one-counter nets*, one-counter automata that cannot check their counter value during runtime. Alternatively, these may be seen as Petri nets with reachability as the acceptance condition in which all places but one only contain one token in total. The decidability for OCN is interesting because regular separability is non-trivial in this case: There are disjoint languages from OCN that are not regularly separable. In the paper, checking the existence of a regular separator is reduced to checking whether a specific approximation is a separator for some constant determining the precision of the approximation. The latter can be reduced to reachability in 2-dimensional vector addition systems (or, equivalently, Petri nets with 2 unbounded places), which is PSPACE-complete [BFGHM15]. Regular separability inherits this complexity.

To the best of the author's knowledge, it has not yet been determined whether regular separability is decidable in the case of Petri net reachability languages. However, there are subclasses for which the problem has been shown to be decidable. We have already elaborated on the result for one-counter nets. In [CCLP17a], the authors prove that the problem is decidable for *Parikh automata*, or equivalently, *integer Petri nets* with reachability as the acceptance condition. An integer Petri net is a Petri net that is executed on what we have called pseudo-markings, markings that may assign negative numbers of tokens. Additionally, the regular separability problem has

been shown to be decidable for the class of commutative closures of Petri net reachability languages [CCLP17b]. Equivalently, it is decidable whether two Petri net reachability languages can be separated by the commutative closure of a regular language.

The rest of this part of the thesis is dedicated to showing results related to regular separability for Petri net coverability languages, another important subclass of the Petri net reachability languages. In fact, we will show that under mild assumptions, regular separability is decidable even for the languages of WSTSes, a class that extends the Petri net coverability languages and is incomparable to the Petri net reachability languages [GRV07]. We will discuss these results, which stem from the publication [CLMMKS18] in the next section.

Related problems

We conclude our overview of the related work from the literature by comparing regular separability to other decision problems for language classes. There are two candidate problems that seem to be closely related to regular separability. The first one is the intersection-emptiness problem for which we have already extensively discussed why it is related to regular separability. The second one is the *regularity problem*, the problem of checking whether the language of a given system is regular. Its relation to regular separability stems from the fact that if a language and its complement are from the same class of languages, then these languages are regular if and only if they are regularly separable. This is because in that case, the language itself is the only possible candidate for a regular separator; we will later formally prove this statement in the form of Corollary 13.1.3. However, it has been shown that both problems are independent of regular separability in general. The decidability or undecidability of regularity or intersection-emptiness does not imply the decidability or undecidability of regular separability and vice versa.

The result by Kopczynski [Kop16] on visibly pushdown languages exhibits a case in which intersection-emptiness is decidable, but regular separability is undecidable. Recently, Thinniyam and Zetzsche [TZ19] have constructed a setting in which regular separability is decidable while intersection-emptiness is undecidable. A notable property of their construction is that they consider the intersection-emptiness problem and regular separability problem for two classes of input languages that are not equal. Formally defining the classes of languages for which they show the result is rather involved and beyond the scope of this thesis.

Regarding regularity, it has been observed that regular separability is undecidable for deterministic one-counter automata [CL17], while regularity is decidable even for deterministic pushdown automata [Val75]. For the other direction, one observes that regularity is undecidable for Parikh automata [CFM11], while regular separability is decidable [CCLP17a].

11.3 Results

The main result that we will show in this part of the thesis is the following: *Under mild assumptions, any two disjoint WSTS languages are regularly separable*. The mild assumptions are in the case of upward-compatible WSTSes that one of the languages is the language of a WSTS that is finitely branching, deterministic or ω^2. The result also applies in the case of downward-compatible WSTSes (DWSTSes). Here, we need that one of the languages is the language of a deterministic DWSTS or the language of an ω^2-DWSTS. We have not yet formally defined the restriction of being ω^2; we will do so in Section 12.1. For now, it suffices to know that virtually all WSTSes that occur in practice are ω^2-WSTSes. In particular, our result applies to Petri net coverability languages.

In Chapter 12, we will study several restricted versions of WSTSes and prove inclusions among their languages classes. We obtain that the aforementioned restriction have in common that they allow us to determinize the corresponding WSTS.

We will use this fact in Chapter 13 when proving our results. Firstly, we show a technical core result that relates certain invariants on the level of configurations to the existence of a regular separator on the level of languages. This result is stated on the level of upward-compatible LTSes, which then allows us to apply it to both DWSTSes and to WSTSes.

Our result on regular separability implies that checking regular separability amounts to checking disjointness, which can be done under mild assumptions. Furthermore, it is constructive in the sense that from a certain type of proof of disjointness, one can construct a regular separator. Hence, it provides the desired certificate for intersection-emptiness, which was our initial motivation for studying regular separability.

We will demonstrate the construction in Chapter 14 in the case of Petri nets with coverability as the acceptance condition. We obtain a triply exponential upper bound for the state complexity of a regular separator. Furthermore, we complement this by providing a doubly exponential lower bound. We will comment on the fact that the bounds do not match and on several other open questions regarding our results throughout this part of the thesis.

12 WSTS expressiveness

Contents

12.1 ω^2-WSTSes 223
12.2 Results on WSTS expressiveness 226

In this chapter, we study the relations among various classes of WSTS languages that are defined by imposing restrictions on the WSTSes. This correlates with our analysis of separability for WSTS languages, the main goal of this part of the thesis, in the following sense: Many of the results that we will prove in this chapter will widen the applicability of our results on WSTS separability. We will comment on this in more detail in the next chapter.

12.1 ω^2-WSTSes

In Section 4.1, we have defined finitely branching and deterministic LTSes. These restrictions also apply to WSTSes, leading to the notions of finitely branching and deterministic WSTSes. In this section, we will introduce ω^2-WSTSes, another restricted version of WSTSes. In contrast to deterministic and finitely branching WSTSes, ω^2-WSTSes are defined exclusively by restricting the underlying order. To define what an ω^2-WQO is, we will need some notation. For a quasi-order $(X, \leqslant)$, we denote by $\mathcal{P}^{\downarrow}(X)$ the downward-closed subsets of X,

$$\mathcal{P}^{\downarrow}(X) = \{D \subseteq X \mid D \text{ is downward closed}\} = \{Y{\downarrow} \mid Y \subseteq X\}\,.$$

Similarly, we define $\mathcal{P}^{\uparrow}(X)$ to be the upward-closed subsets of X.

The formal definition of ω^2-WQOs, see e.g. [Mar94], is technical. Instead of giving it, we use the following characterization that was proven by Jancar [Jan99].

12.1.1 Lemma
$(X, \leq)$ is an ω^2-WQO iff $(\mathcal{P}^{\downarrow}(X), \subseteq)$ is a WQO.

Phrased differently, a WQO is ω^2 if applying the powerset construction results in a WQO. This property holds for all WQOs that we have introduced so far, but not for all WQOs. Jancar [Jan99] has shown that any WQO that is not ω^2 embeds an isomorphic copy of the *Rado order* [Rad54] $(\mathbb{N}^2, \leqslant_{\text{Rado}})$, an ordering on tuples of natural numbers that is a WQO but not ω^2. We define a WSTS to be an ω^2-*WSTS* if the underlying order is an ω^2-WQO, similar for ω^2-*DWSTSes*. The class of ω^2-WQOs and the corresponding WSTSes provides a framework underlying the forward

analysis of WSTSes that was developed in [GRB06; FG09; FG12]. In our case, we will use the fact that applying the powerset construction to an ω^2-WQO results in a WQO to determinize WSTSes, similar to the well-known powerset construction for finite automata.

Let us provide some examples for ω^2-WQOs.

12.1.2 Example

a) Consider $(\Sigma, =)$, a finite alphabet ordered by equality. This order is not only a WQO, see Example 6.5.1, but also ω^2: The set $\mathcal{P}^{\downarrow}(\Sigma)$ is finite, so any quasi-order on it is a WQO. Also note that $\mathcal{P}^{\downarrow}(\Sigma)$ is equal to $\mathcal{P}(\Sigma)$.

b) Consider $(\mathbb{N}, \leqslant)$. A downward-closed subset of $\mathbb{N}$ falls in one of three cases: (1) it is empty, or (2) it is infinite, in which case it is necessarily $\mathbb{N}$ itself, or (3) it is finite, in which case it is of the shape $n{\downarrow}$ for some number $n \in \mathbb{N}$. Hence, $(\mathcal{P}^{\downarrow}(\mathbb{N}), \leqslant)$ is isomorphic to $(\mathbb{N} \cup \{\bot, \top\}, \leqslant)$, the natural numbers extended by a bottom and a top element, where each number n represents $n{\downarrow}$, $\bot$ represents $\varnothing$, and $\top$ represents the set $\mathbb{N}$. This order is a WQO, which can be shown similar to Part b) of Example 6.5.1. Hence, $(\mathbb{N}, \leqslant)$ is an ω^2-WQO.

One can also show that ω^2-WQOs are closed under taking Cartesian products and under taking the subsequence ordering on finite sequences [FG09; FG12]. In particular, the product ordering $(\mathbb{N}^k, \leqslant)$ and the subword ordering $(\Sigma^*, \preceq)$ are ω^2-WQOs.

Note that the class of ω^2-WQOs is not closed under applying the powerset construction: There is a WQO $(X, \leqslant)$ that is ω^2 such that $(\mathcal{P}^{\downarrow}(X), \subseteq)$ is not ω^2, meaning that $(\mathcal{P}^{\downarrow}(\mathcal{P}^{\downarrow}(X)), \subseteq)$ is not a WQO [Jan99]. To remedy this issue, one can consider the class of *better-quasi orderings (BQOs)* [Nas68]. Any BQO $(X, \leqslant)$ is an ω^2-WQO, and $(\mathcal{P}^{\downarrow}(X), \subseteq)$ is a BQO again. The formal definition of BQOs is beyond the scope of this thesis. Let us just mention that the orders from Example 12.1.2 are BQOs and that BQOs are also closed under taking the Cartesian product and the subsequence ordering.

Parts of the literature do not consider the downward-closed subsets ordered by inclusion, but $(\mathcal{P}^{\uparrow}(X), \supseteq)$, the upward-closed subsets ordered by reverse inclusion, or $(\mathcal{P}(X), \sqsubseteq)$, where $Y \sqsubseteq Z$ iff $Y{\downarrow} \subseteq Z{\downarrow}$. The former is clearly isomorphic to $(\mathcal{P}^{\downarrow}(X), \subseteq)$, as witnessed by the order-preserving bijective map $D \mapsto X \setminus D$. The latter becomes equal to $(\mathcal{P}^{\downarrow}(X), \subseteq)$ after identifying sets that have the same downward closure (and hence are equivalent with respect to $\sqsubseteq$).

In the following, we will have to apply the powerset construction to a WQO that may not be ω^2. To obtain a WQO, we use a restricted version of the powerset construction. Let $(X, \leqslant)$ be a quasi-order and define $\mathcal{P}^{\downarrow\text{fin}}(X) = \{Y{\downarrow} \mid Y \subseteq X, Y \text{ is finite}\}$ to be the set of downward-closed subsets of X that occur as the downward closure of finitely many elements. In contrast to $(\mathcal{P}^{\downarrow}(X), \subseteq)$, the order $(\mathcal{P}^{\downarrow\text{fin}}(X), \subseteq)$ always inherits the property of being a WQO. However, it may be missing some elements. For example, $\mathbb{N}$ is not an element of $(\mathcal{P}^{\downarrow\text{fin}}(\mathbb{N}), \subseteq)$.

12.1.3 Lemma
$(X, \leqslant)$ is a WQO iff $(\mathcal{P}^{\downarrow\text{fin}}(X), \subseteq)$ is a WQO.

Proof:
Assume that $(\mathcal{P}^{\downarrow\text{fin}}(X), \subseteq)$ is a WQO. We show that every infinite sequence $x_1, x_2, \ldots$ of elements of X contains an increasing pair. The infinite sequence $\{x_1\}\downarrow, \{x_2\}\downarrow, \ldots$ in $\mathcal{P}^{\downarrow\text{fin}}(X)$ contains an increasing pair $i < j$ with $\{x_i\}\downarrow \subseteq \{x_j\}\downarrow$ by assumption. We conclude $x_i \leqslant x_j$ as desired.

For the other direction, assume that $(X, \leqslant)$ is a WQO. Using Higman's lemma, Lemma 7.1.1, $(X^*, \leqslant^*)$ is a WQO. Now consider any infinite sequence in $\mathcal{P}^{\downarrow\text{fin}}(X)$, which we may write as $Y_1\downarrow, Y_2\downarrow, \ldots$, where each $Y_i = \{y_1^i, \ldots, y_{n_i}^i\}$ is a finite set. We represent each $Y_i\downarrow$ by the finite sequence $y_1^i \ldots y_{n_1}^i \in X^*$ and consider the infinite sequence

$$y_1^1 \ldots y_{n_1}^1,\ y_1^2 \ldots y_{n_2}^2,\ \ldots$$

in X^*. Using the fact that $(X^*, \leqslant^*)$ is a WQO, we obtain $i < j$ such that $y_1^i \ldots y_{n_i}^i \leqslant^* y_1^j \ldots y_{n_j}^j$. From the definition of $\leqslant^*$ we obtain $Y_i \subseteq Y_j$. ■

12.2 Results on WSTS expressiveness

With the notion of ω^2 introduced, we can state our results on WSTS expressiveness. In the following, we identify a type of WSTS with the class of languages of such WSTSes. For example, when we write "deterministic WSTS = finitely branching WSTS", we mean that each language of a deterministic WSTS is also the language of a finitely branching WSTS and vice versa. We do not claim that every finitely branching WSTS is deterministic.

To be able state the result, we will also need the following notation. The *reverse* of a word $a_1 \dots a_k$ is defined as expected, $\mathrm{rev}(a_1 \dots a_k) = a_k \dots a_1$. We extend the notion to languages, $\mathrm{rev}(\mathcal{L}) = \{\mathrm{rev}(w) \mid w \in \mathcal{L}\}$. Two LTSes M, M' with $\mathcal{L}(M) = \mathrm{rev}(\mathcal{L}(M'))$ are called *reversely equivalent*. We will use the symbol $\subseteq^{\mathrm{rev}}$ between languages classes to express that for every language $\mathcal{L}$ language in the class on the left-hand side of the symbol, the class on the right-hand side contains $\mathrm{rev}(\mathcal{L})$.

The following theorem summarizes our results on the relations among various classes of languages defined by WSTSes and their restrictions.

12.2.1 Theorem
The following relations hold among the WSTS language classes:

$$\begin{aligned}
\omega^2\text{-WSTS} &\subseteq \text{deterministic WSTS} = \text{finitely branching WSTS} \subseteq \text{WSTS}\,,\\
\omega^2\text{-DWSTS} &\subseteq \text{deterministic DWSTS} \subseteq \text{finitely branching DWSTS} = \text{DWSTS}\,,\\
\omega^2\text{-WSTS} &\subseteq^{\mathrm{rev}} \text{deterministic DWSTS}\,,\\
\omega^2\text{-DWSTS} &\subseteq^{\mathrm{rev}} \text{deterministic WSTS}\,.
\end{aligned}$$

In words, ω^2-(D)WSTSes determinize and reversely determinize; finitely branching WSTSes determinize too, and unrestricted DWSTSes are equivalent to finitely branching DWSTSes.

The inclusions

- deterministic WSTS ⊆ finitely branching WSTS ⊆ WSTS, and
- deterministic DWSTS ⊆ finitely branching DWSTS ⊆ DWSTS

follow directly from the definitions. We state and prove each of the other inclusions as a separate lemma:

- ω^2-WSTS ⊆ deterministic WSTS is Lemma 12.2.4,
- finitely branching WSTS ⊆ deterministic WSTS is Lemma 12.2.6,
- ω^2-DWSTS ⊆ deterministic DWSTS is Lemma 12.2.5,

- DWSTS $\subseteq$ finitely branching DWSTS is Lemma 12.2.7,
- ω^2-WSTS$\subseteq^{\text{rev}}$deterministic DWSTS is Lemma 12.2.8, and
- ω^2-DWSTS$\subseteq^{\text{rev}}$deterministic WSTS is Lemma 12.2.9.

At the end of this chapter, we will comment on whether the inclusions are strict.

Internal downward closure

Before proving the results that constitute Theorem 12.2.1, we introduce a normal form for WSTSes that will simplify some proofs. We say that a WSTS W is *internally downward closed* if the following two conditions are met. (1) The set of initial configuration is downward closed. (2) If $c \xrightarrow{a} d$ for some configurations c, d, then there is also a transition $c \xrightarrow{a} d'$ for all $d' \in d{\downarrow}$. The latter condition can be rephrased by requiring $\text{post}(a, c)$ to be downward closed.

If a WSTS $W = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ is not downward closed, we may replace it by its *internal downward closure* $W_\nabla = (\Gamma, \leqslant, T', \Gamma_{\text{init}}{\downarrow}, \Gamma_{\text{final}})$ where the transitions are defined by

$$T' = \left\{ c \xrightarrow{a} d' \;\middle|\; \exists d\colon d' \leqslant d \text{ and } c \xrightarrow{a} d \text{ in } T \right\}.$$

As the name suggest, the internal downward closure is internally downward closed. It is easy to check that the internal downward closure indeed is a WSTS. The operation preserves the language of the WSTS, i.e. we always have $\mathcal{L}(W) = \mathcal{L}(W_\nabla)$. The inclusion $\mathcal{L}(W) \subseteq \mathcal{L}(W_\nabla)$ is implied by $\Gamma_{\text{init}} \subseteq \Gamma_{\text{init}}{\downarrow}$ and $T \subseteq T'$, the other one follows from the following lemma.

12.2.2 Lemma
Let $w \in \Sigma^*$. If $c \leqslant c'$ and $c \xrightarrow{w} d$ in W_∇, then $c' \xrightarrow{w} d'$ in W for some configuration d' with $d \leqslant d'$.

Proof:
We proceed by induction on w. The base case is $w = \varepsilon$, and the statement is trivial since we have $c = d$ and may pick $d' = c'$. For the induction step, consider $w.a$ with $c \xrightarrow{w} d \xrightarrow{a} e$ in W_∇, and let $c \leqslant c'$. By induction, there is some d' with $d \leqslant d'$ such that $c' \xrightarrow{w} d'$ in W. Since we have $d \xrightarrow{a} e$ in W_∇, there is some e' with $e \leqslant e'$ such that $d \xrightarrow{a} e'$ in W. We use $d \leqslant d'$ and the upward compatibility of WSTS to obtain that there is some e'' with $e' \leqslant e''$ (and hence $e \leqslant e''$) so that $d' \xrightarrow{a} e''$. Altogether, we obtain $c' \xrightarrow{w} d' \xrightarrow{a} e''$ with $e \leqslant e''$ as desired. ■

Using the lemma, we can now prove $\mathcal{L}(W_\nabla) \subseteq \mathcal{L}(W)$. Assume that $c \xrightarrow{w} d$ is an accepting run of W_∇, which means $c \in \Gamma_{\text{init}}{\downarrow}$ and $d \in \Gamma_{\text{final}}$. By definition, there is some $c' \in \Gamma_{\text{init}}$ with $c \leqslant c'$. Applying the lemma yields a run $c' \xrightarrow{w} d'$ of W with $d \leqslant d'$. Since the final configurations are upward closed, we obtain $d' \in \Gamma_{\text{final}}$, which completes the proof.

Remark
We have used the term *internal downward closure* to express that we apply the downward closure to the internal behavior of the system, i.e. the configurations and transitions, while preserving the language. This has nothing to the with the construction introduced in Section 7.2 that takes e.g. a Petri net and yields one whose language is the downward closure.

12.2.3 Example

Consider a labeled Petri net instance $(N, M_{\text{init}}, M_{\text{final}})$ and the WSTS $W = (\mathbb{N}^P, \leqslant, T_N, M_{\text{init}}, M_{\text{final}}\uparrow)$ that corresponds to the coverability language of that net. This WSTS is not internally downward closed in general; constructing the internal downward closure results in the WSTS $W_\triangledown = (\mathbb{N}^P, \leqslant, T'_N, M_{\text{init}}\downarrow, M_{\text{final}}\uparrow)$ where

$$T'_N = \left\{ M \xrightarrow{a} M' \;\middle|\; M \boxed{t}\!\rangle\; M'' \text{ for some } t, M'' \text{ with } \lambda(t) = a \text{ and } M'' \geqslant M' \right\}.$$

In the original system, the goal was to cover M_{final}, or phrased differently, to reach M_{final} by dropping some superfluous tokens at the end of the computation. In $W_\triangledown$, tokens can also be dropped at the beginning of the computation (by starting from a marking in $M_{\text{init}}\downarrow$ that is not M_{init}) and after each transition. The monotonicity of the transition relation of Petri nets ensures that any computation that drops tokens early can be simulated by one in which all tokens are dropped at the end.

The internal downward closure preserves the underlying order of the original WSTS, and hence also the property of being ω^2. In general, it does not preserve the properties of being deterministic or finitely branching.

We can define a similar notion for downward-compatible WSTSes. We call a DWSTS *internally upward closed* if its set of initial configurations and $\text{post}(a, c)$ are upward closed for all symbols a and all configurations c. For a given DWSTS M, we may enforce these properties by constructing its *internal upward closure* $M_\triangle$, a DWSTS that has $\Gamma_{\text{init}}\uparrow$ as its set of initial configurations and in which the a-successors of a configuration c are $\text{post}_M(a, c)\uparrow$. The languages of the two DWSTS coincide, $\mathcal{L}(M) = \mathcal{L}(M_\triangle)$. The proof from the case of WSTS carries over: To see that this is true, note that the proof of Lemma 12.2.2 has not used the WQO property, so a dualized version holds for DWSTSes.

Proof of Theorem 12.2.1

We now have all preliminaries at hand to state and show a sequence of lemmas that prove the non-trivial inclusions stated in Theorem 12.2.1.

We start with the inclusion of the class of languages of ω^2-WSTSes and ω^2-DWSTSes in the class of languages of deterministic WSTSes and DWSTSes, respectively. In both cases, we use a pow-

erset construction to determinize the given WSTS. The given (D)WSTS being ω^2 ensures that the result is well-quasi-ordered.

12.2.4 Lemma
Every ω^2-WSTS is language equivalent to a deterministic WSTS.

Proof:
Let $W = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ be a WSTS. We may assume that W is internally downward-closed, since taking the internal downward closure does not change the order and hence preserves the property of being ω^2.

We construct a deterministic WSTS $W' = (\mathcal{P}^{\downarrow}(\Gamma), \subseteq, T', \{\Gamma_{\text{init}}\}, \Gamma'_{\text{final}})$ by what essentially is a powerset construction. Note that $(\mathcal{P}^{\downarrow}(\Gamma), \subseteq)$ is a WQO since we assumed $(\Gamma, \leqslant)$ to be ω^2. The final configurations in the new WSTS are sets of configurations of the original WSTS that contain a final configuration,

$$\Gamma'_{\text{final}} = \left\{ C \in \mathcal{P}^{\downarrow}(\Gamma) \;\middle|\; C \cap \Gamma_{\text{final}} \neq \varnothing \right\},$$

and the deterministic transition relation is defined by the post operation,

$$C \xrightarrow{a} \text{post}_W(a, C)\,.$$

Note that W' is well-defined, i.e. $\{\Gamma_{\text{init}}\} \in \mathcal{P}^{\downarrow}(\Gamma)$ and $\text{post}_{W'}(a, C) \in \mathcal{P}^{\downarrow}(\Gamma)$ for all $C \in \mathcal{P}^{\downarrow}(\Gamma)$, because we assumed W to be internally downward closed.

WSTS W' is deterministic by definition. To see that it indeed accepts the same language, one can prove that for each word $w \in \Sigma^*$, we have $\text{reach}_{W'}(w) = \{\text{reach}_W(w)\}$. The base case as well as the inductive step follow directly from the definition of W'. ■

12.2.5 Lemma
Every ω^2-DWSTS is language equivalent to a deterministic DWSTS.

Proof:
The proof is very similar to the proof of Lemma 12.2.4. Let $M = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ be the given DWSTS, and assume that it is internally upward closed. Otherwise, replace it by its internal upward closure. We construct $M' = (\mathcal{P}^{\uparrow}(\Gamma), \supseteq, T', \{\Gamma_{\text{init}}\}, \Gamma'_{\text{final}})$ with $C \xrightarrow{a} \text{post}_W(a, C)$ for all $C \in \mathcal{P}^{\uparrow}(\Gamma)$, and $C \in \Gamma'_{\text{final}}$ if $C \cap \Gamma_{\text{final}} \neq \varnothing$. To see that $(\mathcal{P}^{\uparrow}(\Gamma), \supseteq)$ is a WQO (and hence M' is a DWSTS), note that $(\mathcal{P}^{\uparrow}(\Gamma), \subseteq)$ and $(\mathcal{P}^{\downarrow}(\Gamma), \subseteq)$ are isomorphic, as witnessed by the order-preserving bijective function defined by $C \mapsto \Gamma \setminus C$. Hence, $(\mathcal{P}^{\uparrow}(\Gamma), \supseteq)$ is a WQO since we assume $(\Gamma, \leqslant)$ to be ω^2. To see that the language is preserved, $\mathcal{L}(M) = \mathcal{L}(M')$, one shows $\text{reach}_{M'}(w) = \{\text{reach}_M(w)\}$ by induction. ■

We continue with the inclusion of the class of languages of finitely branching WSTS in the class of deterministic WSTS, which proves that the two classes are equal.

12.2.6 Lemma
Every finitely branching WSTS is language equivalent to a deterministic WSTS.

In principle, the proof is a version of the proof of Lemma 12.2.4 that uses the finitely represented downward closed subsets $(\mathcal{P}^{\downarrow\text{fin}}(\Gamma), \subseteq)$ instead of $(\mathcal{P}^{\downarrow}(\Gamma), \subseteq)$. The assumption that the given WSTS is finitely branching guarantees that considering these subsets is sufficient.

However, there is a technical difficulty that we have to overcome. In the proof of Lemma 12.2.4, we have assumed the given WSTS to be internally downward closed, a property that can easily be enforced by applying the internal downward closure. However, the internal downward closure of a finitely branching WSTS may not be finitely branching. Hence, we will have to essentially incorporate a proof similar to the one of Lemma 12.2.2 here.

Proof:
Let $W = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ be the given finitely branching WSTS. We construct a deterministic WSTS $W' = (\mathcal{P}^{\downarrow\text{fin}}(\Gamma), \subseteq, T', \{\Gamma_{\text{init}}\downarrow\}, \Gamma'_{\text{final}})$ that has the finitely represented downward-closed subsets of Γ as configurations. Since Γ_{init} is finite by assumption, $\Gamma_{\text{init}}\downarrow$ is such a set. A set $C \in \mathcal{P}^{\downarrow\text{fin}}(\Gamma)$ is in Γ'_{final} if it contains a final configuration of the original WSTS, $C \cap \Gamma_{\text{final}} \neq \varnothing$. The transition relation is defined by applying the post-operation and then taking the downward closure,

$$C \xrightarrow{a} \text{post}_W(a, C)\downarrow \ .$$

If $C = \{c_1, \ldots, c_k\}\downarrow$, and for each c_i, we have that $c_{i1}, \ldots, c_{im_i}$ are the finitely many configurations such that $c_i \xrightarrow{a} c_{ij}$ in W, then we have $\text{post}_W(a, C)\downarrow = \{c_{11}, \ldots, c_{1m_1}, \ldots, c_{k1}, \ldots c_{km_k}\}\downarrow$ using that W is finitely branching and upward-compatible. Hence, W' is a well-defined transition system. Checking upward-compatibility and that W' is deterministic is straightforward. Since $(\mathcal{P}^{\downarrow\text{fin}}(\Gamma), \subseteq)$ is always a WQO if $(\Gamma, \subseteq)$ is, see Lemma 12.1.3, we conclude that W' is a WSTS.

It remains to show that W and W' are language equivalent. We first show the following claim by induction: For each $w \in \Sigma^*$, we have

$$\text{reach}_{W'}(w) = \left\{\text{reach}_{W_\nabla}(w)\right\},$$

where W_∇ is the internal downward closure of W.

The base case is by definition, since $\mathrm{reach}_{W'}(\varepsilon) = \{\Gamma_{\mathrm{init}}{\downarrow}\} = \mathrm{reach}_{W_\nabla}(\varepsilon)$. For the inductive step, consider a word $w.a$. We have

$$\begin{aligned}\mathrm{reach}_{W'}(w.a) &= \mathrm{post}_{W'}(a, \mathrm{reach}_{W'}(w)) \\ &= \mathrm{post}_{W'}\big(a, \{\mathrm{reach}_{W_\nabla}(w)\}\big) \\ &= \big\{\mathrm{post}_W\big(a, \mathrm{reach}_{W_\nabla}(w)\big){\downarrow}\big\}\,,\end{aligned}$$

where we use the definition of reach, the induction hypothesis, and the definition of the transition relation of W'. To complete the proof of the claim, we have to show

$$\mathrm{post}_W\big(a, \mathrm{reach}_{W_\nabla}(w)\big){\downarrow} = \mathrm{reach}_{W_\nabla}(w.a)\,.$$

Assume that $d \in \mathrm{post}_W(a, \mathrm{reach}_{W_\nabla}(w))\downarrow$, i.e. $d \leqslant d'$ for some $d \in \mathrm{post}_W(a, \mathrm{reach}_{W_\nabla}(w))$. This in turn means that there is some $c' \in \mathrm{reach}_{W_\nabla}(w)$ with $c' \xrightarrow{a} d'$ in W. Hence, we have $c \xrightarrow{a} d$ in W_∇ by the definition of being internally downward closed. We conclude $d \in \mathrm{post}_{W_\nabla}(a, \mathrm{reach}_{W_\nabla}(w)) = \mathrm{reach}_{W_\nabla}(w.a)$ as required.

For the other direction, let $d \in \mathrm{reach}_{W_\nabla}(w.a)$, i.e. there is some $c \in \mathrm{reach}_{W_\nabla}(w)$ with $c \xrightarrow{a} d$ in W_∇. By the definition of W_∇, there is some d' with $d \leqslant d'$ such that $c \xrightarrow{a} d'$ in W. We conclude $c \in \mathrm{post}_W(a, \mathrm{reach}_{W_\nabla}(w))$, which implies the desired statement (since the downward closure of any set contains that set).

With the claim established, we get that $\mathcal{L}(W') = \mathcal{L}(W_\nabla)$. The latter language is equal to $\mathcal{L}(W)$. ■

Let us now show that every DWSTS language is also the language of a finitely branching DWSTS. The idea is to represent the set of configurations that can be reached along a certain word in the original DWSTS by the finitely many minimal configurations from that set. Since it is not our goal to determinize, we do not need to introduce a powerset construction.

One might wonder why we do not use the finitely represented upward-closed sets, similar to $\mathcal{P}^{\downarrow\mathrm{fin}}(\Gamma)$ in Lemma 12.2.6. By Property (3) from Lemma 6.5.3, every upward-closed set can be finitely represented. However, the upward-closed sets are not well-quasi-ordered by reverse inclusion (which is the order that we would need to consider) unless we require the underlying order to be ω^2.

12.2.7 Lemma
Every DWSTS is language equivalent to a finitely branching DWSTS.

Proof:
Let $M = (\Gamma, \leqslant, T, \Gamma_{\mathrm{init}}, \Gamma_{\mathrm{final}})$ be a given DWSTS. We assume wlog. that it is internally upward closed. We construct $M' = (\Gamma, \leqslant, T', \min \Gamma_{\mathrm{init}}, \Gamma_{\mathrm{final}})$ where the order, the set of configurations, and the set

of final configurations are unchanged. The initial configurations of the new DWSTS are a set of minimal initial configurations of the old one. The transition relation T' is defined by

$$\mathrm{post}_{M'}(a, c) = \min \mathrm{post}_M(a, c) .$$

Indeed, M' is finitely branching since $\min C$ is finite for any upward-closed set $C{\uparrow} \subseteq \Gamma$ of configurations, Property (3) of Lemma 6.5.3, and Γ_{init} and $\mathrm{post}_M(a, c)$ are upward closed by the assumption that M is internally upward closed.

To show $\mathcal{L}(M) = \mathcal{L}(M')$, we first prove the following for all words $w \in \Sigma^*$ using induction:

$$\mathrm{reach}_{M'}(w){\uparrow} = \mathrm{reach}_M(w) .$$

In the base case, we have $\mathrm{reach}_{M'}(\varepsilon){\uparrow} = (\min \Gamma_{\mathrm{init}}){\uparrow} = \Gamma_{\mathrm{init}} = \mathrm{reach}_M(\varepsilon)$. The inductive step is

$$\begin{aligned} \mathrm{reach}_{M'}(w.a){\uparrow} &= \mathrm{post}_{M'}(a, \mathrm{reach}_{M'}(w)){\uparrow} \\ &= \min \mathrm{post}_M(a, \mathrm{reach}_{M'}(w)){\uparrow} \\ &= \min \mathrm{post}_M(a, \mathrm{reach}_M(w){\uparrow}){\uparrow} , \end{aligned}$$

using the definition of reach, the definition of the transition relation of M', and the induction hypothesis.

We observe that in an internally upward-closed DWSTS, we have $\mathrm{post}(a, Y) = \mathrm{post}(a, Y{\uparrow})$ for any set Y. One direction is immediate; to see $\mathrm{post}(a, Y{\uparrow}) \subseteq \mathrm{post}(a, Y)$, we first deduce $\mathrm{post}(a, Y{\uparrow}) \subseteq \mathrm{post}(a, Y){\uparrow}$ using downward compatibility, and then use that $\mathrm{post}(a, Y){\uparrow} = \mathrm{post}(a, Y)$ since the DWSTS is assumed to be internally upward closed.

Applying this equality, we obtain

$$\begin{aligned} \mathrm{reach}_{M'}(w.a){\uparrow} &= \min \mathrm{post}_M(a, \mathrm{reach}_M(w){\uparrow}){\uparrow} \\ &= \min \mathrm{post}_M(a, \mathrm{reach}_M(w)){\uparrow} \\ &= \min \mathrm{reach}_M(w.a){\uparrow} = \mathrm{reach}_M(w.a) \end{aligned}$$

as desired.

Finally, we conclude $\mathcal{L}(M) = \mathcal{L}(M')$: A word w is in $\mathcal{L}(M')$ if and only if $\mathrm{reach}_{M'}(w)$ contains a final configuration. Because the set of final configurations of a DWSTS is downward closed, this is the case iff $\mathrm{reach}_{M'}(w){\uparrow} = \mathrm{reach}_M(w)$ contains a final configuration. The latter condition is equivalent to $w \in \mathcal{L}(M)$. ■

It remains to prove that ω^2-WSTSes can be reversely determinized to DWSTSes and vice versa. The construction simulates the system in the reverse direction to convert from WSTSes to DWSTSes and combines this with a powerset construction to determinize.

12.2.8 Lemma
Every ω^2-WSTS is reversely equivalent to a deterministic DWSTS.

Proof:
Let $W = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ be the given ω^2-WSTS. We construct a DWSTS that simulates W reversely on upward-closed subsets. We define $M = (\mathcal{P}^{\uparrow}(\Gamma), \supseteq, T', \{\Gamma_{\text{final}}\}, \Gamma'_{\text{final}})$, where the configurations of M are upward-closed sets of configurations of W, ordered by reverse inclusion. The unique initial configuration is the set of final configurations of W. A configuration of M is final if it contains an initial configuration of W: $C \in \mathcal{P}^{\uparrow}(\Gamma)$ is in Γ'_{final} if $C \cap \Gamma_{\text{init}} \neq \varnothing$. It remains to define the (deterministic) transition relation T': We have

$$C \xrightarrow{a} \text{pre}_W(a, C) .$$

Note that if C is final, then so is any larger set C' with $C' \subseteq C$. By upward compatibility, $\text{pre}_W(a, C)$ is necessarily upward closed if C is upward closed. We verify that M' satisfies downward compatibility: If C' is larger than C, meaning $C' \subseteq C$, then we have $\text{pre}_W(a, C') \subseteq \text{pre}_W(a, C)$ as required. The order $(\mathcal{P}^{\uparrow}(\Gamma), \supseteq)$ is a WQO since it is isomorphic to $(\mathcal{P}^{\downarrow}(\Gamma), \subseteq)$ and we assume that $(\Gamma, \leqslant)$ is ω^2, as in the proof of Lemma 12.2.5. Hence, M is indeed a well-defined DWSTS.

To conclude the proof, we need to show that $\mathcal{L}(W) = \text{rev}(\mathcal{L}(M))$. We prove that for each $w \in \Sigma^*$, the following holds:

$$\text{reach}_M(w) = \left\{\text{reach}_W^{-1}(\text{rev}(w))\right\} .$$

We proceed by induction on w. In the base case, we have

$$\text{reach}_M(\varepsilon) = \{\Gamma_{\text{final}}\} = \left\{\text{reach}_W^{-1}(\varepsilon)\right\} = \left\{\text{reach}_W^{-1}(\text{rev}(\varepsilon))\right\} .$$

For the induction step, consider $w.a$. We have

$$\begin{aligned}
\text{reach}_M(w.a) &= \text{post}_M(a, \text{reach}_M(w)) \\
&= \text{post}_M\left(a, \left\{\text{reach}_W^{-1}(\text{rev}(w))\right\}\right) \\
&= \text{pre}_W\left(a, \left\{\text{reach}_W^{-1}(\text{rev}(w))\right\}\right) \\
&= \left\{\text{reach}_W^{-1}(a.\text{rev}(w))\right\} \\
&= \left\{\text{reach}_W^{-1}(\text{rev}(w.a))\right\}
\end{aligned}$$

by using induction, the definition of T', and the fact that $\text{rev}(w.a) = a.\text{rev}(w)$.

Using the proven equality, we can establish the following sequence of equivalences for any word $w \in \Sigma^*$:

$$\begin{aligned} w \in \mathcal{L}(M) &\text{ iff } \mathrm{reach}_M(w) \in \Gamma'_{\mathrm{final}} \\ &\text{iff } \mathrm{reach}_W^{-1}(\mathrm{rev}(w)) \in \Gamma'_{\mathrm{final}} \\ &\text{iff } \mathrm{reach}_W^{-1}(\mathrm{rev}(w)) \cap \Gamma_{\mathrm{init}} \neq \varnothing \\ &\text{iff } \mathrm{rev}(w) \in \mathcal{L}(W)\,. \end{aligned}$$

■

12.2.9 Lemma
Every ω^2-DWSTS is reversely equivalent to a deterministic WSTS.

Proof:
The proof is the dual of the proof of Lemma 12.2.8. Given the DWSTS $M = (\Gamma, \leqslant, T, \Gamma_{\mathrm{init}}, \Gamma_{\mathrm{final}})$, we construct the WSTS $W = (\mathcal{P}^{\downarrow}(\Gamma), \subseteq, T', \{\Gamma_{\mathrm{final}}\}, \Gamma'_{\mathrm{final}})$ with $C \in \Gamma'_{\mathrm{final}}$ if $C \cap \Gamma_{\mathrm{init}} \neq \varnothing$ and T' is defined by $C \xrightarrow{a} \mathrm{pre}_W(a, C)$.

The proofs that show that W is a well-defined WSTS and that $\mathcal{L}(W) = \mathrm{rev}(\mathcal{L}(M))$ are as in Lemma 12.2.8. ■

With all lemmas stated and proven, the proof of Theorem 12.2.1 is completed.

The question of whether the inclusions among language classes are strict is left open. In particular, we do not know whether there is a WSTS language that cannot be generated by a finitely branching (or, equivalently, deterministic) WSTS. We know that if such a language exists, it can only be generated by infinitely branching WSTSes whose underlying order is not ω^2. We were unable to construct counterexamples to show that the inclusions are strict, as well as unable to extend the constructions to show that some of the inclusions are in fact equalities. It is imaginable that the differences e.g. between ω^2-WQOs and non-ω^2-WQOs cannot be exhibited using languages of WSTSes consisting of finite words, and hence the corresponding classes are equal.

However, these open problems are largely of theoretical nature. It has been observed that almost all WQOs that occur in practice are BQOs, which implies that they are ω^2. In fact, the paper [Fin16] states that finite graphs ordered by the graph minor relation are the only WQO of practical interest that is not known to be a BQO. This means that for all (D)WSTSes that occur in practice, we can apply the results on ω^2-(D)WSTSes from Theorem 12.2.1 to determinize or reversely determinize these systems.

13 Regular separability for WSTSes

Contents

13.1 The results and their consequences 235
13.2 Technical core 239
13.3 Ideals and the ideal completion 247

The goal of this chapter is proving the results that we have outlined in Section 11.3. We will start by formally stating the results and discussing their consequences. Then, we will prove a technical core theorem from which we can directly conclude our result on the separability of languages of DWSTSes. Finally, we will introduce ideals and the ideal completion of a WSTS, concepts that will allow us to also apply our core result to upward-compatible WSTSes.

13.1 The results and their consequences

The main result regarding upward-compatible WSTSes that we will prove in this chapter is the following.

13.1.1 Theorem
Two disjoint WSTS languages, at least one of them the language of a deterministic WSTS, are regularly separable.

In Section 11.3, we have promised proving that under mild assumptions, any two disjoint WSTS languages are regularly separable. Requiring one of the two WSTSes to be deterministic seems to be an assumption that is quite restricting. However, we may use the expressiveness result from Chapter 12 to see that it is in fact enough to require one of the two WSTS to be finitely branching or ω^2. In particular, recall that virtually all WSTSes that are of practical interest satisfy the ω^2-condition. Hence, the following generalization of the result applies to almost all WSTSes.

13.1.2 Corollary
Two disjoint WSTS languages, at least one of them the language of a finitely branching WSTS or of an ω^2-WSTS, are regularly separable.

Once we have proven Theorem 13.1.1, the corollary follows immediately using Theorem 12.2.1.

One should emphasize that we simply require both languages to be WSTS languages. We do not require them to come from the same class of WSTSes. For example, one could be the language of a Petri net, the other could be the language of a lossy channel system.

It is also noteworthy that our result means that regular separability as a decision problem is essentially equivalent to intersection-emptiness. Luckily, intersection-emptiness can be decided under mild-assumption. To be precise, it has been shown that the coverability problem for unlabeled WSTSes, the problem of checking whether a final configuration is reachable from an initial one, is decidable under mild assumptions [AJ93; GRB06]. In the world of labeled WSTSes, this corresponds to checking language-emptiness. Checking intersection-emptiness amounts to checking language-emptiness for the product system. Luckily, if both given WSTSes satisfy the requirements for the decidability of coverability, then so does the product system, meaning we can decide intersection-emptiness.

Our result is constructive for most WSTSes. We will construct a finite automaton whose language is the required regular separator from the so-called ideal decomposition of an invariant. Such a set is effectively computable in many cases, as demonstrated e.g. in [Fin16; LS15a].

Furthermore, our result has some interesting language-theoretic consequences. In Section 11.2, we have mentioned that separability is related to the regularity problem. The following corollary to our result makes this relationship precise.

13.1.3 Corollary
If a language is a deterministic WSTS language and its complement is a WSTS language (or vice versa), then both are necessarily regular.

Proof:
Assume that $\mathcal{L}(W)$ is the language of a finitely branching WSTS W, and its complement $\overline{\mathcal{L}(W)} = \mathcal{L}(W')$ is the language of a WSTS. These languages are disjoint, so by Theorem 13.1.1, there is a regular separator $\mathcal{R}$. From $\mathcal{L}(W) \subseteq \mathcal{R}$ and $\overline{\mathcal{L}(W)} \cap \mathcal{R} = \varnothing$, we conclude $\mathcal{R} = \mathcal{L}(W)$. Hence, $\mathcal{L}(W)$ is regular and so is its complement. ■

Again, we may apply Theorem 12.2.1 to obtain the following generalization.

13.1.4 Corollary
If a language is the language of a WSTS that is finitely branching or ω^2, and its complement is a WSTS language, then both are necessarily regular.

It has been shown before that Petri net coverability languages have this property [MKRS98b; MKRS98a]. A more pointed phrasing of our result is the following weaker statement.

13.1.5 Corollary
No subclass of the finitely branching or ω^2-WSTS languages beyond REG is closed under complement.

Proof:
Towards a contradiction, assume $\mathcal{F}$ is such a class. Take any non-regular language $\mathcal{L}$ from $\mathcal{F}$. By assumption, $\overline{\mathcal{L}}$ is also in $\mathcal{F}$. Both are finitely branching (or ω^2) WSTS languages, so Corollary 13.1.4 yields that $\mathcal{L}$ is regular, a contradiction. ■

Results for DWSTS

We will show similar results for downward-compatible WSTSes.

13.1.6 Theorem
Two disjoint DWSTS languages, at least one of them the language of a deterministic DWSTS, are regularly separable.

With Theorem 12.2.1, we can generalize the result and require one of the DWSTSes to be ω^2 instead of requiring it to be deterministic. Unlike in the case of upward-compatible WSTSes, it does not seem to be sufficient to require one of the two DWSTSes to be finitely branching.

13.1.7 Corollary
Two disjoint DWSTS languages, at least one of them the language of a deterministic DWSTS or an ω^2-DWSTS, are regularly separable.

We also obtain corollaries that are similar to the Corollaries 13.1.3, 13.1.4 and 13.1.5. The proofs of these corollaries are as in the case of (upward-compatible) WSTSes.

13.1.8 Corollary

a) If a language is the language of a DWSTS that is deterministic or ω^2 and its complement is a DWSTS language (or vice versa), then both are necessarily regular.

b) No subclass of the deterministic or ω^2-DWSTS languages beyond REG is closed under complement.

We leave the question of whether the assumptions that we make are necessary unanswered. It would be desirable to prove that any two disjoint WSTS languages are regularly separable, without requiring one of the two generating WSTSes to be ω^2 or finitely branching. This problem is closely related to the unknown strictness of the inclusions in our result on WSTS expressiveness, Theorem 12.2.1, on which we have commented at the end of Section 12.2. If one could extend

Theorem 12.2.1, e.g. by showing that any WSTS language is the language of a finitely branching WSTS, this also immediately extends our result on regular separability. If it turns out that the inclusions among WSTS language classes are strict, whether the results on regular separability can be extended remains an open problem. Although, one would hope that the proof of strictness allows us to gain new insights. Let us emphasize that virtually all WSTSes of practical interest are known to be ω^2, meaning that our results can be applied.

13.2 Technical core

Both main results, Theorem 13.1.1 and Theorem 13.1.6, will follow from a technical core result. We will prove this technical core in this section and then immediately obtain the regular separability of language-disjoint DWSTSes. The result in the case of upward-compatible WSTSes will require more work. To state the technical core, we need the notion of an inductive invariant that we will introduce in the following.

An *invariant* is a property that holds for all reachable configurations. We may see it the set of configurations that satisfy the property. We will take this view in the rest of the section and see an invariant as a set of configurations containing the reachable ones. In particular, the set of reachable configurations itself is an invariant.

We will be only interested in invariants that are *safe*. Normally, the notion of safety means that a certain set of *bad* configurations cannot be reached. Here, we will see the set of final configurations of an LTS as the set of configurations that should not be reachable: A safe invariant is a set of configurations that contains all reachable configurations, but no final configuration. Hence, a safe invariant is a certificate for language-emptiness. If the language of an LTS is empty, then the set of reachable configurations itself is a safe invariant.

In the following, whenever we use the notion *invariant*, we imply that it is safe. The above discussion yields the following characterization: For an LTS $W = (\Gamma, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ with empty language, the invariants are precisely the sets X so that

$$\text{reach}_W \subseteq X \subseteq \Gamma \setminus \Gamma_{\text{final}} .$$

In particular, reach_W and the complement of Γ_{final} are invariants.

When checking whether a given set X (representing some property) is an invariant, it is useful to impose a restriction that makes this task easier. We call an invariant *inductive* if for any configuration in the invariant, all its successors are also in the invariant: If $c \in X$, then $\text{post}(\Sigma, c) \subseteq X$. This property simplifies the check because it can now be conducted in a local fashion: One checks that if a configuration satisfies the property defining the invariant, then the property still holds after doing a step in the transition system. Invariants, and inductive invariants in particular, are a standard technique used for safety verification of programs [MP95].

Checking whether a set X is an inductive invariant amounts to checking (1) that X contains the initial configurations, (2) that if $c \in X$, then $\text{post}(\Sigma, c) \subseteq X$, and that (3) $X \cap \Gamma_{\text{final}} = \varnothing$. Properties (1) and (2) together imply that X contains the reachable configurations.

While reach_W is always an inductive invariant if the given LTS has empty language, the same is not true for $\Gamma \setminus \Gamma_{\text{final}}$. It may contain the predecessor of a final configuration, which violates inductivity. We exclude all ancestors of final configurations and obtain that $\Gamma \setminus \text{reach}_W^{-1}$ is the

greatest inductive invariant. The inductive invariants of an LTS with empty language are exactly the inductive sets X with

$$\mathrm{reach}_W \subseteq X \subseteq \Gamma \setminus \mathrm{reach}_W^{-1} .$$

In the case of ordered LTS, we can additionally require X to be downward closed. In fact, if X is an inductive invariant, then so is its downward closure $X{\downarrow}$. Condition (1) is maintained when adding configurations, Condition (3) is maintained since we assume the set of final configurations to be upward closed. For inductivity, assume that $c \in X{\downarrow}$, i.e. $c \leqslant c'$ for some $c' \in X$. Using upward compatibility, we have that for any $c \xrightarrow{a} d$, there is some $c' \xrightarrow{a} d'$ with $d \leqslant d'$. We have $d' \in \mathrm{post}(\Sigma, c') \subseteq X \subseteq X{\downarrow}$ by assumption and conclude $d \in X{\downarrow}$ since $X{\downarrow}$ is downward closed.

The above discussion justifies the following definition of inductive invariants in the case of ordered LTS.

13.2.1 Definition
An *inductive invariant* for an ordered LTS $W = (\Gamma, \leqslant, T, \Gamma_{\mathrm{init}}, \Gamma_{\mathrm{final}})$ is a downward-closed set $X \subseteq \Gamma$ with (1) $\Gamma_{\mathrm{init}} \subseteq X$, (2) Safety: $X \cap \Gamma_{\mathrm{final}} = \varnothing$, and (3) Inductivity: $\mathrm{post}_W(\Sigma, X) \subseteq X$.

An inductive invariant exists if and only if $\mathcal{L}(W)$ is empty. If $\mathcal{L}(W)$ is empty, then the downward-closure of the reachable configurations $\mathrm{reach}_W{\downarrow}$ is the least, and the configurations that are not the predecessor of a final configuration $\Gamma \setminus \mathrm{reach}_W^{-1}$ is the greatest inductive invariant. Note that the latter set is always downward closed by upward compatibility, while reach_W may not be downward closed if W is not internally downward closed.

In the following, we want to transform an inductive invariant for the product of two language-disjoint WSTSes into a regular separator for their languages. This will only work if the inductive invariant has a certain finite representation.

13.2.2 Definition
An inductive invariant X is *finitely represented* if $X = Q{\downarrow}$ for a finite set Q of configurations.

We have explained in Chapter 12 that requiring a downward-closed set to be the downward closure of finitely many elements imposes a restriction. Consider for example a Petri net with two places p_1, p_2. The initial marking assigns no tokens, the final marking requires a token on the second place. This means that the set of final configuration is $\mathbb{N} \times (\mathbb{N} \setminus \{0\})$. There is a single transition that requires no tokens and produces a token on the first place. Obviously, the set of configurations reachable in the associated WSTS is $\mathbb{N} \times \{0\}$. To be precise, $\mathbb{N} \times \{0\}$ is a downward-closed safe inductive invariant, and it is the only safe invariant. Any smaller set does not contain all reachable configurations, any bigger set contains a final configuration. However,

this invariant is not finitely represented as there is no finite set of numbers whose downward closure is $\mathbb{N}$.

Not every inductive invariant being finitely represented is a problem that we will need to overcome later. For now, let us use the notion to state the core result.

13.2.3 Theorem
Let W and W' be language-disjoint ordered LTSes, one of them deterministic, such that their product $W \times W'$ admits a finitely represented inductive invariant $Q{\downarrow}$. Then W and W' are regularly separable by the language of a finite automaton with Q as its set of states.

One might wonder why the result talks about ordered LTSes instead of WSTSes. The reason is that when we apply the result, we will first enforce the existence of a finitely represented invariant by applying a preprocessing step. This step will turn the given (D)WSTSes into language-equivalent ordered LTSes that may not be (D)WSTSes in general.

We turn to the proof of Theorem 13.2.3. Let $W = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ and $W' = (\Gamma', \leqslant', T', \Gamma'_{\text{init}}, \Gamma'_{\text{final}})$ be the given WSTSes. We assume that their languages are disjoint, $\mathcal{L}(W) \cap \mathcal{L}(W') = \varnothing$, and that W' is deterministic. We consider their product

$$W_\times = W \times W' = (\Gamma_\times, T_\times, \leqslant_\times, \Gamma_{\times\text{init}}, \Gamma_{\times\text{final}}) \,,$$

and note that $\mathcal{L}(W_\times) = \varnothing$ by assumption. Let $Q \subseteq \Gamma_\times$ be a finite set such that $Q{\downarrow}$ is an inductive invariant for $W_\times$.

We construct an automaton A that has Q as its set of states. Its language will be a regular separator for the languages of the two given WSTSes. To be precise, it will contain $\mathcal{L}(W)$ and be disjoint from $\mathcal{L}(W')$. Before elaborating on its properties, we explain the construction. Intuitively, A overapproximates the product WSTS $W_\times$ using the configurations from Q. Note that the configurations of $W_\times$ (and hence the states of A) are tuples of configurations of the original WSTSes.

13.2.4 Definition
The *separating automaton induced by* Q is $A = (Q, \to, Q_{\text{init}}, Q_{\text{final}})$. A state is initial if it is larger than some initial configuration of $W_\times$,

$$Q_{\text{init}} = \left\{(q, q') \in Q \;\middle|\; (c, c') \leqslant_\times (q, q') \text{ for some } (c, c') \in \Gamma_{\times\text{init}}\right\} .$$

A state is final if its W-component is final,

$$Q_{\text{final}} = \left\{(q, q') \in Q \;\middle|\; q \in \Gamma_{\text{final}}\right\} .$$

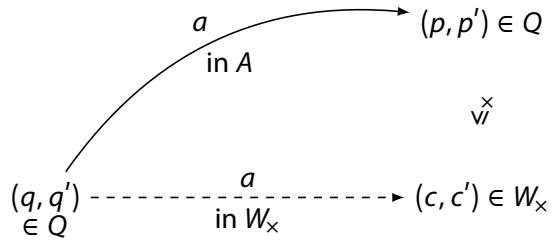


Figure 13.2.a: The transition relation of the separating automaton A.

The transition relation of A overapproximates the transition relation of $W_\times$ as follows:

$$(q, q') \xrightarrow{a} (p, p') \text{ in } A \quad \text{iff} \quad (q, q') \xrightarrow{a} (c, c') \text{ in } W_\times \text{ for some } (c, c') \leqslant_\times (p, p') .$$

The latter part of the definition means that if $(q, q') \xrightarrow{a} (c, c')$ is a transition of the product and $(q, q') \in Q$, then A can overapproximate this transition using any state $(p, p') \in Q$ with $(c, c') \leqslant_\times (p, p')$. This is depicted in Figure 13.2.a. We do not impose any precision requirement, but we could obviously require Q to be an antichain. The fact that $Q{\downarrow}$ is an inductive invariant will be sufficient to prove the correctness of the construction.

Remark
Note that we are dealing with an NFA with multiple initial states here. We have already explained in Section 4.3 that such an NFA can be transformed into an NFA with a unique initial state while introducing a single new state.

To prove that A is indeed a separating automaton, we need to show $\mathcal{L}(W) \subseteq \mathcal{L}(A)$ and $\mathcal{L}(W') \cap \mathcal{L}(A) = \varnothing$. We will prove the following:

- Any computation of W can be embedded into a computation of $W_\times$, and this computation in turn is overapproximated by a run of A.
- Any run of A on some word w, projected to the W'-component, overapproximates the unique computation of W' for w.

From the first property, we get the inclusion $\mathcal{L}(W) \subseteq \mathcal{L}(A)$. Here, it is sufficient to assume that W' is complete to obtain that any computation of W can be embedded into a computation of the product system. From the second property, we get the disjointness $\mathcal{L}(W') \cap \mathcal{L}(A) = \varnothing$. For this statement to hold, it is important that W' is deterministic. Otherwise, the run of A could overapproximate a non-accepting computation of W' although an accepting one exists.

We proceed to formally state and prove the two properties, and conclude that $\mathcal{L}(A)$ is a regular separator, which completes the proof of Theorem 13.2.3. We start with a technical lemma that will be needed to establish $\mathcal{L}(W) \subseteq \mathcal{L}(A)$.

13.2.5 Lemma

a) If $c \in \text{reach}_W(w)$, then $(c, c') \in \text{reach}_{W_\times}(w)$ for some c'.

b) If $(c, c') \in \text{reach}_{W_\times}(w)$, then $(q, q') \in \text{reach}_A(w)$ for some $(q, q') \in Q$ with $(c, c') \leqslant_\times (q, q')$.

Proof:
For Part a), we have that c' is in fact predetermined to be the unique configuration $\text{reach}_{W'}(w)$ since W' is deterministic. Formally, we prove the statement by induction on w. In the base case, we have that $c \in \Gamma_{\text{init}} = \text{reach}_W(\varepsilon)$ is an initial configuration of W. We choose $c' = \text{reach}_{W'}(\varepsilon)$ as the unique initial configuration of W'. By the definition of the synchronized product, we have $(c, c') \in \Gamma_{\times\text{init}} = \text{reach}_{W_\times}(\varepsilon)$ as desired. For the inductive step, consider $w.a$. A configuration $c \in \text{reach}_W(w.a)$ is an a-successor of some configuration $d \in \text{reach}_W(w)$. By induction, we have $(d, d') \in \text{reach}_{W_\times}(w)$. Since W' is deterministic, there is a unique configuration $c' = \text{reach}_{W'}(w.a)$ that is the a-successor of d'. By the definition of the transition relation of the synchronized product, we have $(d, d') \xrightarrow{a} (c, c')$ in $W_\times$, and hence $(d, d') \in \text{reach}_{W_\times}(w.a)$ as desired.

For Part b), we use that an inductive invariant has to contain all reachable configurations of the system. Formally, we show the statement by induction on w. In the base case, we have that $(c, c') = \text{reach}_{W_\times}(\varepsilon)$ is an initial configuration of the product system. By Property (1) of being an invariant, this means $(c, c') \in Q{\downarrow}$. Hence, Q contains some state (q, q') with $(c, c') \leqslant_\times (q, q')$. By the definition of the initial state of A, we have $(q, q') \in Q_{\text{init}} = \text{reach}_A(\varepsilon)$ as required. For the inductive step, consider $w.a$. Any configuration $(d, d') \in \text{reach}_{W_\times}(w.a)$ can be written as an a-successor of a configuration $(c, c') \in \text{reach}_{W_\times}(w)$, i.e. $(c, c') \xrightarrow{a} (d, d')$ in $W_\times$. By induction, we obtain that there is some state $(q, q') \in \text{reach}_A(w)$ with $(c, c') \leqslant_\times (q, q')$. Using upward compatibility, we get that there is some (e, e') with $(q, q') \xrightarrow{a} (e, e')$ in $W_\times$ and $(d, d') \leqslant (e, e')$ By inductivity, Property (3) of being an inductive invariant, we have $\text{post}_{W_\times}(\Sigma, Q{\downarrow}) \subseteq Q{\downarrow}$. Since $(q, q') \in Q \subseteq Q{\downarrow}$, we also obtain $(e, e') \in Q{\downarrow}$. Hence, there is some $(p, p') \in Q$ with $(d, d') \leqslant_\times (e, e') \leqslant_\times (p, p')$. By the definition of the transition relation of A, we have $(q, q') \xrightarrow{a} (p, p')$ in A as this transition overapproximates the transition $(q, q') \xrightarrow{a} (e, e')$ of $W_\times$. We conclude $(p, p') \in \text{post}_A(a, \text{reach}_A(w)) = \text{reach}_A(w.a)$ and $(d, d') \leqslant_\times (p, p')$ as desired. ■

With this lemma at hand, we can immediately conclude that the language of A contains the language of W.

13.2.6 Proposition
$\mathcal{L}(W) \subseteq \mathcal{L}(A)$.

Proof:
Assume that $w \in \mathcal{L}(W)$, i.e. $c \xrightarrow{w} d$ for some $c \in \Gamma_{\text{init}}$ and $d \in \Gamma_{\text{final}}$. By using Statement (1) of Lemma 13.2.5, we obtain that $(d, d') \in \text{reach}_{W_\times}(w)$ for some d'. Using Statement (2) of the same lemma, we get that $(q, q') \in \text{reach}_A(w)$ for a tuple $(q, q') \in Q$ with $(d, d') \leqslant_\times (q, q')$. Since

$d \in \Gamma_{\text{final}}$ and Γ_{final} is upward closed, we have $q' \in \Gamma_{\text{final}}$ and hence $(q, q') \in Q_{\text{final}}$. We conclude that A has an accepting run on word w as desired. ■

We now show that every run of the automaton on some word overapproximates in its second component the unique run of W' ob that word.

13.2.7 Lemma
For every $w \in \Sigma^*$ and every $(q, q') \in \text{reach}_A(w)$ we have $\text{reach}_{W'}(w) \preccurlyeq' q'$.

Proof:
We proceed by induction on w. In the base case, we have that $(q, q') \in \text{reach}_A(\varepsilon) = Q_{\text{init}}$ is an initial state of A. By definition, this means $(c, c') \leqslant_\times (q, q')$ for some initial configuration (c, c') of $W_\times$. Again by definition we have that $c' \in \Gamma'_{\text{init}}$ is the unique initial configuration of W'. Hence, we have $\text{reach}_{W'}(\varepsilon) \preccurlyeq' q'$.

For the inductive step, consider $w.a$. We may write $(p, p') \in \text{reach}_A(w.a)$ as an a-successor of some state $(q, q') \in \text{reach}_A(w)$. This means that there is some transition $(q, q') \xrightarrow{a} (d, d')$ of the product system with $(d, d') \leqslant (p, p')$. Using induction, we get $\text{reach}_{W'}(w) \preccurlyeq q'$. By the definition of the transition relation of the product system, we have that $d' = \text{reach}_{W'}(w.a)$ is the unique configuration reached by W' when reading $w.a$, and we have $d' \preccurlyeq p'$ as required. ■

We use this lemma to conclude that the languages of A and W' are disjoint.

13.2.8 Proposition
$\mathcal{L}(A) \cap \mathcal{L}(W') = \varnothing$.

Proof:
Towards a contradiction, assume that $w \in \mathcal{L}(A) \cap \mathcal{L}(W')$ is a counterexample to disjointness. Since $w \in \mathcal{L}(A)$, there is some accepting state $(q, q') \in \text{reach}_A(w) \cap Q_{\text{final}}$ of A that can be reached by processing w. Note that $(q, q') \in Q_{\text{final}}$ means that $q \in \Gamma_{\text{final}}$ is a final configuration of the first WSTS by definition. Similarly, $w \in \mathcal{L}(W')$ means that the unique configuration $\text{reach}_{W'}(w)$ that W' reaches after processing w is final, i.e. $\text{reach}_{W'}(w) \in \Gamma'_{\text{final}}$.

Using Lemma 13.2.7, we obtain that $\text{reach}_{W'}(w) \preccurlyeq' q'$. The set of final configuration of a WSTS is upward closed, so we can conclude $q' \in \Gamma'_{\text{final}}$. Altogether, we obtain that $(q, q') \in \Gamma_{\text{final}} \times \Gamma'_{\text{final}} = \Gamma_{\times\text{final}}$ is a final configuration of the product system. Furthermore, (q, q') is an element of Q since it is a state of A. This is a contradiction to the assumption that $Q{\downarrow}$ is an inductive invariant that satisfies safety, i.e. $Q{\downarrow} \cap \Gamma_{\times\text{final}} = \varnothing$. ■

The Propositions 13.2.8 and 13.2.8 together show that the language of A is a regular separator for the languages of W and W'. Hence, the proof of Theorem 13.2.3 is completed.

Applying the core result to obtain regular separability for DWSTSes

We conclude this section by showing how we can use Theorem 13.2.3 to show our main result in the case of DWSTSes. Recall that Theorem 13.1.6 states that the languages of any two language-disjoint DWSTSes, one of them deterministic, are regularly separable.

Our core result speaks of upward-compatible LTSes. To bridge the gap, we dualize the given DWSTSes by considering the opposite orders. This process turns a downward-compatible DWSTS into an upward-compatible LTS that may not be well-quasi ordered, which is not a problem in this case. Additionally, the dualization means that any downward-closed set in the product of the dualized DWSTSes is an upward closed set in the product of the original DWSTSes. We can use that upward-closed sets in a WQO can always be written as the upward closure of finitely many elements, which makes the requirement of a finitely represented invariant trivial.

Proof of Theorem 13.1.6:
Let $M = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ and $M' = (\Gamma', \leqslant', T', \Gamma'_{\text{init}}, \Gamma'_{\text{final}})$ be the given language-disjoint DWSTSes where M' is deterministic. We consider the opposite orders $\geqslant$ and $\geqslant'$, obtaining the (upward-compatible) ordered LTS $M^{-1} = (\Gamma, \geqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ and $M'^{-1} = (\Gamma', \geqslant', T', \Gamma'_{\text{init}}, \Gamma'_{\text{final}})$. Note that these indeed satisfy the requirements: For example, Γ_{final} being downward closed with respect to $\leqslant$ means that it is upward closed with respect to the opposite order. The downward-compatibility of T (wrt. $\leqslant$) implies the upward-compatibility of T wrt. $\geqslant$. Obviously, the languages and the fact that M'^{-1} is deterministic remain unchanged. Hence, M^{-1} and M'^{-1} are language-disjoint ordered LTS and one of them is deterministic.

To be able to apply Theorem 13.2.3, we need to find a finitely represented inductive invariant for $M_\times^{-1} = M^{-1} \times M'^{-1}$. We claim that the downward closure of the set of reachable configurations

$$X = \text{reach}_{M_\times^{-1}}{\downarrow}$$

is such a finitely represented invariant, where the downward closure is taken with respect to the product of the orders $\geqslant$ and $\geqslant'$. When we did introduce invariants, we have already discussed that for an ordered LTS with empty language, this set will always be an inductive invariant. To see that it is finitely represented, we first observe that X is also the downward closure (with respect to the product of $\geqslant$ and $\geqslant'$) of the reachable configurations in the product of the original DWSTSes. In a second step, we note that the opposite order of the product of $\geqslant$ and $\geqslant'$ is actually the product of $\leqslant$ and $\leqslant'$. Hence, the downward closure with respect to the product of $\geqslant$ and $\geqslant'$ is the upward closure with respect to the product of the original orders. We may write $X = \text{reach}_{M\times M'}{\uparrow_{\leqslant_\times}}$.

Since $\leqslant$ and $\leqslant'$ are WQOs, so is their product by Lemma 6.5.2. By Property (3) of being a WQO, see Lemma 6.5.3, every upward-closed set can be written as the upward closure of finitely many elements. We obtain that $X = \{x_1, \ldots, x_k\}{\uparrow_{\leqslant_\times}}$ for suitable $x_1, \ldots, x_k \in \text{reach}_{M\times M'}$. In terms of

the product of the opposite orders, we obtain that X is the downward closure of $\{x_1, \ldots, x_k\}$, proving that X is finitely represented.

Hence, we have that X is a finitely represented inductive invariant for $M_\times^{-1}$, so Theorem 13.2.3 yields that $\mathcal{L}(M^{-1}) = \mathcal{L}(M)$ and $\mathcal{L}(M'^{-1}) = \mathcal{L}(M')$ are regularly separable as desired. ■

13.3 Ideals and the ideal completion

The goal of this section is proving Theorem 13.1.1. We want to apply our technical core result, Theorem 13.2.3, but we have to overcome the problem that the existence of a finitely represented invariant is not guaranteed. To this end, we introduce ideals and the ideal completion of a WSTS, well-known techniques that have been proven to be very useful in the forward analysis of WSTSes [FG09; FG12; Fin16].

Consider a WQO $(\Gamma, \leqslant)$. An *ideal* is a non-empty downward-closed subset $\mathcal{I} \subseteq \Gamma$ that is *directed*. Being directed means that every pair of elements has an upper bound within the set, i.e. $\forall x, x' \in \mathcal{I}\, \exists y \in \mathcal{I}\colon x \leqslant y, x' \leqslant y$.

Sets of the shape $y{\downarrow}$ are special cases of ideals. These sets are necessarily directed because they have *global* upper bound. In particular, y is an upper bound for any pair of elements in the set $y{\downarrow}$. In a sense, the notion of ideals replaces the requirement of the existence of such a global upper bound within the set by requiring the existence of *local* upper bounds within the set for each pair of elements. Hence, while any set of the shape $y{\downarrow}$ is an ideal, there can be ideals that are not of this shape.

We briefly consider the shape of ideals in some of the WQOs that we have introduced.

13.3.1 Example

a) Because $(\mathbb{N}, \leqslant)$ is a total order, any non-empty downward closed subset of $\mathbb{N}$ is an ideal. This means that the ideals are precisely the sets of the shape $n{\downarrow}$ for some $n \in \mathbb{N}$ and the set $\mathbb{N}$ itself. The latter is commonly written as $\omega{\downarrow}$.

b) The ideals in $(\Sigma^*, \leqslant)$, where $\leqslant$ is the subword ordering are precisely the languages that can be expressed by regular expressions that consist only of concatenations of expressions of the shape $(a + \varepsilon)$ (where $a \in \Sigma$) and Σ'^* (where $\Sigma' \subseteq \Sigma$ is a subalphabet) [FG09]. Note that this is in fact the original definition of a product in a simple regular expression [ACBJ04].

To determine the ideals in e.g. $\mathbb{N}^k$, one observes that ideals in the product of two WQOs are products of ideals in the original WQOs. This property will also turn out to be very helpful later when we are considering the product of two WSTSes.

13.3.2 Lemma (see e.g. Finkel and Goubault-Larrecq [FG09])

Let $(\Gamma, \leqslant)$ and $(\Gamma', \leqslant')$ be WQOs, and let $(\Gamma_\times, \leqslant_\times)$ be their product. The ideals in $(\Gamma_\times, \leqslant_\times)$ are precisely the sets of the shape $\mathcal{I} \times \mathcal{I}'$ where $\mathcal{I}$ and $\mathcal{I}'$ are ideals of $(\Gamma, \leqslant)$ and $(\Gamma', \leqslant')$, respectively.

13.3.3 Example
The ideals of $\mathbb{N}^k$ can be represented as sets of the shape $M{\downarrow}$, where $M \in (\mathbb{N}_\omega)^k$ is a generalized marking. More formally, we define

$$M{\downarrow} = \left\{ M' \in \mathbb{N}^k \;\middle|\; \forall i \in [1,k] \colon M'(i) \leqslant M(i) \right\}$$

where $n \leqslant \omega$ holds for all natural numbers n.

The usefulness of ideals comes from the property that any downward closed set is a finite union of ideals. This crucial property has been observed and used many times in the past in various contexts. We refer to Section 3.2 of [BFM17] for a more detailed discussion and further references. Here, we restrict ourselves to mentioning that in the recent years, ideals have been used by Finkel and others to design techniques for the forward analysis of WSTSes. Many well-known algorithmic techniques for WSTSes are backward analyses, based on the fact that an upward-closed set can be represented by a basis, i.e. as a finite union of sets of the shape $c{\uparrow}$. Similarly, designing forward analyses requires an effective representation for downward closed sets. With the following lemma, finite sets of ideals can be used as such a representation.

13.3.4 Lemma (see e.g. Blondin, Finkel, and McKenzie [BFM17])
In a WQO, every downward-closed set is a finite union of its inclusion-maximal ideals.

For a WQO $(\Gamma, \leqslant)$ and a downward-closed set $X \subseteq \Gamma$, we write $\mathrm{decomp}_\Gamma(X)$ for the *ideal decomposition*, the set of inclusion-maximal ideals $\mathcal{I} \subseteq X$,

$$\mathrm{decomp}_\Gamma(X) = \left\{ \mathcal{I} \subseteq \Gamma \;\middle|\; \mathcal{I} \text{ ideal of } (\Gamma, \leqslant), \mathcal{I} \subseteq X, \nexists \mathcal{I}' \text{ ideal with } \mathcal{I} \subsetneq \mathcal{I}' \subseteq X \right\}.$$

Using the above lemma, $\mathrm{decomp}_\Gamma(X)$ is always finite, and we have

$$X = \bigcup \mathrm{decomp}_\Gamma(X).$$

Ideals are irreducible in the sense that any ideal contained in a downward-closed subset is contained in an ideal that occurs in the ideal decomposition.

13.3.5 Lemma (Kabil and Pouzet [KP92])
If $\mathcal{I} \subseteq X$ is an ideal, then $\mathcal{I} \subseteq \mathcal{I}'$ for some $\mathcal{I}' \in \mathrm{decomp}_\Gamma(X)$.

Given a WQO $(\Gamma, \leqslant)$, we may consider its *ideal completion* $(\mathrm{ideals}(\Gamma), \subseteq)$, the quasi order of ideals ordered by inclusion. Calling this order a *completion* is justified since the elements x of the original WQO can be identified with their downward closure $x{\downarrow}$, which is always an ideal as

discussed above (and indeed $x \leqslant y$ if and only if $x{\downarrow} \subseteq y{\downarrow}$). The ideal completion is not a WQO in general. Just as the order $(\mathcal{P}^{\downarrow}(\Gamma), \leqslant)$, it is a WQO if and only if the original WQO was ω^2 [FG12].

For some set $X \subseteq \Gamma$, $\mathrm{decomp}_\Gamma(X)$ is the set of inclusion-maximal ideals contained in X. We may consider the downward closure $\mathrm{decomp}_\Gamma(X){\downarrow}$ in the order $(\mathrm{ideals}(\Gamma), \subseteq)$ to obtain the set of all ideals $\mathcal{I} \subseteq X$ that are contained in X. Unlike $\mathrm{decomp}_\Gamma(X)$, this downward closure is not necessarily a finite set of ideals. We have $X = \bigcup \mathrm{decomp}_\Gamma(X) = \bigcup \mathrm{decomp}_\Gamma(X){\downarrow}$.

In [FG12; BFM17], the notion of the ideal completion has been lifted from orders to WSTSes. Given a WSTS, one constructs an ordered LTS whose configurations are ideals of the original underlying order. Its runs approximate runs of the original WSTS in the sense that at each point in time, the ideal that forms the current configuration in the ideal completion contains a configuration of the original WSTS that can be reached by the same word. Due to upward-compatibility, taking the ideal completion of an WSTS does not change the language.

The ideal completion can be seen as an alternative to the construction from Lemma 12.2.4 that we have used to show that an ω^2-WSTS can be transformed into a deterministic WSTS with the same language. The drawback that using $(\mathrm{ideals}(\Gamma), \subseteq)$ has compared to using $(\mathcal{P}^{\downarrow}(\Gamma), \leqslant)$ is that we can only ensure that the new LTS is finitely branching, but not that it is deterministic. The fundamental advantage is that the ideal completion will guarantee that any downward-closed set of configurations of the original WSTS induces a finitely represented downward-closed set in the ideal completion.

The formal definition of the ideal completion is as follows.

13.3.6 Definition
Let $W = (\Gamma, \leqslant, T, \Gamma_{\mathrm{init}}, \Gamma_{\mathrm{final}})$ be a WSTS. Its *ideal completion*

$$\mathrm{compl}(W) = (\mathrm{ideals}(\Gamma), \subseteq, T', \mathrm{ideals}(\Gamma)_{\mathrm{init}}, \mathrm{ideals}(\Gamma)_{\mathrm{final}})$$

is the ordered LTS whose configurations are ideals of $(\Gamma, \leqslant)$, whose initial ideals are the inclusion-maximal ideals in Γ_{init}, $\mathrm{ideals}(\Gamma)_{\mathrm{init}} = \mathrm{decomp}(\Gamma_{\mathrm{init}}{\downarrow})$, and whose final ideals are those that contain a final configuration of W, $\mathrm{ideals}(\Gamma)_{\mathrm{final}} = \{\mathcal{I} \mid \mathcal{I} \cap \Gamma_{\mathrm{final}} \neq \varnothing\}$. The transition relation is defined by applying the post-operation of W and then considering the ideal decomposition. If the ideal decomposition of a post-set consists of multiple ideals, the ideal completion is nondeterministic in that it has a transition to each of them. Formally, we have

$$\mathrm{post}_{\mathrm{compl}(W)}(a, \mathcal{I}) = \mathrm{decomp}_\Gamma(\mathrm{post}_W(a, \mathcal{I}){\downarrow}) \,,$$

i.e. $\mathcal{I} \xrightarrow{a} \mathcal{I}'$ if $\mathcal{I}'$ is an inclusion-maximal ideal of $\mathrm{post}_W(a, \mathcal{I}){\downarrow}$.

Note that for the initial configurations and the post-sets, we need to apply the downward closure in $(\Gamma, \leqslant)$ since we have not required these sets to be downward closed. We have, however,

already exploited in the proof of Theorem 12.2.1 that we could make this assumption without changing the language of the WSTS.

The following lemma summarizes the properties of the ideal completion that we will need in the following. Even though this is not original work, we will give the proof to emphasize the importance of the lemma and to show how the properties of ideals come into play.

13.3.7 Lemma (Finkel and Goubault-Larrecq [FG12]; Blondin, Finkel, and McKenzie [BFM17])

a) The ideal completion compl(W) of a WSTS W is a finitely branching ordered LTS.

b) $\mathcal{L}(W) = \mathcal{L}(\text{compl}(W))$.

c) If WSTS W is ω^2, then compl(W) is a WSTS.

d) If WSTS W is deterministic, then so is compl(W).

Proof of L3MM4 1337:
We first show Part a), i.e. that compl(W) is an ordered LTS. We have to argue that ideals(Γ)$_{\text{final}}$ is upward closed with respect to $\subseteq$: If $\mathcal{I} \cap \Gamma_{\text{final}} \neq \varnothing$ and $\mathcal{I} \subseteq \mathcal{I}'$, then $\mathcal{I}' \cap \Gamma_{\text{final}} \supseteq \mathcal{I} \cap \Gamma_{\text{final}} \neq \varnothing$. Upward compatibility follows from Lemma 13.3.5: Assume $\mathcal{I} \subseteq \mathcal{I}'$ and $\mathcal{I} \xrightarrow{a} \mathcal{J}$ in compl(W). This means $\mathcal{J} \in \text{decomp}(\text{post}_W(a, \mathcal{I})\downarrow)$, which also means that $\mathcal{J} \subseteq \text{post}_W(a, \mathcal{I}')\downarrow$ since $\text{post}_W(a, \mathcal{I}') \supseteq \text{post}_W(a, \mathcal{I})$. Using Lemma 13.3.5, there is some ideal $\mathcal{J}' \in \text{decomp}(\text{post}_W(a, \mathcal{I}')\downarrow)$ with $\mathcal{J} \subseteq \mathcal{J}'$. Hence, $\mathcal{I}' \xrightarrow{a} \mathcal{J}'$ as required. The ideal completion is finitely branching since decomp(X) is finite for any set and so for $\text{post}_W(a, \mathcal{I})\downarrow$ in particular.

For Part c), observe that if the original WSTS is ω^2, then the ideal completion of the underlying order on configurations is a WQO [FG12]. Hence, the ideal completion of W is a WSTS.

To show Part d), that $W = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ being deterministic implies compl(W) being deterministic, we prove that after reading word w, compl(W) is in the unique configuration $\text{reach}_W(w)\downarrow$. We prove this claim by induction. In the base case, we consider an ideal in $\text{decomp}(\Gamma_{\text{init}}\downarrow)$. Since W is assumed to be deterministic, we have $\Gamma_{\text{init}} = \{c_{\text{init}}\}$ for some configuration c_{init}. Hence, $c_{\text{init}}\downarrow$ is the unique inclusion-maximal ideal of $\Gamma_{\text{init}}\downarrow$ and $\text{reach}_{\text{compl}(W)}(\varepsilon) = \text{decomp}(\Gamma_{\text{init}}\downarrow) = \{c_{\text{init}}\downarrow\} = \{\text{reach}_W(\varepsilon)\}$. For the inductive step, consider $w.a$. By induction, $\text{reach}_{\text{compl}(W)}(w)$ is the unique ideal $\mathcal{I} = \text{reach}_W(w)\downarrow$. We have that $\text{reach}_{\text{compl}(W)}(w.a)$ is an element of $\text{decomp}(\text{post}_W(a, \mathcal{I})\downarrow)$. We claim that $\text{post}_W(a, \mathcal{I})\downarrow$ is the ideal $\text{reach}_W(w.a)\downarrow$, and hence its ideal decomposition only consists of this single ideal. It is clear by definition that $\text{reach}_W(w.a)\downarrow$ is a subset of $\text{post}_W(a, \mathcal{I})\downarrow$. For the other direction, consider $d \in \text{post}_W(a, \mathcal{I})\downarrow$. This means there is some $c' \in \mathcal{I}$ so that $c' \xrightarrow{a} d'$ for some d' with $d \leqslant d'$. Since $\mathcal{I} = \text{reach}_W(w)\downarrow$, we in turn have $c' \leqslant c''$ with $c'' = \text{reach}_W(w)$. Using upward compatibility, there are some d'' with $c'' \xrightarrow{a} d''$, $d' \leqslant d''$. We conclude $d'' = \text{reach}_W(w.a)$ and $d \leqslant d' \leqslant d''$, so $d \in \text{reach}_W(w.a)\downarrow$ as required.

To complete the proof, it remains to show Part b), i.e. that the language is preserved. We claim that for each word $w \in \Sigma^*$, the union of the ideals reachable along w in the ideal completion is the set of configurations reachable along w in the original WSTS. Formally, we claim

$$\bigcup \mathrm{reach}_{\mathrm{compl}(W)}(w) \;=\; \mathrm{reach}_W(w)\!\downarrow \;.$$

We prove this claim using induction

In the base case, we have

$$\bigcup \mathrm{reach}_{\mathrm{compl}(W)}(\varepsilon) \;=\; \bigcup \mathrm{ideals}(\Gamma)_{\mathrm{init}} \;=\; \bigcup \mathrm{decomp}(\Gamma_{\mathrm{init}}\!\downarrow) \;=\; \Gamma_{\mathrm{init}}\!\downarrow = \mathrm{reach}_W(\varepsilon)\!\downarrow$$

as required.

For the inductive step, consider $w.a$. Using upward compatibility, induction, and the fact that unions commute with the post-operation, we obtain

$$\begin{aligned}
\mathrm{reach}_W(w.a)\!\downarrow &= \mathrm{post}_W(a, \mathrm{reach}_W(w))\!\downarrow \\
&= \mathrm{post}_W(a, \mathrm{reach}_W(w)\!\downarrow)\!\downarrow \\
&= \mathrm{post}_W\Big(a, \bigcup \mathrm{reach}_{\mathrm{compl}(W)}(w)\Big)\!\downarrow \\
&= \mathrm{post}_W\left(a, \bigcup_{\mathcal{I} \in \mathrm{reach}_{\mathrm{compl}(W)}(w)} \mathcal{I}\right)\!\downarrow \\
&= \bigcup_{\mathcal{I} \in \mathrm{reach}_{\mathrm{compl}(W)}(w)} \mathrm{post}_W(a, \mathcal{I})\!\downarrow \;.
\end{aligned}$$

Using Lemma Lemma 13.3.4, $\mathrm{post}_W(a, \mathcal{I})\!\downarrow$ is equal to $\bigcup \mathrm{decomp}(\mathrm{post}_W(a, \mathcal{I})\!\downarrow)$. With this equality, the definition of the transition relation in $\mathrm{compl}(W)$, the fact that the post-operation commutes with unions, and the definition of reach, we obtain

$$\begin{aligned}
\bigcup_{\mathcal{I} \in \mathrm{reach}_{\mathrm{compl}(W)}(w)} \mathrm{post}_W(a, \mathcal{I})\!\downarrow &= \bigcup_{\mathcal{I} \in \mathrm{reach}_{\mathrm{compl}(W)}(w)} \bigcup \mathrm{decomp}(\mathrm{post}_W(a, \mathcal{I})\!\downarrow) \\
&= \bigcup_{\mathcal{I} \in \mathrm{reach}_{\mathrm{compl}(W)}(w)} \bigcup \mathrm{post}_{\mathrm{compl}(W)}(a, \mathcal{I}) \\
&= \bigcup \mathrm{post}_{\mathrm{compl}(W)}\big(a, \mathrm{reach}_{\mathrm{compl}(W)}(w)\big) \\
&= \bigcup \mathrm{reach}_{\mathrm{compl}(W)}(w.a)
\end{aligned}$$

as desired.

Using the claim that we just have established, proving language equivalence is easy. For any word w, the condition $w \in \mathcal{L}(W)$ is equivalent to the existence of a configuration $c \in \mathrm{reach}_W(w) \cap \Gamma_{\mathrm{final}}$. Since Γ_{final} is upward closed, this is equivalent to $\mathrm{reach}_W(w)\!\downarrow \cap \Gamma_{\mathrm{final}} \neq \varnothing$. Using the claim, this is equivalent to the existence of an ideal $\mathcal{I} \in \mathrm{reach}_{\mathrm{compl}(W)}(w)$ such that $\mathcal{I} \cap \Gamma_{\mathrm{final}} \neq \varnothing$, which means $\mathcal{I} \in \mathrm{ideals}(\Gamma)_{\mathrm{final}}$. This in turn means $w \in \mathcal{L}(\mathrm{compl}(W))$. ■

Recall that the advantage of the ideal completion of a WSTS, in comparison to its determinization using the operator $\mathcal{P}^{\downarrow}(-)$, is that it guarantees the existence of finite representations of downward-closed sets. We formalize this is in the following by showing that any inductive invariant X of a WSTS induces a finitely represented inductive invariant in the ideal completion. This will enable us to apply the technical core result Theorem 13.2.3 to prove Theorem 13.1.1.

13.3.8 Proposition
If $X \subseteq \Gamma$ is an inductive invariant of WSTS W, then $\mathrm{decomp}(X)\downarrow$ is a finitely represented inductive invariant for the ideal completion $\mathrm{compl}(W)$.

Proof:
Assume X to be an inductive invariant for W. Recall that this implies (1) $\Gamma_{\mathrm{init}} \subseteq X$, (2) $\Gamma_{\mathrm{final}} \cap X = \varnothing$, and (3) $\mathrm{post}_W(\Sigma, X) \subseteq X$. We claim that $\mathrm{decomp}(X)\downarrow$ is a finitely represented inductive invariant for $\mathrm{compl}(W)$. Observe that $\mathrm{decomp}(X)\downarrow$ is finitely represented since $\mathrm{decomp}(X)$ is finite by Lemma 13.3.4 It remains to verify the properties of being an invariant.

(1) First, we show $\mathrm{ideals}(\Gamma)_{\mathrm{init}} \subseteq \mathrm{decomp}(X)\downarrow$. For any ideal $\mathcal{I} \in \mathrm{ideals}(\Gamma)_{\mathrm{init}}$, we have $\mathcal{I} \subseteq \Gamma_{\mathrm{init}}$ by definition. Since $\Gamma_{\mathrm{init}} \subseteq X$ by Property (1) of X being an inductive invariant, we conclude that $\mathcal{I}$ is an ideal in X, and hence $\mathcal{I} \in \mathrm{decomp}(X)\downarrow$ since $\mathrm{decomp}(X)\downarrow$ is the set of all such ideals.

(2) Towards a contradiction, assume that there is an ideal $\mathcal{I} \in \mathrm{ideals}(\Gamma)_{\mathrm{final}} \cap \mathrm{decomp}(X)\downarrow$. This means that $\mathcal{I} \subseteq X$ and that $\mathcal{I} \cap \Gamma_{\mathrm{final}} \neq \varnothing$. We conclude $X \cap \Gamma_{\mathrm{final}} \neq \varnothing$, a contradiction to the safety of X.

(3) We need to check the inclusion $\mathrm{post}_{\mathrm{compl}(W)}(\Sigma, \mathrm{decomp}(X)\downarrow) \subseteq \mathrm{decomp}(X)\downarrow$. Pick any $\mathcal{J} \in \mathrm{post}_{\mathrm{compl}(W)}(\Sigma, \mathrm{decomp}(X)\downarrow)$. By definition, there is some $\mathcal{I} \in \mathrm{decomp}(X)\downarrow$ and some letter $a \in \Sigma$ such that $\mathcal{I} \xrightarrow{a} \mathcal{J}$ in $\mathrm{compl}(W)$. This means that $\mathcal{J} \in \mathrm{decomp}(\mathrm{post}_W(a, \mathcal{I})\downarrow)$, so in particular $\mathcal{J} \subseteq \mathrm{post}_W(a, \mathcal{I})\downarrow$. Since we have $\mathcal{I} \subseteq X$ and $\mathrm{post}_W(\Sigma, X) \subseteq X$ holds by assumption, we get $\mathcal{J} \subseteq X$. Hence, $\mathcal{J}$ is contained in $\mathrm{decomp}(X)\downarrow$ as required. ■

Before we can finally prove our main result for WSTSes, we need to overcome a minor challenge. Theorem 13.2.3, our technical core results, expects a finitely represented invariant for the product of two ordered LTSes. Our intention is to use the products of the ideal completions of the given WSTSes. Proposition 13.3.8, however, guarantees the existence of a finitely represented invariant in the ideal completion of a single WSTS. Luckily, the operations of taking the product and taking the ideal completion commute: The product of the ideal completions is the ideal completion of the product.

13.3.9 Lemma
Let W, W' be two WSTSes. The ideal completion of their product $\mathrm{compl}(W \times W')$ is the product of their ideal completions $\mathrm{compl}(W) \times \mathrm{compl}(W')$.

Proof:
The proof essentially follows from the fact that every ideal in the product domain is a product of ideals and vice versa, see Lemma 13.3.2. Hence, the configuration sets of $\mathrm{compl}(W \times W')$ and $\mathrm{compl}(W) \times \mathrm{compl}(W')$ are equal. Using the definition of the synchronized product, it is straightforward to check that the identity also preserves the order and the initial and final sets of configurations. It remains to show that the transition relation is preserved. Consider a product of ideals $\mathcal{I} \times \mathcal{I}'$. The a-successors of this product in $\mathrm{compl}(W \times W')$ are the ideals (which themselves are products) in $\mathrm{decomp}(\mathrm{post}_{W\times W'}(a, \mathcal{I}\times\mathcal{I}')\downarrow)$; the a-successors of $\mathcal{I} \in \mathcal{I}'$ in $\mathcal{I}\times\mathcal{I}'$ are the products of ideals in $\mathrm{decomp}(\mathrm{post}_W(a, \mathcal{I})\downarrow) \times \mathrm{decomp}(\mathrm{post}_{W'}(a, \mathcal{I}')\downarrow)$. Using the definition of the transition relation of the synchronized product and the fact that taking the downward closure and taking the ideal decomposition both commute with the product operation, we can prove the equality of these sets:

$$\begin{aligned}\mathrm{decomp}\big(\mathrm{post}_{W\times W'}\big(a, \mathcal{I} \times \mathcal{I}'\big)\downarrow\big) &= \mathrm{decomp}\big(\big(\mathrm{post}_W(a, \mathcal{I}) \times \mathrm{post}_{W'}\big(a, \mathcal{I}'\big)\big)\downarrow\big)\\ &= \mathrm{decomp}\big(\mathrm{post}_W(a, \mathcal{I})\downarrow \times \mathrm{post}_{W'}\big(a, \mathcal{I}'\big)\downarrow\big)\\ &= \mathrm{decomp}(\mathrm{post}_W(a, \mathcal{I})\downarrow) \times \mathrm{decomp}\big(\mathrm{post}_{W'}\big(a, \mathcal{I}'\big)\downarrow\big)\,.\end{aligned}$$

This completes the proof. ■

We can now finally prove the main result. Recall that Theorem 13.1.1 states that two disjoint WSTS languages, at least one of them the language of a deterministic WSTS, are regularly separable.

Proof of Theorem 13.1.1:
Assume that W, W' are WSTSes with disjoint languages, $\mathcal{L}(W) \cap \mathcal{L}(W') = \varnothing$, and W' is deterministic. We consider their ideal completions $\mathrm{compl}(W)$ and $\mathrm{compl}(W')$. Using Lemma 13.3.7, $\mathrm{compl}(W')$ is guaranteed to be deterministic. Furthermore, $\mathcal{L}(W) = \mathcal{L}(\mathrm{compl}(W))$ and $\mathcal{L}(W') = \mathcal{L}(\mathrm{compl}(W'))$.

Since the languages of W and W' are disjoint, the language of their product $W \times W'$ is empty. Hence, there is an inductive invariant for $W \times W'$, e.g. $\mathrm{reach}_{W\times W'}\downarrow$, that is not necessarily finitely represented. With Proposition 13.3.8, the ideal completion of the product $\mathrm{compl}(W \times W')$ has a finitely represented inductive invariant, e.g. $\mathrm{compl}(\mathrm{reach}_{W\times W'}\downarrow)\downarrow_{\subseteq}$. With Lemma 13.3.9, the ideal completion of the product is equal to the product of the ideal completions. Hence, we have found finitely represented inductive invariant for the product of the ideal completions.

By applying the technical core result Theorem 13.2.3, we obtain that the languages of the ideal completions – which are equal to the languages of the original WSTSes – are regularly separable as desired. ■

We conclude this section by demonstrating that with the concepts that we have introduced in this thesis, it does not seem possible to get rid of the assumption that one of the two WSTSes is deterministic. The above proof combines various state-of-the-art techniques, ones from the literature and freshly developed ones, in a specific order. It is assembled with care to achieve the intended result. Various ways to re-arrange the ingredients of the proof that might seem promising at first glance turn out to be incorrect.

Consider for example the idea of first using e.g. the operator $\mathcal{P}^{\downarrow}(-)$ to make one of the WSTSes deterministic before applying the development detailed in this section. This might seem promising since our technical core result only requires ordered LTSes instead of WSTSes. However, this approach makes the given systems lose the WQO property *too early*: After applying the operator $\mathcal{P}^{\downarrow}(-)$ as in Lemma 12.2.4 to a WSTS that is not guaranteed to be ω^2 will yield an ordered LTS that may not be a WSTS. However, Lemma 13.3.4, the fact that all downward-closed sets decompose into finitely many ideals, is reliant on the finite antichain property of WQOs, Property Item 4 in Lemma 6.5.3. Hence, applying the ideal completion to an LTS that is not a WSTS may yield a system which sets of the shape $\mathrm{decomp}(X)$ can be infinite and Proposition 13.3.8 breaks. A similar problem occurs if we try to start by making one of the given systems finitely branching, e.g. using the ideal completion.

Another idea that comes to mind is replacing the ideal completion by the operator $\mathcal{P}^{\downarrow}(-)$ throughout this section. The resulting system shares many of the properties with the ideal completion: It preserves the language and ensures the existence of finitely represented invariants: If $X \subseteq \Gamma$ is a downward-closed inductive invariant, then so is $X{\downarrow_{\subseteq}} \subseteq \mathcal{P}^{\downarrow}(\Gamma)$. The problem here is that an equivalent of Lemma 13.3.9 does not hold in this case: The downward closed subsets in $\mathcal{P}^{\downarrow}(\Gamma \times \Gamma')$ are not simply products of downward closed subsets in $\mathcal{P}^{\downarrow}(\Gamma)$ and $\mathcal{P}^{\downarrow}(\Gamma')$, but rather unions of such products.

The author conjectures that the key to overcoming the requirement of one of the given systems having to be deterministic, or realizing that the requirement is in fact necessary, is understanding whether the inclusions in our result on WSTS expressiveness, Theorem 12.2.1, are strict. This would require a deeper understanding of the expressive power of WSTSes that are neither ω^2 nor finitely branching.

14 Bounds on the separator size for Petri nets

Contents

14.1 Upper bound 255
14.2 Lower bound 265

The goal of this chapter is to demonstrate that the separator construction outlined in Chapter 13 is constructive. For many types of WSTSes, it is possible to actually construct an invariant in the ideal completion which then can be turned into a separator. Here, we consider the case of Petri nets with coverability as the acceptance condition. We show how to construct an NFA whose language is a regular separator with a number of states that is at most triply exponential. We complement this result by a doubly exponential lower bound. Unfortunately, the bounds are not tight, a fact on which we will briefly comment later.

14.1 Upper bound

Our upper bound is the following theorem.

14.1.1 Theorem
Two labeled Petri net instances with disjoint languages can be separated by a regular language with triply exponential state complexity.

Our proof of the theorem will be constructive in the sense that it provides an invariant in the ideal completion from which one can construct a separating automaton as specified by Definition 13.2.4. Before we present the construction, we need to deal with two problems. The first one is the minor technicality that we have allowed ε-transitions in Petri nets, but not in WSTSes. The second one is the requirement of one of the systems being deterministic. We will discuss the latter in more detail after solving the first problem.

Throughout the rest of this section, let $(N_1, M_{\text{init}1}, M_{\text{final}1})$ and $(N_2, M_{\text{init}2}, M_{\text{final}2})$ be two Petri net instances over some alphabet Σ of size n_1 and n_2, respectively. We assume that the coverability languages of the two nets are disjoint.

Eliminating ε-transitions

Unlike in Petri nets, we have not allowed ε-transitions in WSTSes for reasons that we explained in Remark 6.5.5. The construction that we will apply to determinize one of the nets will also eliminate ε-transitions in that net as a by-product. To handle ε-transitions in the other net, we apply a preprocessing step. It is designed so that it does not change the fact that the nets are language-disjoint, and it has no substantial influence on the size of the separating automaton.

Assume that net N_1 contains transitions labeled by ε. To eliminate them, we consider a fresh letter $a \notin \Sigma$ and define $\Sigma_a = \Sigma \uplus \{a\}$. We relabel all ε-transitions in N_1 by letter a, resulting in the net N_1'. The initial and final marking remain unchanged.

To account for the changes to N_1 in the other net, we add a fresh transition t_a to N_2 that is labeled by a and that does neither consume nor produce any tokens, $\mathrm{in}(t_a) = \mathrm{out}(t_a) = \vec{0}$. Let the resulting net be N_2'. Again, we do not change the initial or the final marking.

Firstly, observe that the size of N_1 and N_2 is polynomial in the size of N_1 and N_2, respectively. Secondly, we will prove that a separator for the original nets can be turned into a separator for the modified nets and vice versa, where both transformations impose a blow up of the separating automaton that is at most polynomial.

14.1.2 Lemma
The languages of N_1 and N_2 are regularly separable iff the languages of N_1' and N_2' are.

Proof:
Assume that A is an NFA over Σ whose language $\mathcal{L}(A)$ is a regular separator for $\mathcal{L}(N_1, M_{\text{init1}}, M_{\text{final1}})$ and $\mathcal{L}(N_2, M_{\text{init2}}, M_{\text{final2}})$. We define A' over Σ_a by adding an a-labeled self-loop $q \xrightarrow{a} q$ to every state q of A. Clearly, the size of A' is polynomial in the size of A. It remains to show that its language separates $\mathcal{L}(N_1', M_{\text{init1}}, M_{\text{final1}})$ and $\mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}})$.

We first show $\mathcal{L}(N_1', M_{\text{init1}}, M_{\text{final1}}) \subseteq \mathcal{L}(A')$. Consider a covering computation $M_{\text{init1}} \boxed{\sigma} M \geqslant M_{\text{final1}}$ of N_1'. It is also a covering computation of N_1, because the two nets differ only in their transition labels. Hence, $\lambda(\sigma) \in \mathcal{L}(A)$ since $\mathcal{L}(A)$ is assumed to be a separator. By using the a-labeled self-loops that are present in A', we turn an accepting run of A on $\lambda(\sigma)$ in into an accepting run of A' on $\lambda'(\sigma)$, which completes the argument.

To see that $\mathcal{L}(A') \cap \mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}}) = \varnothing$, assume towards a contradiction that some word $w' \in \Sigma_a^*$ is contained in the intersection. Define word $w \in \Sigma^*$ by projecting w' to Σ, i.e. by removing all occurrences of letter a. We claim that $w \in \mathcal{L}(A) \cap \mathcal{L}(N_2, M_{\text{init2}}, M_{\text{final2}})$. The only way of generating letter a in automaton A' is using the a-labeled self-loops. By removing all of them, we turn an accepting run of A' on w' into an accepting run of A on w. Similarly, a covering computation of N_2' for w' can be turned into a covering computation of N_2 for w by removing all occurrences of transition t_a. Hence, $w \in \mathcal{L}(A) \cap \mathcal{L}(N_2, M_{\text{init2}}, M_{\text{final2}})$ which is a contradiction to the assumption that $\mathcal{L}(A)$ is a separator.

For the other direction, assume that the language of some NFA A' separates $\mathcal{L}(N_1', M_{\text{init1}}, M_{\text{final1}})$ and $\mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}})$. We define A by relabeling all a-transitions of A' to ε. The resulting automaton has the same size as A', but may contain ε-transitions. However, it is well-known that these transitions can be eliminated without introducing additional states.

We show that $\mathcal{L}(N_1, M_{\text{init1}}, M_{\text{final1}}) \subseteq \mathcal{L}(A)$. Assume $w \in \mathcal{L}(N_1, M_{\text{init1}}, M_{\text{final1}})$, and consider a covering computation $M_{\text{init1}} \xrightarrow{\sigma} M \geqslant M_{\text{final1}}$ of N_1 with $\lambda(\sigma) = w$. We may see this computation as a computation of N_1' that produces the word $w' = \lambda'(\sigma)$, and use the assumption that $\mathcal{L}(A')$ is a separator to conclude $w' \in \mathcal{L}(A')$. Because w is obtained from w' by removing all occurrences of letter a, we obtain $w \in \mathcal{L}(A)$ as desired.

Finally, we show $\mathcal{L}(N_2, M_{\text{init2}}, M_{\text{final2}}) \cap \mathcal{L}(A) = \varnothing$. Assume that word w is contained in $\mathcal{L}(A)$. We define w' to be a word that is obtained from w by inserting occurrences of letter a so that $w' \in \mathcal{L}(A')$. It has to exist by the definition of the automaton A. Using that $\mathcal{L}(A')$ is a separator, we obtain that w' is not contained in $\mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}})$. Hence, the word w that is re-obtained by removing all occurrences of a cannot be contained in the language of N_2: A covering computation of N_2 producing w would imply the existence of a covering computation of N_2' producing w' by inserting occurrences of t_a. This finishes the proof. ■

Altogether, we have shown how to eliminate ε-transitions from one of the nets while only introducing a polynomial blowup to the size of the nets and the size of the separator. In the rest of this section, we can assume without loss of generality that both Petri nets do not contain ε-labeled transitions.

Enforcing determinism

Theorem 13.1.1, our result that shows that the languages of disjoint WSTSes are regularly separable requires one of the two WSTSes to be deterministic. In general, the WSTSes induced by a labeled Petri net will not satisfy this requirement. Because the underlying order $(\mathbb{N}^k, \leqslant)$ of such a WSTS is ω^2, we could use our results on expressiveness, Lemma 12.2.4 in particular, to obtain a deterministic language-equivalent WSTS. The construction from the proof of Lemma 12.2.4 would provide a WSTS whose underlying order is $(\mathcal{P}^{\downarrow}(\mathbb{N}^k), \subseteq)$. This poses a problem. The ideals of $(\mathcal{P}^{\downarrow}(\mathbb{N}^k), \subseteq)$ are well-understood: They are of the shape $D{\downarrow_{\subseteq}} = \{D' \subseteq D \mid D' \text{ is downward closed in } \mathbb{N}^k\}$, where D ranges over the non-empty downward-closed subsets of $\mathbb{N}^k$ [LS15a]. However, we then would need to obtain a bound on the size of the ideal decomposition of an invariant in this order to obtain a bound on the size of the separating automaton. The author did not succeed in proving such a result. It seems that the methods that have been developed by Lazic and Schmitz [LS15a] in the context of *alternating* and *branching vector addition systems* do not apply here.

To circumvent the outlined problems, we do not use the construction from the proof of Lemma 12.2.4. Instead, we apply another preprocessing step to one of the Petri net which leads to the induced WSTS becoming deterministic. This leads to a separator for the modified nets,

but it is not straightforward to obtain the desired separator for the original ones. We will discuss this problem in detail after giving the construction.

Enforcing uniqueness

Assume again that $(N_1, M_{\text{init1}}, M_{\text{final1}})$ and $(N_2, M_{\text{init2}}, M_{\text{final2}})$ are the given Petri net instances. The main challenge in enforcing the WSTS induced by N_1 to be deterministic is the requirement that the transition relation has to be unique. This requirement is violated as soon as N_1 contains two transitions that have the same label, because there is necessarily a marking that is large enough so that both transitions are enabled. Hence, we have to equip N_1 with the *free labeling*, a labeling that labels each transition by its own name. In the freely labeled variant N_1' of N_1, the set of transitions T_1 is the alphabet and the labeling function is the identity $\mathrm{id}\colon T_1 \to T_1$.

It remains to create a modification N_2' of N_2 that also uses T_1 as the alphabet. In the product of N_1 and N_2, a transition t_1 of N_1 can synchronize with all transitions of t_2 that have the same label, $\lambda_1(t_1) = \lambda_2(t_2)$. The idea behind the construction of N_2' is to preserve this property. To this end, we replace each transition t_2, say with label $\lambda_2(t_2) = a$, by a bunch of copies $t_2(t_1)$ of t_2 in N_2', one copy for each transition t_1 of N_1 with label $\lambda_1(t_1) = a$. The incoming and outgoing multiplicities of each copy $t_2(t_1)$ are equal to those of t_2. The places, the initial marking, and the final marking of N_2' coincide with N_2. Net N_2' uses T_1 as its alphabet, and each transition $t_2(t_1)$ is labeled by t_1.

We argue that this construction accomplishes its intended purpose. Firstly, observe that if two transitions t_1 in N_1 and t_2 in N_2 can synchronize in the product of the nets, meaning that they have the same label, then the transition t_1 in N_1' can synchronize with transition $t_2(t_1)$ in N_2', the t_1-labeled copy of t_2.

Secondly, the transition relation of N_1' (and the WSTS induced by N_1') is unique since there is only one transition with a specific label. However, it is not yet deterministic, since we are missing completeness: There may be markings in which the unique transition with a specific label is not enabled. This is a minor issue that we will handle later.

Thirdly, there is a strong connection between the languages of N_1 and N_2 and the languages of N_1' and N_2'. Consider the labeling function of N_1', extended into a homomorphism $\lambda_1\colon T_1^* \to \Sigma^*$. We have that $\mathcal{L}(N_1, M_{\text{init1}}, M_{\text{final1}}) = \lambda_1(\mathcal{L}(N_1', M_{\text{init1}}, M_{\text{final1}}))$ and $\mathcal{L}(N_2, M_{\text{init2}}, M_{\text{final2}}) = \lambda_1(\mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}}))$. This in particular means that the languages of N_1 and N_2 are disjoint if and only if the languages of N_1' and N_2' are. It is easy to see that a word $\sigma \in T_1^*$ that is in the languages of N_1' and N_2' leads to $\lambda_1(\sigma)$ being in the language of N_1 and N_2. For the other direction, consider a word $w \in \Sigma^*$ in the intersection of the languages of N_1 and N_2, and consider firing sequences $\sigma \in T_1^*, \tau \in T_2^*$ that induce the covering computation. We immediately get that σ is in the language of N_1'. To see that σ is also in the language of N_2', one can use the covering computation induced by the firing sequence τ_σ whose i^{th} transition is $\tau_i(\sigma_i)$, the σ_i-labeled copy of τ_i in N_2'.

The most important property is that a regular separator for the languages N_1' and N_2' can be turned in to a regular separator for the languages of the original nets. One might expect that if $\mathcal{R} \subseteq T_1^*$ is a regular language separating the languages of the modified nets, then $\lambda_1(\mathcal{R})$ is the desired separator over Σ^*. However, this turns out to be wrong. Applying a homomorphism to two disjoint languages can result in two languages with a non-empty intersection. Assume that $\mathcal{R}$ is disjoint from the language of N_2', $\mathcal{R} \cap \mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}}) = \varnothing$. Nevertheless, there may be a word $w \in \lambda_1(\mathcal{R}) \cap \lambda_1(\mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}}))$, which means that $\lambda_1(\mathcal{R})$ is not disjoint from $\lambda_1(\mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}})) = \mathcal{L}(N_2, M_{\text{init2}}, M_{\text{final2}})$. Since we assume that $\mathcal{R}$ and the language of N_2' are disjoint, we know that w is not the image of a word over T_1 that is both in $\mathcal{R}$ and in the language of N_2. However, there may be two words that both have w as their image under λ_1 such that one of them is in $\mathcal{R}$ and the other is in the language of N_2'. Phrased differently, we need that any word in $\lambda_1(\mathcal{R})$ has no preimage in the language of N_2', which is a stronger property than disjointness.

This problem can be overcome by using complementation. More precisely, let $\mathcal{R} \subseteq T_1^*$ be a regular language that separates N_2' and N_1' in the sense that $\mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}}) \subseteq \mathcal{R}$ and $\mathcal{L}(N_1', M_{\text{init1}}, M_{\text{final1}}) \cap \mathcal{R} = \varnothing$. Then $\lambda_1(\overline{\mathcal{R}})$ is a regular separator for the languages of N_1 and N_2. Note that the order of the two input languages is swapped to account for the complementation.

14.1.3 Proposition
If $\mathcal{R}$ regularly separates the languages of N_2' and N_1', then $\lambda_1(\overline{\mathcal{R}})$ regularly separates the languages of N_1 and N_2.

Proof:
Firstly, observe that our candidate separator is indeed a regular language because the class of regular languages is closed under complementation and homomorphisms. Secondly, we prove $\mathcal{L}(N_1, M_{\text{init1}}, M_{\text{final1}}) \subseteq \lambda_1(\overline{\mathcal{R}})$. Since $\mathcal{R}$ separates the languages of N_2' and N_1', its complement $\overline{\mathcal{R}}$ separates the languages of N_1' and N_2'. This in particular means that $\mathcal{L}(N_1', M_{\text{init1}}, M_{\text{final1}}) \subseteq \overline{\mathcal{R}}$. Inclusions are preserved under homomorphisms, so we may use $\mathcal{L}(N_1, M_{\text{init1}}, M_{\text{final1}}) = \lambda_1(\mathcal{L}(N_1', M_{\text{init1}}, M_{\text{final1}}))$ to deduce the desired statement.

Finally, we show that $\lambda_1(\overline{\mathcal{R}})$ is disjoint from $\mathcal{L}(N_2, M_{\text{init2}}, M_{\text{final2}})$. Towards a contradiction, assume that there is a word $w \in \Sigma^*$ in the intersection of the languages. On the one hand, this means that there is some sequence $\tau \in \overline{\mathcal{R}}$ with $\lambda_1(\tau) = w$. Note that this sequence consists of (the names of) transitions of N_1. On the other hand, there is a covering computation $M_{\text{init2}} \boxed{\sigma} M \geqslant M_{\text{final2}}$ of net N_2 with $\lambda_2(\sigma) = w$. We turn this computation into a computation σ_τ of N_2' as follows: We replace the i^{th} transition $\sigma_i \in T_2$ of N_2 by $\sigma_i(\tau_i)$, the copy of transition σ_i in N_2' that is labeled by $\tau_i \in T_1$. This copy indeed exists because after applying the respective labeling functions, we have $\lambda_2(\sigma_i) = \lambda_1(\tau_i) = w_i$. The result is a sequence σ_τ of transitions of N_2' that still induces a covering computation, hence $\lambda_1(\sigma_\tau) \in \mathcal{L}(N_2', M_{\text{init2}}, M_{\text{final2}})$. Furthermore, the construction ensures that the labeling of σ_τ under the labeling function of N_2' is $\tau \in \overline{\mathcal{R}}$. This

is a contradiction to the assumption that $\overline{\mathcal{R}}$ and the language of N_2 are disjoint because $\mathcal{R}$ is a regular separator. ■

Unfortunately, applying complementation and a homomorphism has severe impact on the complexity. We will prove that for two Petri nets, one of them freely labeled, one can construct a separator with a doubly exponential state complexity. This means that we can represent its complement by a DFA with at most triply exponentially many states. Applying a homomorphism to this DFA introduces a polynomial blowup to its size, but we lose determinism. The final result is an NFA of triply exponential size, proving Theorem 14.1.1. A priori, the index of the separator may be quadruply exponential.

We will later prove a doubly exponential lower bound for the state complexity of regular separators for coverability languages of non-freely labeled Petri nets. This matches the upper bound in the case that one of the nets is freely labeled, but not in the general case. The author conjectures that the determinization construction in this section can be improved to avoid the exponential blowup introduced by the complement.

Enforcing completeness

The definition of deterministic WSTSes imposes two restrictions on the transition relation, uniqueness and completeness. The above construction ensures that each marking has at most a-labeled successor in the WSTS associated to a Petri nets. To get determinism, we need also need to guarantee that there is at least one a-labeled successor. Luckily, enforcing completeness is typically much easier than enforcing uniqueness. For example, an NFA can be completed by introducing an additional *error state*. A similar trick works in the case of WSTSes.

Let $W = (\Gamma, \leqslant, T, \Gamma_{\text{init}}, \Gamma_{\text{final}})$ be a labeled WSTS over some alphabet Σ. We define the WSTS $W' = (\Gamma \uplus \{\bot\}, \leqslant, T', \Gamma_{\text{init}}, \Gamma_{\text{final}})$ that is obtained by adding a special bottom configuration $\bot$. The order $\leqslant$ on Γ is extended to $\Gamma \uplus \{\bot\}$ by defining $\bot \leqslant c$ for all $c \in \Gamma \uplus \{\bot\}$. This order is a WQO assuming that $(\Gamma, \leqslant)$ is a WQO. The new configuration is neither initial nor final.

The transition relation T' is a superset of T defined as follows. If $c \in \Gamma$ is a configuration that has no a-successor in T, then we have $c \xrightarrow{a} \bot$ in T'. Additionally, we have $\bot \xrightarrow{a} \bot$ in T' for all letters a. Upward compatibility is satisfied since $\bot \leqslant d$ for all $d \in \Gamma$. The definition of T' ensures completeness, i.e. each configuration has at least one a-successor. If T is unique, then T' is deterministic.

Since the new configuration $\bot$ is absorbing, meaning a computation that enters it cannot leave, but it is not final, we have $\mathcal{L}(W) = \mathcal{L}(W')$. If W is the WSTS associated to a freely labeled Petri net, then W' is a deterministic WSTS with the same language. With respect to regular separability, the two can be used interchangeably.

When considering the size of the separator, however, it is important to understand the implications that the construction of W' has on the ideal completion. If $\mathcal{J}$ is an ideal in $(\Gamma \uplus \{\bot\}, \leqslant)$,

then it is of the shape $\mathcal{I} \uplus \{\bot\}$, where $\mathcal{I} \subseteq \Gamma$ is an ideal of $(\Gamma, \leqslant)$ or the empty set. Similarly, if $\mathcal{I}$ is an ideal of $(\Gamma, \leqslant)$, then $\mathcal{I} \uplus \{\bot\}$ is an ideal of $(\Gamma \uplus \{\bot\}, \leqslant)$. To see that both statements are true, we use the fact that $\bot$ is a bottom element, which implies that it is contained in every non-empty downward-closed subset.

Consequently, if X is an invariant for W, and its ideal decomposition $\mathrm{decomp}_\Gamma(X)$ has size k, then $X \cup \{\bot\}$ is an invariant for W' and its ideal decomposition has size $k+1$. The latter consists if the ideals $\mathcal{I} \cup \{\bot\}$ where $\mathcal{I} \in \mathrm{decomp}_\Gamma(X)$ and the additional ideal $\{\bot\}$. This means that applying the processing that makes a WSTS complete only adds a single ideal to the ideal decomposition of an invariant.

An upper bound assuming determinism

The final step in proving Theorem 14.1.1 is showing that if N_1 and N_2 are language-disjoint Petri nets where N_1 is freely labeled, then their languages can be separated by the language of an NFA with doubly exponentially many states. To this end, we will consider the size of the ideal decomposition of an invariant for the products of the nets.

The main ingredient to showing a doubly exponential bound for the size of this invariant is the following result by Bozzelli and Ganty [BG11]. It provides a doubly exponential bound on the size of a representation of $\mathrm{reach}_{W_N}^{-1}$, where W_N is the WSTS associated to a Petri net N. The result has been obtained by inspecting Abdulla's backward search [ACJT96].

14.1.4 Theorem (Bozzelli and Ganty [BG11])
Consider a Petri net instance $(N, M_{\mathrm{init}}, M_{\mathrm{final}})$ and the associated WSTS W_N. One can construct a representation $\mathrm{reach}_{W_N}^{-1} = \{M_1, \ldots, M_k\}\uparrow$, where k as well as all $||M_i||_\infty$ are bounded by

$$g = (|T| \cdot (||N||_\infty + ||M_{\mathrm{init}}||_\infty + ||M_{\mathrm{final}}||_\infty + 2))^{2^{\mathcal{O}(|P| \cdot \log |P|)}}.$$

Here, $||N||_\infty = \max\{||\mathrm{in}||_\infty, ||\mathrm{out}||_\infty\}$ is the maximum multiplicity of any transition. Our challenge in the following will be to convert this bound on a representation of the upward closed set $\mathrm{reach}_{W_N}^{-1}$ into a bound on the size of the ideal decomposition of its complement $\mathbb{N}^P \setminus \mathrm{reach}_{W_N}^{-1}$. This set is guaranteed to be an invariant since W_N has empty language.

We have already briefly mentioned in Section 13.3 that the ideals in $\mathbb{N}^d$ are of the shape $M\downarrow$, where $M \in \mathbb{N}_\omega^d$ is a generalized marking. In this section, we will additionally need to consider (1) how to construct a representation of the intersection of two ideals, and (2) how to construct the ideal decomposition of the set $\mathbb{N}^d \setminus M\uparrow$, where M is a marking. The following lemma states the well-known solution to both problems.

14.1.5 Lemma (see e.g. Lazic and Schmitz [LS15a])

a) For ideals $M_1{\downarrow}, M_2{\downarrow}$ of $\mathbb{N}^d$, their intersection $M_1{\downarrow} \cap M_2{\downarrow}$ is the ideal $M{\downarrow}$ where M is specified by $M(i) = \min\{M_1(i), M_2(i)\}$ for all $i \in [1, d]$.

b) For a marking $M \in \mathbb{N}^d$, the ideal decomposition of $\mathbb{N}^d \setminus M{\uparrow}$ is $\{M_i{\downarrow} \mid i \in [1, d]\}$ where $M_i(i) = M(i) - 1$ and $M_i(j) = \omega$ for $j \neq i$.

The proof is straightforward in both cases. The representation of the intersection simply formalizes the fact that an element of the intersection needs to be smaller in every component than the representations of the intersected ideals. For b), each ideal in the ideal decomposition of $\mathbb{N}^d \setminus M{\uparrow}$ describes all markings that are smaller than M in some component, and hence contained in the complement of $M{\uparrow}$.

With the preliminaries at hand, we can state and prove the main result.

14.1.6 Proposition
Let N_1, N_2 be languages-disjoint Petri nets, and let $W_\times$ be the WSTS associated to their product. The ideal decomposition of the invariant $\mathbb{N}^P \setminus \mathrm{reach}_{W_N}^{-1}$ has size at most doubly exponential.

Proof:
Let $(N_1, M_{\text{init}1}, M_{\text{final}1})$ and $(N_2, M_{\text{init}2}, M_{\text{final}2})$ be the given Petri net instances. Let $W_\times$ be the WSTS associated to the product of the nets, and let d be the total number of places. Since the two Petri nets are language disjoint, $X = \mathbb{N}^d \setminus \mathrm{reach}_{W_\times}^{-1}$ is a safe inductive invariant, see Section 13.2. Hence, the downward closure $Y{\downarrow}$ of its ideal decomposition $Y = \mathrm{decomp}(X)$ is a finitely represented invariant in the ideal completion by Proposition 13.3.8.

With Theorem 14.1.4 applied to $W_\times$, we have $\mathrm{reach}_{W_\times}^{-1} = \{M_1, \ldots, M_k\}{\uparrow}$ for suitable markings $M_1, \ldots, M_k$. We may write

$$X = \mathbb{N}^d \setminus \mathrm{reach}_{W_\times}^{-1} = \mathbb{N}^d \setminus \{M_1, \ldots, M_k\}{\uparrow} = \mathbb{N}^d \setminus \bigcup_{i \in [1,k]} M_i{\uparrow} = \bigcap_{i \in [1,k]} \mathbb{N}^d \setminus M_i{\uparrow} \ .$$

With Part b) of Lemma 14.1.5, we can rewrite each $\mathbb{N}^d \setminus M_i {\uparrow}$ to $\bigcup_{j \in [1,d]} M_{i,j} {\downarrow}$, where $M_{i,j}(j) = M_i(j) - 1$ and $M_{i,j}(\ell) = \omega$ for $\ell \neq j$. Altogether, we have

$$X = \bigcap_{i \in [1,k]} \bigcup_{j_i \in [1,d]} M_{i,j_i}{\downarrow} \ .$$

We use distributivity to swap the intersection and the union, resulting in

$$X = \bigcup_{\vec{j} \in [1,d]^k} \bigcap_{i \in [1,k]} M_{i,\vec{j}(i)}{\downarrow} \ .$$

We can use Part a) of Lemma 14.1.5 to compute for each $\vec{j}$ a single ideal $M_{\vec{j}}$ with $M_{\vec{j}}{\downarrow} = \bigcap_{i\in[1,k]} M_{i,\vec{j}(i)}{\downarrow}$. Altogether, we obtain $X = \bigcup_{\vec{j}\in[1,d]^k} M_{\vec{j}}{\downarrow}$. Some ideals in this union might be redundant, but in any case, we obtain that the ideal decomposition of X consists of at most d^k ideals. Note that we have $k \leqslant g$, where g is the doubly exponential bound specified in Theorem 14.1.4. Unfortunately, this only provides a triply exponential bound d^g.

To get the desired bound, we use that Theorem 14.1.4 also specifies $||M_i||_\infty \leqslant g$ for all i. By the construction of the generalized markings $M_{i,j}$, we have $||M_{i,j}||_\infty \leqslant ||M_i||_\infty \leqslant g$, where we extend the infinity norm to generalized markings by treating ω-components as zero. Finally, we have that each $M_{\vec{j}}$ which represents the intersection of some of the $M_{i,j}{\downarrow}$ also satisfies $||M_{\vec{j}}||_\infty \leqslant g$ by definition. Hence, any non-ω component of $M_{\vec{j}}$ is bounded by g from above.

There are only $(g+2)^d$ different generalized d-dimensional marking in which all non-ω components are bounded by g. Hence, after removing redundant ideals, the ideal decomposition of X has size at most $(g+2)^d$. Inserting the definition of g from Theorem 14.1.4 yields the bound

$$|Y| \leqslant (g+2)^d = \left(\left(|T_\times| \cdot \left(||N_\times||_\infty + ||M_{\text{init}\times}||_\infty + ||M_{\text{final}\times}||_\infty + 2 \right) \right)^{2^{\mathcal{O}(|P_\times| \cdot \log |P|)}} + 2 \right)^d .$$

Here $T_\times, N_\times$ and so on specify the components of the product net of N_1 and N_2 that induces the WSTS $W_\times$. Using the power laws, the number $\left(|T_\times| \cdot \left(||N_\times||_\infty + ||M_{\text{init}\times}||_\infty + ||M_{\text{final}\times}||_\infty + 2 \right) \right)^{2^{\mathcal{O}(|P_\times| \cdot \log |P|)}}$ is at most exponential in $|N_1| + |N_2|$, even if $||N_\times||_\infty$, $||M_{\text{init}\times}||_\infty$, and $||M_{\text{final}\times}||_\infty$ may already be exponential in $|N_1| + |N_2|$. Hence, we get the desired doubly exponential bound. ■

14.1.7 Remark

The above proof of Proposition 14.1.6 using the result by Bozzelli and Ganty [BG11] is a modified version of the proof in the original publication [CLMMKS18; CLMMKS18a]. With the results by Lazic and Schmitz [LS15a], Corollary 4.6 in particular, a simpler proof would be possible. The authors of that paper show that one can compute the invariant $\mathbb{N}^d \setminus \text{reach}_{W_N}^{-1}$ as the fixed point of the descending chain

$$\mathbb{N}^d \setminus M_{\text{final}}{\uparrow} \supseteq \mathbb{N}^d \setminus \text{pre}\left(\Sigma^{\leqslant 1}, M_{\text{final}}{\uparrow}\right) \supseteq \mathbb{N}^d \setminus \text{pre}\left(\Sigma^{\leqslant 2}, M_{\text{final}}{\uparrow}\right) \supseteq \ldots ,$$

based on a technique that uses ideals. They establish a doubly exponential upper bound for the smallest number i such that $\mathbb{N}^d \setminus \text{reach}_{W_N}^{-1} = \mathbb{N}^d \setminus \text{pre}(\Sigma^{\leqslant i}, M_{\text{final}}{\uparrow},)$ and mention that this implies that the ideal decomposition of the invariant is also of doubly exponential size.

Recall that our upper bound, Theorem 14.1.1, states that two labeled Petri net instances with disjoint languages can be separated by a regular language with triply exponential state complexity. We can now assemble its proof by using the various techniques that we have presented in this section.

Proof of Theorem 14.1.1:
Let N_1, N_2 be two given language-disjoint Petri nets. We use the above construction to eliminate all ε-transitions in N_2, resulting in the nets N_1', N_2'. We then consider N_1'', the freely labeled variant of N_1', and N_2'', the corresponding modification of N_2'. Note that being freely labeled also implies that N_1' has no ε-transitions. The techniques that we have applied preserve language-disjointness, so the intersection of the languages of N_1'' and N_2'' are disjoint. This means that the language of the WSTS $W_\times$ associated to their product is empty, and the set $\mathbb{N}^d \setminus \mathrm{reach}_{W_\times}^{-1}(M_{\mathrm{final}\times}\uparrow)$ is an inductive invariant. With Proposition 14.1.6, the size of the ideal decomposition of this invariant is at most doubly exponential.

The transition relation of the WSTS associated to N_1'' is unique, but not complete. However, we may apply a preprocessing step as described above and obtain a deterministic WSTS $W_{N_1''\mathrm{complete}}$ with the same language. Compared to the original WSTS, this WSTS has one additional ideal. This means that the ideal decomposition of the invariant for $W_\times$ translates into an ideal decomposition of an invariant for the product of $W_{N_1''\mathrm{complete}}$ and $W_{N_2''}$ that is larger only by a polynomial factor. This product satisfies the assumption of Theorem 13.2.3, proving that the languages of $W_{N_1''\mathrm{complete}}$ and $W_{N_2''}$ can be separated by the language of an NFA with doubly exponentially many states.

Applying Proposition 14.1.3 yields that the languages of N_1' and N_2' can be separated by a regular separator with triply exponential state complexity. Finally, we apply the construction from the proof of Lemma 14.1.2 to obtain separator for the languages of the original nets N_1 and N_2 with the desired triply exponential state complexity. This completes the proof. ■

14.2 Lower bound

We complement our findings on the regular separability of Petri net coverability languages by a lower bound.

14.2.1 Theorem
Regular separators for Petri net coverability languages may have doubly exponential state complexity and triply exponential index.

The lower bound does not match the upper bound: Theorem 14.1.1 provides a separator with triply exponential state complexity (and hence quadruply exponential index). We have already commented on the fact that this additional exponent is caused by the construction that we have introduced for determinizing one of the nets. The author conjectures that using a suitable technique, this additional blowup can be avoided. This would imply that the lower bound that we are going to prove in this section is optimal.

To prove the theorem, we show a proposition that provides for each $n \in \mathbb{N}$ two language-disjoint Petri nets of size polynomial in n. The regular separators of their coverability languages can be shown to have minimal state complexities and indices as required by the lower bound.

14.2.2 Proposition
For all $n \in \mathbb{N}$, there are Petri nets $N_0(n), N_1(n)$ of size polynomial in n with disjoint languages such that any regular separator for the coverability languages $\mathcal{L}(N_0(n))$ and $\mathcal{L}(N_1(n))$ has state complexity at least 2^{2^n} and index at least $2^{2^{2^n}}$.

The statement on the index is a strictly stronger statement than the one on the state complexity. If we show that any DFA separating the languages has at least 2^k states, this implies that any NFA with this property has at least k states. Otherwise, we could take an NFA with less than k states and determinize it to obtain a DFA with less than 2^k states. Hence, it will be sufficient to prove the statement on the index of a separator in the following.

The proof consists of two ingredients. The first ingredient are two special Petri net instances $(N_{\text{inc}}, M_{\text{initinc}}, \vec{0})$ and $(N_{\text{dec}}, \vec{0}, M_{\text{finaldec}})$ that come from Lipton's proof of the EXPSPACE-hardness of coverability [Lip76]. We have stated the properties of these nets in Proposition 6.2.3.

With these two nets, it would be possible to show that the languages $\{a^m \mid m < k\}$ and $\{a^m \mid m \geqslant k\}$ for $k = 2^{2^n}$ are the coverability languages of Petri nets of size polynomial in n. These languages are regular languages that are the complement of each other and have state complexity k. Hence, they could be used to show the statement on the state complexity of separators.

To also show the stronger statement on the index of separators requires a second ingredient. It is the classical result that a language that can be represented by a *small* NFA might need a *large* DFA that we have briefly mentioned in Example 4.3.2. We consider the binary alphabet $\{0, 1\}$. For $x \in \{0, 1\}$ and a number $k \in \mathbb{N}$, $k > 0$, the language $\mathcal{L}_{x@k}$ is the set of all words whose k-last letter is x,

$$\mathcal{L}_{x@k} = \left\{ w \in \{0, 1\}^{\geq k} \;\middle|\; w_{|w|-(k-1)} = x \right\}.$$

It is well-known that this language has state complexity $k + 1$, but index 2^k. An NFA for the language can guess the occurrence of the k-last letter, check that it is indeed x, and then verify that it is followed by exactly $k - 1$ letters. A DFA for the language, however, cannot the occurrence of the k-last letter. It needs to store at each point in time the k last letters, which requires 2^k states. Kozen [Koz06] mentions this example without proof. We will formally prove it in the context of the proof of Proposition 14.2.2.

Combining the two ingredients yields two Petri nets $N_0(n), N_1(n)$ such that the language of $N_x(n)$ is essentially $\mathcal{L}_{x@k}$ for the doubly exponential number $k = 2^{2^n}$. This means that any DFA whose language separates the two coverability languages is a DFA for $\mathcal{L}_{x@k}$, which means that it needs at least 2^k states.

By using ε-transitions, we could enforce that the language of $N_x(n)$ equals $\mathcal{L}_{x@k}$. To show that the lower bound holds even if we do not allow ε-transitions, we consider the extended alphabet $\{0, 1, a, b\}$. The language of net $N_x(n)$ will be a subset of $a^*.b.\mathcal{L}_{x@k}.b.a^*$, where the prefix a^* and the suffix a^* correspond to a computation of N_{inc} and N_{dec}, respectively. Since the prefix and the suffix are the same for $N_0(n)$ and $N_1(n)$, they do not influence the size of the separator. In the following, we will first explain the construction of the nets $N_x(n)$ and then prove that they satisfy the required properties.

The construction of $N_x(n)$

We state the construction of the Petri net instance $(N_x(n), M_{\text{init}}, M_{\text{final}})$. It is parametric in the size n and in $x \in \{0, 1\}$. Let $(N_{\text{inc}}, M_{\text{initinc}}, \vec{0})$ and $(N_{\text{dec}}, \vec{0}, M_{\text{finaldec}})$ be the Petri net instances for the chosen n whose properties are specified in Proposition 6.2.3. These nets are independent of x; the rest of the construction will be independent of n. The net $N_x(n)$ consists of the disjoint union of N_{inc}, N_{dec}, and a constant number of additional places and transitions. It is depicted schematically, i.e. with the internal behavior of N_{inc} and N_{dec} hidden, in Figure 14.2.a.

The places of $N_x(n)$ are the places of N_{inc}, the places of N_{dec}, and four places $p_1, \ldots, p_4$. These additional places act as control states: The computations of the net will proceed in four phases such that in phase i, place p_i carries one token and the places p_j for $j \neq i$ do not carry any tokens.

The initial marking assigns M_{initinc} to the places of N_{inc}, one token to p_1 and no token elsewhere. The final marking that should be covered requires a token on p_4 and requires tokens as specified by M_{finaldec} on the places of N_{dec}.

Instead of formally specifying the transitions of $N_x(n)$, we describe the phases that form its computations. The transitions that can be used in each phase are disjoint. Each transition t that is used in phase i checks that the control place p_i carries a token, i.e. $\mathrm{in}(t, p_i) = \mathrm{out}(t, p_i) = 1$.

1. The first phase is a computation of N_{inc}. We assume that all transitions of N_{inc} are labeled by a and that they check the existence of a token on p_1.

   Recall that the copy of $N_{\mathrm{inc}}(n)$ is initialized with the correct initial marking. From Proposition 6.2.3 we get that N_{inc} has two special places $p_{\mathrm{haltinc}}, p_{\mathrm{outinc}}$ such that if the computation reaches a marking that assigns a token to p_{haltinc}, p_{outinc} carries exactly 2^{2^n} tokens.

   A b-labeled transition signals the beginning of the second phase. It moves a token from p_1 to p_2 and consumes a token from p_{haltinc}. This means that p_{outinc} carries 2^{2^n} tokens.

2. The second phase uses two transitions that are labeled by 0 and 1, respectively. Besides checking for the control token on p_2, they have no effect.

   Intuitively, the second phase generates a prefix of a word from $\mathcal{L}_{x@k} \subseteq \{0,1\}^*$, namely all but the last k letters of a word from this language.

x. The second phase ends with an x-labeled transition that moves the control from p_2 to p_3. It also moves one token from the place p_{outinc} of N_{inc} to the place p_{indec} of N_{dec}. Note that this is the only transition that depends on $x \in \{0,1\}$, i.e. it is the only part of the construction in which $N_0(n)$ and $N_1(n)$ differ.

   Intuitively, firing this transition corresponds to the k-last letter of a word from $\mathcal{L}_{x@k}$.

3. The third phase consist of two transitions labeled by 0 and 1, respectively. These transitions check for the control token on p_3 and move one token from p_{outinc} to p_{indec}.

   Intuitively, the third phase generates the suffix of length $k-1$ of a word from $\mathcal{L}_{x@k}$.

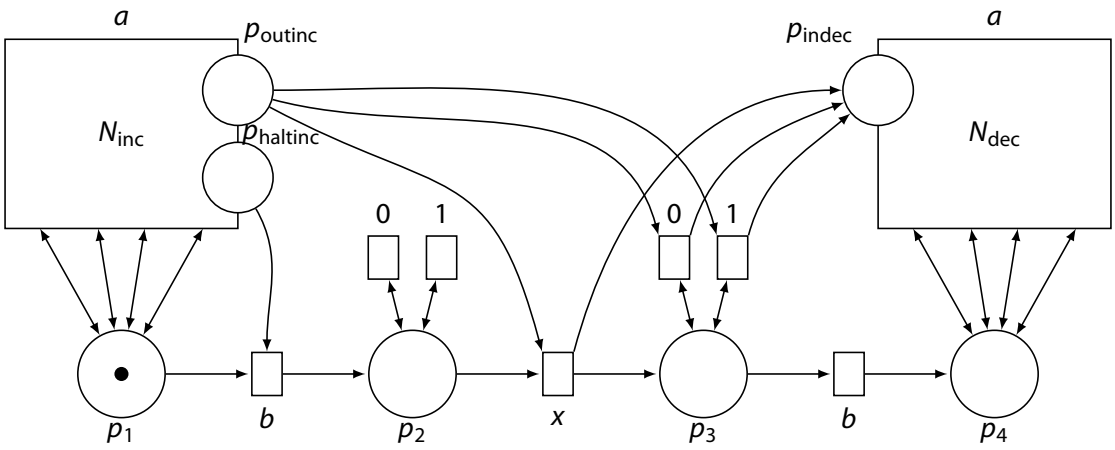


Figure 14.2.a: A schematic representation of the Petri net $N_x(n)$.

The last phase starts with a b-labeled transition that moves the control from p_3 to p_4.

4. The last phase is essentially a computation of N_{dec}. We assume that all transitions of N_{dec} are labeled by a and that they check the existence of a token on p_4.

   Recall that a computation of N_{dec} can only cover the final marking M_{finaldec} if initially, we had at least 2^{2^n} tokens on p_{indec}.

We claim that the language of $N_x(n)$ is

$$\mathcal{L}(N_x(n), M_{\text{init}}, M_{\text{final}}) = \mathcal{L}_{\text{inc}}.b.\mathcal{L}_{x@2^{2^n}}.b.\mathcal{L}_{\text{dec}}$$

where $\mathcal{L}_{\text{inc}}, \mathcal{L}_{\text{dec}} \subseteq a^*$ correspond to computations of $N_{\text{inc}}, N_{\text{dec}}$.

By the description of the phases of $N_x(n)$ above, any word in the language of $N_x(n)$ has the shape $w_{\text{inc}}.b.w.b.w_{\text{dec}}$ where $w_{\text{inc}}, w_{\text{dec}} \in a^*$. To see that we also have $w \in \mathcal{L}_{x@k}$, we argue that the structure of the net ensures that the k-last letter of w is x. We may write $w'.x.w''$, where $w', w'' \in \{0,1\}^*$ correspond to the letters generates by Phase 2 and Phase 3, respectively. To complete the argument, we show that w'' has length $k-1$.

At the beginning of Phase 2, p_{outinc} carries $k = 2^{2^n}$ tokens and p_{indec} carries no token. At the beginning of Phase 3, i.e. after generating $w'.x$, place p_{outinc} carries $k-1$ tokens and p_{indec} carries one token. The final marking of N_{dec}, and hence the final marking of $N_x(n)$, can only be covered if p_{indec} carries k token at the beginning of Phase 4. Thus, w'' needs to have length at least length $k-1$ to move at least $k-1$ tokens from p_{outinc} to p_{indec}. Since there are only $k-1$ tokens on available on p_{outinc}, this also limits the length of w'' by k_1.

We have shown that any word from the language of $N_x(n)$ is of the shape $a^*.b.\mathcal{L}_{x@2^{2^n}}.b.a^*$. For the other direction, one would need to argue that for any word $w \in \mathcal{L}_{x@2^{2^n}}$, the word $w_{\text{inc}}.b.w.b.w_{\text{dec}}$ obtained by pre- and appending suitable pre- and suffixes is in the language of $N_x(n)$. This can be shown easily by picking suitable transitions in the Phases 2 and 3.

To finish the proof of Proposition 14.2.2, we have to show that the languages of $N_0(n)$ and $N_1(n)$ cannot be separated by the regular language of index less than triply exponentially. Since the languages share the prefix $a^*.b$ and the suffix $b.a^*$, this boils down to proving that $\mathcal{L}_{1@2^{2^n}}$ and $\mathcal{L}_{0@2^{2^n}}$ cannot be separated by a DFA with less than triply exponentially many states. We proceed to show this using the well-known fooling-set technique [Bir92; Bir93].

Proof of Proposition 14.2.2:
Let $k = 2^{2^n}$ and assume that $\mathcal{L}(A)$ is the language of a DFA with strictly less than 2^k states that separates the given languages, i.e. $\mathcal{L}(N_0(n), M_{\text{init}}, M_{\text{final}}) \subseteq \mathcal{L}(A)$ and $\mathcal{L}(N_1(n), M_{\text{init}}, M_{\text{final}}) \cap \mathcal{L}(A) = \varnothing$. Towards a contradiction, we consider the set of words $\{0,1\}^k$. Since its size is larger than the number of states of the DFA A, there are distinct words that lead to the same state, a fact that we will exploit later.

Let $w_{\text{inc}} \in a^*$ be a word generated by a computation of N_{inc} that creates a token on p_{haltinc}. Similarly, let $w_{\text{dec}} \in a^*$ be a word generated by a covering computation of N_{dec}, assuming that it starts from a marking that places k tokens on p_{indec}.

We use the set of words $\{w_{\text{inc}}.b.w\}w \in \{0,1\}^k$ as the fooling set. For each word $w_{\text{inc}}.b.w$, let q_w be the unique state in which A is after reading it. Since $|\{0,1\}^k| = 2^k$ is strictly larger than the number of states of A, there are two distinct words w, w' with $w \neq w'$ such that $q_w = q_{w'}$. Since $w \neq w'$, there is some position i with $w_i \neq w'_i$. Wlog., we assume $w_i = 0, w'_i = 1$. We define $w_{\text{fill}} = 0^{i-1}$ as a suitable suffix such that the i^{th} position of w and w' becomes the k-last position in $w.w_{\text{fill}}$ and $w'.w_{\text{fill}}$. In particular, we have $w.w_{\text{fill}} \in \mathcal{L}_{0@k}$, $w'.w_{\text{fill}} \in \mathcal{L}_{1@k}$. This implies

$$w_{\text{inc}}.b.w.w_{\text{fill}}.b.w_{\text{dec}} \in \mathcal{L}(N_0(n), M_{\text{init}}, M_{\text{final}}) \subseteq \mathcal{L}(A)\,,$$

meaning that the unique run of A on that word is accepting. Since the state q_w after reading $w_{\text{inc}}.b.w$ respectively $w.b.w'$ is the same and the suffixes $w_{\text{fill}}.b.w_{\text{dec}}$ are equal, we also get

$$w_{\text{inc}}.b.w'.w_{\text{fill}}.b.w_{\text{dec}} \in \mathcal{L}(A)\,.$$

However, this word contained in $\mathcal{L}(N_1(n), M_{\text{init}}, M_{\text{final}})$, a contradiction to the assumption that A is a separating automaton with $\mathcal{L}(A) \cap \mathcal{L}(N_1(n), M_{\text{init}}, M_{\text{final}}) = \varnothing$. ■

With the proof of the lower bound completed, all results outlined in Section 11.3 have been shown. Our study of the regular separability of WSTS languages has been completed.

Part V.

Games

Contents

15 Games with perfect information 275

16 Effective denotational semantics 285
16.1 Systems of equations and domains . . . 287
16.2 Effective denotational semantics for regular inclusion . . . 298
16.3 ω-regular inclusion . . . 303

17 Context-free games 313
17.1 Context-free regular inclusion games . . . 314
17.2 Solving context-free games . . . 316
17.3 Computing the winner . . . 327
17.4 Computing the strategies . . . 333
17.5 Complexity . . . 339
17.6 Related work . . . 353
17.7 Algorithmics . . . 357
17.8 Deterministic target languages . . . 363
17.9 ω-context-free inclusion games . . . 366

18 Higher-order games 375
18.1 Higher-order games . . . 376
18.2 A model template for deterministic schemes . . . 378
18.3 Fixed-point semantics for higher-order games . . . 389
18.4 Framework for exact fixed-point transfer . . . 403
18.5 Solving higher-order inclusion games . . . 417

19 Valence games 433
19.1 Valence systems over graph monoids . . . 433
19.2 Valence games . . . 441
19.3 Bounded context switching for valence systems . . . 463

Part V.
Games

This part of the thesis is concerned with solving games. We have argued in Section 1.3 of the introduction that games whose game arenas are defined by automata are related to the area of program verification. In particular, the task of a solving synthesis problem can be approached by translating it into an equivalent game.

Outline

We start by formally define board games for two players with perfect information in Chapter 15. We discuss the intricacies that arise from considering inclusion games where the winning condition is membership in a given target language.

Before we actually turn to solving games, we explain effective denotational semantics in Chapter 16. Effective denotational semantics is an approach to solving decision problems that is based on computing the least solution to a system of equations, and it will be our preferred course of action throughout the rest of this part. We demonstrate how to use effective denotational semantics by applying it to the regular inclusion problem for context-free languages and the ω-regular inclusion problem for ω-context-free languages.

In Chapter 17, we consider the problem of solving context-free inclusion games, games whose game arena is defined by a context-free grammar and whose winning condition is membership in a regular target language. We present an algorithm solving such games with the optimal time complexity that is based on effective denotational semantics.

In Chapter 18, we extend this approach to games defined by higher-order recursion schemes. In addition to solving such games, we study effective denotational semantics for verification problems based on higher-order schemes on a more general level, proving an exact fixed-point transfer result that is of independent interest.

Finally, Chapter 19 is concerned with studying the frontier of the decidability of games. To this end, we use the model of valence systems, a type of automaton that generalizes well-known models like pushdown automata and Petri nets. We will prove a classification result that shows that context-free games are essentially the only decidable type of reachability games that can be modeled using valence systems over graph monoids. We propose using bounded context switching as an approach to obtain decidability in a wider range of cases.

Publication

This part of the thesis is based on the publications [HMM16] (resp. its full version [HMM16a]), [HMM17] (resp. its full version [HMM17a]), [MMZ18] (resp. its full version [MMZ18a]), and [MMN17]. At the beginning of each chapter, we will provide more detailed information about which paper the chapter is based on. In Chapter 20, we will discuss the contribution the author made to these publications.

15 Games with perfect information

The goal of this part of the thesis is studying certificate-generating algorithms for solving games. We will use this chapter to give some basic definitions regarding games and to introduce the corresponding notation. Most of the material is standard, it can be found e.g. in [KN01], but the presentation does not follow any particular source.

Games with perfect information

A *game* is a system whose behavior is influenced by several independent entities, called *players*. Games can be divided into two categories, namely games with perfect and games with imperfect information. In a game with perfect information, each of the players knows the rules of the game, and, whenever it is their turn to pick a move, they know the history and the current state of the game. We also require perfect-information games to be sequential; we do not allow concurrent games in which multiple players have to make a choice simultaneously like in the well-known *prisoner's dilemma* [Pou92]. We will exclusively consider perfect-information games in this thesis since they provide a sufficiently expressive model for the problems in verification and synthesis that we are interested in. Games with imperfect information, which are predominantly studied in economic sciences for their capability of modeling a market, and the corresponding notions like equilibria are beyond the scope of this thesis.

A consequence of the players having perfect information is that it is typically sufficient to consider at most two players. Assume that in a k-player game, there is a global winning condition that a coalition of players tries to satisfy, while the opposition, the rest of players, is trying to prevent this from happening. This game is equivalent to the two-player game in which all players in the coalition and opposition have been merged into a single player each. Here, it is crucial that the players do not have any private information that is hidden from the other players in the same group.

Finally, we require games to be discrete in the sense that they are turn-based. This allows us to model games as *board games*, i.e. based on directed graphs. The nodes of the graph represent states of the game and the choices of a player in a certain state are represented by the choice among the outgoing edges of the corresponding node.

In Section 1.3 of the introduction, we have already given several examples of where such games play a role in computer science. We have discussed a proof of Rabin's tree theorem that relies on games, we have briefly mentioned game semantics, and we have talked extensively about how games can be used to model synthesis problems.

Basic definitions

We turn to providing formal definitions for the aforementioned concepts. A *sequential two-player board game with perfect information*, shortly referred to as *game*, consists of a *game arena* and a *winning condition*. A *game arena* is a transition system $S = (\Gamma, T)$ together with a function

$$\text{owner}\colon \Gamma \to \{\bigcirc, \square\}$$

that assigns each configuration its owner, either the *universal player* $\square$ or the *existential player* $\bigcirc$. In the context of games, we often refer to configurations $c \in \Gamma$ as *positions* and to transitions $t \in T \subseteq \Gamma \times \Gamma$ as *moves* of the game.

The owner function induces a partitioning of the positions $\Gamma = \Gamma_\square \uplus \Gamma_\bigcirc$ into the sets $\Gamma_\square = \{c \in \Gamma \mid \text{owner}(c) = \square\}$ and $\Gamma_\bigcirc = \{c \in \Gamma \mid \text{owner}(c) = \bigcirc\}$. We usually write a game arena as $S = (\Gamma_\square \uplus \Gamma_\bigcirc, T)$, where the owner function is implicitly given by the partitioning of the set of all positions $\Gamma = \Gamma_\square \uplus \Gamma_\bigcirc$.

Before we can formally define winning conditions, we need to understand how a game is played. A *play* (from position c_{init}) is a finite or infinite path in the game arena that starts with c_{init}. Formally, an infinite play is a sequence of positions $p = p_0 p_1 \ldots$ such $p_i \in \Gamma$ for all i, $p_0 = c_{\text{init}}$, and for each $i \in \mathbb{N}$, $(p_i, p_{i+1}) \in T$ is a valid move in the game. For finite plays, the definition is adapted accordingly. Intuitively, we think of a token that is initially placed on position c_{init}. The game then proceeds in sequential steps. In each step, the owner of the current position c is active and can move the token from c to a new position by picking a move of the game that originates in c.

We call a play *maximal* if it is either infinite or it is finite and ends in a position that is a *deadlock*, i.e. it has no successors in the arena. The set $\text{Plays}_{\text{max}} \subseteq \Gamma^* \cup \Gamma^\omega$ is the set of all such plays. Note that here, we use Γ^* and Γ^ω to refer to the set of finite and infinite sequences of positions, respectively, even if Γ is not finite. For a finite play p, we use the notation p_{last} to denote the position $p_{|p|-1}$.

A *winning condition* is a function $\text{win}\colon \text{Plays}_{\text{max}} \to \{\bigcirc, \square\}$ that assign to each maximal play a winner. A *game* is of the shape (G, win) where G is a game arena and win is a winning condition for plays on that arena.

The winning condition allows us to determine the winner of a play. Our main interest, however, is to determine whether one of the players has a systematic way of playing so that she wins all resulting plays, independent of the choices of her opponent. If so, we say that this player *wins* the game. To formalize this idea of playing systematically, we introduce the notion of strategies. A *strategy* for player $\bigstar \in \{\bigcirc, \square\}$ is a function $s_\bigstar$ that takes a finite play p that ends in an active position of player $\bigstar$ and assigns a successor of the current position. Formally, we can see it as a function $s_\bigstar\colon \Gamma^* \Gamma_\bigstar \to \Gamma$ such that if p is a finite play where $p_{\text{last}} \in \Gamma_\bigstar$ is not a deadlock, then $(p_{\text{last}}, s_\bigstar(p)) \in T$ is a valid move. Intuitively, a strategy tells a player whenever she is active

which move she should make. Accordingly, we say that a play p *conforms* to strategy $s_{\star}$ if for all p_i with $p_i \in \Gamma_{\star}$, $p_i \neq p_{\text{last}}$, we have $p_{i+1} = s_{\star}(p_0 \dots p_i)$.

A strategy $s_{\star}$ is called a *winning strategy* for a game from position c_{init} if any play that starts in c_{init} and conforms to $s_{\star}$ is won by player $\star$. The set of positions c from which player $\star$ has a winning strategy is called the *winning region* $W_{\star}$ of that player.

It is easy to see that the winning regions of the two players have to be disjoint. Assume both players have a winning strategy from the same position. We can inductively construct the unique maximal play that conforms to both strategies by querying after each move the strategy for the active player. If both strategies are winning, the resulting play has to be winning for both players. This is a contradiction to the winning condition assigning a unique winner to all maximal plays. The question arises whether the set of positions is partitioned into the winning regions. A game that has this property, i.e. a game with $\Gamma = W_{\square} \cup W_{\bigcirc}$, is called *determined*. In theory, it is possible to construct undetermined games in which positions exist from which none of the players has a winning strategy [MS62]. However, all the games that we will consider in this thesis are determined, which can be proven using the Borel determinacy theorem [Mar75].

Hence, we can turn from the theoretical question of determinacy to more practical questions. Given a game and an initial position in that game, can we compute which player wins the game? When we talk about *solving* or *deciding* a game, we are interested in an algorithm that takes (a description of) the game arena and the winning condition. It should either also take a given initial position and return the winner from that position, i.e. the player who has a winning strategy, or it should produce a description of the winning regions of one or both of the players.

In addition to just computing the winner, we also want to compute the corresponding winning strategies. These strategies serve as certificates as explained in Section 1.3. This also means that we are interested in *simple* winning strategies. One particularly simple type of strategies are *positional strategies* that assign the next move only depending on the current position and not on the history. Formally, a strategy $s_{\star}$ is positional if for any two plays p, p' that end in the same position, $p_{\text{last}} = p'_{\text{last}}$, we have $s_{\star}(p) = s_{\star}(p')$. Note that we may see a positional strategy as a function with signature $s_{\star} \colon \Gamma_{\star} \to \Gamma$. A game is called *positionally determined* if for each initial position, exactly one of the players has a positional winning strategy. We will elaborate on other types of simple strategies at the end of this chapter.

Reachability games

We consider the two types of winning conditions that are most commonly considered in the literature and will play a large role in the rest of this part of the thesis, starting with the reachability winning condition.

A *reachability game* is given by a game arena together with a subset $\Gamma_{\text{final}} \subseteq \Gamma$ of the position, the so-called *target set*. A play p is won by the existential player if and only if it visits a position from that set, $\text{win}(p) = \bigcirc$ if there is $i \in \mathbb{N}$ such that $p_i \in \Gamma_{\text{final}}$. Reachability games are the

game-theoretic analogue of the usual acceptance condition in an automaton that processes finite words. However, infinite plays are allowed here, although any play that is won by ○ can already be identified as such after a finite prefix.

Reachability games are well-known to be positionally determined. In fact, both the winning regions and the positional winning strategies can be constructed using the well-known attractor construction.

The attractor of the target set Γ_{final} is the smallest subset of Γ that contains Γ_{final} and satisfies the following property. If a position owned by the existential player has a successor contained in the attractor, it is also contained in the attractor. The same is true for a position owned by the universal player that is not a deadlock and has all its successor contained in the attractor. For games with finite out-degree, the attractor can be constructed as the fixed point of an inductive backwards construction that starts with Γ_{final} and adds in each step all positions owned by ○ that have some successor in the set and all positions owned by □ that are not deadlocks and have all their successors in the set.

The attractor is exactly the set of positions from which the existential player can enforce visiting the target set within finitely many steps. Hence, it is her winning region and its complement is the winning region of the universal player. The corresponding winning strategy for the universal player simply picks for each position that is not in the attractor a successor that is also not in it. The definition of the attractor ensures that this is possible. The winning strategy for the existential player is more involved since it needs to ensure that Γ_{final} is reached after finitely many steps. To this end, if the play is currently in a position that was added to the attractor in the i^{th} step of the aforementioned inductive construction, the strategy needs to pick a successor that was added in the $(i-1)^{\text{st}}$ step of the construction or earlier. This ensures that the play eventually reaches a position that is contained in the zero-step attractor, which is the target set.

Parity games

In the same way that reaching a final state is an acceptance condition that is insufficiently expressive for automata that process infinite words, the reachability winning conditions is not useful whenever a winner should be assigned to an infinite play. Most of the acceptance conditions that have been defined for automata on infinite words have been used to define analogue winning conditions for games, including the Büchi, parity, and Muller conditions. Here, we exclusively consider the parity condition since it is very commonly used in the literature and in tools (see e.g. the seminal paper by Zielonka [Zie98]). Oftentimes, games with other winning conditions can be transformed into parity games. (However, the transformation may introduce a blowup. For example, Muller games can be converted into equivalent parity games with exponentially more positions [DJW97].)

A *parity game* is given by a game arena and a *priority function* $\Omega\colon \Gamma \to \mathbb{N}$ that assigns to each position of the game one of finitely many priorities. For the sake of simplicity, we assume that the game arena is deadlock-free, which means that every maximal play is infinite. Such a play p

is won by the existential player if and only if the largest priority that occurs infinitely often in the sequence $\Omega(p) = \Omega(p_1)\Omega(p_2)\ldots$ is odd.

If we only allow finitely many priorities, choosing smaller or larger priorities to be dominating is arbitrary. Considering a more general setting is beyond the scope of this thesis.

Similar to reachability games, parity games are positionally determined [Mos91; EJ91]. However, the proof is much more involved. Zielonka [Zie98] observed that from the proof of positional determinacy, one can extract an algorithm that constructs the winning regions and corresponding positional strategies. This improves an earlier algorithm by McNaughton [McN93] that was based on the weaker result that parity games admit finite-memory winning strategies [GH82] which we will define below.

Zielonka's algorithm is well known to consume exponential time in the worst case. Determining the precise complexity of solving parity games is one of the most important unsolved problems in theoretical computer science. The problem is known to be in (a subclass of) NP ∩ coNP, since one can guess a positional strategy for one of the players and verify whether it is winning in polynomial time [EJS01; Jur98]. The membership in this intersection strongly suggests that the problem is not NP-complete for complexity-theoretic reasons. To this date, no polynomial-time algorithm is known. A recent breakthrough [CJKLS17] has provided the first algorithm that is both quasi-polynomial and fixed-parameter tractable. The former property means that the algorithm solves parity games in time $\log(2^{n^k})$, where n is the size of the input and k is a constant. The latter means that the algorithm is exponential only in the highest occurring priority, but not in size of the game arena.

Games on the transitions systems induced by automata

In general, it is impossible to compute the winner of a game on an infinite game arena within finite time. However, games on infinite arenas occur in some applications like the synthesis problem for programs that we mentioned in the introduction. To overcome the problem, we use the concept of automata.

In Section 4.1, we have defined automata as finite descriptions for potentially infinite transition systems. More precisely, we have considered transition systems whose configurations are of the shape (q, m), consisting of one of finitely many control states and a potentially unbounded memory value. The transitions of the systems are induced by a finite set of rules whose applicability is depending only on a bounded amount of information on the current memory value. We apply this concept to games and define *games on the transitions systems induced by automata*. Consider the finite syntax of an automaton together with a partitioning $Q = Q_\Box \uplus Q_\bigcirc$ of its control states. The partition turns the semantics of the automaton, i.e. the transition system induced by it, into a game arena. The owner of some position (q, m) in that arena is induced by the partitioning of the control states and independent of the memory value. This concept gives us a chance of computing the winner of a game that is played on such an arena by working with the finite description.

Similar to the definition of the arena, one can define winning conditions based on the control state: In a control-state reachability game, one specifies a target set of control states that should be reached with arbitrary memory value. In a parity game on the transition system induced by an automaton, one typically assumes a priority assignment to the control states that then induces a priority assignment to all configurations based on their control states. However, we will later also consider games in which the goal is reaching a specified configuration, i.e. we fix both the target control state and the target memory value.

For each type of automaton, solving games with a certain type of winning condition is harder than solving verification problems with the same type of acceptance condition. We may see a deterministic automaton as a special case of a nondeterministic automaton, and a nondeterministic automaton as a special case of a game in which all control states are owned by the existential player. Consequently, there is no hope of solving reachability games on the transition systems induced by Turing machines. We will later see that context-free games, games whose arenas are induced by pushdown automata or by context-free grammars, can be solved. Petri nets form an interesting case: While coverability and reachability are decidable, we will prove that games defined by Petri nets with the corresponding winning conditions are undecidable.

Remark
In Section 3.2 we have provided a definition of alternating Turing machines and later introduced nondeterministic Turing machines as special cases. With the notions introduced in this section, we could take the opposite approach: An alternating Turing machine is a nondeterministic Turing machine with a partitioning of the control states, and its semantics is the corresponding control-state reachability game.

Inclusion games

We have argued before, e.g. in Section 4.1, that it is beneficial to consider labeled systems instead of unlabeled ones. By taking this approach, properties of the behavior of a system translate into properties of its language. Correspondingly, instead of checking properties of the behavior of a system, solving a verification problem amounts to checking properties of a language. For example, solving the reachability problem for a system typically translates into checking language-emptiness. Additionally, the language-theoretic approach allows us to compare different systems by considering problems like language inclusion and intersection-emptiness as long as the systems are labeled over the same alphabet. It seems like an obvious choice to also do this for games for exactly the same reasons, although this approach to games is less common in the literature.

Formally, we assume that the game arena under consideration is equipped with a labeling function $\lambda\colon T \to \Sigma^*$ that assigns to each transition a finite word. Note that we do not allow multiple transitions that differ only in their label for the sake of simplicity. The labeling of the transitions induces a labeling of plays: If $p = p_0 p_1 p_2 \dots$ is a play, then $\lambda(p) = \lambda(p_0, p_1)\lambda(p_1, p_2)\dots$ is the

word obtained by concatenating the labels of the transitions used in the play. This word is finite if p is finite or if p is infinite, but only finitely many transitions are not labeled by ε. Otherwise, $\lambda(p)$ is an infinite word.

In the case of a game arena based on the transition system induced by an automaton, the transition labels in the game come from the labeling of the transitions of the automaton as expected.

We will be mostly interested in *inclusion games*. Their winning condition is specified by a *target language* $\mathcal{L}$ over the same alphabet that is used by transition labels. For now, let us consider the case that $\mathcal{L} \subseteq \Sigma^*$ is a language of finite words. A maximal play p of the inclusion game is won by the existential player if its label $\lambda(p)$ is a finite word not contained in $\mathcal{L}$. Otherwise, the play is won the universal player. The motivation for this definition is that we think of the existential player as a representation of the demonic nondeterminism and of the language $\mathcal{L}$ to represent valid behavior. To prove that the system is incorrect, the existential player has to enforce a play that exhibits illegal behavior.

If the target language $\mathcal{L} \subseteq \Sigma^\omega$ is a language of infinite words, we follow a similar convention. The existential player wins a play if it generates an infinite word not in the language. If a play generates a finite word or an infinite one contained in $\mathcal{L}$, the universal player wins.

In Section 1.3 of the introduction, we had considered synthesis as one application for perfect-information games. If we model the game representing a synthesis problem according to the definitions in this section, the roles of the players are as follows. The universal player is the synthesis players. Her goal is to ensure that all executions that are terminating or non-terminating – depending on which type of program we consider – are valid with respect to the specification. An execution being valid with respect to the specification means that the corresponding play generates a word that is in the target language. The existential player represents the environment. She wins plays that generate words that are of the required shape (i.e. finite or infinite) but not in the target language. On the level of executions, this means that an illegal execution exists. In general, which player represents the controllable, angelic nondeterminism, i.e. the type of nondeterminism that helps us meet the objective[1], and which player represents uncontrollable, demonic nondeterminism will depend on the application.

Succinctness

Solving an inclusion game generalizes the problem of deciding inclusions among languages in the same way that solving reachability games generalized the reachability problem for nondeterministic systems. This in particular means that whenever we allow non-regular target languages, solving inclusion games typically becomes undecidable: Inclusion games with a context-free target language are undecidable because already the problem of checking whether a regular language is contained in a context-free one is undecidable (see e.g. [HU79]

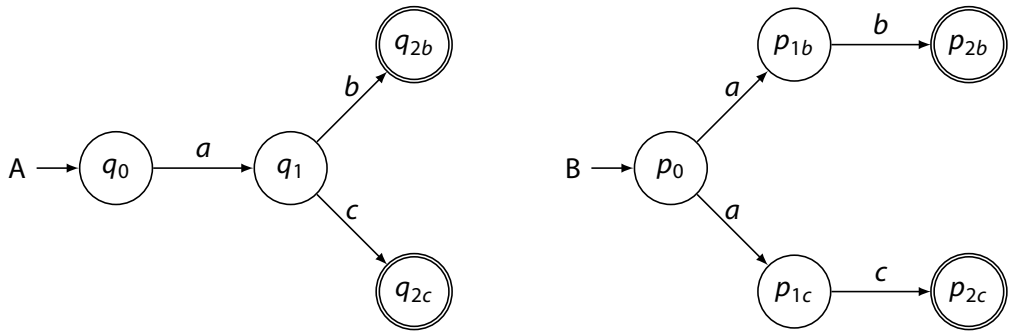
[1] In this thesis, we assume that angelic means good – this is not *a cruel angel's thesis*.

for a proof). In Section 19.2, we will show a result that implies that games with a Petri net coverability target language are undecidable, although the corresponding verification problem is decidable (as we have mentioned in Section 10.1). For these reasons, we will limit ourselves to regular inclusion games, where the target language is regular or ω-regular. In this case, inclusion games can be reduced to reachability games that are simultaneously played on the game arena and the automaton. However, we argue in the following that inclusion games provides a more succinct representation. We demonstrate this for a target language of finite words.

Consider an inclusion game played on the game arena induced by some automaton A that is not necessarily finite state and with regular target language $\mathcal{L}(B)$, where B is an NFA. One might think that equivalently, one can consider the reachability game on the game arena induced by $A \times B$, the product of the automata A and B. The goal in this new game is to reach a position in which the A-component is a deadlock and the B-component is a non-accepting. This reasoning is valid if B is deterministic: If we reach such a position, we have produced a maximal finite play of A defining a word that is not in the language of $\mathcal{L}(B)$. However, it does not work if B is nondeterministic. In this case, we need to prevent the selection of a non-accepting run of the automaton B on a word for which an accepting run exists. There are three types of nondeterminism at play here: The two types of nondeterminism in the automaton A represented by the players of the game, and the nondeterminism of the automaton B that generates the target language. Unfortunately, it is impossible to merge the nondeterminism in the automaton with the nondeterminism in the game by letting one of the players control the automaton. Obviously, we cannot let the existential player control the automaton. The goal of the existential player is to enforce a play generating an illegal word, but for a word to be illegal, all its runs in the automaton need to be non-accepting. If we give the existential player control of the automaton, we modify the semantics of the game in her favor, making it too easy for her to win.

Seeing that letting the universal player control the automaton also does not yield the desired result is more involved. To demonstrate this, we give a brief example without giving the formal definitions. Consider the finite automata depicted in Figure 15.0.a. They are well-known examples that are commonly used to show that language equality does not imply *bisimilarity* in the case of nondeterministic automata. (This is in contrast to the case of deterministic automata, where bisimilarity and language equality are equivalent, which can be used e.g. for the minimization of automata [HK71].) Consider A as a game arena in which all positions are owned by the existential player on which we play an inclusion game with target language $\mathcal{L}(B)$. Both automata have the same language, $\mathcal{L}(A) = \mathcal{L}(B) = \{ab, ac\}$. Hence, the existential player is unable to win the inclusion game starting from position q_0: Both maximal plays that she can choose from yield words in the language of B.

Let us consider the product $A \times B$ in which we give the universal player the control over the automaton B. A play from (q_0, p_0) starts with the existential player picking a move in the game, but her only choice is to use the transition $q_0 \xrightarrow{a} q_1$ of A. Now, the universal player has to pick among the two a-labeled transitions of B. The result is either the position (q_1, p_{1b}) or (q_1, p_{1c}).



i) Automaton A with a unique transition relation. ii) Automaton B with a nondeterministic transition relation.

Figure 15.0.a: Two automata with language $\{ab, ac\}$.

In both positions, the existential player can win by picking a transition in A which cannot be simulated in the state of B that the universal player has selected. We see that the existential player wins the reachability game on the product although she did not win the inclusion game. The reason for the invalidity of the construction is twofold: The universal player has to resolve the nondeterminism of the automaton during the run without knowing the full word that will be generated. When she picks the first transition in B, she does not know whether c or b will follow. Additionally, she makes her choices visible to the existential player, e.g. if she has picked the transition to p_{1b}, the existential player can react by using the c-labeled transition.

To avoid the problem, there are several possibilities. We could construct a game arena in which each play consists of two parts: The first is a play of A that derives some finite word w; the second part is a run of automaton B on w. Here, we can let the universal player control the automaton because she knows the full word that has been generated in the first part. The drawback of this concept is that it introduces an infinite number of positions required for storing the word of unbounded length that is produced in the first part of the game. A more reasonable choice is to determinize automaton B. When we consider a determinized version of B, the product construction becomes valid. However, applying the construction may introduce an exponential blowup. Actually, it would be sufficient to compute a so-called good-for-games automaton [HP06] that may be nondeterministic, but in which the nondeterminism can always be resolved based on prefixes of the word. Unfortunately, such automata may be just as large as deterministic ones [KS15].

In any case, the pair (A, B) consisting of the game automaton A and the NFA B is a representation of the inclusion game on the arena induced by A with target language $\mathcal{L}(B)$ that is more succinct than any of the other choices. We will make this formal in the case of context-free inclusion games by showing that solving such games is exponentially harder if the target language is given by a nondeterministic instead of a deterministic automaton. Our procedure to solve such games will mitigate this problem by using an on-the-fly determinization that, instead of computing a determinization of B upfront, determinizes B along the words that actually occur as labels of plays.

It might seem that the fact that the nondeterminism in B cannot be resolved by one of the players contradicts our earlier statement that a k-player perfect information game can always be transformed into a two player game by merging players. However, if we want the nondeterminism in the automaton to be resolved during the play (instead of resolving it after the play is complete as proposed above), this has to be done in a way that is not visible to the existential player. Hence, we are not dealing with a perfect-information game anymore.

Strategy automata

To conclude this chapter, we come back to the notion of a simple strategy that we mentioned earlier. In the case of games on finite graphs, positional strategies are sufficiently simple. Such a strategy can be stored using space linear in the size of the game arena by storing for each position of a player its designated successor.

The case of games on infinite graphs is more complicated. The results on the positional determinacy of reachability and parity games still hold, but a positional strategy for such a game cannot be represented using finite space. To overcome this problem, we consider strategies that can be finitely represented by automata. To make this concept formal, we assume that the game arena is induced by some automaton whose finite set of transition rules is δ, meaning that each of the (potentially infinitely many) transitions in the game arena is induced by one of the finitely many transition rules $t \in \delta$. We assume that for each position c of the game and each transition rule $t \in \delta$, there is at most one successor that can be reached from c using a transition induced by rule t. A *strategy automaton* is a deterministic automaton with δ as the input alphabet together with an output function out: $Q \to \delta$ that assigns to each control state a designated move. It induces a strategy $s_{\star}$ as follows. Given a play p, we consider the unique control state q in which the automaton is after reading the sequence of transition rules that induce the sequence of moves that has been used in p. If play p ends with a position of player $\star$, the strategy selects the successor that is reached by the move induced by the transition rule out(q) that is the output of the automaton for the current control state.

Different types of automata induce different types of strategies, e.g. deterministic pushdown automata induce *pushdown strategies*, DFAs induce *finite-memory strategies*. The latter type of strategies can be classified further by considering the number of states of the automaton. For games on infinite game arenas, these types of strategies are incomparable to positional strategies. Without imposing further restrictions, the strategies induced by strategy automata depend on the history of the play and are not positional. However, unlike positional strategies, they can be represented in a finite way using the syntax of the defining automaton. In the case of games on finite arenas, strategies defined by automata are strictly weaker (and thus more general) than positional ones. They can be used for some types of games where positional determinacy does not hold. For example, it has been shown that Muller games are not positionally determined, but each player can win on her winning region by using a finite-memory strategy defined by a DFA with at most $n!$ states, where n is the size of the game arena [DJW97]. We will come back to the concept of strategies induced by various types of automata when we study winning strategies for context-free games in Section 17.4.

16 Effective denotational semantics

Contents

16.1 Systems of equations and domains 287
16.2 Effective denotational semantics for regular inclusion 298
16.3 ω-regular inclusion 303

Our goal in this part of the thesis is solving various types of games on the configurations graphs of automata. We aim for a method that also computes certificates in the form of representations of winning strategies for each player. We will achieve this by using *effective denotational semantics* [Aeh07; Sum77; SW15b]. This means that we will translate the problem of solving a game into the problem of finding the least solution to a system of equations over a domain that represents the behavior of the game. Before we employ this method to solve context-free games in Chapter 17 and higher-order games in Chapter 18, we give the basic definitions in this chapter. Furthermore, we demonstrate the technique by applying it to the verification problems of deciding regular inclusion for context-free and ω-regular inclusion for ω-context-free languages.

The name *effective denotational semantics* was introduced by Salvati and Walukiewicz in [SW15b]. They define it as follows: "By effective denotational semantics we mean semantic spaces in which the denotation of a term can be computed" [SW15b], where "term" is referring to a term in the simply typed λY-calculus. This model is similar to higher-order recursion schemes which we will consider in Chapter 18, but, among other differences, includes a collection of Y-operators, one for each type. In order to evaluate such a term, one needs a model that defines the interpretation of the syntactical elements of the term. In particular, one needs an interpretation of the Y-operators, which are typically interpreted as fixed point operators that take a function and compute a fixed point.[2] We will come back to the work by Salvati and Walukiewicz at the end of Section 17.9.

When we speak of effective denotational semantics, we mean an approach to solving verification problems that is not restricted to problems phrased using the λY-calculus, which can be rather technical. However, we restrict ourselves in the sense that we only consider least-fixed-point semantics throughout this thesis. We briefly illustrate how our flavor of effective denotational semantics is supposed to work. Assume we are given a system P and a property φ. The task is to check whether all possible executions of P satisfy φ.

[2] Depending on the type of model, the operator may compute the least, greatest, or a non-extremal fixed point.

To solve the problem using effective denotational semantics, we proceed as follows:

(1) We construct a system of equations (or inequalities) reflecting the runtime behavior of P.

(2) We find a domain $\mathcal{D}$ that captures the behavior of executions that is relevant for deciding whether φ holds. We find an interpretation of the syntax used in the system of equations on the domain $\mathcal{D}$.

(3) We solve the interpreted system of equations over $\mathcal{D}$.

(4) We can now read off the answer to the verification problem from the solution.

Like in *denotational semantics* [SS71], the solution to the system of equations is usually defined as a fixed point. However, the domain $\mathcal{D}$ does not represent the whole program behavior. It only captures the part that is relevant for determining whether of property φ is satisfied. Our aim is to choose $\mathcal{D}$ such that $\mathcal{D}$ can be handled algorithmically (e.g. choosing $\mathcal{D}$ as a finite set), hence the name *effective* denotational semantics.

This approach has a key advantage over designing an algorithm that solves the verification problem of interest directly. By translating the problem into the task of solving a system of equations, we reduce it to a well-known *master problem*. As we will discuss in the next section, this makes available to us both an extensive theory on solving systems of equations and a collection of optimizations that enable us to solve this master problem efficiently.

Sources

The first section presents material that is standard in the literature. We will give references later. The content of the second section is taken from the publication [HM15] by Holík and Meyer. The final section presents material from the paper [MMN17] to which the author has contributed.

16.1 Systems of equations and domains

We start by formally introducing systems of equations, their interpretations and solutions. We then consider ordered domains, which results in the notion of the least solution to a system of equation. The material presented in this section is standard and can be found e.g. in [NNH99].

Systems of equations

Let $\mathcal{V} = \{X_1, \ldots, X_n\}$ be a finite set of *variables*. Let Fun $= \{f_{1/a_1}, \ldots, f_{k/a_m}\}$ be a set of *function symbols*, each symbol f_i annotated with its *arity* $a_i \in \mathbb{N}$. The set of *terms* over $\mathcal{V}$ and Fun is defined by the following BNF,

$$t ::= X \mid f(t, \ldots, t)\,,$$

where $X \in \mathcal{V}$, $f \in$ Fun, and the number of parameter terms matches the arity of f. For a term t, we call the variables that were used to build t the *free variables* of t.

A system of equations provides for each variable one defining term.

16.1.1 Definition
A *system of equations* for the set of variables $\mathcal{V}$ and the set of function symbols Fun is an assignment of one term t_i for each variable $X_i \in \mathcal{V}$. We commonly write it as

$$\begin{aligned} X_1 &= t_1, \\ &\vdots \\ X_n &= t_n\,, \end{aligned}$$

or simply as $\vec{X} = \vec{t}$. We call the term t_i the *right-hand side* for variable X_i.

Before we can solve a system of equations, we first need to associate a meaning to its syntax. Let $\mathcal{D}$ be a *domain*, which at this point should just mean that it is a set of values. An *interpretation* $\mathcal{I}$ of the function symbols over $\mathcal{D}$ is a function that assigns to each function symbol f, say with arity k, a function

$$f^{\mathcal{I}} : \mathcal{D}^k \to \mathcal{D}\,.$$

A *model* is a tuple $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ consisting of a domain and an interpretation over that domain.

A *(variable) assignment* is a function $\sigma : \mathcal{D} \to \mathcal{V}$. We often see such an assignment as a $|\mathcal{V}|$-dimensional vector over $\mathcal{D}$ and write $\sigma \in \mathcal{D}^{\mathcal{V}}$.

Given a model, we can lift the interpretation from function symbols to arbitrary terms. For each term t, we obtain a function

$$\mathcal{M}[\![t]\!] \colon \mathcal{D}^{\mathcal{V}} \to \mathcal{D} \,,$$

called the *semantics* of term t. It takes a variable assignment $\sigma \in \mathcal{D}^{\mathcal{V}}$ and returns the value $\mathcal{M}[\![t]\!]\sigma$ obtained by evaluating t after all variables X have been replaced by $\sigma(X)$ and all function symbols f have been replaced by $f^{\mathcal{I}}$. Note that it is customary to omit the parentheses around the argument σ. Formally, the function is defined by induction over the structure of the term,

$$\begin{aligned} \mathcal{M}[\![X]\!]\sigma &= \sigma_X \,, \\ \mathcal{M}[\![f(t_1, \ldots, t_k)]\!]\sigma &= f^{\mathcal{I}}(\mathcal{M}[\![t_1]\!]\sigma, \ldots, \mathcal{M}[\![t_1]\!]\sigma) \,. \end{aligned}$$

We may see $\mathcal{M}[\![-]\!] \colon \text{Terms} \times \mathcal{D}^{\mathcal{V}} \to \mathcal{D}$ as a function that takes both a term t and an assignment σ and returns $\mathcal{M}[\![t]\!]\sigma$.

Let $\vec{X} = \vec{t}$ be a system of equations and let $\mathcal{M}$ be a model for the set of terms used in that system. Applying the lifted interpretation to the defining terms t_i yields for each variable a function $\mathcal{M}[\![t_i]\!] \colon \mathcal{D}^{\mathcal{V}} \to \mathcal{D}$. We can combine these into a single function

$$\mathit{rhs} \colon \mathcal{D}^{\mathcal{V}} \to \mathcal{D}^{\mathcal{V}}$$

such that $\mathit{rhs}(\sigma)(X_i) = \mathcal{M}[\![t_i]\!]\sigma \in \mathcal{D}$. The function *rhs* takes an assignment of the variables and produces a new assignment by evaluating the defining term for each variable using the given values. We call this function the *interpreted system of equations* defined by $\vec{X} = \vec{t}$ and $\mathcal{M}$.

In the rest of this thesis, we will often drop the distinction between an uninterpreted system of equations and its interpreted version when the interpretation is clear from the context. Nevertheless, we sometimes make use of the fact that we can interpret a system of equations over various domains: We start by interpreting it over a domain where the soundness of the verification approach outlined at the start of this chapter is obvious. Later, we then move to a domain where the soundness is less obvious, but that is optimized e.g. in the sense that it is smaller.

For an interpreted system of equations, we can now define what it means to solve the system of equations. A *solution* for an interpreted system of equations *rhs* is an assignment $\sigma \in \mathcal{D}^{\mathcal{V}}$ such that $\sigma = \mathit{rhs}(\sigma)$. In other words, σ actually satisfies the equations given by the system.

The definition makes clear the correspondence between solutions of systems of equations and fixed points: A solution to a system of equations is defined to be a fixed point of the function *rhs*. Hence, the task of computing a solution to a system of equations is equivalent to the task of computing a fixed point of a certain function.

Context-free grammars as systems of equations

We give an example for the basic definitions. Let $G = (N, P, S)$ be a context-free grammar over the terminal alphabet Σ. There is a natural way to see G as a system of equations. We see N as the set of variables. The function symbols consist of one constant a for each $a \in \Sigma \cup \{\varepsilon\}$ and of the binary symbols $|$ for choice and . for concatenation. The term t_X for each nonterminal X is obtained by collecting all rules $X \to \eta_{X,i}$ for X, seeing each $\eta_{X,i}$ as a term involving constants (terminals), variables (nonterminals), and concatenation. Then, we conjoin these using choice. We obtain for each nonterminal X the equation

$$X = \eta_{X,1} \mid \ldots \mid \eta_{X,\ell_X} .$$

One possible interpretation of this system of equations is over the so-called *language semiring*. The language semiring is a model with domain $\mathcal{P}(\Sigma^*)$, i.e. its elements are languages over Σ, and the function symbols are interpreted in the expected way: Constants are interpreted as singleton languages containing the corresponding one-letter words resp. the empty word, concatenation is the concatenation of languages, and choice is the union of languages.

A solution σ for a context-free grammar, seen as a system of equations interpreted over the language semiring, assigns to each nonterminal a language. In particular, it assigns a language to the initial symbol S, so we might expect $\sigma(S)$ to be the language of the grammar. We will see in the following example that while the language of the grammar is a solution, it is not necessarily unique.

16.1.2 Example

Consider the grammar G with terminals $\Sigma = \{a, b, c\}$, the single nonterminal X and the rules

$$S \to aX \mid bX , \qquad\qquad X \to S .$$

We have $\mathcal{L}(G) = \varnothing$, as there is no terminating derivation process. It is easy to verify that the assignment σ_1 with $\sigma_1(X) = \sigma_1(S) = \varnothing$ satisfies the associated interpreted system of equations.

The assignment σ_2 with $\sigma_2(S) = a\Sigma^* \cup b\Sigma^* = \Sigma^*$ and $\sigma_2(X) = \Sigma^*$ is also a solution to the interpreted system of equations. In fact, for any language $\mathcal{L}$, we have that the assignment $\sigma_{\mathcal{L}}$ with $\sigma_{\mathcal{L}}(X) = \sigma_{\mathcal{L}}(S) = a.\mathcal{L} \cup b.\mathcal{L}$ is a solution. Unless we chose $\mathcal{L}$ to be $\varnothing$ to recover the solution σ_1, $\sigma_{\mathcal{L}}$ does not correspond to the language of the CFG.

The example shows that there can be many solutions to an interpreted system of equations. As also indicated in the example, we are usually interested in a solution that is *precise* in a certain sense. To make the notion of being precise formal, we need to introduce *ordered domains*. Formally, we equip the set of value with a partial order that satisfies certain properties.

Complete Partial Orders.

Let $(\mathcal{D}, \leqslant)$ be a *partial order*, i.e. a set $\mathcal{D}$ of data elements and a partial ordering $\leqslant \subseteq \mathcal{D} \times \mathcal{D}$ on $\mathcal{D}$. We call $(\mathcal{D}, \leqslant)$ *pointed* if there is a least element $\bot \in \mathcal{D}$, called the *bottom element*, satisfying $\bot \leqslant x$ for all $x \in \mathcal{D}$. Recall that an *ascending chain* in $\mathcal{D}$ is a sequence $(d_i)_{i\in\mathbb{N}}$ of elements in $\mathcal{D}$ such that $d_i \leqslant d_{i+1}$ for all $i \in \mathbb{N}$. We call $(\mathcal{D}, \leqslant)$ *ω-complete* if every ascending chain in $\mathcal{D}$ has a least upper bound, called the *join* or the *supremum*, and denoted by $\bigsqcup_{i\in\mathbb{N}} d_i$. Formally, we require that $\bigsqcup_{i\in\mathbb{N}} d_i \geqslant d_j$ for all $j \in \mathbb{N}$, and that for any element $d \in \mathcal{D}$ that also satisfies this property, $\bigsqcup_{i\in\mathbb{N}} d_i \leqslant d$ holds.

If $(\mathcal{D}, \leqslant)$ is pointed and ω-complete, we call it an *ω-complete pointed partial order (CPPO)*.[1] In the context of solving system of equations, we will only consider partial orders that are CPPOs. When the order on the set $\mathcal{D}$ that makes it a CPPO is clear, we simply say that $\mathcal{D}$ is a CPPO.

In the following, we want to solve systems of equations over models whose domains are CPPOs. However, requiring the domain to be a CPPO is insufficient to ensure the existence of a unique least solution. We also need to require the function to satisfy special properties.

Let $\mathcal{D}$ be a CPPO. A function $f\colon \mathcal{D} \to \mathcal{D}$ on $\mathcal{D}$ is *monotonic* if for all $d, d' \in \mathcal{D}$, $d \leqslant d'$ implies $f(d) \leqslant f(d')$. A function $f\colon \mathcal{D} \to \mathcal{D}$ is *join-continuous* if for all ascending chains $(d_i)_{i\in\mathbb{N}}$ we have $f(\bigsqcup_{i\in\mathbb{N}} d_i) = \bigsqcup_{i\in\mathbb{N}} f(d_i)$, i.e. the function value of the join is equal to the join of the function values. Note that join-continuity implies monotonicity: If $d \leqslant d'$ and f is join-continuous, then

$$f(d') = f(d \sqcup d') = f(d) \sqcup f(d') \geqslant f(d)\,,$$

where the first equality holds since $d \leqslant d'$, the second is join-continuity, and the last inequality is the fact that the join is an upper bound. Formally, we have only defined joins for infinite ascending chains, but the definition can be extended to finite chains easily, e.g. $d \sqcup d'$ can be defined as $\bigsqcup_{i\in\mathbb{N}} d_i$ with $d_0 = d$ and $d_i = d'$ for $i > 0$.

Note that if f is a monotonic function, then the chain

$$(f^i(\bot))_{i\in\mathbb{N}},$$

i.e. the chain $\bot \leqslant f(\bot) \leqslant f^2(\bot) \leqslant f^3(\bot) \leqslant \ldots$ is an ascending chain, where f^0 is defined to be the identity and $f^{i+1}(d) = f(f^i(d))$ is the $(i+1)$-fold application of f. The following theorem shows that the join of this chain is the least fixed point of f. The theorem is often attributed to Kleene [Kle52], but its actual origin seems to be unknown, see [LNS82] for a discussion.

[1] In the literature, the notion of CPPO is often used to denote the dual concept, i.e. partial orders that have a greatest element and in which infima of descending chains exist.

16.1.3 Theorem: Kleene's theorem
Let $f\colon D \to D$ be a join-continuous function over a CPPO $(\mathcal{D}, \leqslant)$. Then the join

$$\bigsqcup_{i \in \mathbb{N}} f^i(\bot)$$

is the least fixed point of f.

Formally, this means that it is a fixed point, i.e.

$$f\left(\bigsqcup_{i \in \mathbb{N}} f^i(\bot)\right) = \bigsqcup_{i \in \mathbb{N}} f^i(\bot)$$

holds, and for any other element $d \in \mathcal{D}$ with $f(d) = d$, we have

$$\bigsqcup_{i \in \mathbb{N}} f^i(\bot) \leqslant d\ .$$

We argue that Kleene's theorem can be used to compute the fixed point of monotonic functions if the underlying CPPO satisfies an additional condition. A CPPO $(\mathcal{D}, \leqslant)$ satisfies the *ascending chain condition (ACC)* if every ascending chain in $\mathcal{D}$ is stationary, i.e. for every $(d_i)_{i \in \mathbb{N}}$, there is some $i_0 \in \mathbb{N}$ with $d_i = d_{i+k}$ for all $k \in \mathbb{N}$ and $i \geqslant i_0$. If this is the case, any monotonic function over $\mathcal{D}$ is already join-continuous (see below), and the least fixed point $\bigsqcup_{i \in \mathbb{N}} f^i(\bot)$ is equal to $f^{i_0}(\bot)$ for some $i_0 \in \mathbb{N}$. Indeed, this chain is guaranteed to be stationary by the ACC. In fact, we obtain that the first index i_0 that satisfies $f^{i_0}(\bot) = f^{i_0+1}(\bot)$ is the desired index and $f^{i_0}(\bot) = f^{i_0+k}(\bot)$ holds for all k. Hence, we can obtain the least fixed point of any computable monotonic function over a domain that satisfies the ACC: Starting with the least element $\bot$, iteratively apply f until the result does not change anymore.

In this thesis, we will usually consider domains that satisfy an even stronger property. We say that $(\mathcal{D}, \leqslant)$ has *bounded height* if there is some $j \in \mathbb{N}$ such that any *strictly ascending chain* (an ascending chain $(d_i)_{0 \leqslant i \leqslant b}$ with $d_i < d_{i+1}$ for all $i < b$) has length at most j. Here, the length is defined to be the number of entries minus one. In this case, $(\mathcal{D}, \leqslant)$ satisfies the ACC and the bound i_0 from the definition of ACC is at most j. Consequently, $f^j(\bot)$ is the least fixed point of any monotonic function over $\mathcal{D}$.

Note that $(\mathcal{D}, \leqslant)$ having bounded height is a property strictly stronger than it satisfying the ACC. In theory, there are domains that satisfy the ACC but do not have bounded height, e.g. a domain that consists of an infinite collection of arbitrarily long (but finite) strictly ascending chains. However, we will only need to consider domains that are of bounded height throughout this thesis. In most cases, we will even consider domains $(\mathcal{D}, \leqslant)$ such that $\mathcal{D}$ is finite. Such domains are trivially of bounded height, as $|\mathcal{D}|$ is an upper bound for the length of any strictly ascending chain.

Proving an upper bound tighter than $|\mathcal{D}|$ will be crucial for some of the complexity-theoretic considerations in this part of the thesis.

16.1.4 Remark

We have argued before that join-continuity implies monotonicity. If a domain $\mathcal{D}$ satisfies the ACC, the other direction is also true and the two notions are equivalent. Assume that $f\colon \mathcal{D} \to \mathcal{D}$ is a monotonic function. Consider an ascending chain $(d_i)_{i\in\mathbb{N}}$ and note that this chain has to be stationary. Hence, its join $\bigsqcup_{i\in\mathbb{N}} d_i$ is equal to d_{i_0} for some i_0, and the function value of the join is simply $f(d_{i_0})$. We claim that $f(d_{i_0})$ is also the join of the chain of function values $(f(d_i))_{i\in\mathbb{N}}$. Firstly, observe that $d_i \leqslant d_{i_0}$ holds so the monotonicity of f yields $f(d_i) \leqslant f(d_{i_0})$ for all i. Secondly, the value $f(d_{i_0})$ is an element of the chain of function values. Thus, it is indeed the join, which proves that f is join-continuous.

Applying the theory

We discuss how to apply the theory of CPPOs and join-continuous functions to find the least solution to a system of equations. Assume that $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ is a model where $\mathcal{D}$ is a CPPO and for each function symbol, $f^{\mathcal{I}}$ is a join-continuous function. For functions with multiple arguments, we lift the definition of monotonicity and join-continuity by requiring it to hold for every argument. The key fact is that both monotonicity and join-continuity are preserved under compositions: If f, g are monotonic resp. join-continuous, then so is $f \circ g$. Hence, the interpretation of any term in this setting is a join-continuous function.

We need to argue that also the function $rhs\colon \mathcal{D}^{\mathcal{V}} \to \mathcal{D}^{\mathcal{V}}$ associated to a system of equations interpreted over $\mathcal{M}$ is join-continuous. We first observe that the set $\mathcal{D}^{\mathcal{V}}$ is a CPPO. To this end, we may see $\mathcal{D}^{\mathcal{V}}$ as a product of $|\mathcal{V}|$ copies of $\mathcal{D}$. We have argued in Section 3.1 that the product of partial orders is again a partial order. We extend this result and show that the product of CPPOs equipped with the product order is again a CPPO: The least element in the product is the tuple consisting of the least elements, i.e. it is the assignment $\bot \in \mathcal{D}^{\mathcal{V}}$ with $\bot_X = \bot \in \mathcal{D}$ for all X, where we intentionally use the same symbol $\bot$ in both cases. Joins in the product order can be obtained component-wise: If $(\sigma_i)_{i\in\mathbb{N}}$ is an ascending chain in $\mathcal{D}^{\mathcal{V}}$, then the sequence of values $(\sigma_{i,X})_{i\in\mathbb{N}}$ occurring in component X is an ascending chain in $\mathcal{D}$. The join $\bigsqcup_{i\in\mathbb{N}} \sigma_i$ in $\mathcal{D}^{\mathcal{V}}$ is the vector whose X component is $\bigsqcup_{i\in\mathbb{N}} \sigma_{i,X}$. Correspondingly, a function over $\mathcal{D}^{\mathcal{V}}$ is join-continuous if it is join-continuous in every component. We have argued before that if we assume that the interpretation of all function symbols is join-continuous, then so are the functions $\mathcal{M}[\![t]\!]$. With the above observation, we obtain that also the function $rhs\colon \mathcal{D}^{\mathcal{V}} \to \mathcal{D}^{\mathcal{V}}$ is join-continuous. This enables us to apply Kleene's theorem to it.

We define the i^{th} *approximant* sol^i to be

$$\mathrm{sol}^i = rhs^i(\bot)$$

the variable assignment obtained by an i-fold application of *rhs* to $\bot \in \mathcal{D}^{\mathcal{V}}$, the least element of the product domain. Using Kleene's theorem, the value

$$\mathrm{sol} = \bigsqcup_{i \in \mathbb{N}} \mathrm{sol}^i = \bigsqcup_{i \in \mathbb{N}} rhs^i(\bot)$$

is the least fixed point of the function *rhs*, and hence the least solution to the interpreted system of equations. For convenience, we lift the function $\mathrm{sol}: \mathcal{V} \to \mathcal{D}$ from variables to arbitrary terms by defining $\mathrm{sol}(t) = \mathcal{M}[\![t]\!]\mathrm{sol}$, similar for each sol^i.

If the domain $\mathcal{D}$ satisfies the ACC or is of bounded height, then the same holds true for $\mathcal{D}^{\mathcal{V}}$. To this end, observe that if $\mathcal{D}$ has height j, then $\mathcal{D}^{\mathcal{V}}$ has height $|\mathcal{V}| \cdot j$. In this case, there is some index i_0 so that $\mathrm{sol} = \mathrm{sol}^{i_0}$. Hence, Kleene's theorem allows us to compute the least solution to a system of equations assuming that it is interpreted over a CPPO satisfying the ACC and that the interpretations of the functions are monotonic. Starting with the least element, we iteratively the function *rhs* until the value remains unchanged. This process is called *Kleene iteration*.

Systems of inequalities and lattices

Often, our goal is not to solve a system of equations. Instead, we are given a system of inequalities, where each variable has one or more defining constraints. It turns out that this setting, while arguably sounding more complicated, can in fact be solved more efficiently, as long as the underlying domain satisfies additional properties.

Formally *system of inequalities* for a set of variables $\mathcal{V}$ and a set of function symbols Fun is defined as a finite set of inequalities of the shape

$$X \geqslant t\,,$$

where $X \in \mathcal{V}$ and $t \in$ Terms. The key difference to systems of equations is that we allow multiple inequalities for the same variable. We interpret systems of inequalities over a CPPO $\mathcal{D}$ and using join-continuous functions in the expected way. A solution $\mathrm{sol} \in \mathcal{D}^{\mathcal{V}}$ to such a system has to satisfy $\mathrm{sol}(X) \geqslant \mathrm{sol}(t)$ for all inequalities $X \geqslant t$ in the system.

To enforce the existence of a unique least solution to a system of inequalities, we need to impose additional restrictions on the domain. A *join-semilattice* is a partial order $\mathcal{D}$ so that any subset $X \subseteq \mathcal{D}$ has a least upper bound, the *join* or *supremum* $\bigsqcup X$ of X. The formal definition is a straightforward extension of the definition in the case of chains. We require that $\bigsqcup X$ is an upper bound, i.e. $\bigsqcup X \geqslant x$ for all $x \in X$, and that is smaller than any other upper bound: If y satisfies $y \geqslant x$ for all x, then $\bigsqcup X \leqslant y$.

For the dual concept of a *meet-semilattice*, we require the existence of the *meet* or *infimum* $\bigsqcap X$, the greatest lower bound, for any subset X.

A partial order that is a join- or a meet-semilattice always has a least element $\bot$ and a greatest element $\top$. These elements can be obtained as both the join and the meet of a suitable subset each,

$$\bot = \bigsqcup \varnothing = \bigsqcap \mathcal{D}\,, \qquad \top = \bigsqcup \mathcal{D} = \bigsqcap \varnothing\,.$$

A partial order that is both a join-semilattice and meet-semilattice is called *complete lattice*. Every join-semilattice is a CPPO, so Kleene's theorem guarantees the existence of least fixed points for join-continuous functions. We can dualize this result to obtain that meet-continuous functions over meet-semilattices have greatest fixed points.

To solve a system of inequalities that is interpreted over a complete lattice using monotonic functions, we convert it into a system of equations. We add a new binary function symbol $\sqcup$ that is interpreted as the join. Then, we obtain the defining equation for each variable X as

$$X = t_{X,1} \sqcup \ldots \sqcup t_{X,\ell_X}\,,$$

where $X \geqslant t_{X,1}, \ldots, X \geqslant t_{X,\ell_X}$ are all inequalities with X as their left-hand side. One can show that the least solution to the resulting system of equations is exactly the least solution to the original system of inequalities. In particular, Kleene's theorem guarantees its existence if the interpretations of the functions are join-continuous (or the domain satisfies ACC, in which case monotonicity implies join-continuity).

To formally show this, one uses a famous theorem by Knaster and Tarski [Kna28; Tar49; Tar55].

16.1.5 Theorem (Knaster & Tarski)
Let $f\colon \mathcal{D} \to \mathcal{D}$ be a monotonic function over a lattice $(\mathcal{D}, \leqslant)$. Then f has a least fixed point, namely

$$\bigsqcap \{d \in \mathcal{D} \mid d \geqslant f(d)\}\,.$$

Consider the least solution $\mathrm{sol} = \bigsqcup_{i\in\mathbb{N}} rhs^i(\bot)$ to the system of equations as specified by Kleene's theorem. First note that $\mathrm{sol}(X) = \mathcal{M}[\![t_{X,1} \sqcup \ldots \sqcup t_{X,\ell_X}]\!]\mathrm{sol} = \mathcal{M}[\![t_{X,1}]\!]\mathrm{sol} \sqcup \ldots \sqcup \mathcal{M}[\![t_{X,\ell_X}]\!]\mathrm{sol}$ implies $\mathrm{sol}(X) \geqslant \mathcal{M}[\![t_{X,i}]\!]$ for all X and i. All inequalities are satisfied by sol.

It remains to show that sol is the least solution to the system of inequalities. Assume that some assignment σ satisfies all inequalities. Hence, $\sigma(X) \geqslant \mathcal{M}[\![t_{X,i}]\!]\sigma$ for all i. Since the symbol $\sqcup$ is interpreted as the join, we have $\sigma(X) \geqslant \mathcal{M}[\![t_{X,1} \sqcup \ldots \sqcup t_{X,\ell_X}]\!]\sigma$ and thus $\sigma \geqslant rhs(\sigma)$. We conclude that σ satisfies the defining property of the set $\{d \in \mathcal{D} \mid d \geqslant rhs(d)\}$. The meet of this set is the least fixed point of *rhs* using Knaster's and Tarski's theorem. By Kleene's theorem, this least fixed point is sol. Since the meet of a set is a lower bound for the elements of the set, we obtain $\mathrm{sol} \leqslant \sigma$. As proclaimed, sol is indeed the least solution to the system of inequalities.

Chaotic iteration

We introduce *chaotic iteration* as described e.g. in [SWH12], an algorithm that solves systems of inequalities directly, i.e. without taking the detour via converting it into a system of equations described above. Consider a system of inequalities $\vec{X} \geqslant \vec{t}$ interpreted over a complete lattice using monotonic functions. If the lattice satisfies ACC, the least solution to the system can be found as follows. First initialize the candidate solution $\mathrm{sol} = \bot \in \mathcal{D}^{\mathcal{V}}$ as the least element of the product domain. While there is an inequality that is not satisfied by the current value of sol (i.e. $\mathrm{sol}(X) \not\geqslant \mathcal{M}[\![t_X]\!]\mathrm{sol}$), pick one such inequality $X \geqslant t_X$ and redefine the X-component of sol,

$$\mathrm{sol}(X) \leftarrow \mathrm{sol}(X) \sqcup \mathcal{M}[\![t_X]\!]\mathrm{sol}\,.$$

After performing this update, the inequality is satisfied. Once we have obtained a value for sol that satisfies all inequalities, we have arrived at the least solution. Under the assumptions outlined above, this is guaranteed to happen after finitely many steps.

Because the algorithm allows the inequalities to be evaluated in an arbitrary order, it is called chaotic iteration. For termination, we only need to guarantee that if an unsatisfied inequality exists, we consider it after finite time.

The key advantage of chaotic over Kleene iteration is the fact that it can be implemented using a *worklist*, reducing unnecessary computations. Before solving the system, we analyze its dependencies: We store for each variable X the set of inequalities $Y \geqslant t_Y$ such that an occurrence of X is contained in t_Y. We then initialize the worklist, a queue, by adding all inequalities to it in arbitrary order. In each step, we remove the first inequality from the queue and evaluate it. If it does not hold, we update the corresponding variable X as described above. Then, we enqueue all inequalities $Y \geqslant t_Y$ that depend on X.

The result is an algorithm that re-evaluates inequalities only when there is a chance that they have become invalid by an update of the candidate solution. In contrast to this, Kleene iteration works over the product domain and evaluates all inequalities in each step.

We finish this chapter by coming back to our earlier example of context-free grammars, see e.g. Example 16.1.2. The language semiring $\mathcal{P}(\Sigma^*)$ can be turned into a complete lattice by equipping it with inclusion as the order. The join in this lattice is the union, the meet is the intersection of languages. The least and greatest elements are $\varnothing$ and Σ^*, respectively. All functions that we have considered earlier are join-continuous.

Note that the interpretation of the symbol | is the union of languages and hence the join in the lattice. Our earlier construction that gathered all rules for each nonterminal and combined them into a single equation is an instantiation of the approach to solving systems of inequalities by transforming them into a system of equations.

The language semiring is a special case of a powerset lattice $(\mathcal{P}(M), \subseteq)$ for some set M. If M is finite, then the height of $\mathcal{P}(M)$ is bounded by $|M|$: An ascending chain

$$M_0 \subseteq M_1 \subseteq M_2 \subseteq \dots$$

of subsets of M can only be strictly ascending if each M_i contains at least one element that was not contained in M_{i-1}. This is only possible $|M|$ times until all elements from M has been added. Note that $|M|$ is a much better bound than the trivial bound $|\mathcal{P}(M)| = 2^{|M|}$.

If M is infinite, $\mathcal{P}(M)$ does neither have bounded height, nor does it satisfy the ACC. This in particular applies to the language semiring. While our theory works and we know that $\bigsqcup_{i\in\mathbb{N}} \mathrm{sol}^i = \bigsqcup_{i\in\mathbb{N}} rhs^i(\bot)$ is the least fixed point, it is not guaranteed that we can reach this fixed point within finitely many iterations.

In our example of a system of equations obtained from a context-free grammar, interpreted over the language semiring, we obtain the following correspondence. The least solution associates to each nonterminal the set of finite terminal words derivable from that nonterminal. In particular, it assigns the language of the grammar to the initial symbol. Considering the least solution instead of an arbitrary one allows us to recover the language of the grammar.

16.1.6 Example

Consider again the grammar G from Example 16.1.2 with the production rules

$$S \to aX \mid bX\,, \qquad\qquad X \to S\,.$$

We have that $\mathrm{sol}^0 = rhs^0(\bot) = \bot$ is the tuple that assigns each nonterminal the empty language. It is not hard to see that $\mathrm{sol}^1 = \mathrm{sol}^0$, so $\mathrm{sol}^0 = \mathrm{sol}$ is the least solution. Indeed, $\mathrm{sol}(S) = \varnothing$ is the language of the grammar.

Assume we add the production rule $S \to \varepsilon$, and note that this changes the language of the grammar to be Σ^*. If we start to solve the associated system of equations via Kleene iteration, we obtain the following values.

$\mathrm{sol}^i(-)$	S	X
$i = 0$	$\varnothing$	$\varnothing$
$i = 1$	$\{\varepsilon\}$	$\varnothing$
$i = 2$	$\{\varepsilon\}$	$\{\varepsilon\}$
$i = 3$	$\{a, b, \varepsilon\}$	$\{\varepsilon\}$
$i = 4$	$\{a, b, \varepsilon\}$	$\{a, b, \varepsilon\}$
$i = 5$	$\{aa, ab, ba, bb, a, b, \varepsilon\}$	$\{a, b, \varepsilon\}$

One can observe that for each n, we have that $\mathrm{sol}^{2n+1}(S) = \Sigma^{\leq n}$ is the set of all words of length at most n. In particular, the chain of the sol^i does not get stationary and the least fixed point sol with $\mathrm{sol}(S) = \Sigma^*$ is not reached within finitely many steps.

This example may give the false impression that the i^{th} approximant sol^i corresponds to words that can be derived within i steps. We will see in the next section that instead, sol^i corresponds to words that can obtained from derivation trees of height at most i.

While the least solution to the system of equations associated to a context-free grammar is indeed the language of the grammar, it cannot be computed within finite time using Kleene iteration or chaotic iteration. In the next section, we will see how overcome this problem by using a finite domain. The corresponding least solution will not correspond to the language of the CFG, but it will characterize it precisely enough to be able to solve a specific verification problem.

16.2 Effective denotational semantics for regular inclusion

In this section, we want to demonstrate effective denotational semantics by giving an in-depth example. We consider the regular inclusion problem for context-free languages. The idea of applying effective denotational semantics to this problem is not a contribution by the author of this thesis. The content of this section is taken from a paper by Holík and Meyer [HM15], and our presentation mostly follows that publication.

Formally, the problem we consider is the following.

Regular inclusion for context-free languages (CFL-REGINCLUSION)	
Given:	Context-free grammar G, NFA A.
Question:	Does $\mathcal{L}(G) \subseteq \mathcal{L}(A)$ hold?

The importance of this problem stems from the fact that context-free grammars can be used to represent recursive programs, as we have explained in Section 5.1, assuming that each level of the recursion only stores a bounded amount of data. This means that the problem of checking whether all possible executions of such a program satisfy a given property corresponds to an instance of CFL-REGINCLUSION. Even in the case of programs that use unbounded storage at each level of the recursion, one may use techniques like a language-theoretic version of counterexample guided abstraction refinement that repeatedly solves instances of CFL-REGINCLUSION.

The problem CFL-REGINCLUSION is well-known to be PSPACE-hard[1] and in EXP. To obtain an exponential-time algorithm, one can use that the inclusion $\mathcal{L}(G) \subseteq \mathcal{L}(A)$ holds if and only if the intersection $\mathcal{L}(G) \cap \overline{\mathcal{L}(A)}$ is empty. To check the latter, we first compute an NFA $\overline{A}$ for the complement language. Secondly, we use the well known triple-construction [BPS61] to obtain a new grammar $G \times \overline{A}$. Finally, we apply an emptiness check to this grammar to determine whether $\mathcal{L}(G \times \overline{A}) = \mathcal{L}(G) \cap \overline{\mathcal{L}(A)}$ is empty. The runtime of this algorithm is in $\text{poly}(|G|) \cdot 2^{\text{poly}(|A|)}$. The exponential part corresponds to the determinization of the automaton A that is needed to compute $\overline{A}$.

The drawback of this algorithm is that it requires an upfront determinization of automaton A. The grammar G might not explore the full behavior of A, e.g. if there are states of the determinization of A that are not visited in any run on any word in the language of G. Nevertheless, the algorithm needs to compute a full representation of the determinization before the product with the grammar can be computed.

In the following, we present an algorithm by Holík and Meyer [HM15] that avoids this upfront determinization. Instead, it only determinizes A along the words that occur in $\mathcal{L}(G)$. To this end,

[1] CFL-REGINCLUSION is a generalization of the problem of deciding whether an inclusion among regular languages, represented by finite automata, holds. This problem in turn is a generalization of the *universality problem* for finite automata, i.e. deciding whether $\mathcal{L}(A) = \Sigma^*$ holds for an NFA A. This problem can be proven to be PSPACE-complete using the techniques introduced in [SM73].

it uses effective denotational semantics with a domain that is based on the transition monoid, which we have introduced in Section 4.5.

Assume that we are given a context-free grammar $G = (N, P, S)$ over the terminal alphabet Σ and an NFA A over the same alphabet. We first set up a system of inequalities representing G. Similar to what we did in the last section, we see the nonterminals as variables. Each terminal a as well as ε constitute constant function symbols. Additionally, we have composition . as an operation. We obtain a system of inequalities in which each rule $X \to \eta$ of the grammar translates into an inequality $X \geqslant \eta$ by seeing the right-hand side as a term that is a composition of the constants (terminals) and the variables (nonterminals).

Let $\mathbb{M}_A$ be the transition monoid of A. We interpret the above system over the powerset lattice $(\mathcal{P}(\mathbb{M}_A), \subseteq)$ over the transition monoid. Each of the constants $a \in \Sigma \cup \{\varepsilon\}$ is interpreted as the singleton set $\{\rho_a\}$. The product operation is interpreted as element-wise relational composition: For sets $R_1, R_2 \subseteq \mathbb{M}_A$, we have

$$R_1 \,.\, R_2 = \{\rho \cdot \tau \mid \rho \in R_1, \tau \in R_2\} \,.$$

The powerset lattice $(\mathcal{P}(\mathbb{M}_A), \subseteq)$ has height $|\mathbb{M}_A| \leqslant 2^{|Q|^2}$, where Q is the set of states of A. This in particular means that it satisfies ACC. Additionally, the interpretation of each of the function symbols is monotonic, which is clear by definition. Hence, we can apply the techniques in the previous section to obtain the least solution sol to the system of inequalities within finitely many steps. This solution assigns to each nonterminal X a set of boxes $\mathrm{sol}(X) \subseteq \mathbb{M}_A$. Recall that a rejecting box is an element of $\mathbb{M}_A$ that does not contain a transition from an initial to a final state. This means that the words represented by it are not in the language of A. We claim that $\mathcal{L}(G) \subseteq \mathcal{L}(A)$ holds if and only if the least solution $\mathrm{sol}(S)$ associated to the initial symbol does not contain a rejecting box.

Once this claim is proven, we obtain an algorithm for CFL-REGINCLUSION that sets up the above system of inequalities, determines its least solution, and reads off the answer to the verification problem. This algorithm is amenable to algorithmic improvements, including the worklist procedure that we discussed in the last section and further optimizations that we will mention below.

In the following, we will prove the soundness of the algorithm. The proofs are adapted from [HM15]. We choose to explicitly present these proofs, even though they are not original work by the author of this thesis. They serve as an example of how to show the soundness of an approach to a verification problem based on effective denotational semantics.

The crucial step in proving the soundness of the algorithm is establishing the following lemma.

16.2.1 Lemma
We have $\mathrm{sol}(S) = \{\rho_w \mid w \in \mathcal{L}(G)\}$.

The easiest way to prove this lemma is to assume that we solve the system of inequalities not using chaotic iteration, but by transforming it into a system of equations and applying Kleene iteration. Both procedures arrive at the same result. Kleene iteration has the advantage of providing us with sol^i, the i^{th} approximant.

We show that $\mathrm{sol}^i(X)$ is the set of boxes associated to words that can be obtained by a derivation of height at most i from nonterminal X. Here, the *height* of a derivation is the height of the corresponding derivation tree, i.e. the number of edges from the root note to the most distant child. Recall that a derivation tree (from X) is an ordered tree whose root node, inner nodes, and leaves are labeled by X, nonterminals, and terminals, respectively. A replacement $X \to \eta_1 \dots \eta_n$ with $\eta_i \in N \cup \Sigma$ in the derivation process corresponds to the associated node X of the tree having $\eta_1, \dots, \eta_n$ as children in that order. We only consider complete derivation trees in which all leaves are terminals. The word produced by the derivation process is then exactly the *yield* of the associated derivation tree, i.e. its leaves read from left to right.

The following lemma makes this correspondence formal.

16.2.2 Lemma
For each nonterminal X and each $i \in \mathbb{N}$, we have

$$\mathrm{sol}^i(X) = \left\{\rho_w \;\middle|\; X \Rightarrow^* w \text{ with a derivation of height} \leqslant i\right\}.$$

Proof:
We prove the statement simultaneously for all nonterminals, proceeding by induction on i.

In the base case $i = 0$, we have $\mathrm{sol}^i = \bot$ and hence $\mathrm{sol}^i(X) = \varnothing$. Indeed, no terminal word can be derived from X with a derivation of height 0.

Assume that the statement holds for i and consider $i + 1$. Recall that in order to apply Kleene iteration, we have to transform the system of inequalities into a system of equations. We collect all rules $X \to \eta^{(1)}, \dots, X \to \eta^{(\ell_X)}$ with X and obtain that

$$\mathrm{sol}^{i+1}(X) = \mathrm{sol}^i\left(\eta^{(1)}\right) \sqcup \dots \cup \mathrm{sol}^i\left(\eta^{(\ell_X)}\right).$$

Since the symbol $\sqcup$ is the join, which is interpreted as the union in a powerset lattice, we can rewrite the right-hand side and get

$$\mathrm{sol}^{i+1}(X) = \mathrm{sol}^i\left(\eta^{(1)}\right) \cup \dots \cup \mathrm{sol}^i\left(\eta^{(\ell_X)}\right).$$

Using this equality, we prove $\mathrm{sol}^{i+1}(X) \subseteq \{\rho_w \mid X \Rightarrow^* w \text{ with a derivation of height} \leqslant i+1\}$. If $\rho \in \mathrm{sol}^{i+1}(X)$, then there is a rule $X \to \eta$ (where η is one of the $\eta^{(k)}$ in the above union) so that $\rho \in \mathrm{sol}^i(\eta)$. Let $\eta = \eta_1 \dots \eta_m \in (N \cup \Sigma)^*$ be the decomposition of η into its letters. We have that

$$\mathrm{sol}^i(\eta) = \mathrm{sol}^i(\eta_1) \dots \mathrm{sol}^i(\eta_m)\,,$$

where we use that the interpretation of concatenation is element-wise relational composition. Since $\rho \in \mathrm{sol}^i(\eta)$, each $\mathrm{sol}^i(\eta_j)$ contains some ρ_j so that $\rho = \rho_1 \cdots \rho_m$. Using induction, we can associate to each ρ_j a word w_j such that $\eta_j \Rightarrow^* w_j$ with a derivation of height at most i and ρ_j is the box associated to word w_j, $\rho_j = \rho_{w_j}$. If some letter η_k of η is a terminal symbol, the statement trivially holds.

Finally, we construct a derivation $X \Rightarrow^* w_1 \dots w_m$ by first applying the rule $X \to \eta$ and then using $\eta_j \Rightarrow^* w_j$ for each j. Since each derivation process $\eta_j \Rightarrow^* w_j$ has height at most i, $X \Rightarrow^* w_1 \dots w_m$ has height at most $i+1$ as desired. Its yield is the word $w_1 \dots w_m$ with $\rho_{w_1 \dots w_m} = \rho_{w_1} \cdots \rho_{w_m} = \rho_1 \cdots \rho_m = \rho$.

For the other inclusion, one proceeds similarly. For derivations of size strictly less than $i+1$, we use induction and the fact that $\mathrm{sol}^i(X) \subseteq \mathrm{sol}^{i+1}(X)$. For a derivation of a word w of height exactly $i+1$, we consider the associated derivation tree. It can be decomposed into the application of a production rule $X \to \eta$ to the root node and derivation trees for each of the letters of η. For each η_j, we extract a derivation $\eta_j \Rightarrow^* w_j$ of height at most i for each of the letters of β so that $w = w_1 \dots w_m$. By induction, we have that $\rho_{w_j} \in \mathrm{sol}^i(\eta_j)$, so $\rho_w = \rho_{w_1} \cdots \rho_{w_m} \in \mathrm{sol}^i(\eta) \subseteq \mathrm{sol}^{i+1}(X)$. ■

With this lemma at hand, it is easy to show Lemma 16.2.1.

Proof of Lemma 16.2.1:
Consider $w \in \mathcal{L}(G)$, i.e. $S \Rightarrow^* w$. We need to show $\rho_w \in \mathrm{sol}(S)$. There is some height i of the associated derivation tree, so by Lemma 16.2.2, we have $\rho_w \in \mathrm{sol}^i(S)$. Now we observe that the approximants form an ascending chain $\mathrm{sol}^0(S) \subseteq \mathrm{sol}^1(S) \subseteq \dots$, and the fixed point solution is an upper bound of that chain. Hence, $\rho \in \mathrm{sol}(S)$.

For the other direction, consider $\rho \in \mathrm{sol}(S)$. Since the powerset lattice under consideration satisfies the ACC, there is some $i_0 \in \mathbb{N}$ such that $\mathrm{sol} = \mathrm{sol}^{i_0}$. Hence, $\rho \in \mathrm{sol}^{i_0}(S)$, and by Lemma 16.2.2, there is a derivation process $S \Rightarrow^* w$ with $\rho = \rho_w$. Thus, $\rho = \rho_w$ with $w \in \mathcal{L}(G)$ as required. ■

With both lemmas proven, the soundness of the algorithm follows directly.

16.2.3 Proposition
The inclusion $\mathcal{L}(G) \subseteq \mathcal{L}(A)$ holds if and only if sol(S) contains no rejecting box.

Proof:
Assume that sol(S) contains the rejecting box ρ. With Lemma 16.2.1, we have $\rho = \rho_w$ for some word $w \in \mathcal{L}(G)$. Since ρ_w is rejecting, we have $w \notin \mathcal{L}(A)$ and the inclusion does not hold. The other direction is similar. ■

With the soundness of the algorithm proven, we consider some tricks that can speed it up. Firstly, the system of inequalities can be solved using a worklist procedure as explained in the previous section. Secondly, the implementation can be *lazy* in that it stops the algorithm as soon as sol(S) in the current candidate solution contains a rejecting box. Both chaotic iteration and Kleene iteration guarantee that the candidates for sol(S) that occur throughout the run of the algorithm form an ascending chain. If some candidate contains a rejecting box, then the final solution will contain it too. Finally, the paper by Holík and Meyer [HM15] considers an *antichain optimization*. Here, the key observation is that the elements of the transition monoid itself can be seen as boxes, which are subsets of $Q \times Q$. Boxes can be ordered by inclusion, and the operations that we apply as well as the property of being non-rejecting is well-behaved with respect to inclusion. Hence, we can consider instead of the powerset lattice, whose elements are arbitrary sets of boxes, the *antichain lattice* whose elements are sets of boxes are incomparable. This domain is smaller, so there is a chance that the algorithm will terminate after fewer steps. The antichain optimization have been shown to be highly efficient e.g. for checking the universality of finite automata [CJKLS17]. However, the difference in size is not substantial enough to improve the asymptotic worst-case complexity, which remains exponential.

16.3 ω-regular inclusion

In this section, our goal is to extend the results in the previous section from finite to infinite words. This means we apply effective denotational semantics to the problem of deciding ω-regular inclusion for ω-context-free languages, demonstrating the versatility of the approach. In contrast to the previous section, this section contains contributions by the author of this thesis. The content is taken from the publication [MMN17].

ω-regular inclusion

Formally, the problem that we aim to solve is the following.

ω-**regular inclusion for** ω-**context-free languages** (ωCFL-ωREGINCLUSION)	
Given:	Context-free grammar G, NBA A.
Question:	Does $\mathcal{L}^\omega(G) \subseteq \mathcal{L}^\omega(A)$ hold?

In the previous section, we have argued that the problem CFL-REGINCLUSION corresponds to the verification of the terminating executions of a recursive program. Similarly, ωCFL-ωREGINCLUSION corresponds to verifying the non-terminating executions.

Let us recall the definition of the ω-context-free language $\mathcal{L}^\omega(G)$. We base our explanation on the techniques that we have developed in the context of the proof of the characterization of ω-context-free languages, Theorem 5.2.2. A word $w \in \Sigma^\omega$ in $\mathcal{L}^\omega(G)$ is obtained from a left-derivation process that can be split into two parts: The first is an infinite chain of left-derivations of the form $X_i \Rightarrow_\ell \eta^{(i)} X_{i+1}$ for all $i \in \mathbb{N}$, where $X_0 = S$ is the initial symbol. The second part is a collection of finite left-derivation processes $\eta^{(i)} \Rightarrow_\ell^* w^{(i)}$ for all $i \in \mathbb{N}$ so that $w = w^{(0)}w^{(1)}\ldots$ To make this formal, we have introduced the spinal graph SG of a grammar in Section 5.2, a finite graph whose nodes correspond to nonterminals and whose set of labels is a finite collection of sentential forms. In SG, we have $X \xrightarrow{\eta} Y$ iff $X \to \eta.Y$ is a rule of the grammar. Additionally, we have defined $\mathcal{L}_G(X)$ to be the language of finite words of the grammar G with the initial symbol replaced by X. We have extended the definition to finite and infinite sentential forms by setting $\mathcal{L}_G(a) = \{a\}$ for $a \in \Sigma \cup \{\varepsilon\}$ and $\mathcal{L}_G(\beta_0.\beta_1\ldots) = \mathcal{L}_G(\beta_1).\mathcal{L}_G(\beta_2)\ldots$ In Lemma 5.2.6, we have argued that $\mathcal{L}^\omega(G)$ is exactly the set of infinite words in some $\mathcal{L}_G(\beta)$, where $\beta \in (N \cup \Sigma)^\omega$ is the concatenation of the labels along an infinite path in the spinal graph that starts in S.

Our goal is to construct a system of inequalities whose least solutions provides information about both parts of the derivation process as described above. To handle the finite derivation processes of the shape $\eta^{(i)} \Rightarrow_\ell^* w^{(i)}$, we proceed as in the previous section. For each nonterminal X of the grammar, we introduce a variable of the same name, and each production rule $X \to \eta$ induces an inequality $X \geqslant \eta$. The only difference to the previous section is that we will solve this system using the powerset lattice over the transition monoid of A seen as Büchi automaton (as introduced in Section 4.5).

Capturing the infinite executions directly is difficult. We circumvent this problem by using the fact that the behavior of derivation processes for $\mathcal{L}^\omega(G)$ is periodic in a sense that we will make precise later. For each pair of nonterminals X, Y, we add a fresh variable $\Lambda_{X,Y}$. Intuitively, the least solution for $\Lambda_{X,Y}$ should represent all finite sentential forms β so that $X \Rightarrow_\ell^* \beta.Y$. To get a finite representation, we do not store β, but the set of all boxes ρ_w where w is a finite word that can be obtained from β. Note that $X \Rightarrow_\ell^* \beta.Y$ means that β is the concatenation of the labels along a finite path from X to Y in the spinal graph. Hence, we may use the spinal graph to define the second part of inequalities as follows.

For each edge $X \xrightarrow{\eta} Z$ in the spinal graph associated to the given grammar and each nonterminal Y, we have an inequality

$$\Lambda_{X,Y} \geqslant \eta.\Lambda_{Z,Y} \,.$$

Additionally, for each nonterminal X, there is the inequality

$$\Lambda_{X,X} \geqslant \varepsilon \,.$$

Intuitively, the first inequality states that if we want to get from nonterminal X to the nonterminal Y in the spinal graph and there is a transition $X \xrightarrow{\eta} Z$, we can take this transition. The remaining task is to reach Y from Z. The second inequality states that if we want to reach X from X, we can simply stay where we are.

We interpret the whole system of inequalities over $(\mathcal{P}(\mathbb{M}_A^{\mathrm{NBA}}), \subseteq)$, the powerset lattice over the transition monoid of A, seen as Büchi automaton. Terminals $a \in \Sigma$, are interpreted as the singleton set $\{\rho_a\}$, the symbol ε is interpreted as $\{\rho_\varepsilon\} = \{\mathrm{id}\}$. Concatenation is interpreted as the element-wise composition of such sets, $R_1 \,.\, R_2 = \{\rho_1 \cdot \rho_2 \mid \rho_1 \in R_1, \rho_2 \in R_2\}$.

The powerset lattice over a finite set like $\mathbb{M}_A^{\mathrm{NBA}}$ satisfies the ACC and the interpretation of all functions is monotonic. Therefore, we can solve the system of inequalities by applying chaotic iteration. The following lemma states that the least solution $\mathrm{sol}\colon \mathcal{V} \to \mathcal{P}(\mathbb{M}_A^{\mathrm{NBA}})$ indeed has the aforementioned properties.

16.3.1 Lemma
We have $\mathrm{sol}(X) = \{\rho_w \mid w \in \mathcal{L}_G(X)\}$ and $\mathrm{sol}(\Lambda_{X,Y}) = \{\rho_w \mid X \Rightarrow_\ell^* \beta.Y, \beta \Rightarrow^* w\}$.

The first part of the lemma can be shown exactly as in the proof of Lemma 16.2.1. We conclude that for every finite sentential form β, $\mathrm{sol}(\beta) = \{\rho_w \mid w \in \mathcal{L}_G(\beta)\}$. The second part of the lemma follows using the definition of the spinal graph. We forgo giving a formal proof.

Representing infinite words

It remains to use the solution to this system to decide whether the inclusion $\mathcal{L}^\omega(G) \subseteq \mathcal{L}^\omega(G)$ holds. Before we can explain how to extract this information from the solution, we need to discuss how the behavior of infinite words in a Büchi automaton can be represented by boxes. In the following, we consider pairs of boxes (τ, ρ), where τ describes a finite prefix and ρ describes a behavior that is repeated infinitely. We write such a pair as $\tau\rho^\omega$, and we define $\mathcal{L}(\tau\rho^\omega) = \mathcal{L}(\tau).\mathcal{L}(\rho)^\omega$. The following lemma states that the languages of the shape $\mathcal{L}(\tau.\rho^\omega)$ cover Σ^ω, and that each $\mathcal{L}(\tau.\rho^\omega)$ is either completely contained in $\mathcal{L}(A)$ or disjoint from it.

16.3.2 Lemma (Sistla, Vardi, and Wolper [SVW87])

a) For every $w \in \Sigma^\omega$, there is a pair of boxes $\tau\rho^\omega$ with $w \in \mathcal{L}(\tau\rho^\omega)$.

b) For every pair of boxes $\tau\rho^\omega$, we have $\mathcal{L}(\tau\rho^\omega) \subseteq \mathcal{L}^\omega(A)$ or $\mathcal{L}(\tau\rho^\omega) \subseteq \overline{\mathcal{L}^\omega(A)}$.

To show the first part, one can apply Ramsey's theorem that we will use later to prove the soundness of our algorithm. The second part follows from the fact that boxes present the behavior of a word in a Büchi automaton. In particular, they do so in a way that is precise enough to check acceptance.

For us to be able to use this characterization, we need a way to check for a given pair of boxes $\tau\rho^\omega$ whether $\mathcal{L}(\tau\rho^\omega) \subseteq \mathcal{L}^\omega(A)$. To this end, we need the notion of a *lasso*, which is modified version of the notion of a *proper language* introduced by Sistla, Vardi, and Wolper [SVW87]. A lasso in a Büchi automaton is the concatenation of a finite path from the initial state to some state, and a cycle containing that state which is repeated ad infinitum. To enforce that the word that is read along this infinite path is accepted, the cycle should also contain a final state. We call a pair of boxes a lasso if each word in its language has a run in the Büchi automaton that is a lasso. The following definition makes precise how this property can be checked.

16.3.3 Definition

A pair of boxes $\tau\rho^\omega$ is a *lasso* if either $\rho = \mathrm{id}$ or if there are sequences of states $q_0, q_1, \ldots, q_p$ and $q'_0, \ldots, q'_c$ so that (1) $q_0 = q_{\mathrm{init}}$, (2) $\tau(q_0, q_1) > 0$, (3) for all $i \in [1, p-1]$, $\rho(q_i, q_{i+1}) > 0$, (4) $q_p = q'_0 = q'_c$, (5) for all $j \in [0, c-1]$, $\rho(q_j, q_{j+1}) > 0$, and (6) there is $j \in [0, c-1]$ with $\rho(q_j, q_{j+1}) = 2$. If $\tau = \mathrm{id}$, we replace the Condition (2) by requiring $q_0 = q_1$.

The definition is visualized in the form of Figure 16.3.a. To better understand the definition, it is helpful to see a box ρ as a directed graph with set of nodes Q. It contains the edge $q \to q'$ iff $\rho(q, q') > 0$. The first three conditions state that there should be a path leading from the initial state to some state q_p such that the first transition is contained in τ and all other transitions are contained in ρ. The other conditions require a cycle containing q_p whose transitions are

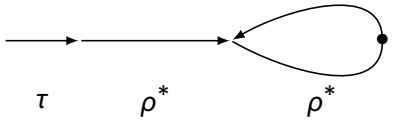


Figure 16.3.a: A schematic depiction of a lasso $\tau\rho^\omega$, where neither τ nor ρ is id. The lasso is a transition in τ, followed by a sequence of transitions in ρ, followed by a cycle of transitions in ρ. One of the transitions along the cycle visits a final state.

contained in ρ. At least one transition along the cycle is labeled by 2, meaning that it visits a final state.

This characterization also explains how to check efficiently whether a pair of boxes is a lasso. It amounts to a polynomial number of reachability checks in directed graphs with $|Q|$ nodes, which is a problem that can be solved in polynomial time.

Let us explain why we consider a pair of boxes $\tau\rho^\omega$ with $\rho = \mathrm{id}$ a lasso. In this case, we have $\mathcal{L}(\tau\rho^\omega) = \mathcal{L}(\tau).\mathcal{L}(\rho)^\omega = \mathcal{L}(\tau).\{\varepsilon\}^\omega = \mathcal{L}(\tau).\varnothing = \varnothing$. Hence, the inclusion $\mathcal{L}(\tau\rho^\omega) \subseteq \mathcal{L}^\omega(A)$ holds.

For a pair of boxes that are not equal to id, we also have that the inclusion holds if and only if the language of the pair is included in the language of the NBA. This is because the definition of a lasso mimics the acceptance condition for Büchi automata: Being a lasso means that every word in the associated language has a run that visits a final state infinitely often. For the formal proof, we would need to invoke Lemma 16.3.2.

Note that here, it is important that we only consider boxes (functions with signature $Q \times Q \to 3$) that actually correspond to elements of the transition monoid, meaning their language is non-empty. If ρ or τ has an empty language, then $\mathcal{L}(\tau\rho^\omega)$ is empty and inclusion in $\mathcal{L}^\omega(A)$ trivially holds, no matter whether $\tau\rho^\omega$ is a lasso.

16.3.4 Lemma (Sistla, Vardi, and Wolper [SVW87])
For boxes ρ, τ with non-empty language, $\mathcal{L}(\rho) \neq \varnothing \neq \mathcal{L}(\tau)$, the pair $\tau\rho^\omega$ is a lasso if and only if $\mathcal{L}(\tau\rho^\omega) \subseteq \mathcal{L}^\omega(A)$.

Lassos characterizing whether a pair of boxes represents a language that is either included or disjoint in the language of an NBA is the final piece we need to solve ωCFL-ωREGINCLUSION using effective denotational semantics. Checking whether the inclusion $\mathcal{L}^\omega(G) \subseteq \mathcal{L}^\omega(A)$ of interest holds amounts to checking whether certain pairs of boxes that are obtained from the least solution to our system of inequalities are lassos.

16.3.5 Theorem
The inclusion $\mathcal{L}^\omega(G) \subseteq \mathcal{L}^\omega(A)$ holds if and only if for each nonterminal X and each $\tau \in \mathrm{sol}(\Lambda_{S,X})$, $\rho \in \mathrm{sol}(\Lambda_{X,X,})$, the pair of boxes $\tau\rho^\omega$ is a lasso.

From the theorem, the solution to ωCFL-ωREGINCLUSION follows immediately. We can compute the least solution to the system of inequalities defined before and obtain the sets of boxes $\Lambda_{S,X}$ and $\Lambda_{X,X}$ for all possible X. It then just remains to check whether all possible pairs of boxes from these sets are lassos.

Before we prove the result, we consider an example.

16.3.6 Example
Consider the alphabet $\{a, r, s, t\}$ whose letters should represent the actions of a server. Letter r represents that the server has received a request, a represents that it has acknowledged a request, s and t are internal actions. A typical liveness property that one would like to verify is that every request gets acknowledged after finite time. Unfortunately, this is not an ω-regular property. Instead, we consider the simpler property that if the server receives infinitely many requests, it also acknowledges infinitely often.

The latter property is ω-regular since it is the language of the Büchi automaton A depicted in Figure 16.3.b.i). Its set of boxes with non-empty language is depicted in Figure 16.3.b.ii).

We consider a server whose behavior is described by the ω-context-free language of the grammar with the production rules

$$X \to rYa \mid XX\,, \qquad\qquad Y \to sYt \mid \varepsilon\,,$$

where X is the initial symbol. The language is $\mathcal{L}^\omega(G) = \{(r(s^{n_i}t^{n_i})a)^\omega \mid n_i \in \mathbb{N} \text{ for all } i\}$.

To set up the system of inequalities associated to that grammar, we first observe that its spinal graph consists of the nodes X and Y and the single edge $X \xrightarrow{X} X$. All transitions but $X \to XX$ are not of the required shape and do not lead to edges in the spinal graph.

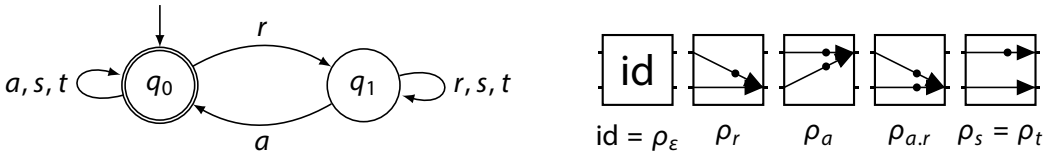
The first part of the system of inequalities for the variables X and Y is the following.

$$X \geqslant r.Y.a \qquad\qquad Y \geqslant s.Y.t$$
$$X \geqslant X.X \qquad\qquad Y \geqslant \varepsilon\,.$$

It can be solved independently of the rest, resulting in the least solution $\mathrm{sol}(X) = \{\rho_a\}$, $\mathrm{sol}(Y) = \{\mathrm{id}, \rho_s\}$. The rest of the system of inequalities describes the variables of the shape $\Lambda_{\_,\_}$:

$$\Lambda_{X,X} \geqslant X.\Lambda_{X,X} \qquad\qquad \Lambda_{X,Y} \geqslant X.\Lambda_{Y,X}$$
$$\Lambda_{X,X} \geqslant \varepsilon \qquad\qquad \Lambda_{Y,Y} \geqslant \varepsilon\,.$$

The variable $\Lambda_{Y,X}$ has no associated inequality because Y has no outgoing edges in the spinal graph. The least solution is $\mathrm{sol}(\Lambda_{X,X}) = \{\mathrm{id}, \rho_a\}$, $\mathrm{sol}(\Lambda_{Y,Y}) = \{\mathrm{id}\}$ and $\mathrm{sol}(\Lambda_{X,Y}) = \mathrm{sol}(\Lambda_{Y,X}) = \varnothing$.



i) Büchi automaton A. ii) The boxes of A with non-empty language.

Figure 16.3.b: An NBA that checks that infinitely many rs implies infinitely many as, and its boxes on the right-hand side. The upper dash on each side of each box ρ represents q_0, the lower one represents q_1. An undotted edge from q to p stands for $\rho(q,p) = 1$, a dotted edge stands for $\rho(q,p) = 1$, i.e. the final state has been visited.

We claim that this solution satisfies the condition in Theorem 16.3.5. There are only two non-trivial cases that we have to check, id $\rho_a^\omega \in \mathrm{sol}(\Lambda_{X,X}) \times \mathrm{sol}(\Lambda_{X,X})$ and $\rho_a \rho_a^\omega \in \mathrm{sol}(\Lambda_{X,X}) \times \mathrm{sol}(\Lambda_{X,X})$. We observe that ρ_a contains the accepting loop $\rho_a(q_0, q_0) = 2$ and that q_0 is the initial state, which means that both pairs of boxes form lassos. This matches the fact that every word in $\mathcal{L}^\omega(G) = \{(r(s^{n_i}t^{n_i})a)^\omega \mid n_i \in \mathbb{N} \text{ for all } i\}$ contains both infinitely many requests and infinitely many acknowledgments.

The rest of this section is dedicated to the proof of Theorem 16.3.5. One part is straightforward.

16.3.7 Lemma
If there is a nonterminal X and boxes $\tau \in \mathrm{sol}(\Lambda_{S,X})$, $\rho \in \mathrm{sol}(\Lambda_{X,X,})$ such that the pair of boxes $\tau\rho^\omega$ is a not a lasso, the inclusion $\mathcal{L}^\omega(G) \subseteq \mathcal{L}^\omega(A)$ does not hold.

Proof:
Assume that $\tau\rho^\omega$ is not a lasso with $\tau \in \mathrm{sol}(\Lambda_{S,X})$, $\rho \in \mathrm{sol}(\Lambda_{X,X,})$. By Lemma 16.3.1, there are finite words v, w with $\rho_v = \tau$, $\rho_w = \rho$ so that $S \Rightarrow_\ell^* \beta X$, $X \Rightarrow_\ell^* \beta' X$ and $\beta \Rightarrow^* v$, $\beta' \Rightarrow^* w$. Consider the infinite word $v.w^\omega$, which is contained in $\mathcal{L}(\tau\rho^\omega)$. Since $\tau\rho^\omega$ is not a lasso, $\mathcal{L}(\tau\rho^\omega)$ is not contained in $\mathcal{L}^\omega(A)$ by Lemma 16.3.4. By Part b) of Lemma 16.3.2, we have $\mathcal{L}(\tau\rho^\omega) \subseteq \overline{\mathcal{L}^\omega(A)}$. This in particular means that $v.w^\omega \notin \mathcal{L}^\omega(A)$. The definitions of v and w yield an infinite left-derivation process, proving that $v.w^\omega \in \mathcal{L}^\omega(G)$. We have obtained $v.w^\omega \in \mathcal{L}^\omega(G) \setminus \mathcal{L}^\omega(A)$, a counterexample to the inclusion. ■

The remaining direction requires more work. In particular, we have to recall Ramsey's theorem which we will need for the proof.

Ramsey's theorem, a generalization of a result by Ramsey [Ram30], is an important result from infinitary combinatorics. It considers undirected complete graphs that are finitely colored, i.e. equipped with a function $\lambda\colon E \to C$ that assign to each pair of distinct nodes $v, v' \in V$, $v \neq v'$ one of finitely many colors $\lambda(\{v, v'\}) \in C$ to the corresponding edge. The coloring is *monochromatic* if all edges are colored by the same color.

16.3.8 Theorem: Ramsey's theorem
An infinite finitely colored undirected complete graph always has an infinite monochromatic complete subgraph.

Ramsey's theorem is standard in the literature, but it seems hard to find a reference that states it in the above form without generalizing it in a way that makes it harder to digest. To solve this issue, we present a proof.

Proof:
Let (V, E) be an infinite undirected complete graph and let $\lambda\colon E \to C$ be a finite coloring of its edges. Using the well-ordering theorem, we may equip the set of nodes V with some well-order $\leqslant$. We define an infinite sequence of triples $\left(V_i, v_i, c_i\right)_{i\in\mathbb{N}}$ so that

- $(V_i)_{i\in\mathbb{N}}$ is a descending chain of infinite subsets of V, i.e. $V_0 \supseteq V_1 \supseteq V_2 \supseteq \ldots$,
- each v_i is a node so that $v_{i+1} \in V_i$, and
- $c_i \in C$ is the color of all edges $\{v_i, v'\}$ for $v' \in V_i$.

We proceed by induction. In the base case, we pick v_0 as the least element of V (with respect to the well-order $\leqslant$) and choose the color c_0 and the set V_0 so that $V_0 = \{v' \in V \mid \lambda(\{v_0, v'\}) = c_0\}$ is infinite. Note that such a color has to exist, because the graph contains infinitely many edges of the shape $\{v_0, v'\}$ but λ assigns only finitely many distinct colors. Also, this choice of V_0 and c_0 means they indeed have the desired properties.

Assume we have defined $(V_0, v_0, c_0), \ldots, (V_n, v_n, c_n)$. We define $(V_{n+1}, v_{n+1}, c_{n+1})$ in the following. We pick v_{n+1} as the least element of V_n, thus satisfying $v_{n+1} \in V_n$. We choose c_{n+1} and V_{n+1} so that $V_{n+1} = \{v' \in V_n \mid \lambda(\{v_{n+1}, v'\}) = c_{n+1}\}$ is infinite. If such a V_{n+1} exists, it obviously is an infinite subset of V_n, and it has to exist using the same line of argumentation as before: V_n is infinite by induction, so the set of edges $\{v_{n+1}, v'\}$ with $v' \in V_n$ is infinite. There must be a color that is assigned to infinitely many of these edges by λ.

The infinite sequence $(c_i)_{i\in\mathbb{N}}$ over the finite set C of colors must contain infinitely many occurrences of some color. Pick one such color c and consider the set of the associated nodes v_i so that the color in the triple (V_i, v_i, c_i) equals c, $V' = \{v_i \mid c_i = c\}$. Since c occurs infinitely often in the sequence of the c_i, the set V' is infinite. We argue that the complete subgraph on the set of nodes V' is monochromatic – all edges between nodes from V' are colored by c.

Consider two distinct nodes from V' and consider the color of the edge between these two nodes. By the definition of V', we may write these nodes as v_n, v_m for some $n, m \in \mathbb{N}$, say with $n < m$. This in particular implies $m > 0$, so we have $v_m \in V_{m-1}$. The sets V_i form a descending chain, so $n < m$ means that $v_m \in V_n$ holds. Now we can use the property of V_n and c_n to obtain that the color of the edge $\{v_n, v_m\}$ is $\lambda(\{v_n, v_m\}) = c_n$. The latter equals c since $v_n \in V'$, which is what we wanted to show. ■

Ramsey's theorem is needed to prove Part a) of Lemma 16.3.2, showing that every word in Σ^ω is in the language of some pair of boxes $\tau\rho^\omega$. The following proof for the remaining part of Theorem 16.3.5 can be seen as an extension of that property.

16.3.9 Lemma
If the inclusion $\mathcal{L}^\omega(G) \subseteq \mathcal{L}^\omega(A)$ does not hold, there is a nonterminal X and boxes $\tau \in \mathrm{sol}(\Lambda_{S,X})$, $\rho \in \mathrm{sol}(\Lambda_{X,X,})$ so that the pair of boxes $\tau\rho^\omega$ is not a lasso.

Proof:
Assume that $w \in \mathcal{L}^\omega(G) \setminus \mathcal{L}^\omega(A)$. We use the nature of right-infinite left-derivation processes: There is an infinite sequence $X_i \to \beta^{(i)}X_{i+1}$ of production rules so that $X_0 = S$ is the initial symbol. Furthermore, there is a decomposition of $w = w^{(0)}w^{(1)}w^{(2)}\ldots$ into finite infixes so that $\beta^{(i)} \Rightarrow^* w^{(i)}$ for all i.

Since there are only finitely many nonterminals, there is at least one nonterminal X that appears infinitely often in the sequence of the X_i. We consider the derivation process for w and merge finite infixes of the above sequence of production rules. The result should be a sequence of sentential forms so that

$$S \Rightarrow^* v^{(0)}X \Rightarrow^* v^{(0)}v^{(1)}X \Rightarrow^* v^{(0)}v^{(1)}v^{(2)}X \Rightarrow^* \ldots,$$

i.e. $w = v^{(0)}v^{(1)}v^{(2)}\ldots$ and each $v^{(i)}$ takes us from one occurrence of X as the rightmost symbol of the sentential form to the next one.

We need to find a pair of boxes $\tau\rho^\omega$ that is not a lasso and whose language contains w. To this end, we use Ramsey's theorem. We construct an undirected complete graph with $\mathbb{N}$ as its set of nodes. The graph is finitely colored by assigning a box to each edge. For $i < j$, we assign $\lambda(\{i,j\}) = \rho_{v^{(i)}.v^{(i+1)}\ldots v^{(j-1)}}$, the box associated to the infix $v^{(i)}.v^{(i+1)}\ldots v^{(j-1)}$ of w. By Ramsey's theorem, Theorem 16.3.8, this graph has a monochromatic infinite complete subgraph, i.e. an infinite subset $M \subseteq \mathbb{N}$ so that there is some box ρ with $\lambda(\{i,j\}) = \rho$ for all $m, m' \in M, m \neq m'$.

We consider a new decomposition of w that is obtained by merging the $v^{(i)}$ according to the subset M. Formally, let $m_0 < m_1 < m_2 < \ldots$ be a strictly ascending sequence that contains exactly the elements of M. We define

$$u^{(0)} = v^{(0)}v^{(1)}\ldots v^{(m_0-1)}, \text{ and}$$

$$u^{(i)} = v^{(m_i)}v^{(m_i+1)}\ldots v^{(m_{i+1}-1)}$$

for $i > 0$. Obviously, $w = u^{(0)}u^{(1)}u^{(2)}\ldots$The first infix $u^{(0)}$ is chosen so that it takes us from the initial symbol to an occurrence of X that corresponds to the least element of M. The other infixes are chosen so that they take us from one occurrence of X to a later one. In particular, we have a sequence of left-derivations $S \Rightarrow^*_\ell \beta^{(0)}.X, X \Rightarrow^*_\ell \beta^{(1)}.X, \ldots$ so that $\beta^{(i)} \Rightarrow^* u^{(i)}$ for all i.

Consider the pair of boxes $\tau\rho^\omega$, where $\tau = \rho_{u^{(0)}}$ is the box associated to $u^{(0)}$ and $\rho = \rho_{u^{(1)}}$ is the box associated to $u^{(1)}$. Since the subgraph on M is monochromatic, ρ is the box associated to $u^{(i)}$ for all $i > 0$. The decomposition $w = u^{(0)}u^{(1)}u^{(2)}\ldots$ is a witness for $w \in \mathcal{L}^\omega(\tau\rho^\omega)$. Since $w \notin \mathcal{L}^\omega(A)$, $\tau\rho^\omega$ cannot be a lasso by Lemma 16.3.4.

To complete the proof, we need to argue that $\tau \in \mathrm{sol}(\Lambda_{S,X,})$, $\rho \in \mathrm{sol}(\Lambda_{X,X})$. With Lemma 16.3.1, we have $\mathrm{sol}(\Lambda_{X,X}) = \{\rho_u \mid X \Rightarrow_\ell^* \beta.X, \beta \Rightarrow^* u\}$. We have argued above that $X \Rightarrow_\ell^* \beta^{(i)}$ and $\beta^{(i)} \Rightarrow^* u^{(i)}$, so $\rho = \rho_{u^{(i)}}$ is indeed contained in $\mathrm{sol}(\Lambda_{X,X})$. The reasoning for $\tau \in \mathrm{sol}(\Lambda_{X,S,})$ is similar. ■

With both lemmas proven, the proof of Theorem 16.3.5 is complete: Each of the lemmas shows one of the required implications by contraposition.

17 Context-free games

Contents

17.1 Context-free regular inclusion games 314
17.2 Solving context-free games 316
17.3 Computing the winner 327
17.4 Computing the strategies 333
17.5 Complexity 339
17.6 Related work 353
17.7 Algorithmics 357
17.8 Deterministic target languages 363
17.9 ω-context-free inclusion games 366

This chapter is dedicated to solving context-free inclusion games. Our goal is to use effective denotational semantics and extend the techniques introduced in the previous chapter. In Chapter 16, we have solved regular inclusion problems (which correspond to verification problems). Here, we want to set up an interpreted system of equations so that its least solution describes the behavior of a game (which corresponds to a synthesis problem, see Section 1.3 of the introduction). The domain that we use for the fixed-point iteration will be so that the certificates, the winning strategies for the game, can be read off from the least solution.

The structure of the chapter is as follows: After providing the formal definitions, we explain how context-free games can be solved using effective denotational semantics by providing the required system of equations and the model. We discuss the properties of algorithm in detail: its soundness, the fact that it has the optimal time complexity, how it can be used to compute winning strategies, and tricks that could be used to speed up an implementation in practice. We also provide an in-depth comparison of our approach to other algorithms that solve similar types of games. Finally, we show that the approach can be extended to the case of ω-context-free games.

Publications

The chapter mostly presents material that has been published in the form of the paper [HMM16] (resp. its full version [HMM16a]). Compared to the paper, the presentation in this thesis has been improved and extended. The last two sections of this chapter present material from the paper [MMN17]. The author's contributions to these publications are discussed in more detail in Chapter 20.

17.1 Context-free regular inclusion games

We define a context-free inclusion game as a game whose plays represent derivation processes of a context-free grammar. The winning condition is membership of the derived word in a regular or ω-regular language. For now, we limit ourselves to the case of a regular target language of finite words.

For the formal development, we use the definition of inclusion games that we have given in Chapter 15. We apply it to the automata-theoretic view on context-free grammars that was based on prefix-growth, see Section 5.1. Formally, this means that we associate to a context-free grammar $G = (N, P, S)$ an LTS whose configurations are either terminal words or sentential forms of the shape $w.X.\beta$, where $w \in \Sigma^*$ is a terminal prefix, $X \in N$ is the leftmost nonterminal and $\beta \in (N \cup \Sigma)^*$ is the rest of the sentential form. There is a transition $w.X.\beta \to w.v.\eta.\beta$ in that LTS if $X \to v.\eta$ is a production rule of the grammar, with v being the terminal prefix of the right-hand side, the largest prefix that exclusively consists of symbols from Σ. The label $\lambda(w.X.\gamma \to w.v.\eta.\gamma) = v$ of this transition is v, the growth of the terminal prefix. Recall that $\mathcal{L}(G)$ is the set of labels that occur along paths from S, the configuration associated to the initial symbol, to terminal words, configurations of the shape $w \in \Sigma^*$. The labels along a path from S to w form exactly the word w.

To turn this LTS in a game arena, we assume that we start with a *game grammar*, a context-free grammar where the nonterminals $N = N_\Box \uplus N_\bigcirc$ are partitioned into the nonterminals $N_\Box$ owned by the universal player and the nonterminals $N_\bigcirc$ owned by the existential player. We usually write such a game grammar as $G = (N_\Box \uplus N_\bigcirc, P, S)$.

This partition of the nonterminals induces a partition of the sentential forms – the configurations of the LTS – based on the leftmost nonterminal. The terminal words have no outgoing transitions in the LTS; we may arbitrarily assign them to one of players, say the existential one. Formally, we have $\Gamma_\bigcirc = \Sigma^*.N_\bigcirc.(N \cup \Sigma)^* \cup \Sigma^*$, $\Gamma_\Box = \Sigma^*.N_\Box.(N \cup \Sigma)^*$. The sentential forms being owned by the owner of the leftmost nonterminal corresponds to the fact that the transitions of the LTS are based on left-derivations. This means that we can think of a play in the game as a left-derivation process in which each player selects the production rules for the nonterminals that she owns.

To equip the game arena with a winning condition, we assume that we are also given a regular target language $\mathcal{L}(A) \subseteq \Sigma^*$ over the same alphabet, where A is a finite automaton. A play is won by the existential player if and only if it produces a finite word that is not contained in $\mathcal{L}(A)$. All other plays are won by the universal player, including all infinite plays.

17.1.1 Definition

A *context-free (regular) inclusion game* is of the shape (G, A), where $G = (N_\Box \uplus N_\bigcirc, P, S)$ is a game grammar and A is an NFA over the same alphabet Σ. The plays are left-derivation processes in which each player picks the replacement rules for her nonterminals. A maximal play is won by the existential player if and only if it is a finite play of the shape $S \to^* w$ with $w \notin \mathcal{L}(A)$.

Most of this chapter is concerned with developing an algorithm that computes the winner of context-free (regular) inclusion games. This is a non-trivial task: While the game can be converted into a reachability game as described in Chapter 15, it is played on an infinite arena that we cannot explicitly store in memory and compute on. Hence, we face the challenge of having to design an algorithm that works with the finite syntax (the game grammar G and the automaton A) describing the infinite game arena in order to solve the game.

Other works in the literature have solved similar types of games, e.g. games played on the transition systems of pushdown automata. We summarize all games in which the game arena is based on a type of system that corresponds to the class of context-free languages, e.g. context-free grammar and pushdown automata, under the term *context-free games*. We defer the discussion of the related work to Section 17.6 in this chapter, after we have presented our algorithm. This will allow us to conduct an in-depth comparison of our solution to other approaches.

17.2 Solving context-free games

To solve context-free games, we want to apply effective denotational semantics. We understand the grammar describing the game arena as a system of equations. We then solve it over a domain that represents the winning condition (the regular target language) of the game. The (least) solution will allow us to read off the winner of the game and the winning strategies.

We start by designing the system of equations.

The system of equations

Let $G = (N_\Box \uplus N_\bigcirc, P, S)$ be the game grammar over Σ that we consider. The construction is similar to the one from Section 16.2. The difference is that we use a system of equations instead of a system of inequalities. In the previous chapter, we were able to create independent inequalities for each of the productions. Conceptually, the iteration solving the system collects them using the join operator of the lattice. In this chapter, the rules for each nonterminal have to be collected and connected using different operators, depending on the owner of the nonterminal. To this end, we use two distinct operators, conjunction $\wedge$ and disjunction $\vee$.

Throughout the rest of this chapter, we take the perspective of the existential player and associate disjunction $\vee$ to her and conjunction $\wedge$ to the universal player. The motivation for this choice is that in order to win, the existential player has the harder task of enforcing a finite derivation.

Formally, the system of equations associated to grammar G is as follows. It uses Σ, the set of terminal symbols and ε as constants, and the binary operators concatenation . , disjunction $\vee$, and conjunction $\wedge$. The system has one variable X for each nonterminal. Its unique defining equation is of the shape

$$X = \eta^{(1)} \overset{\wedge}{\vee} \ldots \overset{\wedge}{\vee} \eta^{(k)},$$

where $X \to \eta^{(1)}, \ldots, X \to \eta^{(k)}$ are all production rules with X as their left-hand side. Here, we see each $\eta^{(i)}$ as a term by using the appropriate constants and concatenation. The operator $\overset{\wedge}{\vee}$ is $\vee$ if $X \in N_\bigcirc$ is owned by the existential player and $\wedge$ otherwise.

If one tried to solve the system of equations with uninterpreted symbols, one would obtain for each nonterminal X a description of the tree of plays from the sentential form X. For example, consider the grammar with the rules $X \to a.Y \mid b.Y$ and $Y \to c \mid d$, where X is owned by the existential and Y by the universal player. The least solution to the resulting system of equations with uninterpreted symbols is $\mathrm{sol}(Y) = c \wedge d$, $\mathrm{sol}(X) = a.(c \wedge d) \vee b.(c \wedge d)$. By being audacious and applying distributivity, even though it does not hold for uninterpreted symbols, the latter can be transformed to $(ac \wedge ad) \vee (bc \wedge bd)$. This formula represents exactly the behavior of the game: First, the existential player can decide whether the first letter should be a or b, then the universal player can pick the second letter. If we are given a target language $\mathcal{L}(A)$, we could find

out the winner of the game from X by checking which of the words ac, ad, bc, bd are contained in that language.

Note that in this toy example, it is possible to explicitly construct this finite representation because the grammar is non-recursive, meaning that it has only finite many derivation processes. In general, the tree of all plays will be infinite and we will not be able to solve the system of equations using uninterpreted symbols. Formally, the domain associated to uninterpreted symbols is a generalization of the language semiring, see Section 16.1. Just as the language semiring, this domain does not satisfy the ascending chain condition and we are unable to obtain the least solution in finite time.

17.2.1 Example

Consider the game grammar G with the production rules $X \to aY \mid \varepsilon$ and $Y \to bX$, where X is owned by the universal player and Y is owned by the universal player. Consider the automaton A with language $(ab)^*$ from Example 4.2.1 resp. Figure 4.2.a. Throughout this chapter, we will consider the context-free inclusion game (G, A) as our running example.

The system of equations associated to (G, A) is

$$X = a.Y \vee \varepsilon\,,$$
$$Y = b.X\,.$$

By substitution, we obtain $X = a.b.X \vee \varepsilon$. This means our system of equations is recursive and we cannot simply explicitly compute its solution with uninterpreted symbols.

It remains to find a model, i.e. a domain and an interpretation of the symbols used in the system of equations. On the one hand, it should be so that $\mathrm{sol}(X)$, the least solution associated to the nonterminal X, still represents the effect of all plays from X. On the other hand, the model should allow us to compute the least solution within finite time. Hence, it should satisfy the ascending chain condition so that we can apply Kleene iteration.

The need for composable information

The key aspect here is that we design the domain so that the information its elements represent is compositional. This is because of the following crucial observation. Assume that $X \to Y.Z$ is the unique production rule for nonterminal X. A play from X that reaches a terminal word will necessarily be of the shape

$$X \Rightarrow_\ell Y.Z \Rightarrow_\ell \beta_1.Z \Rightarrow_\ell \ldots \Rightarrow_\ell \beta_{k-1}.Z \Rightarrow_\ell w.Z \Rightarrow_\ell w.\beta'_1 \Rightarrow_\ell \ldots \Rightarrow_\ell w.\beta'_{m-1} \Rightarrow_\ell w.v\,,$$

where $Y \Rightarrow_\ell \beta_1 \Rightarrow_\ell \ldots \Rightarrow_\ell w$ is a play from Y to a terminal word and similarly, $Z \Rightarrow_\ell \beta'_1 \Rightarrow_\ell \ldots \Rightarrow_\ell v$ is a play from Z to a terminal word. For the other direction, any two

plays from Y resp. Z to terminal words can be combined into a play from X to the concatenation of the terminal words. The reason for this behavior is that we have limited ourselves to left-derivations.

For the system of equations, this means that the solution for the term $Y.Z$ (which will be the same as the solution for X) should be computable using the solutions for Y and Z. Our domain has to be devised so that this compositionality is supported. Suppose for example that we would simply assign $\{0, 1\}$ to each nonterminal, depending on whether the existential player has a winning strategy from the position of the game consisting just of that nonterminal. While this is exactly the information that we want to obtain, there is no hope that we can compute it by a fixed-point iteration over that domain: Consider again $X \to Y.Z$, and assume that we have computed that the existential player wins from Y and Z, which means that she can enforce the derivation of terminal words not contained in $\mathcal{L}(A)$. We get no information on $Y.Z$ (and thus no information on X), however. The fact that we can derive words from Y and Z that are not in the target language does not mean that their concatenation is not in the target language.

The domain

Let A be an NFA representing the regular target language. We interpret a terminal symbol a by the element of the transition monoid ρ_a. Similarly, we interpret ε as $\rho_\varepsilon = \mathrm{id}$. As we have discussed in Section 4.5, the transition monoid element represents the effect that a word has with respect to an NFA, and it does so in a way that can be composed: ρ_a is sufficient to understand whether $w.a.w'$ is contained in the language $\mathcal{L}(A)$ for arbitrary words w, w'.

To be able to interpret the choice operators $\vee$ and $\wedge$, we use positive Boolean formulas as defined in Section 3.4, Boolean formulas without negation. This means that our domain is $\mathrm{pBF}(\mathbb{M}_A)$, the positive Boolean formulas over the transition monoid of A. The transition monoid elements ρ_a are embedded into this domain as atomic formulas in the expected way. Furthermore, this domain allows us to interpret conjunction and disjunction as the respective operations on Boolean formulas.

Recall that $\mathrm{pBF}(\mathbb{M}_A)$ contains the unsatisfiable formula false which we eliminate using the following syntactic rules:

$$F \vee \mathrm{false} = \mathrm{false} \vee F = F \qquad\qquad F \wedge \mathrm{false} = \mathrm{false} \wedge F = \mathrm{false}\,.$$

Consequently, the disjunction or conjunction of two formulas that are not false is never false. (Similar rules would apply for the formula true, but it turns out that true will never occur as a formula when we conduct fixed-point iteration.)

The hard part is interpreting concatenation. Recall from above that a play from $Y.Z$ is essentially a play from Y (with suffix Z) followed by a play from Z (with the outcome of the play from Y as prefix). This in particular means that when the play from Z occurs, the choices that were made when deriving Y are visible to both of the players. Our goal is to interpret concatenation as

formula composition, an operator that mimics this behavior: When composing two formulas, as long as there are choices in the left operand, it resolves these choices while propagating the right operand downwards. As soon as all choices in the left operand are resolved (i.e. the left operand is an atom, an element of the transition monoid), the choices in the right operand are resolved. The atom that is the left operand is propagated downwards. Finally, we obtain a composition of two atoms that we will resolve by composing them using the composition of transition monoid elements.

The formal definition is as follows.

17.2.2 Definition
The composition $F \mathbin{.} H$ of two formulas is inductively defined as follows.

$$\begin{aligned}(F \overset{\wedge}{\vee} F') \mathbin{.} H &= F \mathbin{.} H \overset{\wedge}{\vee} F' \mathbin{.} H \\ \rho \mathbin{.} (H \overset{\wedge}{\vee} H') &= \rho \mathbin{.} H \overset{\wedge}{\vee} \rho \mathbin{.} H' \\ \rho \mathbin{.} \rho' &= \rho \cdot \rho' ,\end{aligned}$$

where $F, F', H, H' \in \mathrm{pBF}(\mathbb{M}_A)$, $\rho, \rho' \in \mathbb{M}_A$, and $\overset{\wedge}{\vee} \in \{\wedge, \vee\}$.

The order

To be able to conduct fixed-point iteration to solve the system of equations, we need to equip the set of positive Boolean formulas over the transition monoid with a partial order. The order should be appropriate in that the operations (conjunction, disjunction, and composition) are monotonic with respect to that order. Furthermore, the least fixed-point solution with respect to this order should indeed capture the effect of the possible plays of the game.

Our idea is that $F \leqslant H$ holds for two formulas if it is easier for the existential player to win the tree of plays described by H than the tree of plays represented by F. More formally, the fixed-point iteration with respect to that order will be set up so that it iteratively explores the outcomes of plays that the existential player can enforce. This means we start with the formula false that represents that the existential player cannot even enforce termination of the play. Later iterations will yield formulas F such that the existential player can enforce visiting an element from any set $M \subseteq \mathbb{M}_A$ if $F(M)$ evaluates to true. Here, we see M as the variable assignment that assign true to ρ if and only if $\rho \in M$. The least fixed point solution associated to a nonterminal will be a formula that characterizes the set of words whose derivation the existential player can enforce from that nonterminal.

However, since the system of equation is set up so that it composes old information to obtain new information, it is not sufficient to define $F \leqslant H$ to hold if the tree of plays represented by H is easier to win than the tree of plays represented by F. We have to require that this holds not only for F and H in isolation, but for F and H in any possible context. More formally, assume that

F represents the plays from X and H represents the plays from Y. Intuitively, we should require that $F \leqslant H$ implies that for any sentential forms β, γ, we have that it is easier for the existential player to win from $\beta.Y.\gamma$ than from $\beta.X.\gamma$. Expressed on the level of formulas, this in particular means that $F \leqslant H$ implies $G' \,.\, F \,.\, G'' \leqslant G' \,.\, H \,.\, G''$ for any two formulas G', G'', which is simply the requirement for composition to be monotonic with respect to our order.

While it would be possible to use the observation in the above paragraph for the formal definition, we will avoid going down this path. An advantage of said definition would be that it is the coarsest order that would be suitable for conducting fixed-point iteration. However, the major drawback is that computing for two formulas whether one is smaller with respect to the order than the other would be very intricate.

We choose to circumvent these issues by considering a more fine-grained order whose definition is not dependent on the notion of context. Then, we will prove that composition is monotonic with respect to that order. As mentioned above, this implies that it has the required properties, i.e. it behaves correctly with respect to all possible contexts. The order that we choose is logical implication. Recall that $F \implies H$ iff for any valuation of the atoms (the transition monoid elements in our case) M, we have $F(M) \leqslant H(M)$, i.e. if $F(M)$ is true, then so is $H(M)$.

Let us argue that implication is a sane choice that matches the intuition provided above. Assume that $F \implies H$ holds. We argue that than indeed H describes a tree of plays in which it is easier for the existential player to win than in the one described by F. Let M be a set of monoid elements so that the existential player can enforce deriving some word w with $\rho_w \in M$. We will later prove that this is equivalent to $F(M) = \text{true}$. Since $F \implies H$, we have $H(M) = \text{true}$ as well, so the existential player can also enforce deriving words whose associated box is in M in the tree of plays described by H.

There is one minor challenge that we need to overcome: Logical implication is not a partial order. In fact, each formula is logically equivalent to infinitely many other formulas. We solve both problems at once, the problem of implication not being a partial order and the problem of our domain being infinite. We simply factorize along logical equivalence $\iff$. Two formulas F, H are logically equivalent, $F \iff H$, if $F \implies H$ and $H \implies F$. If we express the orders as subsets of $\text{pBF}(\mathbb{M}_A)^2$, we may write $\iff \;=\; \implies \cap \impliedby$, where $\impliedby$ is the opposite of $\implies$.

The result of factorizing $\text{pBF}(\mathbb{M}_A)$ with respect to logical implication is $\text{pBF}(\mathbb{M}_A)/\!\iff$. The elements of this set are equivalence classes of positive Boolean formulas over $\mathbb{M}_A$ with respect to $\iff$. On the one hand, this set is finite since the set of atoms $\mathbb{M}_A$ is finite. Each equivalence class of formulas is uniquely determined by the set of subsets of $\mathbb{M}_A$ that satisfy the formulas in the class, and there are only finitely many such sets of subsets. On the other hand, implication is a partial order on these equivalence classes.

We show that the operations of conjunction, disjunction, and composition of formulas are monotonic with respect to logical implication. As a consequence, we obtain that these operations are well-defined on the equivalence classes. In particular, to obtain the conjunction,

disjunction, or composition of two equivalence classes, we can apply the operation to arbitrary representatives and obtain a representative for the equivalence class of the result. Altogether, we will not need to distinguish between equivalence classes and formulas that represent them in the following. This allows us to work with formulas in both the implementation of the resulting algorithm and in the proof of its soundness. However, one has to take into account that in order to check whether the least fixed-point has been reached, one has to check for logical equivalence instead of checking (syntactic) equality.

We summarize the above discussion by providing the formal definition.

17.2.3 Definition
We consider the model $\mathcal{M} = (\mathcal{D}, \mathcal{I})$.

The domain is $\mathcal{D} = (\mathrm{pBF}(\mathbb{M}_A)/\!\iff, \implies)$, equivalence classes of positive Boolean formulas over the transition monoid ordered by implication. It is both a lattice and a monoid. The least element is false, the join operation is disjunction and the meet is conjunction. The monoid operation is formula composition $.$, and the atom $\rho_\varepsilon = \mathrm{id}$ is the neutral element.

The interpretation $\mathcal{I}$ interprets constants $a \in \Sigma \cup \{\varepsilon\}$ as the associated atom ρ_a, conjunction and disjunction as the respective operation, and concatenation as formula composition. All these operations are monotonic with respect to implication.

We have to prove several claims made in this definition. Let us first show that the Boolean formulas equipped with formula composition are a monoid. It is easy to see that the formula ρ_ε is the neutral element. The composition $F \,.\, \rho_\varepsilon$ is a formula with the structure of F in which each atom ρ of F is replaced by itself, since $\rho \,.\, \rho_\varepsilon = \rho \cdot \rho_\varepsilon = \rho$; similar for $\rho_\varepsilon \,.\, F$. Associativity is stated as the following lemma. Its proof is an uninspired and tedious nested induction on the structure of the operands. It is noteworthy that we get associativity as a syntactic equality and not just modulo logical equivalence. Also note that we use G to refer to a formula here. This should not be confused with the game grammar, which we also denote by G.

17.2.4 Lemma
Formula composition is associative: For any formulas F, G, H we have $F \,.\, (G \,.\, H) = (F \,.\, G) \,.\, H$.

Proof:
First note that if any of the three formulas is false, then the result of the composition will be false in both cases. Hence, we will focus on the case that none of the formulas is false in the following. We proceed by a threefold nested induction on the structure of the formulas.

In the base case of the innermost induction, all three formulas are an atom, so $F = \rho, G = \rho', H = \rho''$. We have $F \,.\, (G \,.\, H) = \rho \cdot (\rho' \cdot \rho'')$ and $(F \,.\, G) \,.\, H = (\rho \cdot \rho') \cdot \rho''$. These are equal using the associativity of the composition in $\mathbb{M}_A$.

In the inductive step, we assume that $F = \rho$ and $G = \rho'$ are atomic, while $H = H' \overset{\wedge}{\vee} H''$ is composite (for some operator $\overset{\wedge}{\vee} \in \{\wedge, \vee\}$). We have

$$\begin{aligned} F \,.\, (G \,.\, H) &= \rho \,.\, (\rho' \,.\, H) = \rho \,.\, (\rho' \,.\, (H' \overset{\wedge}{\vee} H'')) \\ &= \rho \,.\, (\rho' \,.\, H' \overset{\wedge}{\vee} \rho' \,.\, H'') = \rho \,.\, (\rho' \,.\, H') \overset{\wedge}{\vee} \rho \,.\, (\rho' \,.\, H'') \\ &= ((\rho \,.\, \rho') \,.\, H') \overset{\wedge}{\vee} ((\rho \,.\, \rho') \,.\, H'') = ((\rho \cdot \rho') \,.\, H') \overset{\wedge}{\vee} ((\rho \cdot \rho') \,.\, H'') \,, \end{aligned}$$

where the penultimate equality uses the induction hypothesis. We also have

$$(F' \,.\, G) \,.\, H = (\rho \,.\, \rho') \,.\, (H' \overset{\wedge}{\vee} H'') = (\rho \cdot \rho') \,.\, (H' \overset{\wedge}{\vee} H'') = ((\rho \cdot \rho') \,.\, H') \overset{\wedge}{\vee} ((\rho \cdot \rho') \,.\, H') \,,$$

so the desired equality holds.

The second induction is on the structure of G. We still assume that $F = \rho$ is atomic, while H is arbitrary. In the base case, G is atomic, so it is already proven by the innermost induction. For the inductive step, consider $G = G' \overset{\wedge}{\vee} G''$. We have

$$\begin{aligned} F \,.\, (G \,.\, H) &= \rho \,.\, ((G' \overset{\wedge}{\vee} G'') \,.\, H) = \rho \,.\, ((G' \,.\, H) \overset{\wedge}{\vee} (G'' \,.\, H)) \\ &= \rho \,.\, (G' \,.\, H) \overset{\wedge}{\vee} \rho \,.\, (G'' \,.\, H) = ((\rho \,.\, G') \,.\, H) \overset{\wedge}{\vee} ((\rho \,.\, G'') \,.\, H) \,, \end{aligned}$$

where the last equality uses the induction hypothesis. Similarly, we get

$$(F \,.\, G) \,.\, H = (\rho \,.\, (G' \overset{\wedge}{\vee} G'')) \,.\, H = ((\rho \,.\, G') \overset{\wedge}{\vee} (\rho \,.\, G'')) \,.\, H = ((\rho \,.\, G')H) \overset{\wedge}{\vee} ((\rho \,.\, G'') \,.\, H) \,,$$

which completes this part of the proof.

In the outermost induction, we induct on the structure of F. The formulas G, H are arbitrary. In the base case, F is atomic, so the second induction proves it. For the inductive step, let $F = F' \overset{\wedge}{\vee} F''$. We have

$$F \,.\, (G \,.\, H) = (F' \overset{\wedge}{\vee} F'') \,.\, (G \,.\, H) = F' \,.\, (G \,.\, H) \overset{\wedge}{\vee} F'' \,.\, (G \,.\, H)$$

and

$$(F \,.\, G) \,.\, H = (F' \,.\, G \overset{\wedge}{\vee} F'' \,.\, G) \,.\, H = ((F' \,.\, G) \,.\, H) \overset{\wedge}{\vee} ((F'' \,.\, G) \,.\, H) \,.$$

Using the induction hypothesis, we obtain the two equalities $F' \,.\, (G \,.\, H) = (F' \,.\, G) \,.\, H$ and $F'' \,.\, (G \,.\, H) = (F'' \,.\, G) \,.\, H$, which concludes the proof. ■

We now proceed to prove that conjunction, disjunction and composition on formulas are monotonic with respect to logical implication. The fact that these operations are well-defined and monotonic on equivalence classes will be a direct consequence of this result.

17.2.5 Proposition
Conjunction, disjunction, and composition of positive Boolean formulas are monotonic with respect to implication.

The proof for conjunction and disjunction is absolutely straightforward. Assume that $F \implies F'$ and $H \implies H'$. To show that $F \wedge H \implies F' \wedge H'$, consider a variable assignment, i.e. a subset $M \subseteq \mathbb{M}_A$ with $(F \wedge H)(M) = \text{true}$. This means $F(M) = H(M) = \text{true}$, and yields $F'(M) = H'(M) = \text{true}$ by assumption. We conclude $(F' \wedge H')(M) = \text{true}$ as desired.

The proof for disjunction is similar. For composition, it is more involved. We state and prove monotonicity in the form of the following lemma.

17.2.6 Lemma
Composition is monotonic with respect to implication: If $F \implies F'$ and $H \implies H'$, then $F \,.\, H \implies F' \,.\, H'$.

Proof:
The structure of the proof is similar to the proof of Lemma 17.2.4. It is a four-fold nested induction on the structure of the formulas. In the outer three inductions, the base case is proven by the previous induction.

In the proof, we let $\{\overset{\wedge}{\vee}, \overset{\vee}{\wedge}\} = \{\wedge, \vee\}$, i.e. $\overset{\wedge}{\vee}$ is one of the operators and $\overset{\vee}{\wedge}$ is the other. We will use $\overset{\wedge}{\vee}$, $\overset{\vee}{\wedge}$, and $\implies$ as syntactic parts of the formula as well as to connect statements in the part of the proof. For example, we may write $(H_1 \implies H') \overset{\vee}{\wedge} (H_2 \implies H')$ to express that $H_1 \implies H'$ and/or $H_2 \implies H'$ hold. This unusual choice prevents the proof from becoming too technical.

Before proceeding with the proof, we make three preliminary observations.

1st Observation: For boxes, $\rho \implies \rho'$ is equivalent to $\rho = \rho'$. Indeed, if $\rho \neq \rho'$, then any assignment that sets ρ to true but ρ' to false is a witness for the implication not holding.

2nd Observation: The formulas $A \implies (B \overset{\wedge}{\vee} C)$ and $(A \implies B) \overset{\wedge}{\vee} (A \implies C)$ are logically equivalent for any three formulas A, B, C. This is simply a consequence of the distributivity of conjunction and disjunction, and can be proven easily by rewriting implications as a disjunctions and vice versa.

3rd Observation: Similarly, the formulas $(A \overset{\wedge}{\vee} B) \implies C$ and $(A \implies C) \overset{\vee}{\wedge} (B \implies C)$ are logically equivalent for any three formulas A, B, C. The proof is similar to the one of the 2nd Observation, but additionally uses De Morgan's laws.

With these preliminaries at hand, we can show the desired statements. Let F, F', H, H' be formulas with $F \implies F'$ and $H \implies H'$. We show $F \,.\, H \implies F' \,.\, H'$.

It is easy to see that if F or H is false, then $F \,.\, H$ will be false and the implication trivially holds. If F' is false, then F has to be false as well since $F \implies F'$, and we are in the previous case, similar for H'. In the rest of the proof, we can assume that none of the formulas is false.

We proceed with the four-fold nested induction.

(1) We induct on the structure of H. In the base case of the innermost induction, all four formulas F, F', H, H' are elements of the transition monoid. Using the 1st Observation, we obtain $F = F'$ and $H = H'$, and we conclude $F \,.\, H = F' \,.\, H'$, which implies the desired implication.

For the step, assume that $H = H_1 \mathbin{\overset{\wedge}{\vee}} H_2$ is composite. Using the 3rd Observation, we obtain $(H_1 \implies H') \mathbin{\overset{\vee}{\wedge}} (H_2 \implies H')$ from $H \implies H'$. We may use the monotonicity of $\overset{\vee}{\wedge}$ together with the induction hypothesis to conclude

$$(F \,.\, H_1 \implies F \,.\, H') \mathbin{\overset{\vee}{\wedge}} (F \,.\, H_2 \implies F \,.\, H') \,.$$

Applying the 3rd Observation in reverse, we obtain $(F \,.\, H_1 \mathbin{\overset{\wedge}{\vee}} F \,.\, H_2) \implies F \,.\, H'$. The premise of that implication is $F \,.\, H$ by the definition of composition, and the conclusion is $F' \,.\, H'$ since $F = F'$ by the 1st Observation.

(2) We induct on the structure of F. In the base case, all formulas but H are atomic, and Part (1) provides the proof. For the step, consider $F = F_1 \mathbin{\overset{\wedge}{\vee}} F_2$. Using the 3rd Observation, we get $(F_1 \implies F') \mathbin{\overset{\vee}{\wedge}} (F_2 \implies F')$. Therefore, the induction hypothesis and the monotonicity of $\overset{\vee}{\wedge}$ yield

$$(F_1 \,.\, H \implies F' \,.\, H') \mathbin{\overset{\vee}{\wedge}} (F_2 \,.\, H \implies F' \,.\, H') \,.$$

Applying the 3rd Observation (in reverse), we get $(F_1 \,.\, H) \mathbin{\overset{\wedge}{\vee}} (F_2 \,.\, H) \implies F' \,.\, H'$. By the definition of formula composition, this is the desired statement.

(3) We proceed by induction on H' while still assuming that F' is atomic. The base case is proven by Part (2). Let $H' = H'_1 \mathbin{\overset{\wedge}{\vee}} H'_2$. We apply the 2nd Observation to get $(H \implies H'_1) \mathbin{\overset{\wedge}{\vee}} (H \implies H'_2)$. We use the induction hypothesis and the monotonicity of $\overset{\wedge}{\vee}$ to obtain

$$(F \,.\, H \implies F' \,.\, H'_1) \mathbin{\overset{\wedge}{\vee}} (F \,.\, H \implies F' \,.\, H'_2) \,.$$

The 2nd Observation in reverse yields $F \,.\, H \implies (F' \,.\, H_1 \mathbin{\overset{\wedge}{\vee}} F' \,.\, H_2)$, where the conclusion is the same as $F' \,.\, H'$. Here, it is crucial that we assume F' to be atomic.

(4) It remains to prove the general case by induction on F'. In the base case, F' is atomic and Part (3) gives us the proof. Now assume $F' = F'_1 \mathbin{\overset{\wedge}{\vee}} F'_2$. Using the 2nd Observation, we get $(F \implies F'_1) \mathbin{\overset{\wedge}{\vee}} (F \implies F'_2)$, use the induction hypothesis and monotonicity to get

$$(F \,.\, H \implies F'_1 \,.\, H') \mathbin{\overset{\wedge}{\vee}} (F \,.\, H \implies F'_2 \,.\, H') \,.$$

We use the 2nd Observation in reverse to get $F \,.\, H \implies (F_1' \,.\, H' \overset{\wedge}{\vee} F_2' \,.\, H')$, and finally use the definition of composition to see that we have indeed shown $F \,.\, H \implies F' \,.\, H'$.

■

Note that the transitivity of implication would allow us to prove the statement by separately showing $F \,.\, G \implies F' \,.\, G$ and $F \,.\, G \implies F \,.\, G'$. We also see this in Step 2 of the proof, since we do not need the assumption that G is atomic.

We conclude that the operations are not only monotonic on formulas, but also monotonic and well-defined on equivalence classes.

17.2.7 Lemma
Disjunction, conjunction and composition are well-defined and monotonic operations on $\mathrm{pBF}(\mathbb{M}_A)/\!\iff$.

Proof:
We formally prove the statement for formula composition. The other two cases are similar.

Let $F, F' \in [F]$ and $H, H' \in [H]$ by two representatives each for two equivalence classes $[F], [H] \in \mathrm{pBF}(\mathbb{M}_A)/\!\iff$. Recall that the composition of equivalence classes is defined by composing arbitrary representatives, $[F] \,.\, [H] = [F \,.\, H]$. This is well-defined: We have that $F' \,.\, H' \iff F \,.\, H$ by the monotonicity of the composition of formulas, and hence $[F \,.\, H] = [F' \,.\, H']$.

Let $[F], [F'], [H], [H']$ be equivalence classes with $[F] \implies [F']$ and $[H] \implies [H']$. Recall that the implication of equivalence classes means that implication holds between arbitrary representatives of these classes. We have that $[F] \,.\, [H]$ is an equivalence class represented by $F \,.\, H$, similar for $[F'] \,.\, [H']$ and $F' \,.\, H'$. The implication $F \,.\, H \implies F' \,.\, H'$ holds by the monotonicity of the composition on formulas. Hence, $[F] \,.\, [H] \implies [F'] \,.\, [H']$ as desired. ■

We have shown that our model satisfies all necessary conditions to solve the interpreted system of equations. To be precise, we have only argued that the interpretation of each function symbols are monotonic. To be able to apply Kleene's theorem, Theorem 16.1.3, we would need that they are join-continuous, which is a stronger requirement. However, we have already observed that our domain is finite, which in particular means that it satisfies the ascending chain condition. As explained in Remark 16.1.4 this means that join-continuity and monotonicity are equivalent. We indeed get that all requirements for applying Kleene iteration are satisfied.

We will discuss in the next section how the solution to the game can be read off from the solution to the system of equations. To complete this section, we continue our example.

17.2.8 Example

Consider (G, A) from Example 4.2.1 resp. Figure 4.2.a. Note that we have depicted all boxes with non-empty equivalence class for automaton A in Figure 4.5.a.

Interpreting the system of equations associated to (G, A) yields

$$X = \rho_a \,.\, Y \;\vee\; \rho_\varepsilon \,,$$
$$Y = \rho_b \,.\, X \,.$$

We use Kleene iteration to compute the first few approximants, obtaining the following values.

$\mathrm{sol}^i(-)$	X	Y
$i = 0$	false	false
$i = 1$	ρ_ε	false
$i = 2$	ρ_ε	ρ_b
$i = 3$	$\rho_{ab} \vee \rho_\varepsilon$	ρ_b
$i = 4$	$\rho_{ab} \vee \rho_\varepsilon$	$\rho_b \,.\, (\rho_{ab} \vee \rho_\varepsilon)$

We resolve the composition to obtain

$$\mathrm{sol}^4(Y) = \rho_b \,.\, (\rho_{ab} \vee \rho_\varepsilon) = (\rho_b \cdot \rho_{ab}) \vee (\rho_b \cdot \rho_\varepsilon) = \rho_{bab} \vee \rho_b \,.$$

We observe that $\rho_{bab} = \rho_b$ for automaton A; hence, $\mathrm{sol}^4(Y) = \rho_b \vee \rho_b$. This formula is syntactically different from ρ_b, but logically equivalent to it. Since we work over the domain of equivalence classes, we obtain $\mathrm{sol}^3(Y) = \mathrm{sol}^4(Y)$. The third iteration corresponds to the least fixed point and $\mathrm{sol} = \mathrm{sol}^3 = \mathrm{sol}^4$ is the least solution to the system of equations.

17.3 Computing the winner

We discuss how to compute the winner of a context-free inclusion game. In the previous section, we have set up a system of equations that represents a game grammar. We then have provided a model, consisting of the domain of equivalence classes of positive Boolean formulas over the transition monoid. We have argued that this model satisfies all requirements that are needed to be able to apply Kleene's theorem and the theory from Section 16.1.

Kleene's theorem proves that the least solution to the system of equations exists, and the fact that our domain is finite means that we can explicitly compute it. The solution is the least fixed point of the function *rhs* obtained by interpreting the right-hand sides of the system of equations. Recall that we have defined $\mathrm{sol}^i = \mathit{rhs}^i(\bot)$ to be the i-fold application of this function to the least element. In our case, the least element is the vector that assigns to each variable the equivalence class of unsatisfiable formulas, i.e. the equivalence class of false. Because the domain satisfies the ACC, the least fixed point sol occurs as $\mathrm{sol}^{i_0} = \mathrm{sol}$ for some $i_0 \in \mathbb{N}$ so that $\mathrm{sol}^{i_0} = \mathrm{sol}^{i_0+1}$. The goal of this chapter is proving that the winner of a context-free inclusion game can be read off from sol.

We consider $M_{\mathrm{rej}} \subseteq \mathbb{M}_A$, the set of *rejecting boxes*

$$M_{\mathrm{rej}} = \{\rho_w \mid w \notin \mathcal{L}(A)\}\,.$$

Recall that if one word in the language $\mathcal{L}(\rho)$ of some box is not contained in $\mathcal{L}(A)$, then no such word is. Furthermore, whether a box is rejecting or not can be determined by checking for the existence of a transition from the initial to a final state.

We call a formula $F \in \mathrm{pBF}(\mathbb{M}_A)$ *rejecting* if it evaluates to true under M_{rej} seen as variable assignment, i.e. the assignment that sets exactly the atoms to true that are contained in M_{rej}. Since being satisfied under a certain assignment is preserved by logical equivalence, this notion can be extended to equivalence classes of formulas by picking an arbitrary representative.

We claim that for each sentential form β, the existential player has a winning strategy for the context-free inclusion game from β if and only $\mathrm{sol}(\beta)$ is rejecting. (Recall that we have extended the definition of sol from variables to arbitrary terms) In particular, the winner of the context-free game when starting from S can be read off from $\mathrm{sol}(S)$.

More formally, we define a partition of the set of sentential forms, the positions of the game,

$$W_{\bigcirc} = \{\beta \in \mathfrak{V} \mid \mathrm{sol}(\beta) \text{ is rejecting } \}$$

and

$$W_{\Box} = \mathfrak{V} \setminus W_{\bigcirc} = \{\beta \in \mathfrak{V} \mid \mathrm{sol}(\beta) \text{ is not rejecting } \}\,.$$

We prove that for each player $\bigstar \in \{\bigcirc, \square\}$, $W_{\bigstar}$ is indeed her winning region.

17.3.1 Theorem
The set $W_{\bigcirc}$ is the winning region of the existential, $W_{\bigcirc}$ is the winning region of the existential player.

17.3.2 Example
We continue Example 17.2.8. The box ρ_b is rejecting while ρ_{ab} and ρ_ε are not. Hence, the formula $\mathrm{sol}(X) = \rho_{ab} \vee \rho_\varepsilon$ is not rejecting and the universal player wins the game (G, A) starting from X. The formula $\mathrm{sol}(Y) = \rho_b$ is rejecting and the existential player wins from Y.

To prove the theorem, we prove a more general statement: For each sentential form β, the behavior of $\mathrm{sol}(\beta)$ under evaluation describes exactly the behavior of the context-free game from β. If $M \subseteq \mathbb{M}_A$ is a set of boxes such that $\mathrm{sol}(\beta)(M) = \mathrm{false}$, then the universal player can prevent the derivation of any word w with $\rho_w \in M$. If $\mathrm{sol}(\beta)(M) = \mathrm{true}$, then the existential player can enforce the derivation of a word w with $\rho_w \in M$. Instantiating these results for the set of rejecting boxes M_{rej} then yields the desired result.

We start by considering the case of the universal player.

17.3.3 Proposition
Let $M \subseteq \mathbb{M}_A$ be a set of boxes and let $\beta \in \vartheta$ be a sentential form. If $\mathrm{sol}(\beta)(M) = \mathrm{false}$, then the universal player has a strategy such that every maximal play conforming to it is either infinite or it ends in a terminal word w with $\rho_w \notin M$.

Proof:
We show that the universal player has a strategy that maintains $\mathrm{sol}(\alpha)(M) = \mathrm{false}$ as an invariant for all positions $\alpha \in \vartheta$ that occur in the play. This invariant guarantees that if a play conforming to it ends after finitely many steps in a terminal word w, we must have $\mathrm{sol}(w)(M) = \rho_w(M) = \mathrm{false}$. This means $\rho_w \notin M$ as desired.

Let us construct a strategy that maintains the invariant. For the initial position, the invariant holds by the assumption. If a sentential form does not contain a nonterminal, it is a deadlock in the game. Hence, we consider $\alpha = w.X.\gamma$, where X is the leftmost nonterminal.

Firstly, we look at the case that the owner of X (and hence the owner of α) is the universal player. We have $\mathrm{sol}(\alpha) = \mathrm{sol}(w.X.\gamma) = \rho_w \,.\, \mathrm{sol}(X) \,.\, \mathrm{sol}(\gamma)$ using the associativity of formula composition. Let $\eta^{(1)}, \ldots, \eta^{(k)}$ be the right-hand sides for nonterminal X, i.e. $X \to \eta^{(1)} \mid \ldots \mid \eta^{(k)}$ are all productions for X. Then we have $\mathrm{sol}(X) = \mathrm{sol}(\eta^{(1)}) \wedge \ldots \wedge \mathrm{sol}(\eta^{(k)})$ by the definition of the

system of equations and the fact that the least solution sol satisfies all equations. Substituting $\mathrm{sol}(X)$ and applying the definition of formula composition yields

$$\mathrm{sol}(\alpha) = \rho_w \,.\, \mathrm{sol}\left(\eta^{(1)}\right) \,.\, \mathrm{sol}(\gamma) \wedge \ldots \wedge \rho_w \,.\, \mathrm{sol}\left(\eta^{(k)}\right) \,.\, \mathrm{sol}(\gamma)\,.$$

Since we had $\mathrm{sol}(\alpha)(M) = \mathrm{false}$ and $\mathrm{sol}(\alpha)$ can be written as the above conjunction, there is at least one j such that $(\rho_w \,.\, \mathrm{sol}(\eta^{(j)}) \,.\, \mathrm{sol}(\gamma))(M) = \mathrm{false}$. We define the strategy so that it picks the move $w.X.\gamma \to w.\eta^{(j)}.\gamma$ induced by the production $X \to \eta^{(j)}$. The formula for the new position is $\mathrm{sol}(w) \,.\, \mathrm{sol}(\eta^{(j)}) \,.\, \mathrm{sol}(\gamma)$, and the construction guarantees that the invariant is maintained.

If the leftmost nonterminal is owned by the existential player, we argue similarly. We have

$$\mathrm{sol}(\alpha) = \rho_w \,.\, \mathrm{sol}\left(\eta^{(1)}\right) \,.\, \mathrm{sol}(\gamma) \vee \ldots \vee \rho_w \,.\, \mathrm{sol}\left(\eta^{(k)}\right) \,.\, \mathrm{sol}(\gamma)\,.$$

This disjunction only evaluates to false if $\rho_w \,.\, \mathrm{sol}(\eta^{(j)}) \,.\, \mathrm{sol}(\gamma)$ evaluates to false for all j. Hence, the invariant is maintained, no matter which move is picked by the existential player. ■

The proof in the case of the existential player is more involved. This is because she has to enforce reaching a word that violates inclusion within finite time.

17.3.4 Proposition
Let $M \subseteq \mathbb{M}_A$ be a set of boxes and let $\beta \in \vartheta$ be a sentential form. If $\mathrm{sol}(\beta)(M) = \mathrm{true}$, then the existential player has a strategy such that every maximal play conforming to it is finite and ends in a terminal word w with $\rho_w \in M$.

Proof:
We prove that the desired property already holds if $\mathrm{sol}^i(\beta)(M) = \mathrm{true}$ for some $i \in \mathbb{N}$. Since we have $\mathrm{sol} = \mathrm{sol}^{i_0}$ for some $i_0 \in \mathbb{N}$ and that $\mathrm{sol}^i(\beta) \implies \mathrm{sol}(\beta)$ for all $i \in \mathbb{N}$, this is a stronger statement. Its advantage is that we can now proceed by induction on i.

In the base case, we consider $i = 0$. Since false is the least element in $\mathrm{pBF}(\mathbb{M}_A)$ with respect to implication, we have $\mathrm{sol}^0(X) = \mathrm{false}$ for all nonterminals. Hence, $\mathrm{sol}^0(\beta)(M) = \mathrm{true}$ is only possible if β does not contain any nonterminal. This means $\beta = w$ is a terminal word and $\mathrm{sol}^0(\beta) = \rho_w$. Thus, position w is a deadlock and $\mathrm{sol}^0(\beta)(M) = \mathrm{true}$ implies $\rho_w \in M$.

For the inductive step, assume that the statement holds for i and consider $i + 1$. Let β be some sentential form. The proof uses an inner induction on the number of nonterminals in β. In the base case, this number is 0 and β is a terminal word. We have already considered this case in the base case of the outer induction.

In the inductive step of the inner induction, we consider $\mathrm{sol}^{i+1}(\beta)$ for some sentential form $\beta = w.X.\gamma$, where X is the leftmost nonterminal in β. We have

$\mathrm{sol}^{i+1}(\beta) = \mathrm{sol}^{i+1}(w.X.\gamma) = \mathrm{sol}^{i+1}(w) \,.\, \mathrm{sol}^{i+1}(X) \,.\, \mathrm{sol}^{i+1}(\gamma) = \rho_w \,.\, \mathrm{sol}^{i+1}(X) \,.\, \mathrm{sol}^{i+1}(\gamma)$ using the associativity of formula composition.

Let us assume that X is owned by the existential player. We may write $\mathrm{sol}^{i+1}(X) = \mathrm{sol}^{i}(\eta^{(1)}) \vee \ldots \vee \mathrm{sol}^{i}(\eta^{(k)})$ where $X \to \eta^{(1)} \mid \ldots \mid \eta^{(k)}$ are all the rules for nonterminal X. Using the definition of formula composition, we obtain

$$\mathrm{sol}^{i+1}(\beta) = \rho_w \,.\, \mathrm{sol}^{i}\left(\eta^{(1)}\right) \,.\, \mathrm{sol}^{i+1}(\gamma) \vee \ldots \vee \rho_w \,.\, \mathrm{sol}^{i}\left(\eta^{(k)}\right) \,.\, \mathrm{sol}^{i+1}(\gamma)\,.$$

Using $\mathrm{sol}^{i+1}(\beta)(M) = \text{true}$, we conclude that there is at least one $j \in [1,k]$ such that $(\rho_w \,.\, \mathrm{sol}^{i}(\eta^{(j)}) \,.\, \mathrm{sol}^{i+1}(\gamma))(M) = \text{true}$.

We would like to apply the induction hypothesis at this point. However, the formula $\rho_w \,.\, \mathrm{sol}^{i}(\eta^{(j)}) \,.\, \mathrm{sol}^{i+1}(\gamma)$ is not of the required shape as we use different approximants for different parts of the formula. To solve this problem, we apply induction only to $\mathrm{sol}^{i}(\eta^{(j)})$. To take the rest of the sentential form $w.\eta^{(j)}.\gamma$ into account, we consider a different variable assignment.

We define the assignment M' by $\rho(M') = (\rho_w \,.\, \rho \,.\, \mathrm{sol}^{i+1}(\gamma))(M)$. Intuitively, it evaluates the given box under M in the context of w on the left-hand side and γ on the right-hand side. By the definition of formula composition, we have that $F(M') = (\rho_w \,.\, F \,.\, \mathrm{sol}^{i+1}(\gamma))(M)$ for any formula F. Indeed, formula F is the leftmost operand of $\rho_w \,.\, F \,.\, \mathrm{sol}^{i+1}(\gamma)$ that may contain conjunctions or disjunctions, which will be resolved first when evaluating the composition. In particular, we have $\mathrm{sol}^{i}(\eta^{(j)})(M') = (\rho_w \,.\, \mathrm{sol}^{i}(\eta) \,.\, \mathrm{sol}^{i+1}(\gamma))(M) = \text{true}$. We may use the induction hypothesis of the outer induction to obtain that the existential player has a strategy s' from $\eta^{(j)}$ to enforce the derivation of a terminal word v with $\rho_v(M') = \text{true}$.

We define the strategy s from β to first pick the move $\beta \to w.\eta^{(j)}.\gamma$ that is induced by the production $X \to \eta^{(j)}$. Then, it imitates s' (by ignoring the prefix w and the suffix γ) until $\eta^{(j)}$ has been derived to a terminal word. After this process has been completed, we end up with $w.v.\gamma$ such that $\rho_v(M') = \text{true}$. By the definition of M', we have $\rho_v(M') = (\rho_w \,.\, \rho_v \,.\, \mathrm{sol}^{i+1}(\gamma))(M) = (\mathrm{sol}^{i+1}(w.v.\gamma))(M) = \text{true}$.

The sentential form $w.v.\gamma$ contains strictly fewer terminals than β, since we have replaced X by a terminal word. We may now use the induction hypothesis of the inner induction to obtain that there is a strategy s'' from $w.v.\gamma$ that is guaranteed to derive a terminal word w' with $\rho_{w'}(M) = \text{true}$, i.e. $\rho_{w'} \in M$, within finitely many steps.

We complete the definition of s by letting s behave as s'' does from $w.v.\gamma$ on.

The case that X is owned by the universal player is similar. The formula $\mathrm{sol}^{i+1}(X)$ is a conjunction with one conjunct $\rho_w \,.\, \mathrm{sol}^{i}(\eta^{(j)}) \,.\, \mathrm{sol}^{i+1}(\gamma)$ for each rule $X \to \eta^{(j)}$. We have that $\mathrm{sol}^{i+1}(X)(M) = \text{true}$ implies that each conjunct evaluates to true under M. For any conjunct j, we proceed as above: After defining a new assignment M'_j with $\rho(M'_j) = (\rho_w \,.\, \rho \,.\, \mathrm{sol}^{i+1}(\gamma))$, we can apply the induction hypothesis of the outer induction. We get a strategy s'_j that enforces

the derivation of a terminal word v_j with $(\mathrm{sol}(i+1)(w.v_j.\gamma))(M) = \mathrm{true}$. Applying the inner induction to $w.v_j.\gamma$ gives us a strategy s''_j that derives a word w' with $\rho_{w'} \in M$. We define the strategy s for the existential player as follows: In position $w.X.\gamma$, the strategy reads the production rule $X \to \eta^{(j)}$ that is picked by the universal player. Afterwards, it imitates s'_j until the sentential form $w.v_j.\gamma$ has been derived. Finally, it behaves as s''_j until the end of the play. The result is again that we guarantee the derivation of a terminal word w' with $\rho_{w'} \in M$ within finitely many steps. ■

17.3.5 Example

We continue Example 17.3.2 to complete our running example. The formula associated to $\mathrm{sol}(Y) = \rho_b$ is rejecting, so the existential player should win the game starting from Y.

In the first move, the universal player has no choice but to derive $b.X$, and indeed $\mathrm{sol}(b.X) = \mathrm{sol}(Y) = \rho_b$ is still rejecting. It is now the existential players choice to replace X either using the production $X \to \varepsilon$ or $X \to a.Y$. The productions lead to the formulas $\mathrm{sol}(b) = \rho_b$ and $\mathrm{sol}(b.a.Y) = \rho_{ba} \,.\, \mathrm{sol}(Y) = \rho_{bab} = \rho_b$, respectively. Either move maintains the invariant of being rejecting. However, this is insufficient to ensure that the existential player wins. If she picks the production $X \to a.Y$ whenever she has to replace X, the resulting play is infinite. Even though it consists entirely of positions whose associated formula is rejecting, the existential player loses. To ensure that she wins, she has to pick the production $X \to \varepsilon$ after finitely many steps, e.g. the first time she replaces X.

We show that we read off this property using the approximants. Recall that the least solution corresponds to the 3rd approximant. The game starts from Y and $\mathrm{sol}^3(Y) = \rho_b$ is rejecting. After one move, the game is in bX and $\mathrm{sol}^2(bX) = \rho_b \,.\, \mathrm{sol}^2(X) = \rho_b \cdot \rho_\varepsilon = \rho_b$ is still rejecting. Applying $X \to \varepsilon$ yields the position b with $\mathrm{sol}^1(b) = \rho_b$ being rejecting. Using $X \to aY$ however yields baY with $\mathrm{sol}^1(b.a.Y) = \rho_{ba} \,.\, \mathrm{sol}^1(Y) = \rho_{ba} \,.\, \mathrm{false} = \mathrm{false}$. The formula false is unsatisfiable and evaluates to false under any assignment. In particular, it is not rejecting. Hence, the production $X \to \varepsilon$ should be used to ensure that the play terminates and that the existential player wins.

With both propositions at hand, it is not difficult to obtain the proof of the theorem.

Proof of Theorem 17.3.1:

We first show that $W_\square = \{\beta \in \mathfrak{I} \mid \mathrm{sol}(\beta) \text{ is not rejecting } \}$ is a subset of the winning region of the universal player. This means that the universal player wins from all positions whose associated formula is not rejecting.

Let $\beta \in \mathfrak{I}$ be so that $\mathrm{sol}(\beta)(M_{\mathrm{rej}}) = \mathrm{false}$. By Proposition 17.3.3 the universal player has a strategy from β that enforces plays that are either infinite or derive a word w with $\rho_w \notin M_{\mathrm{rej}}$. By the definition of M_{rej}, the latter means $w \in \mathcal{L}(A)$. Hence, such a strategy is a winning strategy for the universal player in the context-free regular inclusion game.

It remains to show that the existential player wins from all positions in $W_{\bigcirc} = \{\beta \in \theta \mid \mathrm{sol}(\beta) \text{ is rejecting}\}$. Because $W_{\bigcirc}$ and $W_{\Box}$ form a partition of the set of positions ϑ, this completes the proof. If $\mathrm{sol}(\beta)$ is rejecting, i.e. $\mathrm{sol}(\beta)(M_{\mathrm{rej}}) = \mathrm{true}$, then Proposition 17.3.4 guarantees the existence of a strategy from β that derives a word w with $\rho_w \in M_{\mathrm{rej}}$ within finitely many steps. By the definition of M_{rej}, $\rho_w \in M_{\mathrm{rej}}$ implies $w \notin \mathcal{L}(A)$. Hence, such a strategy is a winning strategy for the existential player. ■

17.3.6 Remark

There is a different approach to proving the soundness of our procedure. For some fixed initial position, one can see the tree of all plays from that position as an infinite formula over $\mathbb{M}_A$. To this end, we see inner nodes of the tree as disjunctions or conjunctions, depending on the owner of the corresponding position. We see a leaf as the atom ρ_w, where w is the unique word that has been derived by the play reaching that leaf. The result is a formula that potentially has infinite depth.

We equip it with an evaluation semantics that gets rid of this problem: We evaluate boxes depending on a variable assignment, i.e. a subset $M \subseteq \mathbb{M}_A$. We then propagate the values for the boxes upwards using the usual rules for conjunction and disjunction. This evaluation semantics essentially ignores all infinite paths, i.e. it evaluates them to false.

The result is that to each infinite formula F, we associate a function from $\mathcal{P}(\mathbb{M}_A) \to \{\mathrm{true}, \mathrm{false}\}$ that evaluates the formula under the given variable assignment. However, it is well known that all functions of this type can be represented by finite formulas. Hence, there is a finite formula that is logically equivalent to the infinite one.

To prove the soundness of our algorithm, one first shows that the evaluation of the infinite formula under the variable assignment M_{rej}, the rejecting boxes, indeed yields the winner of the game. Secondly, one shows that the formula associated to the initial position by the least solution to the system of equations is equivalent to the infinite formula. We have made this approach formal in Section 4 of the full version of our paper [HMM16a].

The approach that we have presented above is not only easier. It also allows us to extract the winning strategies for each of the players, as we will see in the next section.

17.4 Computing the strategies

We show that the information provided by the least solution to the system of equations can also be used to design winning strategies for each of the players. In this section, we will assume that the size of the game grammar G and the size of the automaton A are constants. (We will analyze the computational complexity of solving context-free games, measured in the size of the given grammar and the given automaton in the next section.) This assumption allows us to assume that we can precompute $\mathrm{sol}(a)$ for all $a \in N \cup \Sigma \cup \{\varepsilon\}$ in constant time. Since G and A are of constant size, so is $\mathrm{sol}(X)$ for each $X \in N$. Furthermore, this assumptions means that we can compute the composition of two formulas in constant time.

Representing the winning region

We start by considering a representation of the two winning regions $W_\Box$ and $W_\bigcirc$. Serre [Ser03] has shown that a context-free game with an (ω-)regular winning condition always has an (ω-)regular winning region. Correspondingly, we are able to design for each player $\star \in \{\bigcirc, \Box\}$ a finite automaton over $N \cup \Sigma$ that accepts exactly the sentential forms β from $W_\star$.

Formally, we define the DFA B as follows. Its set of states is $\mathrm{pBF}(\mathbb{M}_A)/\!\iff$, the sets of equivalence class of positive Boolean formulas over $\mathbb{M}_A$. The initial state is the equivalence class of ρ_ε, which is the neutral element of $\mathrm{pBF}(\mathbb{M}_A)/\!\iff$ seen as monoid. If the automaton is in state F, where F is the representative of some equivalence class, and reads the letter $a \in (N \cup \Sigma)$, it goes to state $F \,.\, \mathrm{sol}(a)$. Note that $\mathrm{sol}(a)$ is simply ρ_a for $a \in \Sigma$; otherwise, we take $\mathrm{sol}(a)$ for $a \in N$ from the precomputed solution to the system of equations.

To obtain the automaton $B_\bigcirc$ representing the winning region of the existential player, we make all states F final where F is rejecting, which means that $F(M_{\mathrm{rej}}) = \mathrm{true}$. (Recall that $M_{\mathrm{rej}} = \{\rho_w \mid w \notin \mathcal{L}(A)\}$ is the set of rejecting boxes.) The automaton $B_\Box$ representing the winning region of the universal player is the complement of $B_\bigcirc$, i.e. B with all states representing non-rejecting formulas being final.

Automaton B ensures that after reading sentential form $\beta \in \mathfrak{H}$, it is in state $\mathrm{sol}(\beta)$. Hence, Theorem 17.3.1 proves that $\mathcal{L}(B_\star) = W_\star$ for $\star \in \{\bigcirc, \Box\}$ as desired. Also note that given a sentential form β, automaton B allows us to decide the winner of the game from β in time linear in $|\beta|$ (under the assumption that the size of the grammar and the automaton are constants).

Winning strategies for the universal player

We continue by discussing how winning strategies for both of the players can be obtained. Assume that the play starts from a position whose associated formula is not rejecting. By Proposition 17.3.3, this means that this position is in the winning region of the universal player. Actually, the proof of the proposition also provides a winning strategy: It is a safety strategy that simply maintains the invariant of the formula associated to the current position not being rejecting.

One can implement this strategy in the following way. Assume that the current position is $w.X.\beta$ and the universal player should select a production rule to replace X. For each production rule $X \to \eta$, we compute the formula $\mathrm{sol}(w.\eta.\beta)$ that results from applying the rule. We select the first rule so that $\mathrm{sol}(w.\eta.\beta)$ is not rejecting. The proof of the aforementioned proposition shows that if the formula for $w.X.\beta$ is not rejecting, then one can find such a rule. It also shows that the existential player has no choice but to preserve the invariant of the formula not being rejecting. Hence, this strategy is indeed a winning strategy.

However, this strategy needs linear time for each move: Assume that we consider a play in which already n moves have been made. This means that the length of the sentential form that is the current position is in $\mathcal{O}(n)$. Indeed, in each step, the sentential form grows by at most the length of the longest right-hand side of any production rule, which is less than the size of G which we assume is a constant. Our strategy requires computing and evaluating a constant number of formulas (at most one for each rule of the grammar), each formula associated to a sentential form of size $\mathcal{O}(n)$. The strategy does not reuse the information that has been computed in step $n-1$ to make a decision in step n. It cannot be implemented by a strategy automaton as defined in Chapter 15, because such a strategy is only allowed to use constant time – one step of the strategy automaton – to update its state after each move.

In the following, we design a strategy that is more efficient. It is a *pushdown strategy*, a strategy that is defined by a deterministic pushdown automaton. The input and output alphabet of this pushdown automaton is the set P of production rules of the grammar. Assume that the current sentential form is $w.X.\beta$, where w is the terminal prefix, X is the leftmost terminal, and $\beta = \beta_1 \dots \beta_m$ is the rest of the sentential form. To simplify the notation, we define $\beta_0 = X$. The state of the automaton will be the box $\rho_w \in \mathbb{M}_A$ associated to the terminal prefix w. On its stack, it stores for each symbol β_i with $i \in [0, m]$ that is a nonterminal a tuple $(\beta_i, \mathrm{sol}(\beta_{i+1} \dots \beta_m))$ consisting of the symbol $\beta_i \in \Sigma \cup N$ and the formula $\mathrm{sol}(\beta_i \dots \beta_m)$ associated to the rest of the sentential form (without that symbol). The stack is organized so that the information for symbols that are further to the left of the sentential form occur on the top of the stack, with the tuple for X being the top-of-stack. Assume that the automaton reads a move of the game, i.e. the application of a production rule $X \to \eta$ to the leftmost nonterminal by any of the players. Let $\eta = v.\eta'$, where v is the terminal prefix and $\eta' = \eta'_1 \dots \eta'_\ell$ is the rest of the sentential form. The automaton pops the top-of-stack symbol, say (X, F), and replaces it by

$$(\eta'_1, \mathrm{sol}(\eta'_2 \dots \eta'_\ell) \,.\, F), (\eta'_2, \mathrm{sol}(\eta'_3 \dots \eta'_\ell) \,.\, F), \dots, (\eta'_{\ell-1}, \mathrm{sol}(\eta'_\ell) \,.\, F), (\eta'_\ell, F) \,.$$

If some of the η'_i are terminals, we omit the corresponding stack symbols (but we still include η'_i in the compositions that form the second component of the other stack symbols). Note that because we assume the size of the grammar to be constant, this is a constant number of stack symbols. The automaton also updates its internal state to $\rho_w \cdot \rho_v$, i.e. the box associated to the new terminal prefix. This transition maintains the shape of the stack as described above.

To output a move for the universal player, the automaton peeks at the top-of-stack (X, F), the information for the symbol X that should be replaced. It then computes the formula $\mathrm{sol}(w.\eta.\beta)$ for all possible successors, the number of which is constant. Each such formula is of the shape $\rho_w \,.\, \mathrm{sol}(\beta) \,.\, F$, where ρ_w and F can be read off from the internal state and stack, respectively. It then prints out any move $X \to \beta$ so that the resulting formula is not rejecting. Note that one could modify the automaton so that all information that is needed to determine the next move can be read off from the internal state without peeking at the top-of-stack.

Assuming that the play starts from S, the initial symbol of the grammar, we initialize the pushdown automaton with state ρ_ε, the box associated to the empty terminal prefix, and stack content (S, ρ_ε). Throughout the play, we update the automaton with the moves in the game as detailed above, and whenever the universal player has to make a move, we query the automaton. For each move in the game, the automaton only needs a single transition to update its internal state. Even after the game has already been played for n steps, the automaton can give us the next move in constant time. In principle, the automaton implements the same safety strategy as before, but it stores the computed information in a clever way that allows it to use only constant time instead of linear time.

One can look at the structure of the automaton in more detail and note that it is a *synchronized pushdown automaton*. This concept has been introduced by Walukiewicz [Wal01] in his study of context-free games. Assume that we convert the context-free grammar that describes the game arena into a pushdown automaton P_{arena}. For each finite play of a context-free game, the configurations reached in the strategy automata and in the automaton P_{arena} for the arena have the same height. Namely, the stack of both automata will contain one stack symbol for each nonterminal in the current sentential form. This property allows us to take a product of the strategy automaton $P_{s_\Box}$ and the automaton for the arena, resulting in a single pushdown automaton that on its stack stores tuples of stack symbols. (Note that in general, the product of two pushdown automata results in a multi-pushdown automaton with two stacks that cannot be merged, since the heights of the stacks may not coincide.)

We then use the output function of the strategy automaton to resolve all choices of the universal player in the game. The result is a nondeterministic pushdown automaton, say PDA $P_\bigcirc = P_{\text{arena}} \times P_{s_\Box}$, the nondeterminism of which represents the choices of the existential player. One can verify that the strategy for the universal player that was defined by $P_{s_\Box}$ is indeed a winning strategy by checking that $\mathcal{L}(P_\bigcirc) \subseteq \mathcal{L}(A)$: No finite play of the game that conforms to the strategy defined by $P_{s_\Box}$ produces a counterexample to inclusion.

Winning strategies for the existential player

Let us now consider the case of a sentential form in the winning region of the existential player, i.e. the associated formula is rejecting. The winning strategy for the existential player is necessarily more involved than the one for the universal player. Instead of simply maintaining an invariant, it has to enforce reaching a counterexample to inclusion after finitely many steps.

There is a brute-force approach to computing such a strategy. Consider the infinite tree of plays from the given initial position, and consider the subtree that only contains the plays conforming to a winning strategy for the existential player. The strategy has to enforce that all its plays are finite, and the out-degree of the tree is limited by the number of production rules of the grammar. Kőnig's lemma [Kőn27] proves that a tree with finite height and finite out-degree is necessarily finite. Hence, the subtree of plays associated to a winning strategy is finite.

This result allows us to simply explore the tree of all plays in a breadth-first manner and search for a winning strategy of the existential player. Such a strategy corresponds to a finite subtree that satisfies the following conditions: (1) All the leaves are rejecting in that they are labeled by rejecting boxes. (2) Each inner node owned by the existential player is rejecting in that it has a successor in the subtree that is rejecting as well. (3) Each inner node owned by the universal player is rejecting in that all its successors are present in the subtree and are rejecting. If we start from an initial position whose formula is rejecting, we know that such a finite subtree has to exist, and it can be found by enumeration.

The size of this subtree is immense but constant. We can use i_0, the number of steps after which the fixed-point iteration terminates with the least solution, to estimate its height. One has to take into account that the fixed-point iteration evaluates the equation for every nonterminal in every step. Hence, a bound i_0 on the steps of the iteration leads to a bound $|G|^{i_0}$ on the height of the tree, where $|G|$ is a coarse estimation for the size of any right-hand side in the system of equations. Assuming that $|G|$ and $|A|$ are constant, this number is constant, which allows us to explicitly compute and store a positional winning strategy for the existential player. However, this approach is purely theoretical: We will later discuss that i_0 may be up to doubly exponential in $|A|$, which means that the height of the tree is up to triply exponential in the input size. The strategy is finite, but even assuming that $|A|$ is small (and hence so is i_0), its representation may be too large to store.

To obtain a strategy that is more practical, we use a different approach that is similar to the pushdown safety strategy for the case of the universal player. To understand the construction of the strategy, it is helpful to consider an alternative proof of Proposition 17.3.4 that uses a single induction instead of a nested one. We have provided this proof in [HMM16a], the full version of our paper. Here, we briefly recall the important parts and refer to the publication for the technical details. The proof of Proposition 17.3.4 that we have given in this thesis uses the same line of argumentation, but due to the nested induction, the technical details are somewhat hidden.

The alternative proof uses a single induction over the set $\mathbb{N}^*$ of sequences of natural numbers. The order is defined so that a sequence becomes smaller by replacing a positive number i in it by a sequence of numbers $i_1 \dots i_m$, all of them strictly smaller than i. One can show that this order is well-founded, meaning that all its strictly descending chains are finite, which is needed for a valid proof. Intuitively, we consider sequences of numbers that associate to each symbol β_j in a sentential form β a number i_j. When computing the formula associated to β, we should not use $\mathrm{sol}(\beta_j)$, the least fixed point solution associated to β_j, but rather $\mathrm{sol}^{i_j}(\beta_j)$,

the i_j^{th} approximant. When we replace a nonterminal X by the right-hand side of a production rule $X \to \eta$, this corresponds to one evaluation of the equation for X. Accordingly, we replace the number i associated to X by the sequence $(i-1)\dots(i-1)$ of length $|\eta|$.

We use this proof concept to define a pushdown strategy. As in the case of the strategy for the universal player, the automaton stores the box associated to its terminal prefix in its control state and information on the rest of the sentential form on its stack. For each nonterminal in the current sentential form, we store a triple (X, i, F) consisting of the symbol itself, a number $i \in \mathbb{N}$ that tells us which fixed-point approximant to consider, and a formula that describes the rest of the sentential form in a manner that we will detail below.

Assuming that the play starts from S, the initial control state is ρ_ε and the initial stack content is $(S, i_0, \rho_\varepsilon)$. Here, i_0 is the number of steps after which the fixed-point iteration terminates, i.e. the i_0^{th} approximant equals the least solution. Assume that the produces has arrived in a sentential form $w.X.\beta$. This in particular means that the current control state is ρ_w and that the stack contains for each nonterminal β_j in $X.\beta$ a symbol (β_j, i_j, F_j). When the rule $X \to \eta$ is applied, the stack is updated as follows. We first remove the top-of-stack (X, i_X, F_X) and remember $i_X \in \mathbb{N}$ and the formula F_X. Assume that the rightmost symbol η_ℓ of η is a nonterminal. We then push $(\eta_\ell, i_X - 1, F_X)$ onto the stack. If the penultimate symbol $\eta_{\ell-1}$ is a nonterminal too, we then push $(\eta_{\ell-1}, i_X - 1, \mathrm{sol}^{i_X-1}(\eta_\ell)\ .\ F)$. Assume that $\eta_{\ell-2}$ is a terminal while $\eta_{\ell-3}$ is a nonterminal again. We skip the stack symbol for $\eta_{\ell-2}$ and store $(\eta_{\ell-3}, i_X - 1, \rho_{\eta_{\ell-2}}\ .\ \mathrm{sol}^{i_X-1}(\eta_{\ell-1})\ .\ \mathrm{sol}^{i_X-1}(\eta_\ell)\ .\ F)$. We proceed until all of η but its terminal prefix v has been processed. We then store the terminal prefix by updating the control state to $\rho_w \cdot \rho_v$.

The stack of the automaton stores for each symbol which fixed-point approximant to consider, and the formula that one gets for the rest of the sentential form by using the stored fixed-point approximants for each of the symbols. Replacing a symbol X by a right-hand side η corresponds to evaluating the equation for X once, which means that we go from the i_X^{th} fixed point approximant for X to the $(i_X - 1)^{\text{th}}$ fixed point approximant for the symbols in η.

To output a move, the automaton proceeds as follows. It peeks at (X, i_X, F_X), the top-of-stack that provides information for the symbol X that should be replaced. It then finds a rule $X \to \eta$ so that $\rho_w\ .\ \mathrm{sol}^{i_X-1}(\eta)\ .\ F_X$ is rejecting. To do so, it simply iterates over all production rules and prints the first one that has the desired property. One can prove that such a rule has to exist as in the proof of Proposition 17.3.4.

The automaton maintains the invariant that if the control state is ρ_w and the top-of-stack is (X, i_X, F_X), then $\rho_w\ .\ \mathrm{sol}^{i_X}(X)\ .\ F_X$ is rejecting. Note that this invariant is preserved not only by the moves selected by the automaton, but also by any move picked by the universal player. The index i_Y associated to any nonterminal Y in the sentential form is always a positive number. As soon as the index associated to the top-of-stack X is 1, the next replacement step necessarily replaces X by a terminal word. The fact that the aforementioned order on $\mathbb{N}^*$ is well-founded means that after finitely many steps, all nonterminals have been replaced and the play ends with

a terminal word. The invariant guarantees that this terminal word has a rejecting box, meaning that it is not contained in $\mathcal{L}(A)$.

If the initial position S satisfies the invariant, i.e. $\mathrm{sol}(S) = \mathrm{sol}^{i_0}(S)$ is rejecting, then the automaton indeed implements a winning strategy for the existential player. It has the same properties as the one for the universal player: Each update of the configuration of the automaton is a single move. By querying the automaton, the strategy can output the next move in constant time. Furthermore, the automaton is a synchronized pushdown automaton, having one symbol on its stack for each nonterminal of the sentential form. To verify that it represents a winning strategy, one can compute $P_{\square} = P_{s_{\bigcirc}} \times P_{\text{arena}}$, the product of the strategy automaton $P_{s_{\bigcirc}}$ and the automaton P_{arena} representing the context-free game grammar. The result is a pushdown automaton the nondeterminism of which represents the choices of the universal player. If $P_{s_{\bigcirc}}$ represents a winning strategy, we have that all runs of $P_{\square}$ are finite and that $\mathcal{L}(P_{\square}) \subseteq \overline{\mathcal{L}(A)}$.

17.5 Complexity

We discuss the computational complexity of solving context-free regular inclusion games. Formally, the associated decision problem is defined as follows.

Solving context-free regular inclusion games	
Given:	Game grammar $G = (N_\Box \uplus N_\bigcirc, P, S)$, NFA $A = A = (Q, \delta, q_{\text{init}}, Q_{\text{final}})$.
Question:	Does the existential player have a winning strategy for (G, A) from the position S?

Our main result is that this problem is complete for 2EXP, the class of problems solvable within doubly exponential time, with respect to polynomial-time reductions.

17.5.1 Theorem
Solving context-free regular inclusion games is 2EXP-complete.

As usual, the proof of a result of this type decomposes into showing membership and hardness separately. To prove the upper bound, we will show that our algorithm based on effective denotational semantics solves context-free games in 2EXP. Together with the lower bound, we obtain that our algorithm has the optimal time complexity.

Membership / Upper bound

We prove that context-free regular inclusion games can be solved in doubly exponential time. To this end, we analyze the running time of our algorithm. We start by recapping the algorithm. Given an instance (G, A), it works as follows.

1. Construct the system of equations representing G as described in Section 17.2.
2. Solve the system of equations interpreted over $\mathrm{pBF}(\mathbb{M}_A)$.
   - Initialize $\mathrm{sol}^0(X) = \mathrm{false}$ for all nonterminals X.
   - Starting with $i = 0$, compute sol^{i+1} by evaluating the interpreted right-hand sides of the equations at sol^i:

     $$\mathrm{sol}^{i+1} = rhs\left(\mathrm{sol}^i\right).$$

     While $\mathrm{sol}^{i+1} \neq \mathrm{sol}^i$, i.e. $\mathrm{sol}^{i+1}(X) \not\Longrightarrow \mathrm{sol}^i(X)$ for at least one nonterminal X, increment i and repeat this step.
   - Let i_0 be the first index so that $\mathrm{sol}^{i_0} = \mathrm{sol}^{i_0+1}$, i.e. $\mathrm{sol}^{i_0}(X) \iff \mathrm{sol}^{i_0+1}(X)$ for all nonterminals.

3. The existential player has a winning strategy from S if and only if $\mathrm{sol}^{i_0}(S)$ is rejecting, i.e. if $(\mathrm{sol}^{i_0}(S))(M_{\mathrm{rej}}) = \mathrm{true}$.

Firstly, note that we use Kleene iteration here instead of the more efficient chaotic iteration using a worklist as described in Section 16.1. This will simplify the analysis of the algorithm. Secondly, our domain technically is the set of equivalence classes of formulas. However, we will assume that we represent each equivalence class by some formula in that class. This means that checking equality of equivalence classes amounts to checking logical equivalence of representatives. More precisely, the approximants necessarily form an ascending chain, so $\mathrm{sol}^i(X) \implies \mathrm{sol}^{i+1}(X)$ always holds. The iteration has arrived at the fixed point corresponding to the least solution as soon as $\mathrm{sol}^{i+1}(X) \implies \mathrm{sol}^i(X)$ holds for all nonterminals.

Finally, we have to deal with the problem that the formulas that occur in the approximants may grow in each step. To solve this problem, we assume that formulas are given in conjunctive normal form (CNF) without redundant clauses. This means that each formula is of shape $F = \bigwedge_K \bigvee_{\rho\in K} \rho$ so that no two clauses K, K' are equal. We may also see F as a set of clauses, each of which is a set of boxes.

In order to be able to execute the algorithm on formulas in CNF, we need to discuss how the operations can be implemented. In addition to the three operations conjunction, disjunction, and formula composition, we will also need a criterion for checking implication.

17.5.2 Lemma
Let $F = \bigwedge_K \bigvee_{\rho\in K} \rho$ and $H = \bigwedge_{K'} \bigvee_{\rho'\in K'} \rho'$ be positive Boolean formulas over $\mathbb{M}_A$ in CNF.

a) $F \wedge H = F \cup H$, i.e. $F \wedge H = (\bigwedge_K \bigvee_{\rho\in K} \rho) \wedge (\bigwedge_{K'} \bigvee_{\rho'\in K'} \rho')$.

b) $F \vee H = \{K \cup K' \mid K \in F, K' \in H\}$, i.e. $F \vee H = \bigwedge_{K,K'}((\bigvee_{\rho\in K} \rho) \vee (\bigvee_{\rho'\in K'} \rho'))$.

c) $F \,.\, G = \bigcup_{K\in F} \bigcup_{\zeta:K\to H} \{\bigcup_{\rho\in K} \rho \,.\, \zeta(\rho)\}$, i.e. $F \,.\, G = \bigwedge_{K\in F} \bigwedge_{\zeta:K\to H} \bigvee_{\rho\in K} \rho \,.\, \zeta(\rho)$.

d) $F \implies H$ holds iff every clause of H contains a clause of F, i.e. iff there is a function $\iota\colon H \to F$ so that $\iota(K') \subseteq K'$ for all $K' \in H$.

Proof:
Part a) is obvious. Part b) is obtained by applying distributivity to bring the disjunction of two CNFs back into CNF.

For Part c), observe that the composition of F and H is

$$\bigwedge_K \bigvee_{\rho\in K} \bigwedge_{K'} \bigvee_{\rho'\in K'} \rho \cdot \rho' \,.$$

To obtain a CNF, we need to swap the order of the disjunction $\bigvee_{\rho\in K}$ and the conjunction $\bigwedge_{K'}$ using distributivity. The resulting formula will have one conjunct for each combination of a box

$\rho \in K$ and a clause K' of H. The conjunction $\bigwedge_{\zeta:K\to G}$ over all functions that assign a clause $\zeta(\rho)$ of H to every box ρ in K produces exactly these conjuncts. Additionally, the composition $\rho \,.\, K'$ of a box ρ and a clause K' is the disjunction $\bigwedge \rho' \in K' \rho \cdot \rho'$.

For Part d), assume a function $\iota\colon H \to F$ as specified exists and assume that $F(M) = \text{true}$ for some variable assignment M. We need to argue that every clause K' of H is satisfied under M. Each K' contains a clause $\iota(K')$ of F. Since F is satisfied under M, so are all its clauses. A clause is a disjunction, so if $\iota(K')$ is satisfied, then so is K'.

For the other direction, assume that there is a clause K' of H that does not contain any clause of F. Consider the variable assignment that sets to true all atoms not contained in K'. This variable assignment does not satisfy H since it does not satisfy K'. Since K' does not contain any clause of F, every clause of F contains at least one atom that is set to true. Hence, F is satisfied under this assignment and we obtain $F \not\Longrightarrow H$. ■

With these preliminaries at hand, we can now analyze the running time of the algorithm. We obtain that the time is polynomial in the size of the grammar, but doubly exponential in the size of the automaton.

17.5.3 Proposition
The algorithm solves a given context-free regular inclusion game in time

$$\mathcal{O}\Big(|G|^{c_1}\Big) \cdot 2^{2^{\mathcal{O}(|Q|^{c_2})}}$$

for suitable constants $c_1, c_2 \in \mathbb{N}$.

Proof:
The two crucial factors for the running time of the algorithm are the size of the formulas that need to be manipulated, and the number of steps after which the fixed point iteration terminates. We first observe that constructing the system of equations is polynomial in the size of the grammar.

To analyze the complexity of the fixed-point iteration, we start by analyzing the height of the domain. We argue that its height is at most the height of $(\mathcal{P}(\mathcal{P}(\mathbb{M}_A)), \subseteq)$, a two-fold powerset over $\mathbb{M}_A$. To see this, we identify an equivalence class of formulas $[F]$ with the set of variable assignments $\{M \subseteq \mathbb{M}_A \mid F(M) = \text{true}\} \in \mathcal{P}(\mathcal{P}(\mathbb{M}_A))$ under which F evaluates to true. We observe that if $F \Longrightarrow H$, then any variable assignments satisfying F will also satisfy H and we get that $\{M \subseteq \mathbb{M}_A \mid F(M) = \text{true}\}$ is a subset of $\{M \subseteq \mathbb{M}_A \mid H(M) = \text{true}\}$. If the implication is strict, meaning that the two formulas are not equivalent, then also the subset relation among the associated sets of variable assignments will be strict. Altogether, we obtain that a strict ascending chain of equivalence classes of formulas with respect to implication induces a strict ascending chain in $(\mathcal{P}(\mathcal{P}(\mathbb{M}_A)), \subseteq)$. Hence, the height of the latter partial order bounds the height of our domain.

For any set D, the height of $\mathcal{P}(D)$ is $|D|$, as we have discussed in Section 16.1. Hence, the height of $\mathrm{pBF}(\mathbb{M}_A)$ is bounded by $|\mathcal{P}(\mathbb{M}_A)| = 2^{2^{|Q|^2}}$. The fixed-point iteration is actually conducted over the product domain $N \to \mathrm{pBF}(\mathbb{M}_A)$ that associates one formula to each nonterminal. It is not difficult to check that the height of that domain is

$$|N| \cdot 2^{2^{|Q|^2}} \in \mathcal{O}(|G|) \cdot 2^{2^{|Q|^2}},$$

since any strictly ascending chain in the product domain can be decomposed into $|N|$ ascending chains, one for each of the nonterminals. In each of these chains, there are at most $2^{2^{|Q|^2}}$ steps in which the elements can strictly increase.

The height of the domain gives us the required bound on the steps of the fixed-point iterations. We analyze the cost of one step of the iteration. If we assume that a formula in CNF contains no repeated clauses, we obtain that the size of a single formula is at most doubly exponential. To be precise, it contains at most each of the $2^{2^{|Q|^2}}$ different clauses of size at most $2^{|Q|^2}$.

It remains to discuss the cost of the operations that we perform on these formulas. Using Lemma 17.5.2, the conjunction and the disjunction of two formulas can be computed in time polynomial in the size of the formulas.

The size of the composition $F \,.\, H = \bigwedge_{K \in F} \bigwedge_{\zeta : K \to H} \bigvee_{\rho \in K} \rho \,.\, \zeta(\rho)$ of two formulas is mainly determined by the number of functions $\zeta \colon K \to H$. This number is at most

$$|H|^{|K|} \leqslant \left(2^{2^{|Q|^2}}\right)^{\left(2^{|Q|^2}\right)} = 2^{\left(2^{|Q|^2}\right)^2} = 2^{2^{2 \cdot |Q|^2}},$$

which is doubly exponential in Q. Altogether, the cost of computing the composition is at most doubly exponential.

After each operation, we get rid of redundant clauses, which takes quadratic time (in the size of the formulas). The total number of operations that we need to apply in one step of the fixed-point operation is bounded by the size of the system of equations, which is polynomial in $|G|$.

After each step of the iteration, we need to check whether we have reached the fixed point. To this end, we use Part d) of Lemma 17.5.2. This check can be conducted in quadratic time (in the size of the formulas).

Finally, we observe that reading off the winner of the game requires evaluating a formula, which is polynomial in the size of the formula. Altogether, we obtain the desired result. ■

As a consequence of this result, solving context-free games is easy as long as the automaton representing the target language is small. This is in particular the case when the condition for membership in the target language is a simple one like the absence of a certain symbol in the

produced word. In this case, the size of the automaton is a constant, and the resulting algorithm runs in polynomial time.

Hardness

To complete the proof of Theorem 17.5.1, we show that solving context-free inclusion game is 2EXP-hard.

17.5.4 Theorem
Deciding context-free inclusion games is 2EXP-hard.

The proof uses a well-known technique that was introduced to the best of the author's knowledge in Stockmeyer's and Meyer's proof that deciding language equivalence for regular expressions is PSPACE-hard [SM73]. The proof uses that is sufficient to show that the universality problem, i.e. the task of deciding whether a regular language is equal to Σ^*, is PSPACE-hard. The original paper uses regular expressions to represent regular languages, but we will work with NFAs in the following.

The idea behind that proof is as follows: Given a Turing machine M with polynomial space consumption and an input x for that machine, we construct an NFA that accepts all invalid or non-accepting computations of M on x. To this end, it expects a string of configurations of the machine, and it accepts if one of the encodings of the configurations is invalid or the transition relation of the Turing machine has not been respected. If M does not accept x, M has no valid accepting computation for that input and the language of the automaton is universal. The well-known proof for the universality problem for Turing machines being neither semi-decidable nor co-semi-decidable uses a similar construction.

Walukiewicz [Wal01] has extended this idea to show that a special type of context-free games, namely parity games on the configuration graphs of pushdown automata are EXP-hard. He makes use of the game-aspect in the pushdown to simulate an alternating Turing machine with polynomial space consumption, relying on EXP = APSPACE. Later, Muscholl, Schwentick, and Segoufin [MSS06] have in turn extended Walukiewicz'sproof to show that a type of context-free games that is similar to the one that we consider is 2EXP-hard. We will explain this type of game in detail in Section 17.6. The additional layer provided by having a winning condition that works on the level of languages rather than on the internal state of the system can be used to simulate an alternating Turing machine that has an exponential space bound instead of a polynomial one.

The full proof that we provide in the rest of this section is strongly inspired by the proof of Theorem 4.3 in [MSS06]. It uses the same idea: Given an ATM with exponential space construction and an input for that machine, we construct a game that proceeds in two phases. The first phase

produces a computation of the machine for the given input, represented as a sequence of configurations. We use the players to mimic the alternation between universal and existential control states of the ATM. In the second phase, the players place markers in this computation that help the automaton to detect whether the computation is valid. The automaton is designed so that it accepts all computations that are invalid or non-accepting. If the existential player manages to enforce the derivation of a valid accepting computation, winning the game, the given input is in the language of the machine and vice versa.

Into the proof

The rest of the section is dedicated to making this proof approach formal, including all gory technical details. We provide a polynomial time reduction from the AEXPSPACE acceptance problem. Recall that this problem is, given an alternating Turing machine M and an input x for that machine, to decide whether M accepts x with a computation tree in which each configuration has space consumption at most $2^{|x|}$. We have argued in Lemma 3.3.2 that this problem is AEXPSPACE-complete. Furthermore, it is well known that AEXPSPACE = 2EXP [CKS81].

We also assume that the input alphabet of the Turing machine is $\{a, b\}$, meaning that the tape alphabet $\{a, b, \sqcup\}$. It is well-known that an arbitrary input alphabet can be encoded into a binary one by only polynomially increasing the space consumption.

Let (M, x) be the input for the AEXPSPACE acceptance problem. We define $n = |x|$, and we see the transition relation δ of the Turing machine as a set of transitions of the shape $t = (q, y) \mapsto (q', y', d)$ with q, q' control states, $y, y' \in \Gamma$ tape symbols and $d \in \{L, R\}$ a direction for the head movement.

Our reduction constructs a context-free inclusion game such that its tree of plays from a fixed initial position corresponds to the computation tree of M on input x. The game will be designed so that the existential player has a winning strategy if and only if x is accepted by M with space consumption at most 2^n. More precisely, a play of the game will derive a terminal word that is a branch of the computation tree, i.e. a sequence of successive configurations of M. To this end, the players will write down configurations of the Turing machines, starting with the initial one. The ownership assignment is chosen such that the universal player chooses the transitions that originate in configurations with a universal control state, similar for the existential player.

We combine this approach with two tricks. The first trick is to use the automaton that represents the winning condition of the game to detect invalid computations. For example, it will detect if the head of the ATM is moved in an invalid way or the tape content is not copied properly from one transition to next.

It remains to deal with the problem that the configurations of exponential length are too long for a polynomially sized automaton to properly keep track of the tape content and head movement. The second trick solves this problem by designing the grammar so that the game proceeds in two phases. The first phase is a left-to-right pass in which the computation of the

Turing machine is constructed as discussed above. The second phase is a right-to-left pass in which the players can place markers within the computation. With the help of these markers, the automaton can check whether the computation is valid or not.

The first phase

We start by discussing the technical details of the first phase. The goal of the first phase is to obtain a sentential form of the shape

$$c^{(k)}\, t_k\, c^{(k-1)}\, t_{k-1}\, \dots\, t_1\, c^{(1)}\, t_0\, c^{(0)}$$

so that $c^{(0)} \xrightarrow{t_0} c^{(1)} \xrightarrow{t_1} \dots \xrightarrow{t_{k-1}} c^{(k-1)} \xrightarrow{t_k} c^{(k)}$ is a candidate for an accepting computation of the ATM. By $c^{(i)} \xrightarrow{t_i} c^{(i+1)}$, we mean that transition $t_i \in \delta$ of the ATM was used to get from $c^{(i)}$ o $c^{(i+1)}$. Each configuration $c^{(i)}$ is encoded as its tape content

$$c^{(i)} = c_1^{(i)} \dots c_m^{(i)}$$

of length exactly $m = 2^n$. (If a configuration uses less than m space, we fill up the remaining space with blank symbols). Each $c_j^{(i)}$ is either just a tape symbol, or a tape symbol together with a control state. The latter case occurs exactly once per configuration.

For the verification of the validity of the computation, it will be important that each tape cell is preceded by an index, i.e. its number on the tape. Formally, we assume that $\{0, 1\}$ are two terminal symbols used for the indexing (while $\{a, b, \sqcup\}$ is the tape alphabet of the ATM). Furthermore, each index is preceded by two marker symbols $M_\square M_\bigcirc$ that will be needed during the second phase.

Hence, each $c_j^{(i)}$ is either of the shape $M_\square M_\bigcirc \{0,1\}^n \{a, b\}$, or $M_\square M_\bigcirc \{0,1\}^n Q\{a, b\}$. Note that since each configuration has length $m = 2^n$, a binary number of length n is necessary and sufficient to properly number the cells on the tape. As we can see, the control states are used as terminal symbols as well.

The initial configuration. The derivation starts with rules that implement a process that writes down the initial configuration. The result of this process is the sentential form

$$E_{q_0,x_1} M_\square M_\bigcirc \underbrace{0\dots0}_{\text{encoding of } 0}\, q_0 x_1 \;\dots\; M_\square M_\bigcirc \underbrace{0\dots}_{\text{encoding of } n-1} x_n \; M_\square M_\bigcirc \underbrace{0\dots}_{\text{encoding of } n} \sqcup \;\dots\; M_\square M_\bigcirc \underbrace{1\dots1}_{\text{encoding of } 2^n-1} \sqcup .$$

Here, we use that $x = x_1 \dots x_n$ has length n, meaning we can explicitly encode it via grammar rules. Furthermore, we use the well-known result that a context-free grammar can produce an exponential number of symbols with a polynomial number of rules to derive the $2^n - n$ blank symbols at the end of the tape content. (See the end of Section 9.2 for a similar construction.) For the indexing of each symbol (including the preceding markers), we essentially let the existential player guess the correct indexing. We discuss this mechanism in detail later.

Picking a transition. The symbol E_{q_0,x_1} is a nonterminal whose subscripts indicate that we have seen control state q_0 and tape symbol x_1 at the head position in the last configuration. After every configuration, we will have some nonterminal $E_{q,y}$ as the leftmost symbol of the sentential form where q is the control state and $y \in \{a, b, \sqcup\}$ is the head symbol. This nonterminal is owned by the player that owns state q. Her task is now to pick a transition of the Turing machine that is valid with respect to state q and symbol q. Formally, she has a rule

$$E_{q,y} \to A_{q'}\, t$$

for any transition $t \in \delta$ with $t = (q, y) \mapsto (q', y', d)$. The transition that was used is written down as a terminal symbol.

A special case is that the control state $q = q_{\text{final}}$ is the unique accepting control state of the ATM. In this case, we have the rule

$$E_{q_{\text{final}},y} \to \varepsilon$$

that enforce the termination of the first phase.

Writing the tape content. Whenever A_q occurs (for some state q), the existential player that owns the symbol can iteratively write cells of the tape of the next configuration. For each cell, she can first decide whether this should be the new position of the head by choosing between the rules

$$A_q \to A_q^{\text{nohead}} \mid A_q^{\text{head}}\,.$$

If she picks A_q^{nohead}, the next cell is not the position of the head, and she can first pick a tape content in $\{a, b, \sqcup\}$ and then an index, a string over $\{0, 1\}$ of arbitrary length. Formally, we have the rules

$$A_q^{\text{nohead}} \to A_q^{\text{index}} a \mid A_q^{\text{index}} b \mid A_q^{\text{index}} \sqcup$$

and

$$A_q^{\text{index}} \to A_q M_\square M_\bigcirc \mid A_q^{\text{index}} 0 \mid A_q^{\text{index}} 1\,.$$

The nonterminals A_q^{nohead} and A_q^{index} are owned by the existential player. The markers $M_\square$ and $M_\bigcirc$ that are inserted in front of the index are used for the second phase; we will describe their function later in detail.

Writing the head position. If she picks A_q^{head}, the next tape cell should be the position of the head. This means she is forced to write down control state q (that should be the control state in the current position), but she may select an arbitrary tape content. The control state and the tape content at the head position are tracked as a subscript. The rules are

$$A_q^{\text{head}} \to B_{q,a}^{\text{index}} qa \mid B_{q,b}^{\text{index}} qb \mid B_{q,\sqcup}^{\text{index}} q\sqcup .$$

Writing the rest of the tape content. The rest of the tape content is written while $B_{q,y}$ is the leftmost nonterminal, indicating that the head position has already occurred in the current configuration. In $B_{q,y}$, the existential player can either continue to write cells (namely, the cells that are to the left of the head position). Or, she can derive $E_{q,y}$ as the leftmost nonterminals, which means that the next transition should be picked.

Formally, the rules are

$$B_{q,y} \to E_{q,y} \mid B_{q,y}^{\text{index}} a \mid B_{q,y}^{\text{index}} b \mid B_{q,y}^{\text{index}} \sqcup .$$

The indexing for the symbols in this part of the tape content works just as before with the rules

$$B_{q,y}^{\text{index}} \to B_{q,y} M_{\square} M_{\bigcirc} \mid B_{q,y}^{\text{index}} 0 \mid B_{q,y}^{\text{index}} 1 .$$

This process proceeds either ad infinitum (in this case, the existential player looses by definition), or until the accepting state is reached. In the latter case, we end up with a sentential form in which the only occurrences of nonterminals are the markers $M_{\square}$ and $M_{\bigcirc}$. Before discussing their usage, we explain the parts of the automaton that solely reference the terminals that have been written in first phase.

Summarizing the first phase. If the first phase terminates, it derives a sequence $c^{(k)}\ t_k\ \ldots\ t_0\ c^{(0)}$, where $c^{(0)}$ is the initial configuration. Each $c^{(i)}$ contains exactly one control state, and the last configuration $c^{(k)}$ contains the accepting control state. The transitions of the control states respects the transitions t_i that occur between the configuration, meaning that the new control state indeed results from the old control state and the old symbol at the head position. Each tape symbol is preceded by an indexing from $\{0,1\}^*$ and the two markers. Also note that the transitions originating in universal resp. existential control states have been picked by the universal resp. existential player.

There are two properties that are still required so that $c^{(k)}\ t_k\ \ldots\ t_0\ c^{(0)}$ is indeed a valid computation:

1. The indexing has to be correct: All indices have length n, the first tape symbol of each configuration is indexed by $0 = \underbrace{0 \dots 0}_{n \text{ times}}$, the last tape symbol is indexed by $2^n - 1 = \underbrace{1 \dots 1}_{n \text{ times}}$, and successive tape symbols are indexed with successive binary numbers (from left to right).

2. The tape content has to respect the transition relation of the Turing machine: From one configuration to the next, the tape content does not change at any position but at the former position of the head. Additionally, the head position and the tape content at the former head position have to be modified according to the transition that was picked.

To check the indexing, the information that is present in the sentential form resulting from the first phase is sufficient. To enable the automaton that will represent the target language of the game to check the second property, we need the second phase of the game.

The second phase

The goal of the second phase is to help the automaton checking that the tape content has been manipulated in the correct way. Intuitively, the automaton should compare for each index $j \in [0, 2^n - 1]$ and for each two successive configurations $c^{(i+1)}$, $c^{(i)}$ that the tape content of $c^{(i+1)}$ in cell j has indeed been obtained from $c^{(i)}$ at all cell j in a valid way. To be able to do this, the automaton would have to store j to match the two cells in the configurations. Since there are 2^n possibilities for j, this is not possible using a polynomially sized automaton.

To solve this problem, we use the help of the players. Recall that at the end of the first phase, each cell is of the shape

$$M_\square M_\bigcirc \{0,1\}^* Q^{\leqslant 1} \{a, b, \sqcup\} ,$$

where all symbols but $M_\square$ and $M_\bigcirc$ are terminals. (The expression $Q^{\leqslant 1}$ stands for the optional occurrence of a state, i.e. it is either some $q \in Q$ or ε.)

As we will explain later, we can detect configurations in which the indexing is incorrect without the help of the markers. Hence, we will assume that Property 1 is satisfied in the following. We use the nonterminals $M_\star$ to implement a marking mechanism. After the first phase has finished, the leftmost pair of markers becomes the leftmost nonterminal. After replacing this occurrence of $M_\square M_\bigcirc$, the second occurrence of $M_\square M_\bigcirc$ becomes active and so on. In total, the $M_\star$ markers implement a left-to-right pass (in contrast to the right-to-left writing process of the first phase) over the word.

Intuitively, the universal player can use $M_\square$ to mark cell j of configuration $c^{(i+1)}$ for which she thinks that the existential player has made a mistake when writing down the cell content in the first phase. This can be seen as a challenge for the existential player. The existential player

should defend herself by using $M_{\bigcirc}$ at cell j of configuration $c^{(i)}$. Formally, for each occurrence of $M_{\bigstar}$, its owner $\bigstar$ can select to use the marker $m_{\bigstar}$ or to omit it. We have rules

$$M_{\square} \to m_{\square} \mid \varepsilon \qquad \text{and} \qquad M_{\bigcirc} \to m_{\bigcirc} \mid \varepsilon \,.$$

After all $M_{\bigstar}$ symbols have been replaced, we obtain a terminal word.

The automaton

In the following, we will define the automaton A representing the target language. It will accept a terminal word if and only if it encodes a candidate computation in which one the two properties is violated. Because the goal of the existential player is non-inclusion, this will force her to ensure that the computation is valid.

Checking the indexing. We describe the behavior of the automaton that is related to Property 1. Firstly, we construct automata that accept all words that contain

$$\{a, b, \llcorner\!\lrcorner\} m_{\square}{}^{\leqslant 1} m_{\bigcirc}{}^{\leqslant 1} \{0, 1\}^{\ell} Q^{\leqslant 1} \{a, b, \llcorner\!\lrcorner\}$$

as an infix where $\ell \neq n$. Here, the two occurrences of $\{a, b, \llcorner\!\lrcorner\}$ are two successive tape symbols and $\{0, 1\}^{\ell}$ is an indexing of the wrong length. The other symbols are the optional occurrences of control states and markers. We essentially have to encode this expression for $\ell \in \{[0, n-1]\}$ and $\ell \geqslant n+1$. This can be done using $(n+1)$ automata that are of size polynomial in n.

Secondly, we construct an automaton accepting any word that contains an infix $t\, m_{\square}{}^{\leqslant 1} m_{\bigcirc}{}^{\leqslant 1} w$ where $w \neq 0 \ldots 0$. This enforces that the first cell in each configuration has the binary encoding of 0 (of length n) as indexing. This can be implemented by an automaton that counts the number of zeros. A similar construction ensures that all configurations end with $1 \ldots 1 Q^{\leqslant 1} \{a, b, \llcorner\!\lrcorner\}\, t$, where $1 \ldots 1$ is the binary encoding of $2^n - 1$ and $\{a, b, \llcorner\!\lrcorner\}$ is the tape symbol.

Finally, we have an automaton accepting words that contain as infix

$$m_{\square}{}^{\leqslant 1} m_{\bigcirc}{}^{\leqslant 1}\ w\ Q^{\leqslant 1} \{a, b, \llcorner\!\lrcorner\} \quad m_{\square}{}^{\leqslant 1} m_{\bigcirc}{}^{\leqslant 1}\ v\ Q^{\leqslant 1} \{a, b, \llcorner\!\lrcorner\}$$

so that w is an indexing of number ℓ and v is not the indexing of number $\ell + 1$. We argue that this can be implemented by an automaton of polynomial size. The key observation is that the encoding of $\ell + 1$ can be obtained from the encoding of ℓ as follows: The rightmost 0 is replaced by a 1, and all ones that are to the right of that 0 are replaced by zeroes. To implement this, we use nondeterminism. The automaton assumes that v is not the encoding of $\ell + 1$ and guesses in which bit v differs from the proper encoding, say bit s. It walks through w and stores bit s of w, and whether there is a later bit in w that is zero. It then goes to bit s of v and checks the following. If w contains a later zero, then the s^{th} bit of w and the s^{th} bit of v should coincide (because the increment only affects a suffix of the word that does not contain the s^{th} bit). If w

contains no later zero, then the s^{th} bit of w and the s^{th} should be different (either w at s was the last zero which should be flipped to one in v, or w at s was a one which should be flipped to zero). Whenever this condition is violated, the automaton accepts.

Checking the tape content. It remains to check the tape content. We implicitly assume that the automata we describe in the following only accept words that contain precisely one occurrence of $m_{\square}$. If a word contains no occurrence of $m_{\square}$, this means that the universal player admits that it indeed encodes an accepting computation of the ATM. For the sake of simplicity, we force the universal player to mark exactly one mistake, even if the computation contains several mistakes. She then should choose the earliest mistake (i.e. the rightmost). Note that the grammar does not allow this occurrence to be in the initial configuration, since the tape content of the initial configuration is guaranteed to be valid by the grammar.

Assume that a word contains exactly one occurrence of $m_{\square}$, say in configuration $c^{(i+1)}$. If the word does not contain exactly one occurrence of $m_{\bigcirc}$, we make the automaton accept the word (which corresponds to the existential player admitting defeat). This unique occurrence of $m_{\bigcirc}$ should be in configuration $c^{(i)}$, i.e. in the configuration to the right of the one that contains $m_{\square}$. Intuitively, if the universal player has marked cell j in $c^{(i+1)}$, then the existential player should mark cell j in $c^{(i)}$. This uniquely determines the correct location for her marker $m_{\bigcirc}$.

Assume that the word has an infix of the shape

$$\underbrace{m_{\square}\ w\ qy}_{\text{cell } j \text{ in } c^{(i+1)}}\ \ldots\ \ t\ \ \ldots\ \underbrace{m_{\bigcirc}\ w'\ q'y'}_{\text{cell } j \text{ in } c^{(i)}?},$$

where $w, w' \in \{0,1\}^n$ are indexings, $q, q' \in Q \cup \{\varepsilon\}$ are optional control states and $y, y' \in \{a, b, \sqcup\}$ are tape symbols.

The automaton can easily verify that the markers occur in successive configurations by requiring the occurrence of exactly one terminal symbol $t \in \delta$ corresponding to a transition between the markers. The automaton then checks that $m_{\bigcirc}$ is at the correct position, i.e. at cell j. If this is not the case, w and w' differ. To do so, the automaton guesses a position $s \in [1, n]$, stores the s^{th} bit of w and compares it to the s^{th} bit of w'. If the bits differ, it accepts the input (meaning that the existential player loses).

Assume that the marker $m_{\bigcirc}$ is at the correct position. It remains to verify that the cell content is valid. We distinguish several cases. If control state q' is not present, then cell j was not the head position in configuration $c^{(i)}$. This means $y = y'$ should hold. The automaton checks this and accepts if the condition is violated.

In the second case, we assume that q' is present. This means y should be chosen according to the transition that has been applied. The automaton stores y, reads the transition

$t = (q', y') \mapsto (q'', y'', d)$ and accepts if the symbol y'' that should be written by the transition is not equal to y.

Finally, the position of the head in $c^{(i+1)}$ should be valid with respect to the transition. More formally, the automaton checks that if control state q is present, and the two configurations are separated by a transition $t = (q''', y') \mapsto (q, y'', L)$ that moves the head to the left, then the control state in configuration $c^{(i)}$ is present in cell $j + 1$. Similarly, if the configuration moves the head to the right, then the control state in $c^{(i)}$ should be present in cell $j - 1$. For each of these cases, one can create a polynomially sized automaton.

In total, we obtain a collection of NFAs of polynomial size. If the word violates any of the properties that are required to ensure that it represents an accepting computation of the ATM, at least one of these automata accepts the word. If the word represents an accepting computation, none of the automata accept. The final automaton defining our target language is the union of all aforementioned automata. Since all automata were of polynomial size, so is their union.

The exponential space bound

Recall that an instance (M, x) of the AEXPSPACE acceptance problem is only a yes-instance if the computation tree of M on input x is accepting and its configurations use less than $2^{|x|}$ space. Without loss of generality, we can assume that the tape of the Turing machine is bounded on the left side, and no computation ever goes to the left of the first cell of the input. It remains to check that no cell ever exceeds cell $2^{|x|}$ on the right-hand side. Luckily, this property is already implicitly verified by our construction. We have enforced that each configuration is represented by a tape content of length exactly $2^{|x|}$. If the computation exceeds space $2^{|x|}$, there is a step in which the head position moves from cell $2^{|x|}$ to cell $2^{|x|} + 1$. Since we enforce the encoding of the configurations to be of size $2^{|x|}$, this means that the encoding of the latter configuration is invalid: It either contains an invalid indexing, or the head of the Turing machine has been moved incorrectly. In both cases, the existential player loses the play, as expected.

Soundness

It remains to argue that the construction is correct: The existential player wins the context-free inclusion game if and only if input x is accepted by a computation of the ATM that does not use more than exponential space.

Game to machine. Assume that the existential player has a winning strategy for the game. We consider the tree of all plays conforming to this strategy. In this tree, we cut off each branch after the first phase has ended. Each branch contains such a point, since infinite plays cannot be won by the existential player. Note that the result of the first phase, and hence each branch of the tree, is essentially a computation of the ATM. To be precise, the fact that we start with the winning strategy for the existential player ensures that it is a valid accepting computation. We

obtain a subtree of the computation tree in which in for each configuration owned by the existential player, only one transition has been considered, namely the one chosen by the strategy. In configurations owned by the universal player, all transitions have been taken into account. This subtree is a witness for the full computation tree of the ATM for input x being accepting.

Machine to game. Assume that the configuration tree of M for input x is accepting. In Chapter 15, we have mentioned that alternating Turing machines can be seen as games played on the configuration graphs of Turing machines. To be precise, the computation tree of M on x being accepting means that the existential player wins the reachability game on that computation tree. The ownership is given by the partition of the control states, the winning condition is reaching an accepting configuration. Let us consider a winning strategy for the existential player in that reachability game.

We now construct a winning strategy for the existential player in the context-free inclusion game. Whenever it is the choice of the existential player to write the index of a tape cell, she should pick the correct indexing, i.e. cell number j from the left should be indexed with the n-bit binary encoding of j. Assume that in the game, the encoding of some configuration c has just been written down. If it is the existential players choice to pick the next transition, she does so by picking the transition that leads to successor of c in the computation tree of M as selected by her winning strategy for the reachability game. Otherwise, the universal player picks the transition. In any case, the existential player than proceeds to write down the configuration that results from that transition, with the correct indexing, tape content, and head position. When the first phase of the game has ended, the existential player defends from a challenge of the universal player – say she puts her marker $m_\square$ in cell j of configuration $c^{(i+1)}$ – by putting marker $m_\bigcirc$ in cell j of configuration $c^{(i)}$.

We argue that the resulting play is won by the existential player. Because she picks her transitions conforming to a winning strategy for the reachability game on the computation tree, it is guaranteed that the first phase ends after finitely many steps by reaching an accepting configuration. The assumption that we start with a yes-instance of the AEXPSPACE acceptance problem also means that no configuration exceeds $2^{|x|}$ space. Hence, the existential player can ensure that the indexing, the tape content, and the head movements are correct. If the universal player places a marker, she can successfully defend the challenge. The automaton representing the target language will not accept the outcome of the play and the existential player wins.

This finishes the proof of Theorem 17.5.4 and establishes 2EXP-completeness.

17.6 Related work

We discuss the relation of our approach to context-free games to other works from the literature.

Walukiewicz's reduction

To the best of the author's knowledge, the first to not only consider context-free games but also to solve them was Walukiewicz in [Wal01]. In an earlier work [Wal02], he had discovered the relationship between the model-checking problem for the modal μ-calculus and parity games. The modal μ-calculus is a logic that features operators for least and greatest fixed points. A μ-calculus formula can describe the winning region of a parity game and vice versa. With this motivation in mind, he translated the model checking problem for modal μ-calculus formulas on the computation tree of a pushdown automaton into solving a parity game on the computation tree of a modified pushdown automaton. The tree is turned into a game arena by a partition of the control states into the ones owned by each of the players. Similarly, the configurations inherit their priorities from the control states, for which one obtains a priority assignment from the nesting structure of the μ-calculus formula.

To solve this type of game, Walukiewicz has designed a reduction that outputs an equivalent parity game on a finite arena, i.e. a parity game with the same winner. The idea behind the reduction is to not store the full stack content of the pushdown, but only the top-of-stack. Whenever a push operation, say push a should be executed, the finite-state game uses a guess-and-verify mechanism, aided by the fact that we have two adversarial players. Firstly, one of the players is allowed to choose a prediction, a set of tuples of control state and priority. Intuitively, an entry (q, i) in the prediction set represents a play of the game in which a gets pushed, remains on the stack for some time, and then gets popped again while reaching state q. The priority i is the largest priority that occurred while symbol a was on the stack. Secondly, the other player decides whether to trust or to verify the prediction. If she trusts the prediction, she picks one entry (q, i) from the prediction set, and the play continues with control state q and the current top-of-stack remains as before – we assume that the infix of the play in which the letter a was on the stack has been skipped. If she wants to verify the prediction, the push operation is actually executed and the top-of-stack is replaced by a. When a pop operation, pop a, occurs, the play ends and the player who picked the prediction wins depending on whether the control state that has been reached and the maximum priority that has been visited are contained in the prediction.

Assume that we have a winning strategy for the context-free game. By picking the correct predictions – namely the ones actually describing the plays that can occur according to our winning strategy – we obtain a winning strategy for the game on the finite arena. Similarly, a winning strategy for the finite game yields a strategy for the context-free one. To be precise, it yields a strategy that can be implemented by a synchronized pushdown automaton as in Section 17.4.

To be able to apply the strategy for the finite game, the synchronized pushdowns tracks for each stack level also the prediction set that has been chosen for the corresponding push operation.

Walukiewicz has considered parity games, but the construction can be easily adapted to transform a context-free game with a reachability winning condition defined on the control states of the pushdown automaton into a finite reachability game. In both cases, the construction introduces an exponential blowup, since we need to add the prediction sets to the state space, which is essentially a powerset construction. To be able to apply Walukiewicz's construction to context-free inclusion games, one has to determinize the automaton for the target language as described in Chapter 15. This introduces another exponential blowup, which is to be expected: Walukiewicz has shown EXP-completeness for his type of games, while we have shown 2EXP-completeness.

In her Master's thesis, Elisabeth Neumann [Neu17] has conducted an in-depth comparison of Walukiewicz's approach and ours. Assume that we do not use the standard powerset construction to determinize the automaton, but rather we use a construction based on the transition monoid, as explained in Section 4.5. If one transform a given game grammar to a game pushdown, takes the product with this determinized automaton, applies Walukiewicz's reduction to obtain a finite-state reachability game, one can then solve this game with the well-known attractor construction. From the attractor (resp. its approximants that occur during its computations), one can read off the formulas that are the least solution to the interpreted system of equations in our approach (resp. their approximants). Note that, however, the fact that Walukiewicz's approach can be conducted using the determinization based on the transition monoid does not mean that it actually uses the advantages that the transition monoid provides, e.g. the fact that the behavior of a word in arbitrary contexts is represented by its box.

If one compares Walukiewicz's approach applied to ω-context-free games to our approach that we will present in Section 17.9, one sees that both approaches ultimately require solving a finite-state parity game. The key difference is that in Walukiewicz game, the players can pick an arbitrary prediction set. Finding out which prediction sets are not valid candidates has to be done when the parity game is solved and the winning strategy is computed. In our approach, we use the fixed-point iteration to precompute information about finite subplays. This yields formulas – which we can see as a collection of prediction sets –, and the players are only allowed to choose among these. This should be a more efficient implementation.

Walukiewicz's reduction is inspired by the game semantics for the μ-calculus, see e.g. [BW18]: Here, instead of computing a least or greatest fixed point, one of the player guesses it. This guess is enforced to be correct by giving the other player the chance to verify the guess. This trick has proven to be very versatile. Walukiewicz's reduction has been extended to higher-order pushdown automata by Cachat and Walukiewicz [CW07] and to multi-pushdown automata with certain restrictions by Seth [Set09].

Cachat's saturation algorithm

In [Cac02], Cachat has considered a different type of context-free games. Like in Walukiewicz's case, the game arena is the computation tree of a pushdown automaton with a partition of the control states. However, the winning condition is reaching a configuration that is in a regular target set. To specify such a regular set of configurations of some pushdown P, one uses so-called P-AFAs. A P-AFA is an alternating automaton whose input alphabet is the set of stack symbols and whose set of states contains the set of states of P. A configuration of the pushdown with state q and stack content w is accepted by a P-AFA if the automaton accepts the word w from the control state q. Cachat presents a saturation technique that turns a P-AFA for the target set into a P-AFA for its attractor, the winning region of the reachability game, by iteratively adding transitions. The fact that he is considering alternating automata makes it easy to implement the attractor construction for solving reachability games on the level of automata.

Cachat also presents an extension of his construction to Büchi games. The constructions for both reachability and Büchi games are very much inspired by the seminal paper by [BEM97] on the verification of pushdown automata. Cachat's contribution is extending these constructions from verification problems to games. The main modification consists of using P-AFAs instead of P-NFAs, which are defined similarly but do not feature alternation.

Similar to the case of Walukiewicz's reduction, Cachat's algorithm can be applied to use context-free inclusion games. This also requires determinizing the automaton for the target language. Elisabeth Neumann has also considered Cachat's algorithm in her Master's thesis [Neu17] and shown that if one uses the determinization based on the transition monoid, one obtains a correspondence between the structure of the P-AFA representing the winning region and the formulas forming the least solution in our approach.

Muscholl, Schwentick, and Segoufin: Active context-free games

Muscholl, Schwentick, and Segoufin [MSS06] have considered so-called active context-free games. Like in our definition of context-free games, the game arena for these games are induced by context-free grammars and they have regular target languages. However, these target languages are languages over the set of both terminals and nonterminals. Compared to our context-free games, active context-free games proceed differently. In each round of a play, one of the players selects a nonterminal in the current sentential form, then the other player picks a production rule to replace that nonterminal. The winning condition is obtaining a sentential form in the target language. In its most general form, this type of game is undecidable. However, Muscholl et al. show that it is decidable with a left-to-right restriction: In this case, once the player selecting the nonterminals has picked some nonterminal, she is not allowed to pick nonterminals to the left of it during the rest of the play. With this restriction in place, the game becomes very similar to the context-free inclusion games that we consider. In fact, the paper [MSS06] contains a proof showing that one can incorporate features like symmetric rule

choice, i.e. letting each of the players pick the production rules for a certain set of nonterminals like in our type of game. It is not too difficult to transform a context-free regular inclusion game into an active context-free game with the left-to-right restriction and vice versa.

The key difference between the considerations in the paper [MSS06] and our study is that Muscholl, Schwentick, and Segoufin were mainly concerned with the decidability and computational complexity of active context-free games and their various restrictions. To obtain the upper bounds, they simply reduce their games to the games considered by Cachat. This results in an algorithm that has the optimal asymptotic time complexity, but likely would be inefficient in practice. In contrast to this, we are interested in an approach that in addition to achieving optimal time complexity is also practical.

The proof that we have provide for the 2EXP-hardness of context-free inclusion games in Section 17.5 is based on the proof of the 2EXP-hardness of active context-free games in [MSS06], which in turn is based on Walukiewicz's proof of EXP-hardness in [Wal01].

Active context-free games have been later extended to the case of a visibly context-free specification of the game arena [SS15], which is a restriction, and a visibly context-free target language, which is a generalization. More recently, Coester, Schwentick, and Schuster [CSS19] have studied active context-free games with imperfect information.

Kupferman and Vardi: Two-way tree automata

Kupferman and Vardi [KV00] actually do not consider games on context-free games. Rather, they consider the μ-calculus model checking problem for context-free trees (computation trees of pushdown automata), which by Walukiewicz's findings is equivalent to solving a parity game. They provide a reduction from this model checking problem to checking the emptiness of a two-way alternating parity tree automaton. As in Cachat's work, alternation provides a way to incorporate the game aspect. A two-way automaton is an automaton that can move up and down in the tree. This is used to handle pops: A pop operation of the pushdown can be simulated by a two-way automaton by simply walking up the tree to the configuration before the letter has been pushed. This reduction is then combined with an earlier result of Vardi [Var98] that shows how a two-way alternating parity tree automaton can be transformed into an equivalent regular (one-way) alternating parity tree automaton. The emptiness problem for this automaton can be solved by a variation of the construction that has been used to prove Rabin's tree theorem (which we have briefly described in Chapter 15).

Serre: Regular winning regions of context-free games

Serre [Ser03] has conducted a more general study of the winning regions of context-free games. He has shown that if the winning condition of an (ω-)context-free game is (ω-)regular, then so is its winning region. Our findings certainly confirm his result: As detailed in Section 17.4, for each of the players, we can construct a deterministic finite automaton representing her winning region, proving that it is regular.

17.7 Algorithmics

We discuss a prototype implementation of our procedure solving context-free regular inclusion games. We also list several techniques that can be used to speed it up. Most of these ideas have already been implemented with various degrees of success.

Evaluation of a prototype implementation

In conjunction with the conference publication of our procedure for solving context-free games [HMM16], the author of this thesis has developed a prototype implementation of the algorithm that we have described in the first half of Section 17.5. In the publication, we have compared this implementation to another approach to solving context-free games that works by first computing an equivalent instance of Cachat's context-free games and then applying Cachat's algorithm. Here, we discuss the implementations briefly and recapitulate the evaluation that is present in our conference publication.

Our prototype implementation [Mus16] consists of several parts. The first is a random generator for automata and game grammars. The generator for automata follows the Tabakov-Vardi model [TV05]. The generator for grammars is an extension of the Tabakov-Vardi model to context-free grammars. We use the generators to provide us with randomly generated instances of context-free regular inclusion games for certain parameters, like the alphabet size, the number of states of the automaton and the number of nonterminals of the grammar.

The second and main part of the prototype is an implementation of our procedure for solving context-free regular inclusion games using effective denotational semantics. The procedure constructs the system of equations from the given grammar. It then solves it using Kleene iteration: We first assign false to all variables, i.e. the nonterminals, and then update their values according to the production rules of the grammar until a fixed point has been reached. The equivalence classes of formulas are stored by storing some arbitrary representation in conjunctive normal form, as explained in Section 17.5. Using conjunctive normal form simplifies various parts of the algorithm, e.g. checking whether the fixed point has been reached, but leads to a potential increase in formula size. The prototype contains both a naive implementation of Kleene iteration and a worklist-based implementation of chaotic iteration. In the naive implementation, the value of each variable is updated in every step. Worklist-based chaotic iteration precomputes the dependencies among the variables to speed up the solving process as described in Section 16.1.

As a third part, the prototype contains the reduction to Cachat's type of context-free games. Recall from the previous section that these games are played on the configuration graph of a pushdown automaton with the goal of reaching a stack content that is in a specified regular language. We describe the idea of the reduction in detail. Given a context-free regular inclusion game (G, A), we first determinize and minimize the automaton A using standard techniques. This potentially leads to an exponential blowup in the size of the automaton, which is to be

expected since solving Cachat's context-free games is only EXP-complete, in contrast to the 2EXP-completeness of context-free regular inclusion games. We then construct a pushdown automaton that is essentially the product of the grammar and the determinized automaton. Its control-state stores the state of the automaton after reading the terminal prefix of the sentential form. The stack stores the rest of the sentential form. Formally, for each state q of the determinization of A, the pushdown has two states, $q_{\bigcirc}$ and $q_{\Box}$ owned by the respective player. If the top-of-stack is a terminal symbol a, the pushdown has transitions $q_{\bigcirc} \to p_{\bigcirc}$ and $q_{\Box} \to p_{\Box}$ that pop symbol a, where p is the unique state of the DFA with $q \xrightarrow{a} p$. If the top-of-stack is a nonterminal X that is owned by $\Box$, and the current state is $q_{\bigcirc}$, the automaton has a transition $q_{\bigcirc} \to q_{\Box}$ that hands control to the correct player without modifying the top-of-stack. In case X is owned by $\bigcirc$ and the state is $q_{\Box}$, the transition is similar. If the owner $\bigstar \in \{\bigcirc, \Box\}$ of the top-of-stack X and the state $q_{\bigstar}$ match, player $\bigstar$ may select a production rule $X \to \beta$. Formally, we have a transition that maintains state $q_{\bigstar}$, but replaces the stack symbol X by the sequence β, with the leftmost symbol of β becoming the new top-of-stack. Finally, we also need an automaton that represents the winning condition in the game. This automaton simply accepts any configuration of the pushdown if the control state is not final and the stack is empty. It is not difficult to check that producing a terminal word that is not in the language of A in the grammar-based game is equivalent to reaching a non-accepting control state of the DFA with empty stack in Cachat's type of game. Hence, the winners of both games coincide.

The fourth and final part of our implementation is a prototype implementation of Cachat's algorithm for his types of games, since to the best of the author's knowledge, no such implementation is publicly available. The implementation follows Cachat's description of his algorithm in the paper [Cac02] in a straightforward manner.

It might seem unfair to compare a worklist-based implementation of Kleene iteration to a naive implementation of Cachat. We have actually also implemented a worklist-based implementation of Cachat, but it does not speed up the algorithm. The problem is that the reduction from inclusion games to Cachat's type of games produces instances that are dense in the sense that any state of the pushdown has at least one transition for any possible top-of-stack.

One should be aware of the fact that we are not comparing our algorithm to solve context-free regular inclusion games with Cachat's algorithm to solve his type of games. Rather, we are comparing the two algorithms on context-free regular inclusion games, which requires a reduction to be able to apply Cachat's algorithm. This reduction may result in instances that Cachat's algorithms is not tailored to.

In our conference publication, we have generated random instances of context-free regular inclusion games consisting of linear grammars. This means that all transitions of the grammar are of the shape $X \to a.Y.b$ with $a, b \in \Sigma \cup \{\varepsilon\}$ terminals and $Y \in N \cup \{\varepsilon\}$ a single nonterminal. Both algorithms benefit from this restriction, but our approach does so in a major way. Consider a sentential form $a.Y.b$ where F is the formula that is currently associated to Y. To compute the formula associated to $a.Y.b$, it is sufficient to replace every atom ρ that occurs in F by $\rho_a \cdot \rho \cdot \rho_b$.

In other words, the grammar being linear makes computing the composition of formulas that is needed when evaluating the right-hand sides of the equations simple.

The experimental evaluation in the conference publication shows that the worklist-based implementation of Kleene iteration is typically faster than the naive one by one order of magnitude, i.e. a factor of 10. There are even cases where the naive implementation reaches our 10-second timeout limit, while the worklist-based one finishes in roughly 100 milliseconds. This is partially because we do not minimize the formulas after every step: Doing unnecessary updates of the variables does not change the equivalence class of the associated formula, but it may increase the size of its representation. This increase in size can then propagate in later steps to the values of other variables, substantially slowing down the algorithm. Minimizing the variables in every step could avoid this, at the cost of the quadratic running time of the minimization (in the current size of the formula). Using the worklist procedure is an easy and efficient way to avoid this problem.

Our procedure solves context-free regular inclusion games with linear grammars substantially faster than Cachat's algorithm. Even for very small instances, it is faster by two orders of magnitude, i.e. a factor of about 100. For instances with 10 control states in the NFA, 10 terminal symbols, and 10 nonterminals (5 owned by each player), Cachat's algorithm was not able to solve 46 out of 50 randomly generated instanced within the 10-second time limit. For the 4 instances that it was able to solve, it took about 7.7 seconds on average. Our algorithm (with the worklist-based iteration scheme) applied to the very same set of instances could solve all the instances within the time limit and took an average of 0.2 milliseconds. Increasing the number of terminals to 15 and the number of nonterminals to 30 resulted in Cachat's algorithm to not being able to solve any of 50 randomly generated instances within the time limit. Our algorithm solved all of these instances with an average running time of 1.8 milliseconds. This means that in this case, our algorithm is more efficient by three orders of magnitude. For more data on the evaluation of these algorithms, we refer to the publication [HMM16] and the code [Mus16].

We claim that one of the reasons why our algorithm is superior for solving context-free inclusion games is that it avoids the upfront determinization of the automaton for the target language. If the existential player can enforce the derivation of a finite word w, then our algorithm will eventually compute the behavior of w in the finite automaton in the form of the transition monoid element for w. For all words whose derivation cannot be enforced by the existential player, we never consider their behavior in the automaton. This means that potentially, a large part of the determinization of the automaton remains unexplored, which saves running time. In contrast to this, solving context-free inclusion games by using Cachat's algorithm requires us to determinize the automaton at the start. The result is an instance that is potentially exponentially larger, even if only a fraction of the deterministic automaton will actually be needed.

Non-linear grammars

We have investigated the behavior of the implementations on non-linear grammars, i.e. grammars in which the right-hand side of a production rule may contain several nonterminals. This means that when evaluating the system of equations, we need to compute the composition of non-atomic formulas. Consequently, the formulas that we need to store can get very large within few iterations. In unpublished experiments using our prototype, we have observed that this quickly leads to an exorbitant memory condition. For some instances, the solver consumes the 12 GB of available main memory of the machine we ran the tests on within a few seconds and the program goes out-of-memory before it could exceed the 10-second timeout.

When comparing the performance of our procedure with the other approach that relies on Cachat's algorithm, we still obtain that our procedure is vastly superior. However, in the case of non-linear grammars, our procedure was often unable to solve instances that consisted of an automaton with more than 5 states without exceeding the available main memory, which is unsatisfactory.

Our prototype implementation does not minimize formulas and it keeps all formulas that occur as approximants when conducting Kleene iteration in memory until the fixed point has been computed. Changing these aspects can potentially mitigate the problems with solving games defined by non-linear grammars. We have tasked a group of students, supervised by Roland Meyer and the author of this thesis, to develop a more advanced implementation of the algorithm. While the resulting tool indeed performs better than the prototype implementation, especially on instances with non-linear grammars, it could not fully overcome the problems that arise from the high memory consumption of some instances. This tool is also the basis for several attempts to improve the algorithm that we will report on in the following. The idea behind some of these attempts has already been outlined in the full version [HMM16a] of our publication [HMM16] on context-free games.

Non-CNF formulas

Our prototype implementation stores formulas in conjunctive normal form (CNF). Using Lemma 17.5.2, this allows us to check whether a formula implies another in quadratic time. The result of computing the conjunction of two formulas is additive, computing the disjunctive is multiplicative, and computing the composition is exponential in the size of the given formulas.

In his bachelor's thesis [Stu17], Felix Stutz, supervised by Roland Meyer and the author of this thesis, has explored the option of not normalizing the formulas to conjunctive normal form. This potentially allows us to obtain a smaller representation for formulas, which decreases both the memory consumption and the time needed for computing subsequent operations on these formulas. However, there is a major drawback. Recall that the Kleene iteration terminates as soon as two subsequent approximants are equal, where checking the equality of equivalence classes of formulas amounts to checking logical equivalence for representatives. Since implication in

one direction is guaranteed to hold, checking whether the algorithm terminates means checking an implication among positive Boolean formulas. For formulas in conjunctive normal form, Part d) of Lemma 17.5.2 provides a characterization of implication that can be used to implement a check using quadratic time. For non-normalized formulas, the problem is substantially more involved. Felix Stutz has proven in his bachelor's thesis that checking the implication between positive Boolean formulas is coNP-complete. Intuitively, checking whether $F \implies H$ holds amounts to checking whether $\neg F \vee H$ is a tautology, with is a coNP-complete problem, even if F and H are not allowed to contain negations.

To handle the implication check, Stutz proposes several approaches. One of them is simply encoding the implication as an instance of the unsatisfiability problem and employing a SAT solver. However, an implementation of this approach did not consistently outperform the version that uses formulas in conjunctive normal form. A more promising approach uses that the implication $F \implies H$ holds if and only if all minimal satisfying assignments for F also satisfy H. A minimal satisfying assignment is a variable assignment $M \subseteq \mathbb{M}_A$ under which F evaluates to true so that F does not evaluate to true under any strict subset $M' \subsetneq M$. One can design a procedure that computes a superset of the minimal assignments that satisfy F and checks whether they also satisfy H. Implication holds if and only if this true for all minimal assignments, and as soon as we encounter an assignment that is a counterexample to implication, we can terminate. An implementation of the implication check that uses this approach consistently outperforms the CNF-based implementation. It can solve more instances in a ten-second time frame and in the case that both implementations can solve an instance, the approach based on minimal satisfying assignments is at least twice as fast on average.

Antichain optimizations

A second approach to increasing the performance of our solver for context-free games is the so-called antichain optimization. It was successfully used in the context of transition-based method for universality and inclusion testing for nondeterministic finite automata [DDHR06; ACHMV10]. In his master's thesis [Hai17], Fajar Haifani, supervised by Roland Meyer and the author of this thesis, has explored the antichain optimization for solving context-free games using fixed-point iteration.

In the development throughout this chapter, we have considered the transition monoid elements as distinct and unrelated atoms of positive Boolean formulas. The key observation enabling the antichain optimization is that transition monoid elements carry additionally information that can be used to relate them to each other. Let us see a transition monoid element as a box, i.e. as an element of $\mathcal{P}(Q \times Q)$. If a box is a subset of another, $\rho \subseteq \rho'$, then box ρ' can only be rejecting if ρ is rejecting, since being rejecting is defined as the absence of a transition from the initial to a final state. We can extend this subset relation from boxes to formulas as follows. Instead of considering the standard definition for the implication $F \implies H$, i.e. we require all assignments that satisfy F to also satisfy H, we restrict ourselves to assignments that respect the relation among boxes. More formally, we only consider assignments with the property that

if they set some box ρ' to true, then they also set all its subsets $\rho' \subseteq \rho$ to true. We redefine implication so that $F \implies H$ holds if every assignment that respect the relation among boxes and that satisfies F also satisfies H.

Considering less assignments in the definition of implication means that it is easier for a formula to imply another. When we redefine logical equivalency using our new notion of implication, we get larger equivalence classes. Hence, we expect Kleene iteration to terminate in fewer steps compared to the naive approach that does not consider relations among boxes. However, using the antichain approach also requires a more involved implication check. We have to replace the characterization of implication provided by Part d) of Lemma 17.5.2 by a version that considers the subset relation among boxes.

In addition to just considering subsets relations among boxes, there are relations that are even more powerful, meaning that they relate more boxes. These relations on boxes arise from so-called simulation relations [DHW91] on states of the automaton. Using these relations gives us even larger equivalence classes, so we expect faster termination. However, this comes at the cost of having to precompute simulation relations on states of the automaton.

Using an extension of the aforementioned tool, Fajar Haifani has implemented various relations on boxes and compared their performance to the naive approach. In all cases, the evaluation represents formulas using conjunctive normal form. Some cases also reduce the formulas in the current approximant every few steps, meaning that they try to exploit the relations among boxes to find a smaller representative for the same equivalence class. The hope is that this reduction will improve the space consumption of the formulas and the time needed for subsequent operations on these formulas.

The evaluation in Haifani's thesis suggests that on average, the refined implementation using relations on boxes slightly outperforms the naive approach. In the best case, it cuts the time needed to solve the instances in half. Unfortunately, one also encounters instances where the cost of the precomputation and the more expensive implication check outweighs the reduced number of iterations – the optimization increases the running time of the algorithm. Interestingly, Haifani's data suggests that for almost all game instance, either the naive approach is better than all versions of the antichain optimization (using the subset relation or various relations arising from simulation relations), or every version of the antichain optimization outperforms the naive approach.

One should note that both our publication, and the theses by Stutz and Haifani, use randomly generated instances for the evaluation of the implementations. Randomly generated instances are known from e.g. SAT solving to have much less structure that algorithmic optimizations can exploit to speed up the solving process, compared to instances that occur in practical examples. It would be interesting to see the behavior of the various implementations and optimizations on instances that result from e.g. encoding real program synthesis problems as context-free games.

17.8 Deterministic target languages

We consider the special case of context-free regular inclusion games where the regular target language is given by a deterministic automaton. It can be show that solving such games is EXP-complete, instead of 2EXP-complete as in the case of an NFA representing the target language. We modify our algorithm to achieve a matching running time.

To show EXP-completeness, on can for example use that the context-free games on the configuration graphs of pushdown automata that were considered by Walukiewicz [Wal01] and Cachat [Cac02] can be translated into a grammar-based context-free game with a DFA target language within polynomial time. Hence, Walukiewicz'sproof of EXP-completeness for this type of game applies.

One also can immediately see that our proof of 2EXP-hardness, Theorem 17.5.4, does not work. Dealing with configurations of an alternating Turing machine with exponential space consumption requires dealing with binary strings of polynomial length. An automaton of polynomial size for this task is necessarily nondeterministic. One could adapt the proof to the case of a deterministic target language as follows: We assume that the given alternating Turing machine has polynomial space consumption, using that APSPACE = EXP. Hence, the binary strings indexing cells on the tape are only of logarithmic size. The logarithmic nondeterministic automaton dealing with these strings can be transformed into a polynomially sized deterministic one to complete the proof.

Unfortunately, the algorithm that we have outlined in this chapter cannot exploit the fact that the target language is deterministic to achieve a better running time. The crucial factor in the running time of the algorithm is the size of transition monoid $\mathbb{M}_A$, the number of boxes. If automaton A is a DFA, then there are fewer potential boxes: For each of states, there is only a single successor state along each word. Hence, there are at most $|Q|^{|Q|} = 2^{\log|Q|\cdot|Q|}$ different boxes in $\mathbb{M}_A$. This number is substantially smaller than $2^{|Q|^2}$, the number of boxes in the general case, but it still is exponential.

To overcome this problem, we propose a variation of the algorithm that uses $\mathrm{pBF}(Q)$ as domain. Instead of formulas over the transition monoid, where the atoms track the full behavior of words in the automaton, we have formulas over Q, where each atom corresponds to a target state. To make this approach work, we also need to fix the source state. Formally, we have a system of equations with variables of the shape qX, where q is a (source) state and X is a nonterminal. An atom p in the formula $\mathrm{sol}(qX)$, taken from the least solution to the system, corresponds to a word w derivable from X with $q \xrightarrow{w} p$ in A.

However, this introduces a new challenge. Recall that the composition of two formulas $\mathrm{sol}(Y) \;.\; \mathrm{sol}(Z)$ is defined so that it captures the structure of words derivable from $Y.Z$: Each such word is a word derivable from Y followed by a word derivable from Z. When using formulas over the transition monoid, we can simply compose the atoms corresponding to these

words using relational composition. With the modified system of equations, $\mathrm{sol}(qY)\ .\ \mathrm{sol}(pZ)$ only makes sense if we derive from Y a word that indeed leads to target state p. However, the words derivable from Y can lead to many different target states, and a priori it is not clear to which ones.

The solution to this problem is to compose $\mathrm{sol}(qY)$ not with a single $\mathrm{sol}(pZ)$, but with the family of formulas $(\mathrm{sol}(pZ))_{p\in Q}$ for all $p \in Q$ simultaneously. The composition will be defined so that it first resolves the operators in $\mathrm{sol}(qX)$ until arriving at an atom p. It then selects the appropriate formula $\mathrm{sol}(pZ)$ from the family to complete the composition.

Formally, we define a new composition operator $;$. It is $(|Q|+1)$-ary, but for the sake of readability we write the last $|Q|$ arguments as a family of formulas. This means we write $F\ ;\ (H_q)_{q\in Q}$ where F and H_q for every $q \in Q$ are formulas. The definition is as follows:

$$(F \overset{\wedge}{\vee} F')\ ;\ (H_q)_{q\in Q} = F\ ;\ (H_q)_{q\in Q} \overset{\wedge}{\vee} F'\ ;\ (H_q)_{q\in Q}$$
$$p\ ;\ (H_q)_{q\in Q} = H_p\ .$$

The operation essentially replaces each atom p in the first formula by H_p.

With this definition at hand, we can formally define the system of equations. As variables, we use expressions of the shape qX where $q \in Q$ and $X \in N$. Assume that $X \to \beta^{(1)} \mid \ldots \mid \beta^{(k)}$ are all production rules for nonterminal X. For each state q, the defining equality for qX is

$$qX = \mathrm{preprocess}_q(\beta^{(1)}) \overset{\wedge}{\vee} \ldots \overset{\wedge}{\vee} \mathrm{preprocess}_q(\beta^{(k)})\ .$$

As usual, $\overset{\wedge}{\vee}$ is disjunction if the owner of X is the existential player and conjunction otherwise. The operation $\mathrm{preprocess}_q$ is defined on sentential forms β as follows: If β consists of a single symbol, $\beta \in \mathbb{N} \cup \Sigma \cup \{\varepsilon\}$, we have

$$\mathrm{preprocess}_q(\beta) = q\,\beta\ .$$

If $\beta = \beta_1.\beta_2 \ldots \beta_m$, we prefix the first symbol with q, replace the rest of the symbols with the corresponding families, and connect the results using the new composition operator,

$$\mathrm{preprocess}_q(\beta_1.\beta_2 \ldots \beta_m) = q\,\beta_1\ ;\ (p\,\beta_2)_{p\in Q}\ ;\ \ldots\ ;\ (p\,\beta_m)_{p\in Q}\ .$$

The system of equations uses disjunction, conjunction, the new composition operator and constants of the shape $q\,a$ for $q \in Q$ and $a \in \Sigma \cup \{\varepsilon\}$ as function symbols. We provide a model over which the system can be interpreted. The domain is $\mathrm{pBF}(Q)$ factored by logical equivalence, i.e. equivalence classes of positive Boolean formulas over states of the DFA, ordered by implication. The interpretation of conjunction, disjunction, and the new composition operator is as expected. For each state q, the interpretation of $q\varepsilon$ is $q \in \mathrm{pBF}(Q)$. For state q and terminal $a \in \Sigma$, $q\,a$ is $p \in \mathrm{pBF}(Q)$, where p is the unique state of the DFA with $q \xrightarrow{a} p$.

Given an instance (G, A) of a context-free regular inclusion game with a DFA A representing the target language, one solves the game as follows. We first construct the system of equations, and then compute its least solution using the above interpretation. The least solution provides us with a formula $\mathrm{sol}(q_{\mathrm{init}}S) \in \mathrm{pBF}(Q)$ for the initial state of A and the initial symbol of G. We consider the variable assignment $Q_{\mathrm{rej}} = Q \setminus Q_{\mathrm{final}} \subseteq Q$ that sets to true the non-final states. The existential player has a winning strategy for the game iff $\mathrm{sol}(q_{\mathrm{init}}S)(Q_{\mathrm{rej}})$ evaluates to true.

To show this, one can prove a correspondence between variable assignments $Q' \subseteq Q$ that satisfy a formula $\mathrm{sol}(qX)$ and strategies for the existential player that enforce the derivation of a word w from X such that $q \xrightarrow{w} q'$ with $q' \in Q'$. To make this formal, the theory that we have developed in Section 17.3 can be adapted easily.

The resulting algorithm solves the game in exponential time, which is the optimal complexity. The new system of equations is substantially larger as it features one equation per state and nonterminal, but its size remains polynomial. For solving the interpreted system of equations, using Q as the set of atoms is the determining factor for the complexity. This means that the height of the domain as well as the maximum size of a single formula in conjunctive normal form are singly exponential. This can be made formal by adapting the proof of Proposition 17.5.3.

Altogether, we have shown the following.

17.8.1 Theorem
Context-free regular inclusion games with the target language represented by a DFA are EXP-complete, and an algorithm based on effective denotational semantics solves them in exponential time.

When we are given a context-free regular inclusion game (G, A) where A is an NFA, we now have two procedures with doubly exponential running time for solving it. The first is the algorithm that we have discussed earlier in this the chapter. The second procedure starts by determinizing the automaton, obtaining a new instance (G, A^{det}) that is potentially exponentially larger. However, we can now apply the algorithm described in this section to solve this larger instance in singly exponential time (in its size), resulting in doubly exponential time overall.

While both algorithms solve the problem, we expect the first approach to be much more efficient for the same reasons that we have discussed in Section 17.7. An upfront determinization of the automaton introduces a guaranteed blowup, even if some parts of the automaton may actually never be used by words that can be derived in the grammar. The first approach uses an on-the-fly determinization that may avoid this blowup.

17.9 ω-context-free inclusion games

We complete our study of context-free games by studying ω-context-free ω-regular inclusion games. In the case of context-free inclusions games with a regular language of finite words as the target language, all infinite plays are won by the universal player. In order to be able to win, the existential player had to be able to enforce the termination of any play. Here in this section, we will take the opposite approach. We define the winning condition so that the universal player wins all plays that do terminate. For plays that are infinite, the winner is determined by whether the play corresponds to an infinite word in the given ω-regular target language. To be precise, the existential player wins if the play corresponds to a right-infinite left-derivation process that produces an infinite word that is not contained in the target language.

Formal definition

Formally, an ω-context-free (ω-regular) inclusion games is given by a context-free game grammar $G = (N_\Box \uplus N_\bigcirc, P, S)$ and a nondeterministic Büchi automaton $A = (Q, \delta, q_{\text{init}}, Q_{\text{final}})$. The game arena is defined as for context-free regular inclusion games in Section 17.1: Starting from the initial symbol, a play is a left-derivation process in which each player chooses the production rules applied to the nonterminals owner by her. They key difference is that the winning condition for the existential player is now non-inclusion in an ω-regular language. To this end, she has to enforce that the play of the game corresponds to a right-infinite (left-)derivation process as defined in Section 5.2. Recall that this means that the play is a sequence of sentential forms that have a nonterminal as their rightmost symbol which is replaced infinitely often. Consequently, the derivation of all other nonterminals is a finite process. The existential player wins the play if and only if it constitutes such a right-infinite derivation process, deriving an infinite word w not contained in $\mathcal{L}^\omega(A)$. In all other cases, the universal player wins: This applies when the play is a right-infinite derivation process deriving a word from $\mathcal{L}^\omega(A)$, but it also applies if the play is finite, or if it is infinite, but it is not a right-infinite derivation process, or if it is a right-infinite derivation process that does not derive an infinite word. The latter case could occur if starting at some point, the terminal prefix of the sentential form stops growing. Our hope is to extend the techniques for context-free games in the same way that we extended our algorithm for context-free regular inclusion to ω-context-free ω-regular inclusion in Section 16.3. To do so, there are two major challenges. The first lies in the fact that the automaton representing the ω-regular target language is nondeterministic. The second one is that least fixed-points as used in effective denotational semantics seem to be insufficient to capture the semantics of this type of game. We discuss each of the problems in detail and present our solutions.

Determinizing the automaton

We have explained in Chapter 15 that a nondeterministic automaton representing the target language poses a problem, as it essentially introduces a third player that can be merged with neither of the other two players without changing the semantics of the game. In the case of

context-free inclusion games, we have overcome this problem by using the transition monoid. We have relied on the fact that using the transition monoid is a way of implicitly determinizing the automaton. The same is not true for nondeterministic Büchi automata: Firstly, we cannot determinize an NBA into a deterministic Büchi automaton in general. Secondly, even by considering a more involved acceptance condition for which deterministic automata are sufficiently expressive, like the parity or Muller acceptance conditions, considering the transition monoid is not enough. Unlike in the case of automata on finite words, in general there is no deterministic automaton that uses transition monoid elements (boxes) as states that is equivalent to a given NBA. We have used in Section 16.3 that infinite words can be represented by pairs of boxes $\tau\rho^\omega$. However, during the derivation of a word, it is impossible to decide on-the-fly which pair of boxes $\tau\rho^\omega$ is the correct one and where the split between the $\mathcal{L}(\tau)$ and the $\mathcal{L}(\rho)^\omega$ part happens. In the case of a verification problem, we could overcome this by guessing nondeterministically. In the case of a game, this mechanism is invalid: We cannot let one of the players guess for the same reasons that cannot let one of players pick the moves of the automaton (as explained in Chapter 15). - Our solution to this problem is an upfront determinization of the automaton, the very thing that we successfully avoided in the case of context-free inclusion games. In the rest of this section, we assume that the automaton representing the target language is a deterministic parity automaton $A = (Q, \Sigma, \delta, q_{\text{init}}, \Omega)$. Using a famous result by Safra [Saf88], a given NBA with n states can be transformed into a DPA with $2^{\mathcal{O}(n \cdot \log n)}$ states. These states are so-called Safra-trees, and the construction is a variant of the powerset construction that is rather involved. We also assume that the priority function $\Omega\colon Q \to \mathbb{N}$ does not assign the priority 0 to any state, as we will need this priority later for the empty word. This condition can be easily enforced by incrementing all priorities by 2. Let d be the largest priority assigned to any state of the automaton in the following.

Solving finite subgames

Before actually considering the ω-context-free game, we set up a system of equations whose least solutions characterizes the effect of finite words on the parity automaton. We re-use the system that we have developed in the previous section for context-free games with the target language being represented by a DFA. The only modification that is needed is tracking the priorities in the automaton. Formally, we consider formulas in $\mathrm{pBF}(Q \times [0, d])$ whose atoms are tuples of a target state and the largest priority that has been seen.

We quickly recall the construction. For each state q and each nonterminal X, there is a variable qX. Its defining equation is

$$qX = \mathrm{preprocess}_q(\beta^{(1)}) \mathbin{\overset{\wedge}{\vee}} \ldots \mathbin{\overset{\wedge}{\vee}} \mathrm{preprocess}_q(\beta^{(k)})\,.$$

Here, $X \to \beta^{(1)} \mid \ldots \mid \beta^{(k)}$ are all production rules for nonterminal X, and $\overset{\wedge}{\vee}$ is disjunction if and only if the owner of X is the existential player. The operation $\mathrm{preprocess}_q$ applied to a

sentential form $\beta_1.\beta_2 \dots \beta_m$ prefixes the first symbol by q, replaces all other symbols β_i by the family $(p\,\beta_i)_{p\in Q}$, and connects them using the composition operator ; . Formally, we define

$$\mathrm{preprocess}_q(\beta_1.\beta_2 \dots \beta_m) = q\,\beta_1 \;;\; (p\,\beta_2)_{p\in Q} \;;\; \dots \;;\; (p\,\beta_m)_{p\in Q}\;.$$

This system is interpreted over (equivalence classes of) positive Boolean formulas from $\mathrm{pBF}(Q \times [0,d])$, ordered by implication. Conjunction and disjunction are interpreted in the expected way. For each terminal $a \in \Sigma$ and each state $q \in Q$, the interpretation of the constant $q\,a$ is the formula just consisting of the atom (p, j), where p is the unique state with $q \xrightarrow{a} p$ in the automaton, and $j = \max\{\Omega(q), \Omega(p)\}$. For any state q, the interpretation of the constant $q\,\varepsilon$ is the atom $(q, 0)$. Our rationale behind using priority 0 here (which by assumption is not assigned to any state) is that in the ω-context-free game, the existential player has to guarantee the derivation of an infinite word. If the game derives an infinite sequence of ε starting at some point, the result of the play is not an infinite word, and the play is won by the universal player. Therefore, we should assign an even priority to ε, since even priorities correspond to words that are accepted by the automaton (which is good for the universal player). However, the derivation of infinitely many ε is only problematic if no non-ε word is derived infinitely often – therefore, we assign the lowest even priority to ε.

The composition operator ; is a $(|Q|+1)$-ary operator that composes a formula F with a family of formulas $(H_q)_{p\in Q}$. Compared to the previous section, the definition is altered so that it keeps track of the maximal priority. We inductively define $F \;;\; (H_q)_{p\in Q}$ to be

$$\begin{aligned}(F \mathbin{\overset{\wedge}{\vee}} F') \;;\; (H_q)_{q\in Q} &= F \;;\; (H_q)_{q\in Q} \mathbin{\overset{\wedge}{\vee}} F' \;;\; (H_q)_{q\in Q}\\ (p, j) \;;\; (H_q)_{q\in Q} &= \max\nolimits_j H_p\;,\end{aligned}$$

where $\max_j H_p$ is defined by

$$\begin{aligned}\max\nolimits_j (H \mathbin{\overset{\wedge}{\vee}} H') &= \max\nolimits_j H \mathbin{\overset{\wedge}{\vee}} \max\nolimits_j H'\\ \max\nolimits_j (s, j') &= (s, \max\{j, j'\})\;.\end{aligned}$$

The intuition behind the definition is as follows: Assume the automaton is in state q, and we derive a finite word from the sentential form $Y.Z$. This means we first derive a finite word from Y, tracking the target state p and the maximum priority j that has occurred during the run of the automaton from q to p. We then derive a word from Z while tracking the target state s of this word from the source state p and the maximum priority j' along the way. The information of interest for the concatenation of these words is the target state s and the maximum priority $\max\{j, j'\}$ seen along the run from q to s.

With the interpretation fixed, we can solve the system of equations. We obtain for each pair qX of state and nonterminal a formula $\mathrm{sol}(qX) \in \mathrm{pBF}(Q \times [0,d])$. The atoms (p, j) in that formula

correspond to finite words derivable from X that introduce a run from q to p with maximum priority j in the automaton.

Solving the ω-game

It remains to use the information about the finite subgames to solve the ω-context-free game. Recall that a derivation of an infinite word in the language of an ω-context-free grammar can be seen as a nested process consisting of an infinite path in the spinal graph of the grammar (as defined in Section 4.4) and a sequence of finite derivations from the symbols occurring as transition labels along that path. Similarly, a play of an ω-context-free game consists of an infinite play in the spinal graph constructing an infinite sentential form, and a sequence of plays from the symbols of that sentential form. In order to win, the existential player has to enforce that the latter are finite, i.e. they are plays of a context-free game.

We use this insight to construct a parity game on a finite arena that is induced by the spinal graph and the parity automaton representing the target language. For each nonterminal X and state q of the DPA, (X, q) is a position of the game. The initial position is (S, q_{init}), consisting of the initial symbol and the initial state. When the parity game that we construct is in some position (X, q), it proceeds as follows. Firstly, the owner of X picks some production of the grammar for X. If this production is not of the shape $X \to \beta.Y$, i.e. its rightmost symbol is not a nonterminal, the play does not represent a right-infinite left-derivation process and the universal player wins immediately. Else, we consider the formula $F = \text{sol}(\text{preprocess}_q\beta)$ that describes the behavior of the finite play from β with respect to the initial state q. Here, we have used the preprocess_q operator to insert the states of the automaton into the sentential form. Secondly, we let the players resolve the operators in F in the expected way: The existential player resolves disjunctions, the universal player resolves conjunctions. Finally, the play arrives at an atom of the shape (p, j). From this position, the play continues at (Y, p), the position formed by the rightmost nonterminal of the production and the state that was reached by the finite word that was derived from β. The priority of each position in the parity game is 0, unless the position is of the shape (p, j), in which case it is j.

A play of the parity game in which only transitions of the shape $X \to \beta.Y$ are picked corresponds to a right-infinite left-derivation process. The finite play that unfolds from β is represented by the formula structure. This means we use the formulas to represent a play of finite, but unbounded length by a play on the formula of bounded length. The literal (p, j) that is the result of such a bounded play corresponds to the behavior of the finite word that is derived from β on the DPA. In particular, the priority j corresponds to the maximum priority seen in the run on the word. If the existential player wins the parity game, i.e. she can enforce a play in which the dominating priority is odd, she can enforce the derivation of an infinite word that is not accepted by the DPA. This means she wins the ω-context-free game by proving non-inclusion.

Before giving the proof, it is necessary to make the construction of the parity game formal. For the sake of simplicity, we proceed as in the proof of Proposition 17.5.3 and assume that all for-

mulas are in conjunctive normal form. This means that $\mathrm{sol}(\mathrm{preprocess}_q\beta)$ is a set of clauses, and each such clause is a set of literals. The parity game has as positions a sink $\bot$, positions of the shape (X, q) and positions for the formulas, clauses, and literals,

$$\begin{aligned}\Gamma = \quad & (N \times Q) \cup \{\bot\} \\ & \cup \left\{\left(\mathrm{sol}\left(\mathrm{preprocess}_q\beta\right), Y\right) \;\middle|\; X \to \beta.Y \text{ is a production}\right\} \\ & \cup \left\{(K, Y) \;\middle|\; K \in \mathrm{sol}\left(\mathrm{preprocess}_q\beta\right), X \to \beta.Y \text{ is a production}\right\} \\ & \cup \left\{((p, j), Y) \;\middle|\; (p, j) \in K \in \mathrm{sol}\left(\mathrm{preprocess}_q\beta\right), X \to \beta.Y \text{ is a production}\right\}.\end{aligned}$$

The initial position is (S, q_{init}). The existential player owns all positions of the shape (X, q) for $X \in N_{\bigcirc}$ and all positions (K, Y) corresponding to clauses. All other positions are owned by the universal player. The priority of a position (p, j) is j. The priority of all other positions is 0.

The transitions connect positions of the shape (X, q) to formulas according to the productions for X, or to the sink state. The sink state has a self-loop as its unique outgoing transition. They also connect formulas to their clauses and clauses to their literals. Empty clauses are connected to the sink state. Finally, literals are connected to the positions of the shape (X, q),

$$\begin{aligned}T = \quad & \{(X, q) \to \bot \mid X \to \eta \text{ is a production with } \eta \notin (N \cup T)^*.N\} \\ & \cup \{\bot \to \bot\} \\ & \cup \left\{(X, q) \to \left(\mathrm{sol}\left(\mathrm{preprocess}_q\beta\right), Y\right) \;\middle|\; X \to \beta.Y \text{ is a production}\right\} \\ & \cup \left\{\left(\mathrm{sol}\left(\mathrm{preprocess}_q\beta\right), Y\right) \to (K, Y) \;\middle|\; K \in \mathrm{sol}\left(\mathrm{preprocess}_q\beta\right), X \to \beta.Y \text{ is a production}\right\} \\ & \cup \left\{(K, Y) \to ((p, j), Y) \;\middle|\; (p, j) \in K \in \mathrm{sol}\left(\mathrm{preprocess}_q\beta\right), X \to \beta.Y \text{ is a production}\right\} \\ & \cup \left\{(K, Y) \to \bot \;\middle|\; K = \varnothing, K \in \mathrm{sol}\left(\mathrm{preprocess}_q\beta\right), X \to \beta.Y \text{ is a production}\right\} \\ & \cup \{((p, j), Y) \to (Y, p) \mid Y \in N, p \in Q\}.\end{aligned}$$

With the formal construction at hand, we can prove the main result.

17.9.1 Theorem
The winner of the finite parity game equals the winner of the ω-context-free ω-regular game.

Proof:
It is well-known that parity games are determined: Exactly one of the players has a winning strategy. We prove that this winning strategy induces a winning strategy for the ω-context-free game. We give the formal proof in the case that the existential player wins the parity game. The proof for the universal player is easier as one has to care less about various corner cases.

Assume that some winning strategy for the existential player is fixed. We show how to turn a conforming play of the parity game into a play of the context-free game. We do this in a way

that implicitly defines a winning strategy. In particular, whenever the universal player can make a choice, we consider all of these choices.

Assume that the play of the parity game is in some node (X, q) and the play of the ω-context-free game is in some sentential form $w.X$. Initially, we have $X = S$, $w = \varepsilon$, and $q = q_{\text{init}}$. The owner of X selects a transition $X \to \beta.Y$ in the parity game, and we pick the corresponding production in the ω-context-free game. If $X \in N_{\bigcirc}$, this transition is chosen according to the winning strategy. Note that any production that is not of the shape $X \to \beta.Y$ will never be chosen, since it would lead to the sink state of the parity game in which the universal player wins.

The parity game is now in the state $(\text{sol}(\text{preprocess}_q\beta), Y)$, in which the universal player can choose one of the clauses of the formula. Since we are following a winning strategy, we know that the existential player can react to any choice that the universal player can make. More formally, for each clause $K \in \text{sol}(\text{preprocess}_q\beta)$ that the universal player could select, there is one atom $(p, j) \in K$ contained in that clause so that the winning strategy would select the transition $(K, Y) \to ((p, j), Y)$ in the parity game. Note that this in particular implies that the formula cannot be equivalent to false: A positive Boolean formula in conjunctive formal is unsatisfiable if and only if it contains the empty clause. The empty clause in the game is connected to the sink state, which would lead to the universal player winning the play.

We define a variable assignment that sets the atoms to true that are picked by the winning strategy. More precisely, (p, j) is evaluated to true true if the winning strategy picks the transition $(K, Y) \to ((p, j), Y)$ for some clause K. Since every clause contains at least one such atom, the formula $\text{sol}(\text{preprocess}_q\beta)$ evaluates to true under that assignment. By an analogue of Proposition 17.3.4, adapted to the setting under consideration, one can show that the existential player has a strategy from β that enforces the derivation of a finite word v whose effect is described by (p, j). This means $q \xrightarrow{v} p$ in the DPA, and the maximum priority seen in the run is j. We invoke this strategy in the ω-context-free game to go from sentential form $w.\beta.Y$ (which we obtained from $w.X$ by applying $X \to \beta.Y$) to $w.v.Y$. Now, the play of the ω-context-free game continues from that sentential form, while the play of the parity game continues from (Y, p). We repeat this process ad infinitum.

When following this strategy in the ω-context-free game, we encounter an infinite sequence of sentential forms of the shape

$$S \to^* w^{(1)}X_1 \to^* w^{(1)}w^{(2)}X_2 \to^* \ldots \to^* w^{(1)}w^{(2)}\ldots w^{(k)}X_K \to^* \ldots$$

We argue that this constitutes a right-infinite left-derivation process deriving an infinite word not in the language of the DPA. Firstly, we note that the strategies that we invoke for the subgames enforce deriving a finite word. Thus, every subgame terminates after finitely many steps, and the process is indeed right-infinite. Secondly, each infix $w^{(i)}$ that is derived in such a subgame corresponds to an atom (p_i, j_i) of some formula that is chosen by the winning strategy in the parity game. The fact that the strategy wins the parity game means that the largest priority

occurring infinitely often is odd. In particular, it is not zero, the priority that we have assigned to ε above. Winning the parity game implies that infinitely many of the $w^{(i)}$ are not equal to ε, and we indeed derive an infinite word. Finally, the atoms are witnesses for the unique run

$$q_{\text{init}} \xrightarrow{w^{(1)}} p_1 \xrightarrow{w^{(2)}} p_2 \xrightarrow{w^{(3)}} \dots$$

of the DPA on the infinite word $w^{(1)}w^{(2)}w^{(3)}\dots$ that is derived by the process. Each priority j_i is the maximum priority seen during the run on the finite infix $w^{(i)}$. If the largest priority that occurs infinitely often among the j_i is odd, then so is the largest priority that occurs infinitely often in the run. (Summarizing finite infixes in the sequence of priorities by taking their maximum does not change the dominating priority.) Hence, the run of the DPA on the word is not accepting. The existential player has enforced the derivation of an infinite word that is not in the language of the DPA. ■

With the soundness of the construction proven, we turn to considering its complexity. Assume that we have fixed a grammar G and a DPA with Q as its sets of states. We claim that the running time our algorithm is polynomial in G and exponential in Q. Note that we can assume without loss of generality that the maximum priority d is in $\mathcal{O}(|Q|)$, $d \leqslant |Q| + 4$ to be precise.

Firstly, we can argue similarly to the previous section that computing $\text{sol}(\text{preprocess}_q\beta)$ for each transition $X \to \beta.Y$ can be done in that time. Also, the number of such transitions is bounded by $|G|$. The formulas themselves can be shown to obey a similar bound: The number of atoms is $|Q \times [0, d]| = |Q| \cdot (d + 1)$. This also bounds the size of any clause, and there are at most $2^{|Q|\cdot(d+1)}$ clauses, which bounds the size of any formula. Combining at most $|G|$ of these objects into a parity game leads to a parity game whose size is polynomial in $|G|$, exponential in $|Q|$, and whose maximum priority is in $\mathcal{O}(|Q|)$. We can now use the recent breakthrough result [CJKLS17] that parity games are fixed-parameter tractable: They can be solved in time polynomial in the size of the arena and exponential only in the maximum priority. Since the maximum priority is linear in the size of the original input, solving the parity game overall is exponential in the size of the input. Altogether, we obtain the desired running time.

If we imagine starting with a nondeterministic Büchi automaton instead of a deterministic parity automaton, we first have to apply the Safra construction for determinization. It transforms an NBA with set of states Q into a parity automaton with a set of states of size $2^{\mathcal{O}(|Q|\cdot\log|Q|)}$ and maximum priority in $\mathcal{O}(|Q|)$. Applying the determinization and then the rest of the algorithm leads to a running time that is polynomial in the size of the grammar and doubly exponential in the size of the NBA. This is the optimal time complexity. Obviously, solving ω-context-free games with an NBA representing the target language is 2EXP-hard: A context-free game with an NFA representing the target language can be seen as a special case of such a game, and for this case, we have proven 2EXP-completeness in Theorem 17.5.4. To transform the instance, it is sufficient to modify the grammar and the automaton so that they first produce a finite word as usual, and then proceed to generate an infinite sequence of occurrences of a filler symbol.

Can the parity game construction can be avoided?

We have provided an algorithm for ω-context-free games with the optimal time complexity. The fact that this algorithm relies on solving a parity game might seem undesirable. An algorithm that simply solves a system of equations to compute the winner of the game with no additional steps would be a cleaner solution and potentially more efficient. However, solving parity games is a well-studied topic, so it would be easy enough to actually implement the algorithm by combining a modified version of the solver we have presented in Section 17.7 with a solver for parity game like Oink [Dij18].

In the following, we want to argue that it might be impossible to devise an algorithm for computing the winning of an ω-context-free inclusion game by simply computing a least fixed point. In their papers [SW15b; SW15a], Salvati and Walukiewicz have considered the λY calculus as a tree-generating mechanism which is closely related to higher-order recursion schemes. They have shown that a so-called Scott model (which is similar to the model template for HORSes that we will introduce in Section 18.2) that is based on either least or greatest fixed points is not very expressive when it comes to its capabilities of accepting trees generated by λY-calculus terms. Its expressiveness is equal to boolean combination of Ω-blind automata. The precise definition of Ω-blind automata is beyond the scope of this thesis; it shall suffice to say that an Ω-blind automata will accept all infinite branches of a tree (and its negation will reject all infinite branches of a tree). Hence, a boolean combination of Ω-blind automata is insufficient to check liveness properties (like membership in an ω-regular language) along an infinite branch of the tree generated by a λY-calculus term.

In personal communication with the author of this thesis, Roland Meyer has conjectured that the result by Salvati and Walukiewicz can be used to show that ω-context-free inclusion games cannot be solved with an algorithm based on effective denotational semantics that simply translates the problem of computing the winner into computing the least solution to a system of equations. Recall that a context-free grammar can be seen as a higher-order recursion scheme. It should be possible to transform a game grammar into a term in the λY-calculus that generates a tree so that each possible play of the game corresponds to a branch of that tree. We leave making this correspondence formal for future work.

Walukiewicz's work also provides us with an explanation why this problem can be circumvented by constructing and solving a parity game instead of trying to read off the winner from the solution to the system of equations directly. He has shown in [Wal02] that the winning regions of parity games can be described by formulas in μ-calculus that contain an alternation of least and greatest fixed-point operators. Hence, solving a parity game based on the solution to the system of equations means computing a so-called *non-extremal* fixed point that is obtained by nesting least and greatest fixed-point operators. The expressiveness results by Salvati and Walukiewicz do not apply to semantics featuring non-extremal fixed point operators.

We think that if this part of the computation potentially cannot be avoided, then representing it via parity games, for which there is a plethora of research and numerous solvers like the ones implemented in the aforementioned tool Oink, is the most elegant way to deal with the issue.

18 Higher-order games

Contents

18.1 Higher-order games 376
18.2 A model template for deterministic schemes 378
18.3 Fixed-point semantics for higher-order games 389
18.4 Framework for exact fixed-point transfer 403
18.5 Solving higher-order inclusion games 417

We demonstrate the versatility of our technique for solving inclusion games based on effective denotational semantics by applying to it to games induced higher-order recursion schemes. The setting is similar to the one in the previous chapter: We consider a word-generating recursion scheme whose derivation processes are controlled by the players, and the goal of the game is membership resp. non-membership in a regular target language. The approach that we take, however, is vastly different.

After giving the formal definition of HORS games, we consider the more general case of the system of equations associated to a deterministic HORS. We design a model template that allows us to get a full model for interpreting such a system. To instantiate the template, it is sufficient to provide a domain for kind ground and the interpretations for the nonterminals. We apply this model template to the determinization of a game HORS and show that the associated least solution to the interpreted system of equations characterizes the winner of the game. This leaves us with the problem that we cannot compute this least solution since we have used a domain that does not satisfy the ascending chain condition.

Returning to a more general setting, we present a framework for exact fixed-point transfer. It allows us to transfer the properties of the least fixed point with respect to one model to the least fixed point with respect to another one. We then use this framework to go from the aforementioned infinite domain to a model with a finite domain. The latter allows us to compute the least solution while preserving the information about the winner of the game.

Publication

The chapter presents material that has been published in the form of the paper [HMM17] (resp. its full version [HMM17a]). Compared to the publication, the material has been improved. We will discuss both the improvements and the authors author's contributions to the publication in Chapter 20.

18.1 Higher-order games

We define inclusion games on arenas induced by word-generating higher-order recursion schemes. A *higher-order (regular) inclusion game* is of the shape (G, A), where G is a *game HORS*[1], a word-generating HORS whose nonterminals $N = N_\Box \uplus N_\bigcirc$ are partitioned into the nonterminals owned by each of the players. Recall that being word generating means that the terminals are of the shape $T = \Sigma \uplus \{\$: o\}$, consisting of a set Σ of terminals of kind $o \to o$ and the word-end marker. The second component A of the inclusion game is an NFA over Σ, specifying the regular target language.

A game HORS induces a game arena. The positions in the arena are the terms of the HORS of kind ground, i.e. well-kinded expressions of kind o that can be built using the nonterminals, terminals, and variables. We fix S, the initial nonterminal of kind o, to be the initial position of interest. The moves in the arena are defined by outermost-to-innermost (OI) derivation steps. Recall that this means replacing an outermost redex (a subexpression that starts with a nonterminal and has all parameters present so that it is of kind o) using one of the rules of the HORS.

The restriction to OI derivations in conjunction with the assumption that the HORS is word generating means that each term of kind o has a unique outermost redex. To be precise, such a term will either not contain a nonterminal, being of the shape $a_1(\dots a_n(\$))$, which we identify with the finite word $a_1 \dots a_n \in \Sigma^*$, or it will be of the shape

$$a_1(a_2(\dots a_n(F\ t_1 \dots t_k)\dots))\,,$$

where F is the outermost nonterminal, uniquely determining the outermost redex. To see that this is true, note that the initial term S satisfies this property and that applying a replacement to the outermost redex in a term of this shape will result in another term of this shape. Here, it is crucial that we cannot obtain terms of the shape $a\ (F \dots)\ (G \dots)$, i.e. a terminal with two redexes as parameters, in a word-generating scheme.

The above observation allows us to assign to each term an owner based on its outermost nonterminal: Player ☆ owns the term $a_1(a_2(\dots a_n(F\ t_1 \dots t_k)\dots))$ if $F \in N_\bigstar$. For terms not containing nonterminals, the ownership does not matter.

With the definition of the game arena completed, we observe that a play from S is an OI-derivation process of the HORS in which each player selects the replacement steps for the nonterminals owned by her. This is similar to the definition of context-free games in the previous section.

The winning condition is (non-)membership in the regular target language defined by automaton A. A play that ends in a terminal word $a_1 \dots a_n$ not contained in $\mathcal{L}(A)$ is won by the existential player. All other plays, including all infinite ones, are won by the universal player.

[1] Game HORSes should not be confused with horse games, e.g. polo.

Note that one could also give a definition that fits our framework for games from Chapter 15 by basing it on the growth of the terminal prefix $a_1 \dots a_n$ of a term $a_1(a_2(\dots a_n(F\ t_1 \dots t_k)\dots))$.

The rest of this chapter is dedicated to solving higher-order inclusion games by computing the player that has a winning strategy from the initial position. The difficulty lies in having to compute information about a game on an infinite arena based on the finite syntactical representation of that arena – the HORS and the automaton for the target language.

Related work

Similar to context-free inclusion games, HORS games can be approached from various angles. Instead of considering games on arenas induces by HORSes, one can consider games induced by higher-order pushdown automata [Cac03; BM04; KNUW05; HMOS08; HO07; HO09; BCHS12], which are equivalent to (a subclass of) HORSes [Dam82; DG86; KNU02]. As in Walukiewicz's work on the μ-calculus and context-free games [Wal01], the game aspect can also be present in the form of the specification that is used in a verification problem that takes a HORS as input. For example, the decidability of Monadic Second Order Logic (MSO) resp. the modal μ-calculus over trees generated by recursion schemes can be seen as such a result [KNU02; Cau02; Ong06], especially since the latter result by Ong explicitly uses game semantics. Kobayashi [Kob09; KO09] has pioneered an approach to μ-calculus model checking on trees generated by HORSes that is based on so-called intersection types. The typing algorithm that solves the problem then amounts to computing a least-fixed point, which is similar to our approach using effective denotational semantics.

While our definition of HORS inclusion games with a regular target language seems to be novel, the decidability result that we strive to obtain in the rest of this chapter is not particularly surprising. It would have been possible to obtain the decidability of HORS inclusion games by reducing them to the setting considered in one of the aforementioned papers. Rather than the result itself, the interesting aspect of our study is the way in which we obtain that result. It demonstrates the versatility of our approach based on effective denotational semantics. Furthermore, the technical development will yield results of independent interest, like the framework for exact fixed-point transfer that we are going to present in Section 18.4, as a byproduct.

18.2 A model template for deterministic schemes

Our goal is to solve HORS games using effective denotational semantics. In order to do so, we will design a system of equations, find a suitable model to interpret this system, and then argue that the least solution to the interpreted system allows us to determine the winner of the game. For now, however, we will not focus on HORS games. Instead, we develop a general framework for interpreting the systems of equations associated to HORSes. This framework is not only of independent interest, we will also instantiate it multiple times when we solve HORS games later.

Associating equations to a deterministic HORS

We start with discussing on how to associate a system of equations to a HORS. We will provide this construction for an arbitrary deterministic HORS.

Let $G = (V, N, T, R, S)$ be a deterministic HORS, i.e. each nonterminal F has a unique rule $F \to t$. We design a system of equations representing G. As in the case of context-free grammars, it has one variable for each nonterminal. In particular, the variables of the HORS do not become variables of the system of equations. Each rule $F \to t$ of the HORS yield one equation $F = t$ of the system. To this end, we need to introduce function symbols that allow us to see the HORS term t as a term of the system of equations, where the syntactic fragments that build up the HORS term t yield the syntactic fragments that build the term t of the system of equations.

Following the definition of HORS terms in Section 5.3, each HORS term is obtained by composing three types of constructs: (1) HORS variables, terminals, and nonterminals, (2) function application, and (3) lambda abstraction. As already mentioned, the HORS nonterminals are simply the variables of the system of equations. For everything else, we introduce appropriate function symbols: We see terminals and HORS variables as constants, function symbols with arity zero. Function application is a binary function symbol: If f and t are terms in the system of equation, then $f\ t$ is also a term. As usual for function application in the context of HORSes, we use infix notation and omit brackets. For each variable x, the lambda abstraction λx is a unary function symbol, so if t is a valid term in the system of equations, then so is $\lambda x.t$

Choosing HORS terminals to have arity zero instead of assigning them their arity as specify by the kind might seem peculiar. The reason here is that a terminal in a HORS term does not always have all its parameter present. For example, a terminal $a\colon o \to o$ may occur in a term like $F\ a$ without any parameter. Therefore, it makes sense to let terminals and variables be constants and encode function application as a separate function symbol.

Introducing all of these function symbols allows us to see the set of production rules of the HORS as a system of equations. Note that an (interpreted) system of equations according to our definition from Section 16.1 is essentially a first-order construction, as it just deals with values and functions that transform these values. By associating a system of equations to a HORS,

we have essentially turned a higher-order object into a first-order one. In order to recover the higher-order aspect, we will have to interpret the system of equations accordingly.

The model template

Later, we will consider several interpretations of the same system of equations associated to the determinization of a game HORS. Each interpretation is of a similar shape. In order to make this formal, we define a *model template* for the system of equations associated to a deterministic HORS. This model template can be instantiated using a very restricted amount of information, yielding a full model, i.e. a domain and interpretations for all function symbols.

Formally, the model template is $\mathcal{M}_- = (\mathcal{D}_-, \mathcal{I}_-)$ consisting of a domain $\mathcal{D}_-$ and the interpretation $\mathcal{I}_-$ of the function symbols. Instantiating the template requires a domain $\mathcal{D}_o$ for elements of kind ground and an interpretation for all terminals. The template then provides the rest of the model: The domain for all other kinds and the interpretation of the variables, function application, and lambda abstraction.

The domain

We start by explaining the domain. Assume that a CPPO $\mathcal{D}_o$ has been chosen. For any other kind $\kappa_1 \to \kappa_2$, the domain $\mathcal{D}_{\kappa_1 \to \kappa_2}$ is then defined to be the set of join-continuous functions from $\mathcal{D}_{\kappa_1}$ to $\mathcal{D}_{\kappa_2}$. We will denote this domain by $\mathcal{D}_{\kappa_1} \to_\sqcup \mathcal{D}_{\kappa_2}$ in the following. Note that any such function is monotonic since join-continuity implies monotonicity. (Here, we generalize the definitions of join-continuity and monotonicity from functions with signature $D \to D$ to arbitrary functions where source and target set do not have to coincide.)

It is well-known that if D, D' are CPPOs, then the set $D \to_\sqcup D'$ of join-continuous functions from D to D' is also a CPPO. The ordering is component-wise, i.e. $f \leqslant g$ iff $f(x) \leqslant g(x)$ for all x. The function $f\colon D \to D'$ that maps all elements of D to the bottom-element of D' is the bottom element of the function domain. The join of an ascending chain of functions is the function that takes a value and returns the join of the chain of function values,

$$\Big(\bigsqcup_{i\in\mathbb{N}} f_i\Big)(d) = \bigsqcup_{i\in\mathbb{N}} (f_i(d)) \,.$$

It is easy to verify that this is indeed the join of the chain. Let us briefly argue that $\bigsqcup_{i\in\mathbb{N}} f_i$ is join-continuous. Consider an ascending chain of values $(d_j)_{j\in\mathbb{N}}$. We need to prove that $(\bigsqcup_{i\in\mathbb{N}} f_i)(\bigsqcup_{j\in\mathbb{N}} d_j) = \bigsqcup_{j\in\mathbb{N}}((\bigsqcup_{i\in\mathbb{N}} f_i)(d_j))$.

By the definition of the join, the left-hand side of the equality is $\bigsqcup_{i\in\mathbb{N}}(f_i(\bigsqcup_{j\in\mathbb{N}} d_j)) = \bigsqcup_{i\in\mathbb{N}}\bigsqcup_{j\in\mathbb{N}} f_i(d_j)$, using that each f_i is join-continuous. We now observe that the two joins commute and use the definition of $\bigsqcup_{i\in\mathbb{N}} f_i$ to get the right-hand side of the desired equality.

We have obtained a CPPO $\mathcal{D}_\kappa$ for every kind $\kappa \in K$, and we define $\mathcal{D} = \bigcup_{\kappa\in K} \mathcal{D}_\kappa$ to be their union. However, the domains of the shape $\mathcal{D}_\kappa$ are insufficient to evaluate terms of the kind κ

that contain free variables. The right-hand sides of all rules of the HORS are guaranteed to be closed in that each occurrence of a variable is bound by a preceding lambda abstraction, but when applying structural induction, we will encounter subterms that contain free variables. In order to handle free variables, the elements of our domain are functions that expect a *valuation*, a partial function that assigns a value to each HORS variable that is free in the term of interest. Hence, the domain $\mathcal{D}_-$ that is used by our template is actually

$$\mathcal{D}_- = (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}\,,$$

the set of join-continuous functions from valuations to $\mathcal{D}$. Here $V \to_p \mathcal{D}$, denotes the set of partial functions from the variables to $\mathcal{D}$, functions that may be undefined for some variables.

In order for the notion of join-continuity to make sense here, we need to see $V \to_p \mathcal{D}$ itself as a CPPO. In order for this to succeed, we will need to restrict $(V \to_p \mathcal{D})$ in the following way. For each variable x of kind κ, we assume that a valuation either is undefined or it assigns a value from $\mathcal{D}_\kappa$. Whenever we write $V \to_p \mathcal{D}$ in the following, we implicitly assume that we only consider valuations that respect the kinds of the variables. With this restriction in place, we can define an order on valuations. We define $v \leqslant v'$ by requiring that for each HORS variable x, either $v(x)$ is undefined or both $v(x)$ and $v'(x)$ are defined and $v(x) \leqslant v'(x)$ holds (using the order $\leqslant$ on $\mathcal{D}_\kappa$, where κ is the kind of x). The least element of this CPPO is the function that is undefined everywhere. The join of an ascending chain of valuations $(v_i)_{i\in\mathbb{N}}$ is the valuation $\bigsqcup_{i\in\mathbb{N}} v_i$ with $(\bigsqcup_{i\in\mathbb{N}} v_i)(x) = \bigsqcup_{i\in\mathbb{N}}(v_i(x))$. For the latter expression to make sense, we extend the order on $\mathcal{D}_\kappa$ to an order on $\mathcal{D}_\kappa$ together with the undefined value by seeing undefined as the new least element. In words, $\bigsqcup_{i\in\mathbb{N}} v_i$ is the valuation that is undefined for a variable x if all v_i are undefined for x and that returns the join of the defined values $v_i(x)$ otherwise.

We summarize our definition of the domain.

18.2.1 Definition
Let $\mathcal{D}_o$ be a CPPO. We recursively define $\mathcal{D}_{\kappa_1\to\kappa_2} = \mathcal{D}_{\kappa_1} \to_\sqcup \mathcal{D}_{\kappa_2}$ for all kinds κ_1, κ_2 to be the CPPO of join-continuous functions from $\mathcal{D}_{\kappa_1}$ to $\mathcal{D}_{\kappa_2}$. Let $\mathcal{D} = \bigcup_{\kappa\in K} \mathcal{D}_\kappa$.

We define $\mathcal{D}_- = (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}$ to be the set of join-continuous functions from valuations $v\colon V \to_p \mathcal{D}$ to domain elements $d \in \mathcal{D}$.

In order to enable Kleene iteration later, we need a domain that is a CPPO. Formally, we would need to argue that $\mathcal{D}_- = (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}$ is a CPPO. However, we will make sure that we interpret each term t of kind κ as a value from $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_\kappa$. In particular, we will never need to compare a value from $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_\kappa$ to a value from $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_{\kappa'}$ for distinct kinds $\kappa \neq \kappa'$. Hence, it will suffice to make sure that $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_\kappa$ is a CPPO for each kind κ.

18.2.2 Lemma
For each kind κ, $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_\kappa$ is a CPPO.

Proof:
The set $V \to_p \mathcal{D}$ of valuations is a CPPO under the assumption that we only consider valuations that respect the kinds of the variables. The set $\mathcal{D}_\kappa$ is either $\mathcal{D}_o$ which is a CPPO by assumption or it is a CPPO as the set of join-continuous functions from one CPPO to another. Hence, $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_\kappa$ is the CPPO of join-continuous functions from one CPPO to another.

Note that the least element of $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_\kappa$ is the function that ignores the given valuation and returns the least element of $\mathcal{D}_\kappa$. The join of a chain of functions is the function that returns the join of the chain of function values. ■

The interpretations

After fixing the domain $\mathcal{D}_-$, it remains to provide interpretations for all function symbols. We assume that for each terminal a of kind κ, an interpretation, $a^{\mathcal{I}_-} \in \mathcal{D}_\kappa$ has been provided. We can see $a^{\mathcal{I}_-}$ as a function with signature $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}$ that ignores the provided valuation and always returns the same element. Indeed, in the term a itself, no variable occurs freely.

The template should provide the interpretations for all other types of function symbols: variables, function application, and lambda abstraction. A HORS variable x is interpreted as $x^{\mathcal{I}_-}\colon (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}$, the function that takes a valuation v and returns $v(x)$. This function is indeed join-continuous by the definition of the join on $V \to_p \mathcal{D}$. Note that if the valuation respects the kinds of the variables, $v(x)$ will be an element of $\mathcal{D}_\kappa$, where κ is the kind of variable x.

The interpretation of function application is a function with signature

$$\mathcal{D}_-{}^2 \to \mathcal{D}_- \;=\; \big((V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}\big)^2 \to (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}\,.$$

It expects three parameters: two values $f^{\mathcal{I}}, t^{\mathcal{I}} \in \mathcal{D}_-$ and a valuation v. We define the interpretation $(f^{\mathcal{I}}\ t^{\mathcal{I}})^{\mathcal{I}_-} v$ of function application to be $(f^{\mathcal{I}})v\,(t^{\mathcal{I}}v)$. Intuitively, $f^{\mathcal{I}}$ and $t^{\mathcal{I}}$ are interpretations of terms and v specifies the values of variables that are free in f or t.

18.2.3 Lemma
The interpretation of function application has signature $\mathcal{D}_-{}^2 \to \mathcal{D}_-$: For any $f^{\mathcal{I}}, t^{\mathcal{I}} \in \mathcal{D}_-$, the result of function application is the value $(f^{\mathcal{I}}\ t^{\mathcal{I}})^{\mathcal{I}_-} \in \mathcal{D}_-$.

If $v\colon V \to_p \mathcal{D}$ respects the kinds of the variables, $f^{\mathcal{I}}v \in \mathcal{D}_{\kappa_1 \to \kappa_2}$, and $t^{\mathcal{I}}v \in \mathcal{D}_{\kappa_1}$, then $(f^{\mathcal{I}}\ t^{\mathcal{I}})^{\mathcal{I}_-} v = (f^{\mathcal{I}}v)\,(t^{\mathcal{I}}v) \in \mathcal{D}_{\kappa_2}$.

Proof:
We prove the second part of the statement first. We have that $f^{\mathcal{I}} v$ is a value from $\mathcal{D}_{\kappa_1 \to \kappa_2}$, which is a join-continuous function from $\mathcal{D}_{\kappa_1}$ to $\mathcal{D}_{\kappa_2}$. Similarly, $t^{\mathcal{I}} v$ is in $\mathcal{D}_{\kappa_1}$. Hence, $(f^{\mathcal{I}_-} v)\,(t^{\mathcal{I}_-} v)$ is in $\mathcal{D}_{\kappa_2}$, and κ_2 is indeed the kind of $f\ t$.

To show that the interpretation of function application has the required signature, we need to argue that $(f^{\mathcal{I}}\ t^{\mathcal{I}})^{\mathcal{I}_-}$ is join-continuous on the level of valuations. Consider an ascending chain of valuations $(v_i)_{i\in\mathbb{N}}$. We need to prove

$$(f^{\mathcal{I}}\ t^{\mathcal{I}})^{\mathcal{I}_-}\Big(\bigsqcup_{i\in\mathbb{N}} v_i\Big) = \bigsqcup_{i\in\mathbb{N}} (f^{\mathcal{I}}\ t^{\mathcal{I}})^{\mathcal{I}_-}(v_i)\,.$$

By definition, we have $(f^{\mathcal{I}}\ t^{\mathcal{I}})^{\mathcal{I}_-}(\bigsqcup_{i\in\mathbb{N}} v_i) = (f^{\mathcal{I}}(\bigsqcup_{i\in\mathbb{N}} v_i))(t^{\mathcal{I}}(\bigsqcup_{i\in\mathbb{N}} v_i))$. By assumption, $f^{\mathcal{I}}$ and $t^{\mathcal{I}}$ are elements of $\mathcal{D}_-$, so they are join-continuous. We can rewrite the right-hand side of the equality as $(\bigsqcup_{i\in\mathbb{N}} f^{\mathcal{I}} v_i)(\bigsqcup_{i\in\mathbb{N}} t^{\mathcal{I}} v_i)$. As mentioned before, each $f^{\mathcal{I}} v_i$ is a join-continuous function from $\mathcal{D}_{\kappa_1\to\kappa_2}$. The join on $\mathcal{D}_{\kappa_1\to\kappa_2}$ is defined to be simply the function that returns the join of the function values, so we can rewrite this expression as $\bigsqcup_{i\in\mathbb{N}} ((f^{\mathcal{I}} v_i)(\bigsqcup_{i\in\mathbb{N}} t^{\mathcal{I}} v_i))$. Because each $f^{\mathcal{I}} v_i$ is join-continuous, we finally obtain $\bigsqcup_{i\in\mathbb{N}} \bigsqcup_{i\in\mathbb{N}} (f^{\mathcal{I}} v_i)(t^{\mathcal{I}} v_i)$. Omitting one of the joins does not change the result and applying the definition of the interpretation of function application proves the desired equality. ■

To complete the specification of the model, we have to define the interpretation of lambda abstraction. For each variable x, the interpretation $(\lambda x)^{\mathcal{I}_-}$ is a function with signature

$$\mathcal{D}_- \to \mathcal{D}_- \;=\; \big((V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}\big) \to (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}\,.$$

It takes as parameters a value $t^{\mathcal{I}} : (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}$, which intuitively is the interpretation of a term, and a valuation v. It returns the function $d \mapsto t^{\mathcal{I}}(v[x \mapsto d])$ that expects a value d for x and evaluates $t^{\mathcal{I}}$ at v with the value for x replaced by d.

18.2.4 Lemma
The interpretation of lambda abstraction has signature $\mathcal{D}_- \to \mathcal{D}_-$: For any variable x and any $t^{\mathcal{I}} \in \mathcal{D}_-$, the result of lambda abstraction is $(\lambda x)^{\mathcal{I}_-} t^{\mathcal{I}} \in \mathcal{D}_-$.

If $v : V \to_p \mathcal{D}$ respects the kinds of the variables, x has kind κ_1 and $t^{\mathcal{I}} v \in \mathcal{D}_{\kappa_2}$, then $(\lambda x)^{\mathcal{I}_-} t^{\mathcal{I}} v \in \mathcal{D}_{\kappa_1\to\kappa_2}$.

Proof:
We prove the second part of the statement first. Assume that variable x has kind κ_1, and that term t has kind κ_2. We have that $((\lambda x)^{\mathcal{I}_-} t^{\mathcal{I}})v$ is a function that takes $d \in \mathcal{D}_{\kappa_1}$ and returns $t^{\mathcal{I}}(v[x \mapsto d]) \in \mathcal{D}_{\kappa_2}$. Hence, it is indeed a function from $\mathcal{D}_{\kappa_1}$ to $\mathcal{D}_{\kappa_2}$. This matches the kind $\kappa_1 \to \kappa_2$ of the term $\lambda x.t$. To conclude that the function $d \mapsto t^{\mathcal{I}}(v[x \mapsto d])$ is an element of $\mathcal{D}_{\kappa_1\to\kappa_2}$, we need to argue that it is join-continuous. Let $(d_i)_{i\in\mathbb{N}}$ be an ascending chain in $\mathcal{D}_{\kappa_1}$.

We first observe that the valuation $\nu[x \mapsto \bigsqcup_{i\in\mathbb{N}} d_i]$ equals the join $\bigsqcup_{i\in\mathbb{N}} \nu[x \mapsto d_i]$ by the definition of the join on $V \to_p \mathcal{D}$. We then note that $t^{\mathcal{I}} : (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}$ is join-continuous, and obtain $t^{\mathcal{I}}(\nu[x \mapsto \bigsqcup_{i\in\mathbb{N}} d_i]) = \bigsqcup_{i\in\mathbb{N}} t^{\mathcal{I}}(\nu[x \mapsto d_i])$ as desired.

To prove the first part of the statement, we need to argue that $(\lambda x)^{\mathcal{I}_-} t^{\mathcal{I}}$ is join-continuous on the level of valuations. Let $(\nu_i)_{i\in\mathbb{N}}$ be an ascending chain of valuations. We need to show

$$(\lambda x)^{\mathcal{I}_-}(t^{\mathcal{I}})(\bigsqcup_{i\in\mathbb{N}} \nu_i) = \bigsqcup_{i\in\mathbb{N}} (\lambda x)^{\mathcal{I}_-}(t^{\mathcal{I}})\nu_i \,.$$

The left-hand side of the equality is the function $d \mapsto t^{\mathcal{I}}((\bigsqcup_{i\in\mathbb{N}} \nu_i)[x \mapsto d])$. The valuation $(\bigsqcup_{i\in\mathbb{N}} \nu_i)[x \mapsto d]$ equals $\bigsqcup_{i\in\mathbb{N}}(\nu_i[x \mapsto d])$ using the definition of the join on $V \to_p \mathcal{D}$. Since $t^{\mathcal{I}} : (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}$ is join-continuous, the function equals $d \mapsto \bigsqcup_{i\in\mathbb{N}} t^{\mathcal{I}}(\nu_i[x \mapsto d])$. By the definition of the join on the level of functions, this function is the join of the functions $d \mapsto t^{\mathcal{I}}(\nu_i[x \mapsto d])$, which by the definition of the interpretation of lambda abstraction is the right-hand side of the equality we wanted to show. ■

Join-continuity

We have fully specified our $\mathcal{M}_- = (\mathcal{D}_-, \mathcal{I}_-)$ by providing a domain and interpretations for all function symbols. In order to prove that the model satisfies the requirements for Kleene iteration as presented in Section 16.1, we need to argue that the interpretation of all function symbols are join-continuous.

When we have considered join-continuity in the proofs of the previous lemmas, we were discussion join-continuity on the level of valuations. We were merely arguing that for each term t, its interpretation is a value of $(V \to_p \mathcal{D}) \to \mathcal{D}$, i.e. a join-continuous function that expects a valuation. To satisfy the requirements of Kleene's theorem, we need to argue that the functions are also join-continuous with respect to their arguments from $\mathcal{D}_-$.

Remark
In the previous chapter, it was sufficient to prove that the interpretations are monotonic by relying on finiteness of the domain and Remark 16.1.4. In this chapter, we are interested in instantiating the model template for domains that are not necessarily finite and that do not necessarily satisfy the ACC.

For constants, function symbols without arguments, there is nothing to do. This applies to the interpretations of terminals and HORS variables. It remains to consider the unary interpretation of lambda abstraction and the binary interpretation of function application.

18.2.5 Lemma
The interpretation of lambda abstraction is a join-continuous function with signature $\mathcal{D}_- \to \mathcal{D}_-$.

Proof:
Let us consider a lambda abstraction λx where x is of kind κ_1. Let $(t_i)_{i\in\mathbb{N}}$ be an ascending chain of values in $(V \to_p \mathcal{D}) \to \mathcal{D}_{\kappa_2}$. We have to show $(\lambda x)^{\mathcal{I}_-}(\bigsqcup_{i\in\mathbb{N}} t_i) = \bigsqcup_{i\in\mathbb{N}} (\lambda x)^{\mathcal{I}_-}(t_i)$. The expression on the left-hand side is equal to

$$(v, d) \mapsto \big(\bigsqcup_{i\in\mathbb{N}} t_i\big)(v[x \mapsto d]) \,,$$

a function that takes a valuation v and a value d for x and evaluates $\bigsqcup_{i\in\mathbb{N}} t_i$ at $v[x \mapsto d]$. Similarly, the expression on the right-hand side equals,

$$\bigsqcup_{i\in\mathbb{N}} \big((v, d) \mapsto t_i(v[x \mapsto d])\big) \,.$$

Using the definitions of the join on $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}$ and the join on $\mathcal{D}_{\kappa_1\to\kappa_2}$, this equals

$$(v, d) \mapsto \bigsqcup_{i\in\mathbb{N}} (t_i(v[x \mapsto d])) \,.$$

We use the definition of the join on $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}$ again to establish

$$\big(\bigsqcup_{i\in\mathbb{N}} t_i\big)(v[x \mapsto d]) = \bigsqcup_{i\in\mathbb{N}} (t_i(v[x \mapsto d])) \,,$$

which yields the desired equality. ■

We proceed similarly for function application, which is a binary function. We need to prove join-continuity in both parameters, which we will do simultaneously.

18.2.6 Lemma
The interpretation of function application is a join-continuous function with signature ${\mathcal{D}_-}^2 \to \mathcal{D}_-$.

Proof:
Let $(f_i)_{i\in\mathbb{N}}$ be an ascending chain in $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_{\kappa_1\to\kappa_2}$ and let $(t_j)_{j\in\mathbb{N}}$ be an ascending chain in $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_{\kappa_1}$. We need to prove the following equality

$$\Big(\big(\bigsqcup_{i\in\mathbb{N}} f_i\big)\big(\bigsqcup_{j\in\mathbb{N}} t_j\big)\Big)^{\mathcal{I}_-} = \bigsqcup_{i\in\mathbb{N}} \bigsqcup_{j\in\mathbb{N}} (f_i\ e_j)^{\mathcal{I}_-} \,.$$

Both expressions are functions that expect a valuation $v\colon V \to_p \mathcal{D}$ and return a value from $\mathcal{D}_{\kappa_2}$. Evaluating the left-hand side at some valuation v yields

$$\begin{aligned}\Big(\big(\bigsqcup_{i\in\mathbb{N}} f_i\big)\big(\bigsqcup_{j\in\mathbb{N}} t_j\big)\Big)^{\mathcal{I}_-} v &= \Big(\big(\bigsqcup_{i\in\mathbb{N}} f_i\big)\, v\Big)\Big(\big(\bigsqcup_{j\in\mathbb{N}} t_j\big)\, v\Big) &&= \bigsqcup_{i\in\mathbb{N}}(f_i\, v) \bigsqcup_{j\in\mathbb{N}}(t_j\, v)\\ &= \bigsqcup_{i\in\mathbb{N}}\Big((f_i\, v) \bigsqcup_{j\in\mathbb{N}}(t_j\, v)\Big) &&= \bigsqcup_{i\in\mathbb{N}}\bigsqcup_{j\in\mathbb{N}}\Big((f_i\, v)\,(t_j\, v)\Big),\end{aligned}$$

using the definition of the interpretation of function application, the definition of the join on $(V \to_p \mathcal{D}) \to \mathcal{D}_\kappa$ for all kinds κ, the definition of the join on $\mathcal{D}_{\kappa_1\to\kappa_2}$ and the fact that each $f_i\, v$ is a join-continuous function. Evaluating the right-hand side of the above expression at v gives us

$$\big(\bigsqcup_{i\in\mathbb{N}}\bigsqcup_{j\in\mathbb{N}}(f_i\, e_j)^{\mathcal{I}_-}\big)\, v \;=\; \bigsqcup_{i\in\mathbb{N}}\bigsqcup_{j\in\mathbb{N}}\big((f_i\, e_j)^{\mathcal{I}_-}\, v\big) \;=\; \bigsqcup_{i\in\mathbb{N}}\bigsqcup_{j\in\mathbb{N}}\Big((f_i\, v)\,(t_j\, v)\Big),$$

using the definition of the join on $(V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_{\kappa_2}$ and the definition of the interpretation of function application. The desired equality holds. ■

The semantics of terms

We have completed the definition of the model template $\mathcal{M}_- = (\mathcal{D}_-, \mathcal{I}_-)$. Each term of the HORS that does not contain nonterminals can be interpreted as a value from $\mathcal{D}$. Furthermore, if the kind of term t is κ and v is a valuation that respects the kinds of the variables, then we get that the interpretation of t evaluated at v is a value from $\mathcal{D}_\kappa$.

It remains to consider terms that contain variables of the system of equations, i.e. nonterminals of the HORS. The theory from Section 16.1 fixes a semantics $\mathcal{M}_-[\![t]\!]$ for each term t. In particular, it does so for the terms that occur as the right-hand sides of the rules of the deterministic HORS, which are the right-hand sides of the equations in our system. Let us consider what the signature of $\mathcal{M}[\![t]\!]$ is. Formally, it is a function that expects an assignment of the variables of the system of equations and then evaluates t at this assignment. The variables of the system of equations are the nonterminals of the HORS. We get the signature

$$\mathcal{M}_-[\![t]\!]\colon (N \to \mathcal{D}_-) \to_\sqcup \mathcal{D}_- .$$

Here, we have used the fact that the interpretation of each function symbol is join-continuous (Lemmas 18.2.5 and 18.2.6) and join-continuity is preserved under composition. Hence, the interpretation of each term is also join-continuous.

By substituting $\mathcal{D}_-$ using its definition, this signature equals

$$\big(N \to \big((V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}\big)\big) \to_\sqcup (V \to_p \mathcal{D}) \to_\sqcup \mathcal{D} .$$

Our first observation is that to evaluate the variables (nonterminals), a valuation of the HORS variables is not needed. A term that only consists of a nonterminal does not contain a free variable. Hence, we can simplify the expression to

$$(N \to \mathcal{D}) \to_{\sqcup} (V \to_p \mathcal{D}) \to_{\sqcup} \mathcal{D} .$$

Intuitively, to evaluate a term in order to obtain an element of $\mathcal{D}$, we need an assignment of values to the nonterminals and a valuation of the HORS variables. We merge the assignment and the valuation into a single valuation, a partial function with signature $N \uplus V \to_p \mathcal{D}$. When evaluating a term t, this valuation should be defined for all nonterminals and all HORS variables that are free in t. Furthermore, we will implicitly assume that we only consider valuations ν that respect the kinds (of both HORS variables and nonterminals), i.e. if F resp. x is of kind κ, then $\nu(F)$ resp. $\nu(x)$ is a value from $\mathcal{D}_\kappa$. Under this assumption, a term of kind κ should evaluate to a value from $\mathcal{D}_\kappa$.

Altogether, we have proven the following.

18.2.7 Lemma
The semantics of a HORS term t is a join-continuous function

$$\mathcal{M}_-[\![t]\!] \colon (N \uplus V \to_p \mathcal{D}) \to_{\sqcup} \mathcal{D} .$$

If ν is a valuation that respects the kinds and t is of kind κ, then $\mathcal{M}_-[\![t]\!]\nu \in \mathcal{D}_\kappa$.

Applying Kleene's theorem

Consider the interpretation of the system of equations under a model that results from instantiating the model template. The least solution to the interpreted system is the least fixed point of the function that is obtained by combining the interpretations of the right-hand sides of the equations, i.e. the rules of the deterministic HORS. Because the model satisfies the requirement of Kleene's theorem, the existence of this least fixed point is guaranteed.

18.2.8 Proposition
The system of equations associated to a deterministic HORS interpreted under a model that results from instantiating the model template has a least solution.

Overview: Model template

Domain

Input: CPPO $\mathcal{D}_o$ for kind ground (partially ordered; least element exists; ascending chains have a join).

Recursively define $\mathcal{D}_{\kappa_1 \to \kappa_2} = \mathcal{D}_{\kappa_1} \to_\sqcup \mathcal{D}_{\kappa_2}$, the join-continuous functions from $\mathcal{D}_{\kappa_1}$ to $\mathcal{D}_{\kappa_2}$. This is a CPPO with the component-wise order; the least element is the function that always returns the least element; the join of a chain is the function that, given an input, returns the join of the chain of function values.

We define the domain $\mathcal{D}_- = (V \to_p \mathcal{D}) \to \mathcal{D}$ with $\mathcal{D} = \bigcup_\kappa \mathcal{D}_\kappa$. For each kind κ, $(V \to_p \mathcal{D}) \to \mathcal{D}_\kappa$ is a CPPO. Elements of this domain are functions that take a valuation, an assignment of a value from $\mathcal{D}_\kappa$ to each variable of kind κ.

Interpretations

Input: For each terminal $s\colon\kappa$ of a deterministic HORS G an interpretation $s^{\mathcal{I}_-} \in \mathcal{D}_\kappa$.

Specify $\mathcal{I}_-$ by defining the interpretation of variables ($\mathcal{M}_-[\![x]\!]\nu = \nu(x)$), function application ($\mathcal{M}_-[\![t_1\ t_2]\!]\nu = \mathcal{M}_-[\![t_1]\!]\nu\ \mathcal{M}_-[\![t_2]\!]\nu$), and lambda abstraction ($\mathcal{M}_-[\![\lambda x.t]\!]\nu : d \mapsto \mathcal{M}_-[\![t]\!](\nu[x \mapsto d])$). Altogether, we get a model $\mathcal{M}_- = (\mathcal{D}_-, \mathcal{I}_-)$.

Semantics

We obtain for each term t of kind κ the semantics

$$\mathcal{M}_-[\![t]\!]\colon (N \uplus V \to_p \mathcal{D}) \to_\sqcup \mathcal{D}_\kappa\ ,$$

a join-continuous function that expects a valuation of the nonterminals and the HORS variables that are free in t.

Kleene iteration

For each $i \in \mathbb{N}$, inductively define the valuation sol^i.

For $i = 0$ and each nonterminal F of kind κ,

$\mathrm{sol}^i(F) = \bot_\kappa$, the least element of $\mathcal{D}_\kappa$.

For $i + 1$ and a nonterminal F whose unique rule in the HORS is $F \to t$,

$\mathrm{sol}^{i+1}(F) = \mathcal{M}_-[\![t]\!]\mathrm{sol}^i$, the right-hand side for F evaluated at sol^i.

The system of equations associated to G interpreted with respect to $\mathcal{M}_- = (\mathcal{D}_-, \mathcal{I}_-)$ has a least solution, namely

$$\mathrm{sol} = \bigsqcup_{i \in \mathbb{N}} \mathrm{sol}^i\ .$$

Figure 18.2.a: Overview: Model template.

Proof:
We may design a function

$$rhs\colon (N \to \mathcal{D}) \to_{\sqcup} (N \to \mathcal{D})$$

with $rhs(v, F) = \mathcal{M}_{-}[\![t]\!]v$ for each nonterminal F, where $F \to t$ is the unique rule of the HORS for F. Note that since each such t does not contain any free variables, we can omit the valuation for the HORS variables here. We have argued that for each term t, $\mathcal{M}_{-}[\![t]\!]$ is a join-continuous function on a CPPO. Hence, also *rhs* is a join-continuous function on a CPPO. By Kleene's theorem, Theorem 16.1.3, the least fixed point of *rhs* is the join

$$\mathrm{sol} = \bigsqcup_{i \in \mathbb{N}} \mathrm{sol}^i ,$$

where the i^{th} approximant $\mathrm{sol}^i = rhs^i(\bot)$ is obtained by the i-fold application of *rhs* to the least element of $N \to \mathcal{D}$. This least element is the function that assigns to each nonterminal F of kind κ the least element of $\mathcal{D}_\kappa$. Note that the definition of sol^i implies $\mathrm{sol}^{i+1}(F) = \mathcal{M}_{-}[\![t]\!]\mathrm{sol}^i$ for each nonterminal F whose unique rule is $F \to t$.

Using the theory from Section 16.1, sol is the least solution to the interpreted system of equations. ■

Note that the proposition guarantees the existence of the least solution as the join of an infinite ascending chain. Unlike in the previous chapter, our domain will not satisfy the ascending chain condition in general. Hence, we do not necessarily get that the solution equals sol^i for some fixed i. Kleene iteration does not allow us to compute the fixed point in finite time. When we want to apply our theory to solve HORS games, this is a challenge that we will have to overcome.

We conclude the section with an *overview* in the form of Figure 18.2.a. It specifies what a user of the template has to provide to instantiate it and what the result of the instantiation is.

Remark
The models that result from instantiating our model template are similar to what Salvati and Walukiewicz [SW15a] call *Scott models* for λY-calculus.

18.3 Fixed-point semantics for higher-order games

We come back to the task of solving higher-order inclusion games. We want to take an approach based on effective denotational semantics. To this end, we instantiate the model template that we have introduced in the previous section. The result is the solution to a system of equations from which we can read off the winner of the game. However, Proposition 18.2.8 will only guarantee the existence of the solution. We will not be able to compute it in finite. This is a problem that we will tackle later.

Determinizing HORSes

The first challenge that we have to overcome is that the theory from Section 18.2 only applies to deterministic HORSes. A game HORS is inherently nondeterministic. Hence, the system of equations that we consider will not represent the game HORS of interest, but rather its determinization. We have briefly mentioned that HORSes can be determinized in Section 5.3; it is time to make this explicit.

Let $F \to \lambda x_1 \dots \lambda x_m.e_1, \dots, F \to \lambda x_1 \dots \lambda x_m.e_k$ be an exhaustive list of all rules for nonterminal F in a HORS with $k > 1$. Note that we assume that the e_i are λ-free terms of kind o and that each right-hand side uses the same sequence of variables. The latter property can be enforced by an appropriate renaming of the variables. In order to determinize the HORS, we replace this collection of rules by the single rule

$$F \to \lambda x_1 \dots \lambda x_m.\mathrm{branch}_F\ e_1 \dots e_k\ .$$

Here, $\mathrm{branch}_F : \underbrace{o \to o \to \dots \to o}_{k \text{ times}} \to o$ is a fresh terminal symbol whose arity corresponds to the number of rules for F.

Applying this process to each nonterminal of the HORS with more than one rule results in a new HORS that is deterministic – it has exactly one rule for each nonterminal. However, this process does not preserve the property of being word generating: The new terminal symbols that we have introduced have arity greater than one.

One could formally establish the following correspondence between a word-generating HORS and its determinization: The unique (typically infinite) derivation process of the determinization generates an infinite tree as its limit. Each finite branch of that tree (with the occurrences of the terminals of the shape branch_F removed) is a finite word in the language of the word-generating HORS and vice versa. Since we will not need this correspondence in the following, we forgo giving the formal proof.

Working with the determinization of the game HORS has the advantage that we can encode the semantics of the choices of the two players into the interpretation of the symbols of the shape branch_F for their respective nonterminals.

The concrete model

Assume we apply the determinization procedure to a game HORS. We consider the system of equations associated to the deterministic HORS as defined in Section 18.2. In order to interpret it, we instantiate our model template. The resulting model is what we will call the *concrete model* $\mathcal{M}_c$.

Instantiating the model requires us to provide a domain $\mathcal{D}_{c,o}$ for kind ground. Intuitively, we want to use positive Boolean formulas over words. In contrast to the development in Chapter 17, this means that we consider an infinite set of atoms. As a consequence, it will easier to establish a correspondence between the winner of the game and the least solution to the interpreted system of equations, but we will not be able to compute the least solution in finite time.

Let us consider the set of positive Boolean formulas over words as atoms, factorized by logical equivalence and ordered by implication. Because the set of atoms is infinite, the set of equivalence classes is not finite. Unfortunately, this also means that this domain is not a CPPO. Let $\{w^{(i)} \mid i \in \mathbb{N}\}$ be an infinite set of distinct words. Then the sequence of formulas

$$w^{(0)},\ \ w^{(0)} \vee w^{(1)},\ \ w^{(0)} \vee w^{(1)} \vee w^{(2)},\ \ \ldots$$

is a strictly ascending chain with respect to implication. If the proposed domain were a CPPO, its join $\bigsqcup_{i\in\mathbb{N}} w^{(0)} \vee \ldots \vee w^{(i)}$ would need to exist. Intuitively, this join should be $\bigvee_{i\in\mathbb{N}} w^{(i)}$, but this is an infinite disjunction and not a finite formula.

To overcome the problem, we consider (potentially infinite) sets of formulas that we see as the (potentially infinite) disjunction of all formulas in the set[1]. More formally, we define the domain $\mathcal{D}_{c,o}$ associated to kind ground as

$$\mathcal{D}_{c,o} = \left(\left(\mathcal{P}\left(\mathsf{pBF}\left(\Sigma^*\right)\right) \setminus \varnothing\right)/_{\Longleftrightarrow},\ \Longrightarrow\right),$$

non-empty sets of positive Boolean formulas over words, factorized by logical equivalence and ordered by implication.

In order to formally define implication, we proceed as follows. Since the set of atoms is the set of words over Σ, variable assignments correspond to languages $\mathcal{L} \subseteq \Sigma^*$ by setting $\mathcal{L}(w)$ = true iff $w \in \mathcal{L}$. A language $\mathcal{L}$ satisfies a formula H, $H(\mathcal{L})$ = true, if the formula evaluates to true under the standard evaluation semantics. A language $\mathcal{L}$ satisfies a non-empty set of formulas $\mathcal{H}$, $\mathcal{H}(\mathcal{L})$ = true, if it satisfies at least one formula $H \in \mathcal{H}$. A set of formulas $\mathcal{H}$ implies another set of formulas $\mathcal{H}'$, $\mathcal{H} \Longrightarrow \mathcal{H}'$, if any language that satisfies $\mathcal{H}$ also satisfies $\mathcal{H}'$. Two sets of formulas $\mathcal{H}, \mathcal{H}'$ are logically equivalent, $\mathcal{H} \Longleftrightarrow \mathcal{H}'$, if both $\mathcal{H} \Longrightarrow \mathcal{H}'$ and $\mathcal{H}' \Longrightarrow \mathcal{H}$ hold.

Note that our definition of implication extends the usual notion of implication for formulas to sets of formulas in a straightforward way by seeing them as (potentially) infinite disjunction. For

[1] Note that this contrasts with the convention in logic to see sets of formulas as their (potentially infinite) conjunction.

example, the singleton $\{H \vee H'\}$ is logically equivalent to the set $\{H, H'\}$. In the following, we will proceed as in the previous chapter and work with formulas that represent the corresponding equivalence class.

It remains to argue that $\mathcal{D}_{c,o}$ is indeed a CPPO. Let $(\mathcal{H}_i)_{i\in\mathbb{N}}$ be an ascending chain of sets of formulas. The join of this chain is simply its union, $\bigsqcup_{i\in\mathbb{N}} \mathcal{H}_i = \bigcup_{i\in\mathbb{N}} \mathcal{H}_i$. Using the definition of implication, it is not hard to verify that $\mathcal{H}_i \implies \bigcup_{i\in\mathbb{N}} \mathcal{H}_i$ holds for all i and that any formula that is implied by all $\mathcal{H}_i$ is also implied by their union. The least element of $\mathcal{D}_{c,o}$ is the equivalence class of the singleton set {false}.

With the definition of $\mathcal{D}_{c,o}$ fixed, the model template defines $\mathcal{D}_{c,\kappa}$ for all kinds κ their union $\mathcal{D}_c = \bigcup_\kappa \mathcal{D}_{c,\kappa}$. To complete the instantiation, we still need to provide the interpretation $\mathcal{I}_c$ of the terminals.

Our HORS has three types of terminals. The first is the word-end marker \$ of kind o. We interpret it as the singleton set containing the atom ε as a formula, $\$^{\mathcal{I}_c} = \{\varepsilon\}$. This is an element of $\mathcal{D}_{c,o}$ as expected. The second type are the letters $a\colon o \to o$. We define the interpretation to be $a^{\mathcal{I}_c} = \mathrm{prepend}_a(-)$, a function that is defined as follows: Given a set of formulas, it distributes over that set, $\mathrm{prepend}_a(\mathcal{H}) = \{\mathrm{prepend}_a(H) \mid H \in \mathcal{H}\}$. In a single formula, it distributes over conjunctions and disjunctions until it reaches an atom, where it prepends the word consisting of the letter a,

$$\mathrm{prepend}_a\big(H \overset{\wedge}{\vee} H'\big) = \mathrm{prepend}_a(H) \overset{\wedge}{\vee} \mathrm{prepend}_a\big(H'\big)$$
$$\mathrm{prepend}_a(w) = a.w\ .$$

The prepend-function is an element of $\mathcal{D}_{c,o\to o}$, i.e. a join-continuous function from $\mathcal{D}_{c,o}$ to $\mathcal{D}_{c,o}$. The definition makes it easy to verify join-continuity.

Finally, there are terminals of the shape branch_F (where F is a nonterminal) that were introduced by the determinization. We interpret these symbols as disjunctions or conjunctions, depending on whether the owner of F is the existential or the universal player. We give the definitions in the case that branch_F has arity two. The general case can be easily derived. If F is owned by the existential player, then $\mathrm{branch}_F^{\mathcal{I}_c}$ is the function that takes two sets of formulas and returns their union, which corresponds to the disjunction of the sets:

$$\mathrm{branch}_F^{\mathcal{I}_c}(\mathcal{H}, \mathcal{H}') = \mathcal{H} \vee \mathcal{H}' := \mathcal{H} \cup \mathcal{H}'\ .$$

If F is owned by the universal player, then $\mathrm{branch}_F^{\mathcal{I}_c}$ is the function that takes two sets $\mathcal{H}$ and $\mathcal{H}'$ and returns $\{H \wedge H' \mid H \in \mathcal{H}, H' \in \mathcal{H}'\}$, which corresponds to the conjunction of the sets:

$$\mathrm{branch}_F^{\mathcal{I}_c}(\mathcal{H}, \mathcal{H}') = \mathcal{H} \wedge \mathcal{H}' := \big\{H \wedge H' \;\big|\; H \in \mathcal{H}, H' \in \mathcal{H}'\big\}\ .$$

Using distributivity, one can check that these definitions have the expected properties and satisfy join-continuity.

The domain for kind ground and the interpretations of the terminals satisfy the requirements of the model template. Instantiating it yields the concrete model $\mathcal{M}_c = (\mathcal{D}_c, \mathcal{I}_c)$. By Lemma 18.2.7, we get for each term t of kind κ the *concrete semantics* of term t as a function

$$\mathcal{M}_c[\![t]\!] : (N \uplus V \to_p \mathcal{D}_c) \to_\sqcup \mathcal{D}_{c,\kappa} .$$

Soundness

Assume that we are given a higher-order inclusion game (G, A). We construct the system of equations associated to the HORS and interpret it using the concrete model $\mathcal{M}_c$. Proposition 18.2.8 proves the existence of the *concrete solution* sol_c, the least value so that $\mathrm{sol}_c(F) = \mathcal{M}_c[\![t]\!]\mathrm{sol}_c$ for each nonterminals F, where $F \to t$ is its unique rule. We also get that the concrete solution is the join of the chain of approximants. We will deal with the issue of computing this least solution later.

The following theorem tells us how the concrete solution yields the winner of the game.

18.3.1 Theorem
The existential player wins the inclusion game iff $\mathrm{sol}_c(S)$ is satisfied by $\overline{\mathcal{L}(A)}$.

The value of $\mathrm{sol}_c(S)$ is an (an equivalence class of) a set of formula over words, so the notation of being satisfied by a language makes sense. Note that the concrete solution associates elements of the domain to terms of the determinization of the game HORS. In the following proofs, we will apply sol_c to terms of the game HORS. Since every term of the game HORS is a valid term of its determinization, this is formally valid. In the proof of Theorem 18.3.1 we will have to bridge the gap between the two HORSes by showing that the information about the determinization that is captured by the concrete solution indeed characterizes the winner of the game.

We first show that whenever $\mathrm{sol}_c(S)$ is not satisfied by $\overline{\mathcal{L}(A)}$, then the universal player has a winning strategy. The strategy maintains the property that the solution associated to each term that occurs in the play is not satisfied by $\overline{\mathcal{L}(A)}$.

For the proof of this property, we will need the following *substitution lemma*.

18.3.2 Lemma
For any two terms t, t' and a valuation ν, we have

$$\mathcal{M}_c[\![t[x \mapsto t']]\!]\nu = \mathcal{M}_c[\![t]\!](\nu[x \mapsto \mathcal{M}_c[\![t']\!]\nu]) .$$

Proof:
We proceed by induction on t. If t is a HORS variable other than x, a terminal, or a nonterminal, equality obviously holds. If $t = x$ we get

$$\mathcal{M}_c[\![x[x \mapsto t']]\!]\nu = \mathcal{M}_c[\![t']\!]\nu = \mathcal{M}_c[\![x]\!](\nu[x \mapsto \mathcal{M}_c[\![t']\!]\nu]) .$$

For the induction step, consider function application, i.e. $t = t_1\ t_2$. We have

$$\mathcal{M}_c[\![t_1\ t_2]\!](\nu[x \mapsto \mathcal{M}_c[\![t']\!]\nu]) = \left(\mathcal{M}_c[\![t_1]\!](\nu[x \mapsto \mathcal{M}_c[\![t']\!]\nu])\right)\left(\mathcal{M}_c[\![t_2]\!](\nu[x \mapsto \mathcal{M}_c[\![t']\!]\nu])\right) .$$

We apply induction to t_1 and t_2, obtaining that the latter term is equal to

$$\left(\mathcal{M}_c[\![t_1[x \mapsto t']]\!]\nu\right)\left(\mathcal{M}_c[\![t_2[x \mapsto t']]\!]\nu\right) = \mathcal{M}_c[\![t_1[x \mapsto t']\ t_2[x \mapsto t']]\!]\nu = \mathcal{M}_c[\![(t_1\ t_2)[x \mapsto t']]\!]\nu .$$

Consider the case that $t = \lambda x.t_1$ starts with a lambda abstraction for variable x. We have

$$\mathcal{M}_c[\![(\lambda x.t_1)[x \mapsto t']]\!]\nu = \mathcal{M}_c[\![\lambda x.t_1]\!]\nu = \mathcal{M}_c[\![\lambda x.t_1]\!](\nu[x \mapsto \mathcal{M}_c[\![t']\!]\nu]) .$$

The first equality is because the replacement $x \mapsto t'$ will only replace the free occurrences of x in t. Since t starts with λx, there are no free occurrences. The second equality is because the interpretation of lambda abstraction will discard the value for x that is present in ν.

Finally, consider $t = \lambda y.t_1$ where $y \neq x$. We have that $\mathcal{M}_c[\![\lambda y.t_1]\!](\nu[x \mapsto \mathcal{M}_c[\![t']\!]\ \nu])$ is a function that takes a value d for y and returns $\mathcal{M}_c[\![t_1]\!](\nu[x \mapsto \mathcal{M}_c[\![t']\!]\ \nu, y \mapsto d])$. By induction, this return value is equal to $\mathcal{M}_c[\![t_1[x \mapsto t']]\!](\nu[y \mapsto d])$. If we start from the other side of the desired equality, we obtain that $\mathcal{M}_c[\![(\lambda y.t_1)[x \mapsto t']]\!]\nu$ equals $\mathcal{M}_c[\![\lambda y.(t_1[x \mapsto t'])]\!]\nu$, which is a function that expects a value d and returns $\mathcal{M}_c[\![t_1[x \mapsto t']]\!](\nu[y \mapsto d])$. Equality holds and the proof is complete. ■

With the lemma at hand, we can prove one direction of Theorem 18.3.1.

18.3.3 Proposition
If $\mathrm{sol}_c(S)$ is not satisfied by $\overline{\mathcal{L}(A)}$, then the universal player has a winning strategy.

Proof:
We construct a strategy for the universal player such that every play that starts in a (variable-free) term t' with $\mathrm{sol}_c(t')$ not satisfied by $\overline{\mathcal{L}(A)}$, then for any term t occurring the play, $\mathrm{sol}_c(t)$ not satisfied by $\overline{\mathcal{L}(A)}$.

To see that this is indeed a winning strategy, note that if the play ends after finite time in a terminal word w, then we guarantee that $\mathrm{sol}_c(w) = w$ is not satisfied by $\overline{\mathcal{L}(A)}$. Hence, $w \in \mathcal{L}(A)$, and the universal player wins this play.

Let us now show that whenever term t is not a deadlock (i.e. it contains a nonterminal), then the strategy can maintain the invariant. Let $t = a_1(\ldots(a_n(F\ t_1 \ldots t_m))\ldots)$ be the current term. If F is owned by the universal player, the strategy can pick the next replacement step and it is sufficient so show that a suitable successor (whose associated solution is not satisfied by the complement language) exists. If F, however, is owned by the existential player, we have to show that any applicable replacement step yields a successor that is not satisfied by $\overline{\mathcal{L}(A)}$.

We assume that

$$\begin{aligned}\mathrm{sol}_c(t) = \mathcal{M}_c[\![t]\!]\mathrm{sol}_c &= \mathcal{M}_c[\![a_1(\ldots(a_n(F\ t_1 \ldots t_m)))]\!]\mathrm{sol}_c \\ &= \mathrm{prepend}_{a_1\ldots a_n}((\mathcal{M}_c[\![F]\!]\mathrm{sol}_c)\,(\mathcal{M}_c[\![t_1]\!]\mathrm{sol}_c)\ldots(\mathcal{M}_c[\![t_m]\!]\mathrm{sol}_c))\end{aligned}$$

is not satisfied by $\mathcal{L}(A)$. Here, we have used that the composition of $\mathrm{prepend}_{a_1}$, $\mathrm{prepend}_{a_2}$, …, $\mathrm{prepend}_{a_n}$ equals $\mathrm{prepend}_{a_1\ldots a_n}$. We have also evaluated the function applications in $F\ t_1 \ldots t_m$ using the interpretation of function application.

Let $F \to \lambda x_1 \ldots \lambda x_m.e_1, \ldots, F \to \lambda x_1 \ldots \lambda x_m.e_k$ be an exhaustive list of all rewriting rules for F in the game HORS. Recall that the unique rule for F in the determinization is

$$F \to \lambda x_1 \ldots \lambda x_m.\mathrm{branch}_F\ e_1 \ldots e_k\ .$$

The concrete solution satisfies $\mathrm{sol}_c(F) = \mathcal{M}_c[\![F]\!]\mathrm{sol}_c = \mathcal{M}_c[\![\lambda x_1 \ldots \lambda x_m.\mathrm{branch}_F\ e_1 \ldots e_k]\!]\mathrm{sol}_c$. Using the interpretation of lambda abstraction, we obtain that $\mathcal{M}_c[\![F]\!]\mathrm{sol}_c$ is a function that takes values $d_1, \ldots, d_m$ for the variables $x_1, \ldots, x_m$ and returns

$$\mathcal{M}_c[\![\mathrm{branch}_F\ e_1 \ldots e_k]\!](\mathrm{sol}_c[x_1 \mapsto d_1, \ldots x_m \mapsto d_m])\ .$$

Evaluating the function applications and using the interpretation of branch_F, this value equals

$$\mathrm{branch}_F{}^{\mathcal{I}_c}\ \mathcal{M}_c[\![e_1]\!](\mathrm{sol}_c[x_1 \mapsto d_1, \ldots x_m \mapsto d_m])\ \ldots\ \mathcal{M}_c[\![e_1]\!](\mathrm{sol}_c[x_1 \mapsto d_1, \ldots x_m \mapsto d_m])\ .$$

Consider this case that F is owned by the universal player. This means the interpretation of branch_F is conjunction and we get that the above value equals

$$\mathcal{M}_c[\![e_1]\!](\mathrm{sol}_c[x_1 \mapsto d_1, \ldots x_m \mapsto d_m]) \wedge \ldots \wedge \mathcal{M}_c[\![e_1]\!](\mathrm{sol}_c[x_1 \mapsto d_1, \ldots x_m \mapsto d_m])\ .$$

We plug this definition of $\mathcal{M}_c[\![F]\!]\mathrm{sol}_c$ into the value we have obtained for $\mathrm{sol}_c(t)$ and get

$$\begin{aligned}\mathrm{sol}_c(t) &= \mathrm{prepend}_{a_1\ldots a_n}((\mathcal{M}_c[\![F]\!]\mathrm{sol}_c)\,(\mathcal{M}_c[\![t_1]\!]\mathrm{sol}_c)\ldots(\mathcal{M}_c[\![t_m]\!]\mathrm{sol}_c))\\ &= \mathrm{prepend}_{a_1\ldots a_n}\big(\mathcal{M}_c[\![e_1]\!](\mathrm{sol}_c[x_1 \mapsto \mathcal{M}_c[\![t_1]\!]\mathrm{sol}_c, \ldots x_m \mapsto \mathcal{M}_c[\![t_m]\!]\mathrm{sol}_c])\\ &\qquad\qquad \wedge \ldots \wedge \mathcal{M}_c[\![e_1]\!](\mathrm{sol}_c[x_1 \mapsto \mathcal{M}_c[\![t_1]\!]\mathrm{sol}_c, \ldots x_m \mapsto \mathcal{M}_c[\![t_m]\!]\mathrm{sol}_c])\big)\\ &= \mathrm{prepend}_{a_1\ldots a_n}(\mathcal{M}_c[\![e_1]\!](\mathrm{sol}_c[x_1 \mapsto \mathcal{M}_c[\![t_1]\!]\mathrm{sol}_c, \ldots x_m \mapsto \mathcal{M}_c[\![t_m]\!]\mathrm{sol}_c]))\\ &\quad \wedge \ldots \wedge \mathrm{prepend}_{a_1\ldots a_n}(\mathcal{M}_c[\![e_1]\!](\mathrm{sol}_c[x_1 \mapsto \mathcal{M}_c[\![t_1]\!]\mathrm{sol}_c, \ldots x_m \mapsto \mathcal{M}_c[\![t_m]\!]\mathrm{sol}_c]))\,,\end{aligned}$$

where the latter equality uses that the prepend function distributions over conjunctions.

Since $\mathrm{sol}_c(t)$ is not satisfied by $\overline{\mathcal{L}(A)}$, there must be at least one i such that the conjunct

$$\mathrm{prepend}_{a_1\ldots a_n}(\mathcal{M}_c[\![e_i]\!](\mathrm{sol}_c[x_1 \mapsto \mathcal{M}_c[\![t_1]\!]\mathrm{sol}_c, \ldots x_m \mapsto \mathcal{M}_c[\![t_m]\!]\mathrm{sol}_c]))$$

is not satisfied by $\overline{\mathcal{L}(A)}$. We define the strategy of the universal player to pick the rule $F \to \lambda x_1 \ldots \lambda x_m.e_i$ in the game.

It remains to prove that the strategy maintains the invariant, i.e. we have to argue that the solution associated to the term that results from picking this rule is not satisfied by A. The rewriting rules of HORSes are defined so that replacing F in $a_1(\ldots(a_n(F\ t_1 \ldots t_m))\ldots)$ yields the term

$$t' = a_1(\ldots(a_n(e_i[x_1 \mapsto t_1, \ldots, x_m \mapsto t_m]))\ldots)\,.$$

The solution associated to this term is

$$\mathrm{sol}_c(t') = \mathrm{prepend}_{a_1\ldots a_n}(\mathcal{M}_c[\![e_i[x_1 \mapsto t_1, \ldots, x_m \mapsto t_m]]\!]\mathrm{sol}_c)\,.$$

We apply the substitution lemma, Lemma 18.3.2, m times to obtain the equality

$$\mathcal{M}_c[\![e_i[x_1 \mapsto t_1, \ldots, x_m \mapsto t_m]]\!]\mathrm{sol}_c = \mathcal{M}_c[\![e_i]\!](\mathrm{sol}_c[x_1 \mapsto \mathcal{M}_c[\![t_1]\!]\mathrm{sol}_c, \ldots x_m \mapsto \mathcal{M}_c[\![t_m]\!]\mathrm{sol}_c])\,.$$

Plugging this equality into the expression for $\mathrm{sol}_c(t')$ shows that this value equals the value for the conjunct e_i in $\mathrm{sol}_c(t)$. We have argued before that the value for the conjunct is not satisfied by $\overline{\mathcal{L}(A)}$, so $\mathrm{sol}_c(t')$ is not satisfied by $\overline{\mathcal{L}(A)}$ as desired.

The argumentation is similar in the case that the existential player owns F. In this case, branch_F is interpreted as disjunction, and $\mathrm{sol}_c(t)$ is a disjunction with one disjunct for each e_i. Since $\mathrm{sol}_c(t)$ is not satisfied by $\overline{\mathcal{L}(A)}$, none of the disjuncts is. No matter which rule $F \to \lambda x_1 \ldots \lambda x_m.e_i$ is picked by the existential player, the resulting term t' has an associated solution $\mathrm{sol}_c(t')$ that is not satisfied by $\overline{\mathcal{L}(A)}$ since it corresponds to one of the disjuncts. The strategy maintains the invariant as desired. ■

We consider the case of the existential player. While the result looks very similar, its proof is much more involved. This is because a winning strategy for the existential player needs to enforce that all plays that conform to it terminate after finite many steps.

18.3.4 Proposition
If $\mathrm{sol}_c(S)$ is satisfied by $\overline{\mathcal{L}(A)}$, then the existential player has a winning strategy.

Proof:
We introduce some notation to simplify the rest of the proof. For a domain element $\varphi \in \mathcal{D}_{c,o}$ and a variable-closed term t of kind o, we define *φ to be sound for t*, denoted by $\varphi \vDash t$, if for all words $w = w_1 \dots w_k \in \Sigma^*$ such that $\mathrm{prepend}_w(\varphi)$ is satisfied by $\overline{\mathcal{L}(A)}$, the existential player has a winning strategy from term $w(t)$, where $w(t)$ is the term $w_1(\dots w_k(t)\dots)$. For $w = \varepsilon$, we set $\mathrm{prepend}_\varepsilon(\varphi) = \varphi$ and let $\varepsilon(t) = t$.

We claim that in order to prove the proposition, it is sufficient to show that

$$\mathrm{sol}_c(S) \vDash S\,,$$

i.e. $\mathrm{sol}_c(S)$ is sound for S. Indeed, by choosing w as ε and using the fact that $\mathrm{sol}_c(S) = \mathrm{prepend}_\varepsilon(\mathrm{sol}_c(S))$ is satisfied by $\overline{\mathcal{L}(A)}$, establishing soundness implies showing that the existential player has a winning strategy from $\varepsilon(S) = S$.

In the proof, we will also need the notion of soundness for terms of higher order. For a variable-closed term t of kind $\kappa_1 \to \kappa_2$ and a function $\Phi \in \mathcal{D}_{c,\kappa_1\to\kappa_2}$, we define $\Phi \vDash t$ to hold whenever for all variable-closed terms t' of kind κ_1 and $\Phi' \in \mathcal{D}_{c,\kappa_1}$ such that $\Phi' \vDash t'$ we have $\Phi\,\Phi' \vDash t\,t'$; written as formula

$$\Phi \vDash t \quad \text{iff} \quad \forall\, \Phi', t' \text{ such that } \Phi' \vDash t' \colon\ \Phi\,\Phi' \vDash t\,t'\,.$$

Note that this is an inductive definition that defines the notion of soundness for kind $\kappa_1 \to \kappa_2$ assuming that it has already been defined for κ_1 and κ_2.

We will also need to extend the notion of soundness to terms t with free variables. Assume that the variables $x_1 \dots x_m$ are free in t. For $\Phi : (V \to_p \mathcal{D}_c) \to_\sqcup \mathcal{D}_c$, we define $\Phi \vDash t$ by requiring that for any variable-closed terms $t_1, \dots, t_m$ and any $\Phi_1, \dots, \Phi_m \in \mathcal{D}_c$ with $\Phi_j \vDash t_j$ for all $j \in [1, m]$, we have $\Phi[\forall j\colon x_j \mapsto \Phi_j] \vDash t[\forall j\colon x_j \mapsto t_j]$. Here, we use $[\forall j\colon x_j \mapsto \Phi_j]$ as a shorthand for $[x_1 \mapsto \Phi_1, \dots, x_m \mapsto \Phi_m]$, similar for $[\forall j\colon x_j \mapsto t_j]$.

Our goal is to prove $\mathrm{sol}_c(S) \vDash S$. We first show that each approximant that occurs during the fixed-point iteration is sound for the respective term, i.e. $\mathcal{M}_c[\![t]\!]\mathrm{sol}_c^i \vDash t$ for all i and all terms t of the game HORS. Since the domain is infinite, this is not sufficient to prove $\mathrm{sol}_c(S) \vDash S$. We will also need to show that soundness holds for the least fixed point.

Let us prove that for any term t that may occur in a play of the game and every $i \in \mathbb{N}$, $\mathcal{M}_c[\![t]\!]\mathrm{sol}_c^i$ is sound for t, $\mathcal{M}_c[\![t]\!]\mathrm{sol}_c^i \vDash t$. Note that terms t that occur in the game are terms of the game HORS. They do not contain the terminals of the shape branch_F. Additionally, these terms do not contain lambda abstraction. Indeed, the initial position S does not contain lambda abstraction and the rewriting rules of the HORS maintain this property. Whenever we replace a redex $F\ t_1 \dots t_m$ in the current position using a rule $F \to \lambda x_1 \dots \lambda x_m.e$, we obtain $e[x \mapsto t_1, \dots, x_m \mapsto t_m]$ where e does not contain lambda abstraction.

To prove $\mathcal{M}_c[\![t]\!]\mathrm{sol}_c^i \vDash t$, we proceed with a nested induction: The outer induction is on the iteration count i, the inner is on the structure of the term t. Since we will need to consider non-variable-closed terms during the induction, we need to consider valuations that assign values to the free HORS variables. We use v^i to denote any valuation that coincides with sol_c^i on the nonterminals, i.e. $v^i(F) = \mathrm{sol}_c^i(F)$ for all nonterminals F.

Base of the outer induction, $i = 0$:

In the base case, we have $i = 0$ and $v^i(F) = \bot$ for all nonterminals F. We show $\mathcal{M}_c[\![t]\!]v^i \vDash t$ for all terms t, proceeding by induction on the structure of terms. We emphasize that in almost all cases, the reasoning is independent of i, which will enable us to reuse it later.

The following base cases of the inner induction are independent of the iteration count.

Base case $t = \$$. For all i, we have $\mathcal{M}_c[\![\$]\!]v^i = \varepsilon$. Take any word w such that $\mathrm{prepend}_w(\varepsilon) = w$ is satisfied by $\overline{\mathcal{L}(A)}$. This means that $w \in \overline{\mathcal{L}(A)}$, and the existential player has won the play from $w(\$)$.

Base case $t = a$. Terminal a has kind $o \to o$, so we need to consider an arbitrary variable-closed term t of kind ground and $\Phi \in \mathcal{D}_{c,o}$ so that $\Phi \vDash t$ and show

$$\mathcal{M}_c[\![a]\!]v^i\ \Phi = \mathrm{prepend}_a(\Phi) \vDash a(t)\,.$$

Take any word w such that $\mathrm{prepend}_w(\mathrm{prepend}_a(\Phi)) = \mathrm{prepend}_{w.a}(\Phi)$ is satisfied by $\overline{\mathcal{L}(A)}$. By $\Phi \vDash t$, the existential player has a winning strategy from $wa(t) = w(a(t))$ as required.

Base case $t = x$. For all i and all extensions v^i of sol_c^i, we have

$$\mathcal{M}_c[\![x]\!]v^i\ = v^i(x).$$

Take any $v^i(x) = \Phi$ and any variable-closed term t' with $\Phi \vDash t'; v^i(x) \vDash x[x \mapsto t']$ is immediate.

The only base case of the inner induction that depends on the iteration count is $t = F$.

Base case $t = F$ (and $i = 0$). Assume that $F{:}\kappa$ has arity m. Consider variable-closed terms $t_1, \ldots, t_m$ with corresponding $\Phi_1, \ldots, \Phi_m$ such that $\Phi_j \vDash t_j$ for all j. Note that $F\ t_1 \ldots t_m$ is of kind o. We have

$$\mathcal{M}_c[\![F]\!]\nu^0\ \Phi_1 \ldots \Phi_m = \mathrm{sol}^0_c(F)\ \Phi_1 \ldots \Phi_m = \{\mathrm{false}\}\,,$$

since $\nu^0(F) = \mathrm{sol}^0(F) = \bot$ is the least element of $\mathcal{D}_{c,\kappa}$, the function that takes a suitable number of arguments and returns {false}, the least element of $\mathcal{D}_{c,o}$. Hence, the premise of the statement, $\mathcal{M}_c[\![F]\!]\nu^0\ \Phi_1 \ldots \Phi_m$ being satisfied by some language, cannot be true, and the implication that we wanted to prove trivially holds.

We come to the induction step of the inner induction. As we have discussed, the terms of interest never contain lambda abstraction. We only need to consider function application. Our argumentation is independent of the iteration count i.

Induction step, $t = t'\ t''$. Using induction, we can assume

$$\mathcal{M}_c[\![t']\!]\nu^i \vDash t' \quad \text{and} \quad \mathcal{M}_c[\![t'']\!]\nu^i \vDash t''\,.$$

We need to prove

$$\mathcal{M}_c[\![t'\ t'']\!]\nu^i = (\mathcal{M}_c[\![t']\!]\nu^i)\,(\mathcal{M}_c[\![t'']\!]\ \nu^i) \vDash t'\ t''\,.$$

Assume that $x_1, \ldots, x_n$ are all free variables in $t'\ t''$ and consider terms $t_1, \ldots, t_n$ and $\Phi_1, \ldots \Phi_n$ so that $\Phi_j \vDash t_j$ for all j. Let ν^i map x_j to Φ_j for all $j \in [1, n]$. By the definition of $\vDash$ for terms with free variables, we have $\mathcal{M}_c[\![t']\!]\nu^i \vDash t'[\forall j : x_j \mapsto t_j]$ and $\mathcal{M}_c[\![t'']\!]\nu^i \vDash t''[\forall j : x_j \mapsto t_j]$. Then, by the definition of $\vDash$ for functions, we obtain

$$\begin{aligned}\mathcal{M}_c[\![t'\ t'']\!]\nu^i &= (\mathcal{M}_c[\![t']\!]\nu^i)\,(\mathcal{M}_c[\![t'']\!]\nu^i)\\ &\vDash (t'[\forall j : x_j \mapsto t_j])\,(t''[\forall j : x_j \mapsto t_j]) = (t'\ t'')[\forall j : x_j \mapsto t_j]\,.\end{aligned}$$

This means $\mathcal{M}_c[\![t'\ t'']\!]\nu^i \vDash t'\ t''$ as required.

Induction step of the outer induction, $i \to i + 1$:

We again proceed by induction on structure of the term t. All cases but $t = F$ have already been treated in full generality in the base case of the outer induction. We have to show $\mathcal{M}_c[\![F]\!]\nu^{i+1} \vDash F$.

We first observe that

$$\mathcal{M}_c[\![F]\!]\nu^{i+1} = \mathcal{M}_c[\![t_F]\!]\nu^i$$

where $F \to t_F$ is the unique rule for F in the determinization of the game HORS. For the sake often simplicity, we assume that there are only two rules with left-hand side F, i.e. we have the rules $F \to \lambda x_1 \ldots \lambda x_m.e_1$ and $F \to \lambda x_1 \ldots \lambda x_m.e_2$ in the game HORS. (Proving the general case with an arbitrary number of right-hand sides for F is analogous.) Hence, the unique rule for F in the determinization is $F \to \lambda x_1 \ldots \lambda x_m.\mathrm{branch}_F\ e_1\ e_2$. Consequently,

$$\mathcal{M}_c[\![F]\!]v^{i+1} = \mathcal{M}_c[\![\lambda x_1 \ldots \lambda x_m.\mathrm{branch}_F\ e_1\ e_2]\!]v^i\,.$$

Using the interpretation of lambda abstraction, the latter value is a function that takes values $d_1, \ldots, d_m$ and returns

$$\mathcal{M}_c[\![\mathrm{branch}_F\ e_1\ e_2]\!]v^i[x_1 \mapsto d_1, \ldots, x_m \mapsto d_m]\,.$$

By evaluating the interpretation of function application, this return value equals

$$\mathrm{branch}_F{}^{\mathcal{I}_c}\ \mathcal{M}_c[\![e_1]\!]v^i[x_1 \mapsto d_1, \ldots, x_m \mapsto d_m]\ \mathcal{M}_c[\![e_2]\!]v^i[x_1 \mapsto d_1, \ldots, x_m \mapsto d_m]\,.$$

In order to show $\mathcal{M}_c[\![F]\!]v^{i+1} \vDash F$, consider terms $t_1, \ldots, t_m$ and $\Phi_1, \ldots, \Phi_m$ so that $\Phi_j \vDash t_j$ for all $j \in [1, m]$. We need to prove

$$\mathcal{M}_c[\![F]\!]v^{i+1}\ \Phi_1 \ldots \Phi_m \vDash F\ t_1 \ldots t_m\,.$$

By substituting the definition of $\mathcal{M}_c[\![F]\!]v^{i+1}$ that we have obtained above, the left-hand side of this expression equals d where

$$d = \mathrm{branch}_F{}^{\mathcal{I}_c}\ \mathcal{M}_c[\![e_1]\!]v^i[x_1 \mapsto \Phi_1, \ldots, x_m \mapsto \Phi_m]\ \mathcal{M}_c[\![e_2]\!]v^i[x_1 \mapsto \Phi_1, \ldots, x_m \mapsto \Phi_m]\,.$$

Consider a word $w \in \Sigma^*$. We need to show that if $\mathrm{prepend}_w(d)$ is satisfied by $\overline{\mathcal{L}(A)}$, then the existential player has a winning strategy from $w(F\ t_1 \ldots t_m)$.

Assume that the existential player owns F. In this case, the interpretation $\mathrm{branch}_F{}^{\mathcal{I}_c}$ of branch_F is disjunction. We have

$$\begin{aligned}\mathrm{prepend}_w(d) &= \mathrm{prepend}_w\big(\mathcal{M}_c[\![e_1]\!]v^i[x_1 \mapsto \Phi_1, \ldots, x_m \mapsto \Phi_m] \\ &\qquad \vee\ \mathcal{M}_c[\![e_2]\!]v^i[x_1 \mapsto \Phi_1, \ldots, x_m \mapsto \Phi_m]\big) \\ &= \mathrm{prepend}_w\Big(\mathcal{M}_c[\![e_1]\!]v^i[x_1 \mapsto \Phi_1, \ldots, x_m \mapsto \Phi_m]\Big) \\ &\qquad \vee\ \mathrm{prepend}_w\Big(\mathcal{M}_c[\![e_2]\!]v^i[x_1 \mapsto \Phi_1, \ldots, x_m \mapsto \Phi_m]\Big)\,,\end{aligned}$$

using that the prepend function distributes over disjunctions. If $\mathrm{prepend}_w(d)$ is satisfied by $\overline{\mathcal{L}(A)}$, then at least one of the disjuncts is satisfied by $\overline{\mathcal{L}(A)}$. Wlog., we will assume that the first disjunct $\mathrm{prepend}_w(\mathcal{M}_c[\![e_1]\!]v^i[x_1 \mapsto \Phi_1, \ldots, x_m \mapsto \Phi_m])$ is satisfied by $\overline{\mathcal{L}(A)}$ in the following.

We claim that we obtain a winning strategy from $w(F\ t_1 \dots t_m)$ for the existential player by first applying the move associated to the rule $F \to \lambda x_1 \dots \lambda x_m.e_1$ and then using the winning strategy from the resulting term. The resulting term is $w(t')$ where

$$t' = e_1[x_1 \mapsto t_1, \dots, x_m \mapsto t_m]\,.$$

In order to complete the argument, we need to show that there is a winning strategy from $w(t')$. Using the induction hypothesis of the outer induction, we have that $\mathrm{sol}_c^i(e_i)$ is sound for e_i, $\mathrm{sol}_c^i(e_i) \vDash e_i$. Consider the terms t_j and the Φ_j for $j \in [1, m]$. The definition of soundness for terms with free variables means that $\mathrm{sol}_c^i(e_i) \vDash e_i$ implies

$$\mathrm{sol}_c^i(e_i)[x_1 \mapsto \Phi_1, \dots, x_m \mapsto \Phi_m] \vDash e_1[x_1 \mapsto t_1, \dots, x_m \mapsto t_m]\,,$$

and note that the right-hand side of this expression is t'. Consider the word w as before. Applying $\mathrm{prepend}_w(-)$ to the left-hand side yields

$$\mathrm{prepend}_w\Big(\mathrm{sol}_c^i(e_i)[x_1 \mapsto \Phi_1, \dots, x_m \mapsto \Phi_m]\Big)$$

$$= \mathrm{prepend}_w\Big(\mathcal{M}_c[\![e_i]\!]\Big(v^i[x_1 \mapsto \Phi_1, \dots, x_m \mapsto \Phi_m]\Big)\Big)\,,$$

which is the same value that we have obtained for the disjunct associated to e_1 above. We already know that this value is satisfied by $\overline{\mathcal{L}(A)}$, so we obtain that there is a winning strategy for the existential player from $w(t')$.

If F is owned by the universal player, $\mathrm{prepend}_w(d)$ is a conjunction. We can argue similarly: The value associated to both conjuncts must be satisfied by $\overline{\mathcal{L}(A)}$. No matter which rule $F \to \lambda x_1 \dots \lambda x_m.e_i$ is picked by the universal player, we can apply the induction hypothesis to e_i. We obtain that the existential player has a winning strategy from both positions that result from applying one of the rules. By using one of these winning strategies, depending on which rule is used by the universal player, we obtain a winning strategy from $w(F\ t_1 \dots t_m)$ as required.

From $\mathbf{sol}_c^i$ to $\mathbf{sol}_c$

With both inductions finished, we have shown $\mathcal{M}_c[\![t]\!]\mathrm{sol}_c^i \vDash t$ for all terms t and all $i \in \mathbb{N}$. It remains to show $\mathcal{M}_c[\![t]\!]\mathrm{sol}_c \vDash t$. Since the CPPO under consideration is not finite, this needs to be proven separately (since we do not necessarily have $\mathrm{sol}_c = \mathrm{sol}_c^i$ for some i.)

To this end, we proceed by induction on kinds. Let t be a variable-closed term of kind κ, and let $(\Phi_i)_{i\in\mathbb{N}}$ be an ascending chain in $\mathcal{D}_{c,\kappa}$ so that each Φ_i is sound for t, $\Phi_i \vDash t$. We prove that then the join $\bigsqcup_{i\in\mathbb{N}} \Phi_i$ is sound for t, $\bigsqcup_{i\in\mathbb{N}} \Phi_i \vDash t$.

Base case $t\colon o$. Let $(\Phi_i)_{i\in\mathbb{N}}$ be an ascending chain in $\mathcal{D}_{c,o}$ so that $\Phi_i \vDash t$ for all i. We prove $\bigsqcup_{i\in\mathbb{N}} \Phi_i \vDash t$.

Take any word $w \in \Sigma^*$ and assume that $\mathrm{prepend}_w(\bigsqcup_{i\in\mathbb{N}} \Phi_i)$ is satisfied by $\overline{\mathcal{L}(A)}$, then we need to show by the definition of $\vDash$ that the existential player has a winning strategy from $w(t)$. The prepend function is join-continuous, so we obtain

$$\mathrm{prepend}_w(\bigsqcup_{i\in\mathbb{N}} \Phi_i) = \bigsqcup_{i\in N} \mathrm{prepend}_w(\Phi_i)\,.$$

Recall that the join on $\mathcal{D}_{c,o}$ is union, which corresponds to an (infinite) disjunction, this value being satisfied by $\overline{\mathcal{L}(A)}$ means that $\mathrm{prepend}_w(\Phi_i)$ is satisfied by $\overline{\mathcal{L}(A)}$ for at least one i. By assumption $\Phi_i \vDash t$ holds, so there is a winning strategy for the existential player from $w(t)$. This proves $\bigsqcup_{i\in\mathbb{N}} \Phi_i \vDash t$ as desired.

Induction step, $t\colon \kappa_1 \to \kappa_2$. Let $(\Phi_i)_{i\in\mathbb{N}}$ be an ascending chain in $\mathcal{D}_{c,\kappa_1\to\kappa_2}$ so that $\Phi_i \vDash t$ for all i. We prove $\bigsqcup_{i\in\mathbb{N}} \Phi_i \vDash t$.

Consider a term t' and Φ' so that $\Phi' \vDash t'$. We need to show $(\bigsqcup_{i\in\mathbb{N}} \Phi_i)\,\Phi \vDash t\ t'$. Consider the left-hand side of this expression. Each Φ_i is a function from $\mathcal{D}_{c,\kappa_1\to\kappa_2}$ and by the definition of the join on $\mathcal{D}_{c,\kappa_1\to\kappa_2}$ we get

$$(\bigsqcup_{i\in\mathbb{N}} \Phi_i)\,\Phi \;=\; \bigsqcup_{i\in\mathbb{N}} (\Phi_i\,\Phi)\,.$$

By assumption, $\Phi_i \vDash t$ holds for all i. By the definition of soundness, this means $\Phi_i\,\Phi \vDash t\ t'$ for all i. The sequence $(\Phi_i\,\Phi)_{i\in\mathbb{N}}$ is an ascending chain in $\mathcal{D}_{c,\kappa_2}$. (Indeed, $\Phi_i \in \mathcal{D}_{c,\kappa_1\to\kappa_2}$ is join-continuous and hence monotonic.) We can apply induction and obtain

$$\bigsqcup_{i\in\mathbb{N}} (\Phi_i\,\Phi) \vDash t\ t'\,.$$

This is what we wanted to show and the induction has been completed.

Let t be some term. Our goal is to show that $\mathrm{sol}(t)$ is sound for t. Consider the chain $(\mathrm{sol}_c^i(t))_{i\in\mathbb{N}}$. Using the first half of the proof, each $\mathrm{sol}_c^i(t)$ is sound for t. Using the second half of the proof, then also $\bigsqcup_{i\in\mathbb{N}} \mathrm{sol}_c^i(t)$ is sound for t. We have

$$\mathrm{sol}_c(t) = \mathcal{M}_c[\![t]\!]\mathrm{sol}_c = \mathcal{M}_c[\![t]\!] \bigsqcup_{i\in\mathbb{N}} \mathrm{sol}_c^i = \bigsqcup_{i\in\mathbb{N}} \mathcal{M}_c[\![t]\!]\mathrm{sol}_c^i = \bigsqcup_{i\in\mathbb{N}} \mathrm{sol}_c^i(t)\,,$$

using the definition of $\mathrm{sol}_c(t)$, the fact that $\mathrm{sol}_c = \bigsqcup_{i\in\mathbb{N}} \mathrm{sol}_c^i$, the join-continuity of $\mathcal{M}_c[\![t]\!]$, and the definition of $\mathrm{sol}_c^i(t)$. Hence, $\mathrm{sol}_c(t)$ is sound for t.

In particular, $\mathrm{sol}_c(S)$ is sound for S. As argued at the beginning of the proof, this means that $\mathrm{sol}_c(S)$ being satisfied by $\overline{\mathcal{L}(A)}$ implies the existential player having a winning strategy from S. This is what we wanted to show, finishing the proof of the proposition. ■

A winning strategy for the existential player needs to enforce that all plays that conform to it terminate. In the proof, this fact is hidden in the form of the outer induction on i. If $\mathrm{prepend}_w(\mathrm{sol}^i_c(t))$ is satisfied by $\overline{\mathcal{L}(A)}$, then the existential player can enforce a terminating play that leads to a terminal word in $\overline{\mathcal{L}(A)}$. The play corresponds to a derivation tree, a notion that we have formally defined for context-free grammars, but not for higher-order recursion schemes, whose height is at most i and whose width is bound by the size of the given game HORS. Once we replace a nonterminal F in the proof using a rule $F \to t'$, we go from considering $\mathrm{sol}^i_c(F)$ to considering $\mathrm{sol}^{i-1}_c(t'')$, where t'' is the term resulting from replacing F. This yields the aforementioned bound on the height.

Together, Proposition 18.3.3 and Proposition 18.3.4 prove Theorem 18.3.1. The winner of the inclusion game induced by the HORS can be read off from the concrete solution. We are left with the problem that we cannot compute this solution in finite time.

18.4 Framework for exact fixed-point transfer

Our goal in this section is to develop a framework for the exact transfer of the fixed point with respect to one model to the fixed point with respect to another model. Assume we are given two models $\mathcal{M}_c$ and $\mathcal{M}_a$ and a function α that maps values from one domain to the other. We want to provide a minimal set of criteria that will allow us to conclude that the function applied to the fixed point wrt. $\mathcal{M}_c$ is the fixed point wrt. $\mathcal{M}_a$, $\alpha(\mathrm{sol}_c) = \mathrm{sol}_a$.

The following is a potential application for this framework. Assume the goal is solving a decision problem involving HORSes using effective denotational semantics. We have *concrete* model $\mathcal{M}_c = (\mathcal{D}_c, \mathcal{I}_c)$, for which it is easy to prove that the associated fixed point sol_c captures the answer to the decision problem. In this model, it is hard or even impossible to compute the least solution (e.g. because the domain is infinite and does not satisfy the ascending chain condition). We also have an *abstract* model $\mathcal{M}_a = (\mathcal{D}_a, \mathcal{I}_a)$ that is well-behaved with respect to the fixed-point computation, but proving that sol_a also captures the answer to the decision problem may be much harder. Using the framework, it is sufficient to provide an abstraction function α from the concrete to the abstract domain that satisfies some properties. We get exact fixed-point transfer and have $\alpha(\mathrm{sol}_c) = \mathrm{sol}_a$. If the property that allows us to read off the answer to the decision problem is preserved under α, we conclude that reading off the answer to the decision problem is indeed sound. Hence, we can replace a complicated proof, namely the proof that sol_a captures the solution to the decision problem, by several simpler proofs.

In the next section, we will apply this approach to computing the winner of inclusion games with game arenas defined by game HORSes. Since we think that the framework for exact fixed-point transfer is of independent interest, we develop our theory for a general setting.

Assume that a deterministic HORS G and its associated system of equations are fixed. We consider two models $\mathcal{M}_c = (\mathcal{D}_c, \mathcal{I}_c)$ and $\mathcal{M}_a = (\mathcal{D}_a, \mathcal{I}_a)$ that are instantiations of the model template described in Section 18.2 We call $\mathcal{M}_c$ and the associated semantics *concrete* and $\mathcal{M}_a$ *abstract* to appeal to the intuition of the reader. However, these words have no formal meaning here; in particular, both models could be finite or infinite. Each of the models provides an interpretation $\mathcal{I}_c$ resp. $\mathcal{I}_a$ of all function symbols used in the system of equations, a domain $\mathcal{D}_{c,\kappa}$ resp. $\mathcal{D}_{a,\kappa}$ for each kind κ and a signature $(N \cup V \to_p \mathcal{D}_c) \to_\sqcup \mathcal{D}_c$ resp. $(N \cup V \to_p \mathcal{D}_a) \to_\sqcup \mathcal{D}_a$ for the semantics of HORS terms.

Assume that an *abstraction function* $\alpha\colon \mathcal{D}_{c,o} \to \mathcal{D}_{a,o}$ is given. Note that it acts on the domains for kind ground. The first step in the development of the framework will be to lift α to the domains for other kinds. Intuitively, the abstraction $\alpha(f_c)$ of a function $f_c \in \mathcal{D}_{c,\kappa_1 \to \kappa_2}$ should be the function that takes an abstract value $x_a \in \mathcal{D}_{a,\kappa_1}$, *concretizes* it by taking a preimage $x_c \in \mathcal{D}_{c,\kappa_1}$ with $\alpha(x_c) = x_a$, applies f to the preimage and abstracts the result, $\alpha(x_c)\, x_a = \alpha(f_c\, x_c)$.

This definition has two problems: The abstract value x_a might have no preimage under α, and even if it has one, the preimage may not be unique. To solve the second problem, we introduce

the notion of *compatibility*: A function is compatible (with α) if values with the same abstraction are evaluated to function values with the same abstraction. Hence, the value of the abstraction of such a function will not depend on which preimage is chosen.

Furthermore, we define $\alpha(f)\ x_a$ to be the least element of the domain for the appropriate kind whenever x_a has no α-preimage. This solves the first problem, but causes another one: We now cannot hope that $\alpha(\mathrm{sol}_c) = \mathrm{sol}_a$ will hold for the components of the fixed-point that are functions. Instead, we will only get what is essentially equality on the image of α. We formalize this using an equivalence $\approx$ on the abstract domain that is equality for kind ground and equality on the image of α for function kinds. In the end, our exact fixed-point transfer result will prove $\alpha(\mathrm{sol}_c) \approx \mathrm{sol}_a$ instead of the desired equality. Since $\approx$-equivalence is equality for kind ground, this will still give us equality, $\alpha(\mathrm{sol}_c)(t) = \mathrm{sol}_a(t)$, whenever t is a term of kind ground.

Remark
In our publication [HMM17], we solved the problem of missing preimages differently, namely by simply requiring the abstraction function α to be surjective. This solves the problem, but it has a cost attached to it: Showing surjectivity is an additional proof obligation for the user of the framework. Indeed, it is often natural to use an abstraction that is not surjective. For example, we will consider an abstraction function that is not surjective in Section 18.5. Usually, it would be possible to restrict the abstract domain so that α becomes surjective (and this is in fact the approach we took in the aforementioned publication), but this again is an additional step for the user of the framework. The version of the framework we present in this thesis works without requiring surjectivity.

Basic definition

We formally define the notions that we have introduced before. For each kind κ, we provide (1) a notion of compatibility of elements $x_c \in \mathcal{D}_{c,\kappa}$, (2) an extension of the abstraction function to kind κ, i.e. $\alpha\colon \mathcal{D}_{c,\kappa} \to \mathcal{D}_{a,\kappa}$, and (3) an equivalence relation $\approx$ on $\mathcal{D}_{a,\kappa}$.

These definitions are interleaved with each other, so we have to proceed using induction. We start with kind ground, where the abstraction function $\alpha\colon \mathcal{D}_{c,o} \to \mathcal{D}_{a,o}$ is already fixed.

18.4.1 Definition
We define every element of $\mathcal{D}_{c,o}$ to be compatible. We define the equivalence $\approx$ on $\mathcal{D}_{a,o}$ to be equality.

Now consider kind $\kappa_1 \to \kappa_2$, where we assume that α, $\approx$ and compatibility have been defined for κ_1 and κ_2. Note that for each kind κ, we denote by $\bot_{c,\kappa}$ and $\bot_{a,\kappa}$ the least element of $\mathcal{D}_{c,\kappa}$ and $\mathcal{D}_{a,\kappa}$, respectively.

18.4.2 Definition

(1) A function $f_c \in \mathcal{D}_{c,\kappa_1\to\kappa_2}$ is *compatible* if for every compatible value $x_c \in \mathcal{D}_{c,\kappa_1}$, $f_c\ x_c$ is compatible, and for every compatible $x_c, x_c' \in \mathcal{D}_{c,\kappa_1}$ such that $\alpha(x_c) \approx \alpha(x_c')$, we have $\alpha(f_c\ x_c) \approx \alpha(f_c\ x_c')$.

(2) If $f_c \in \mathcal{D}_{c,\kappa_1\to\kappa_2}$ is compatible, we define its *abstraction* $\alpha(f_c) \in \mathcal{D}_{a,\kappa_1\to\kappa_2}$ as follows: Whenever $x_a \in \mathcal{D}_{a,\kappa_1}$ is $\approx$-related to the α-image of a compatible element, i.e. $x_a \approx \alpha(x_c)$ for some $x_c \in \mathcal{D}_{c,\kappa_1}$, we define the function value $\alpha(f_c)\ x_a = \alpha(f_c\ x_c)$. If no such x_c exists, we define $\alpha(f_c) = \bot_{a,\kappa_1}$ to be the least element of $\mathcal{D}_{a,\kappa_1}$.

For non-compatible f_c, we define $\alpha(f_c) = \bot_{a,\kappa_1\to\kappa_2}$ as the bottom function for kind $\kappa_1 \to \kappa_2$, the function that maps any value from $\mathcal{D}_{a,\kappa_1}$ to $\bot_{a,\kappa_2}$.

(3) The *equivalence* $\approx$ on $\mathcal{D}_{a,\kappa_1\to\kappa_2}$ is defined as follows: For $f_a, f_a' \in \mathcal{D}_{a,\kappa_1\to\kappa_2}$, $f_a \approx f_a'$ holds if for any $x_a \in \mathcal{D}_{a,\kappa_1}$ that is $\approx$-equivalent to the α-image of a compatible element, $x_a \cong \alpha(x_c)$ for some $x_c \in \mathcal{D}_{c,\kappa_1}$, we have $f_a\ x_a \approx f_a'\ x_a$.

Before studying the definition in more detail, we consider an example.

18.4.3 Example

Consider kind $o \to o$. A function $f_c \in \mathcal{D}_{c,o\to o}$ is compatible if for any two $x_c, x_c \in \mathcal{D}_{c,o}$, $\alpha(x_c) = \alpha(x_c)$ implies $\alpha(f_c\ x_c) = \alpha(f_c\ x_c')$. Note that this is just the second part of the definition of compatibility, since the first one is trivially satisfied. Also, the fact that $\approx$ is just equality on $\mathcal{D}_{c,o}$ and that every element of $\mathcal{D}_{c,o}$ is compatible play a rule here.

Such a compatible function f_c can be abstracted, obtaining $\alpha(f_c)$, the function that takes a value $x_a \in \mathcal{D}_{a,o}$ and returns $\alpha(f_c\ x_c)$, where $x_c \in \mathcal{D}_{c,o}$ satisfies $\alpha(x_c) = x_a$. If no such x_c exists, the return value is $\bot_{a,o}$. This in particular means that $\alpha(f_c)\ \alpha(x_c) = \alpha(f_c\ x_c)$ holds for all $x_c \in \mathcal{D}_{c,o}$.

Two functions $f_a, f_a \in \mathcal{D}_{a,o\to o}$ are $\approx$-equivalent if for any $\alpha(x_c)$, their function values coincide.

This characterization can be generalized easily for any kind of order one with arbitrary arity. A concrete function is compatible if swapping its arguments for ones that have the same image under α does not change the α-image of the function value. The abstraction of the function takes a preimage for each abstract value, applies the function, and abstracts the result. Two abstract functions are $\approx$-equivalent if their function values coincide on the image of α.

We turn back to discussing the intricacies of Definition 18.4.2. It is not hard to verify using induction that $\approx$ is indeed an equivalence. However, it quickly turns our there is an issue with the well-definedness of α for compatible functions f_c. The value of $\alpha(f_c)\ x_a$ depends on which preimage x_c we chose. However, assume x_c, x_c' are both compatible elements with $\alpha(x_c) \approx \alpha(x_c') \approx x_a$. Because f_c is compatible, we get $\alpha(f_c\ x_c) \approx \alpha(f_c\ x_c')$. Even if we chose a different preimage for x_a, we end up with an $\approx$-equivalent result. While $\alpha(f_c)$ is technically not well-defined, it is at

least well-defined modulo applying the equivalence on the function domain. We will take care of this issue by making sure that we never compare the function values of $a(f_c)$ using equality. Instead, we always use $\approx$-equivalence in the following.

There is another issue. Recall that $\mathcal{D}_{a,\kappa_1\to\kappa_2}$ is defined to be the set of join-continuous functions from $\mathcal{D}_{a,\kappa_1}$ to $\mathcal{D}_{a,\kappa_2}$. Hence, $a(f_c)$ should be join-continuous, which in particular implies that it should be monotonic. From the definition, it is obvious that monotonicity does not hold: If we feed $a(f_c)$ an abstract value x_a that is not $\approx$-equivalent to some $a(x_c)$, then the result $a(f_c)\,x_a$ will be the least element of $\mathcal{D}_{a,\kappa_1}$. However, we will later show that $a(f_c)$ is indeed join-continuous and hence monotonic if we restrict ourselves to applying it to values of the shape $a(x_c)$ where x_c is compatible. As in the above case, we will circumvent the issue by making sure that we never use the monotonicity or join-continuity of $a(f_c)$ for arguments that are not of this shape.

One could solve both problems – the well-definedness of $a(f_c)$ and its join-continuity – by modifying the abstract domain. Instead of considering $\mathcal{D}_{a,\kappa}$, we should only consider its subset $a(\{x_c \in \mathcal{D}_{c,\kappa} \mid x_c \text{ compatible}\})$, the image of a on compatible elements, and we should additionally factorize modulo the equivalence relation $\approx$. Not taking this approach will require us to be more careful, but it will also improve readability.

Basic properties

Later, we will put additional requirements on a to be able to prove exact-fixed point transfer. For now, we will show some properties of the notions that we have introduced that do not require further restrictions. The first one is that the equivalence $\approx$ is well-behaved with respect to forming limits: The join of an ascending chain remains the same (modulo $\approx$) if we exchange the elements of the chain for $\approx$-equivalent ones.

18.4.4 Lemma
If $(x_{a,i})_{i\in\mathbb{N}}$ and $(x'_{a,i})_{i\in\mathbb{N}}$ are ascending chains in $\mathcal{D}_{a,\kappa}$ for some κ such that $x_{a,i} \approx x'_{a,i}$ for all i, then $\bigsqcup_{i\in\mathbb{N}} x_{a,i} \approx \bigsqcup_{i\in\mathbb{N}} x'_{a,i}$.

Proof:
We proceed by induction on the kind κ. In the base case, the statement is trivial since $\approx$ is equality for kind ground.

Consider kind $\kappa_1 \to \kappa_2$, and let $(f_{a,i})_{i\in\mathbb{N}}$ and $(f'_{a,i})_{i\in\mathbb{N}}$ be two ascending chains from $\mathcal{D}_{a,\kappa_1\to\kappa_2}$ such that $f_{a,i} \approx f'_{a,i}$ for all i. The join of these chains is a function, and the definition of $\approx$ for functions requires us to consider $x_a \in \mathcal{D}_{a,\kappa_1}$ such that $x_a \approx a(x_c)$ for some compatible x_c. We have to argue that

$$\Big(\bigsqcup_{i\in\mathbb{N}} f_{a,i}\Big)\,x_a \approx \Big(\bigsqcup_{i\in\mathbb{N}} f'_{a,i}\Big)\,x_a$$

holds. Using the definition of the join on the function domain, this simplifies to showing

$$\bigsqcup_{i\in\mathbb{N}}(f_{a,i}\ x_a) \approx \bigsqcup_{i\in\mathbb{N}}(f'_{a,i}\ x_a)\,.$$

Since $f_{a,i} \approx f'_{a,i}$, we have $f_{a,i}\ x_a \approx f'_{a,i}\ x_a$ for all i. Hence, we may apply induction to conclude the desired statement. ■

The other result tackles the following problem: The definition of $\approx$ essentially guarantees that $f_a \approx f'_a$ implies $f_a\ x_a \approx f'_a\ x_a$. However, we will later also need that $\approx$ is a congruence, i.e. that $f_a\ x_a \approx f_a\ x'_a$ if $x_a \approx x'_a$. We then get the congruence property that will be required: If $f_a \approx f'_a$ and $x_a \approx x'_a$, then $f_a\ x_a \approx f'_a\ x'_a$.

Unfortunately, $\approx$ is not a congruence on $\mathcal{D}_{a,\kappa}$ in general. Instead, we define the notion of *permeable* elements of $\mathcal{D}_{a,\kappa}$ such that $\approx$ is a congruence on these elements. Our definition of permeability is again by induction on the kind.

For kind ground, an element $x_a \in \mathcal{D}_{a,o}$ is permeable if $x_a = \alpha(x_c)$ for some compatible $x_c \in \mathcal{D}_{c,o}$. (Since $\approx$ is equality for ground kind, this is equivalent to requiring $x_a \approx \alpha(x_c)$.)

A function $f_a \in \mathcal{D}_{a,\kappa_1\to\kappa_2}$ is permeable if (1) $f_a \approx \alpha(f_c)$ for some $f_c \in \mathcal{D}_{c,\kappa_1\to\kappa_2}$, (2) for any permeable $x_a \in \mathcal{D}_{\kappa_1}$, $f_a\ x_a \in \mathcal{D}_{a,\kappa_2}$ is permeable, and (3) for permeable $x_a, x'_a \in \mathcal{D}_{\kappa_1}$ with $x_a \approx x'_a$, we have $f_a\ x_a \approx f_a\ x'_a$.

Part (3) of the definition of permeability is exactly the congruence with respect to $\approx$ that we will need later. We show that permeability is well-behaved with respect to $\approx$.

18.4.5 Lemma
If f_a is permeable and $f_a \approx f'_a$, then so is f'_a.

Proof:
We proceed by induction on the kind, where the base case is trivial since $\approx$ is equality for ground kind. Consider functions $f_a, f'_a \in \mathcal{D}_{a,\kappa_1\to\kappa_2}$ such that f_a is permeable and $f_a \approx f'_a$. We argue that f'_a is permeable. For (1), we have $f'_a \approx f_a \approx \alpha(f_c)$ for some compatible f_c using the fact that f_a is permeable and the transitivity of $\approx$. For (2), consider a permeable $x_a \in \mathcal{D}_{\kappa_1}$. We have $f_a\ x_a \approx f'_a\ x_a$, where the former value is permeable by the assumption that f_a is permeable. We apply induction to get that also the latter value is permeable. For (3), consider permeable $x_a, x'_a \in \mathcal{D}_{\kappa_1}$ with $x_a \approx x'_a$. We have $f'_a\ x_a \approx f_a\ x_a \approx f_a\ x'_a \approx f'_a\ x'_a$, where we use the definition of the equivalence $\approx$ twice and the fact that f_a is assumed to be permeable. ■

The notion of permeability may give the impression that it introduces a new proof obligation on the user of the framework. However, this is not the case: We will show that any α-image $\alpha(x_c)$ of a compatible x_c is permeable. By Lemma 18.4.5, this also means that any element that

is $\approx$-equivalent to an α-image is permeable. Before we can formally prove this, we need to state some additional requirements on α that we will need to make in the rest of this section.

Precision

The exact fixed-point transfer will require α to have some special properties.

18.4.6 Definition
We call an abstraction function $\alpha\colon \mathcal{D}_{c,o} \to \mathcal{D}_{a,o}$ *precise* if (P1) $\alpha(\bot_{c,o}) = \bot_{a,o}$, (P2) α is join-continuous on $\mathcal{D}_{c,o}$: If $(x_{c,i})_{i\in\mathbb{N}}$ is an ascending chain in $\mathcal{D}_{c,o}$, then $\alpha(\bigsqcup_{i\in\mathbb{N}} x_{c,i}) = \bigsqcup_{i\in\mathbb{N}} \alpha(x_{c,i})$, (P3) for each terminal s, $\mathcal{I}_c(s)$ is compatible, (P4) for each terminal s, $\alpha(s^{\mathcal{I}_c}) \approx s^{\mathcal{I}_a}$.

The first requirement is that α maps the least element of the concrete domain to the least element of the abstract domain. The second is that α is join-continuous. Since the least fixed point is, by Kleene's theorem, the join of an ascending chain that starts with the bottom element, requiring these properties is not unexpected. Note that we only require these properties for kind ground. We will show that this is sufficient to be able to lift them to arbitrary the domains for arbitrary kinds.

The latter two requirements are that the concrete interpretation of the terminals are compatible and that under α, they are $\approx$-equivalent to the abstract interpretations. Recall that we have required terminals to be first order. This means that the definitions if α and $\approx$ are simple, as we have explained in Example 18.4.3. We lift the latter two properties from the terminals to arbitrary HORS terms later. This lifting result be the key ingredient for proving exact fixed-point transfer. At the end of the section, we will give an overview of what the user of the framework has to show to be able to use it.

We start by considering the properties of $\bot_{c,\kappa}$ and $\bot_{a,\kappa}$, lifting Property (P1) in the process.

18.4.7 Lemma
Assume that α satisfies Property (P1). For each kind κ, $\bot_{c,\kappa}$ is compatible and $\alpha(\bot_{c,\kappa}) = \bot_{a,\kappa}$.

Proof:
We proceed by induction on the kind κ. In the base case, compatibility of $\bot_{c,o}$ is trivial. Property (P1) guarantees $\alpha(\bot_{c,o}) = \bot_{a,o}$.

Consider kind $\kappa_1 \to \kappa_2$. For compatibility, consider any elements $x_c, x'_c \in \mathcal{D}_{c,\kappa_1}$. We have that $\bot_{c,\kappa_1\to\kappa_2}\ x_c = \bot_{c,\kappa_2}$ is compatible by induction. Similarly, we have $\alpha(\bot_{c,\kappa_1\to\kappa_2}\ x_c) = \alpha(\bot_{c,\kappa_2}) = \alpha(\bot_{c,\kappa_1\to\kappa_2}\ x'_c)$ as required.

To see that $\alpha(\bot_{c,\kappa_1\to\kappa_2})$ is $\bot_{a,\kappa_1\to\kappa_2}$, the function that sends any element $x_a \in \mathcal{D}_{a,\kappa_1}$ to $\bot_{a,\kappa_2}$, consider such an x_a. If x_a is not $\approx$-equivalent to the α-image of some compatible $x_c \in \mathcal{D}_{c,\kappa_1}$, $\alpha(\bot_{c,\kappa_1\to\kappa_2})\ x_a = \bot_{a,\kappa_2}$ holds by definition. Otherwise, we have $\alpha(\bot_{c,\kappa_1\to\kappa_2})\ x_a = \alpha(\bot_{c,\kappa_1\to\kappa_2}\ x_c) = \alpha(\bot_{c,\kappa_2}) = \bot_{a,\kappa_2}$ by induction. ■

With this lemma, we can also tie up a loose end from earlier. It enables us to show that $\alpha(f_c)$ is always permeable. Hence, permeability is not an additional proof obligation; we get it for free for any abstract element that is in the image of α.

18.4.8 Lemma
Assume that α satisfies Property (P1). For any $f_c \in \mathcal{D}_{c,\kappa}$, $\alpha(f_c) \in \mathcal{D}_{a,\kappa}$ is permeable.

Proof:
We proceed by induction on the kind κ. We additionally prove that $\bot_{a,\kappa}$ is permeable. For kind o, permeability just means being in the α-image, which is satisfied by assumption. The least element $\bot_{a,\kappa}$ is in the image of α by Property (P1).

Consider kind $\kappa_1 \to \kappa_2$. We first argue that $\bot_{a,\kappa_1\to\kappa_2}$ is permeable. By Lemma 18.4.7, it is in the image of α. For Part (2) of the definition of permeability, consider a permeable $x_a \in \mathcal{D}_{a,\kappa_1}$ and note that $\bot_{a,\kappa_1\to\kappa_2}\ x_a = \bot_{a,\kappa_2}$, which is permeable by induction. Similarly, for Part (3), consider permeable $x_a, x_a' \in \mathcal{D}_{a,\kappa_1}$ and we get $\bot_{a,\kappa_1\to\kappa_2}\ x_a = \bot_{a,\kappa_2} = \bot_{a,\kappa_1\to\kappa_2}\ x_a'$.

Let us now consider $f_c \in \mathcal{D}_{c,\kappa_1\to\kappa_2}$. We need to prove that $\alpha(f_c)$ is permeable, where the first part of the definition is trivially satisfied. If f_c is not compatible, $\alpha(f_c) = \bot_{a,\kappa_1\to\kappa_2}$, and we have just argued that this value is permeable. Let us assume that f_c is compatible, and let $x_a \approx x_a'$ be permeable values. This in particular means $x_a \approx x_a' \approx \alpha(x_c)$ for some compatible x_c. Consider $\alpha(f_c)\ x_a \approx \alpha(f_c\ x_c)$, using the definition of $\alpha(f_c)$. The latter value is permeable by induction, and so is the former by Lemma 18.4.5. For Part (3) of the definition, note that $\alpha(f_c)\ x_a \approx \alpha(f_c\ x_c) \approx \alpha(f_c)\ x_a'$ as required. ■

As a next step, we lift Property (P2), the join-continuity of α, beyond kind ground. To be precise, we show a restricted version of join-continuity. We only consider ascending chains of compatible elements, and instead of showing that the join of the function value equals the function value of the join, we show that they are $\approx$-equivalent. For the proof by induction to work, we need to strengthen the induction hypothesis and also show that the limit of a chain of compatible elements is compatible.

18.4.9 Lemma
Assume that α satisfies the Properties (P1) and (P2) For all κ: The join $\bigsqcup_{i\in\mathbb{N}} f_{c,i}$ of an ascending chain of compatible elements $(f_{c,i})_{i\in\mathbb{N}}$ from $\mathcal{D}_{c,\kappa}$ is compatible, and α is join-continuous on such chains: $\alpha(\bigsqcup_{i\in\mathbb{N}} f_{c,i}) \approx \bigsqcup_{i\in\mathbb{N}} \alpha(f_{c,i})$.

Proof:
We proceed by induction on κ. Any element of $\mathcal{D}_{c,o}$ is compatible by definition, which in particular applies to the join of a chain. The join continuity of a is simply Property (P2).

Consider kind $\kappa_1 \to \kappa_2$, and let $\bigsqcup_{i\in\mathbb{N}} f_{c,i}$ be an ascending chain of compatible elements $(f_{c,i})_{i\in\mathbb{N}}$ from $\mathcal{D}_{c,\kappa_1\to\kappa_2}$.

For the first part of the definition of compatibility, consider a compatible $x_c \in \mathcal{D}_{c,\kappa_1}$ We have $(\bigsqcup_{i\in\mathbb{N}} f_{c,i})\ x_c = \bigsqcup_{i\in\mathbb{N}}(f_{c,i}\ x_c)$ by the definition of the join on the function domain $\mathcal{D}_{c,\kappa_1\to\kappa_2}$. Since x_c is compatible and each $f_{c,i}$ is compatible, then so is each $f_{c,i}\ x_c$. Additionally, the $f_{c,i}$ forming an ascending chain implies their function values $f_{c,i}\ x_c$ forming an ascending chain. Induction proves that $\bigsqcup_{i\in\mathbb{N}}(f_{c,i}\ x_c)$ is compatible as desired.

For the second part, let $x_c, x_c' \in \mathcal{D}_{c,\kappa_1}$ be compatible so that $a(x_c) \approx a(x_c')$. We need to prove

$$a\Big(\big(\bigsqcup_{i\in\mathbb{N}} f_{c,i}\big)\ x_c\Big) \ \approx\ a\Big(\big(\bigsqcup_{i\in\mathbb{N}} f_{c,i}\big)\ x_c'\Big).$$

Using the definition of the join on both sides, it would be sufficient to show

$$a\Big(\bigsqcup_{i\in\mathbb{N}}(f_{c,i}\ x_c)\Big) \ \approx\ a\Big(\bigsqcup_{i\in\mathbb{N}}(f_{c,i}\ x_c')\Big).$$

As in the first part of the proof, both $(f_{c,i}\ x_c)_{i\in\mathbb{N}}$ and $(f_{c,i}\ x_c')_{i\in\mathbb{N}}$ are ascending chains of compatible elements in $\mathcal{D}_{c,\kappa_2}$. We apply induction and use the join-continuity of a on $\mathcal{D}_{c,\kappa_2}$ to get the desired statement.

For join-continuity, we need to show $a(\bigsqcup_{i\in\mathbb{N}} f_{c,i}) \approx \bigsqcup_{i\in\mathbb{N}} a(f_{c,i})$. By the definition of $\approx$, this means we need to consider some $x_a \in \mathcal{D}_{\kappa_1}$ such that $x_a \approx a(x_c)$ for some compatible x_c. We prove $a(\bigsqcup_{i\in\mathbb{N}} f_{c,i})\ x_a \approx (\bigsqcup_{i\in\mathbb{N}} a(f_{c,i}))\ x_a$. By the definition of a for functions, we have $a(\bigsqcup_{i\in\mathbb{N}} f_{c,i})\ x_a \approx a((\bigsqcup_{i\in\mathbb{N}} f_{c,i})\ x_c)$. By the definition of the join, this value equals $a(\bigsqcup_{i\in\mathbb{N}}(f_{c,i}\ x_c))$. As before, $(f_{c,i}\ x_c)_{i\in\mathbb{N}}$ is an ascending chain of compatible elements, so we apply induction to get $a\big(\bigsqcup_{i\in\mathbb{N}}(f_{c,i}\ x_c)\big) \approx \bigsqcup_{i\in\mathbb{N}} a(f_{c,i}\ x_c)$. Starting from the other side of the desired equality we get $(\bigsqcup_{i\in\mathbb{N}} a(f_{c,i}))\ x_a = \bigsqcup_{i\in\mathbb{N}}(a(f_{c,i})\ x_a)$ using the definition of the join. By the definition of a, we have $a(f_{c,i})\ x_a \approx a(f_{c,i}\ x_c)$, which completes the proof. ■

The previous lemma allows us to tie up another loose end. At the beginning of the section, we have claimed that $a(f_c)$ is join-continuous on the a-images of compatible elements. We can now prove this statement.

Let $f_c \in \mathcal{D}_{c,\kappa_1\to\kappa_2}$ be a function. Consider an ascending chain $(x_{c,i})_{i\in\mathbb{N}}$ in $\mathcal{D}_{c,\kappa_1}$ of compatible elements. By Lemma 18.4.9, a is join-continuous and hence monotonic. We define $x_{a,i} = a(x_{c,i})$ for each i and obtain that $(x_{a,i})_{i\in\mathbb{N}}$ is an ascending chain in $\mathcal{D}_{a,\kappa_1}$. We prove that $a(f_c)\ (\bigsqcup_{i\in\mathbb{N}} x_{a,i}) \approx \bigsqcup_{i\in\mathbb{N}}(a(f_c)\ x_{a,i})$. By the join-continuity of a, we get that the join $\bigsqcup_{i\in\mathbb{N}} x_{a,i}$ of the $x_{a,i}$ is $\approx$-equivalent to the join of the $x_{a,i}$, $\bigsqcup_{i\in\mathbb{N}} x_{a,i} \approx a(\bigsqcup_{i\in\mathbb{N}} x_{c,i})$. Hence, the definition

of $a(f_c)$ gives us $a(f_c)\,(\bigsqcup_{i\in\mathbb{N}} x_{a,i}) \approx a(f_c\,(\bigsqcup_{i\in\mathbb{N}} x_{c,i}))$. The function f_c is join-continuous by our definition of $\mathcal{D}_{c,\kappa_1\to\kappa_2}$, so we can transform the latter expression to $a(\bigsqcup_{i\in\mathbb{N}} f_c\ x_{c,i})$. We use the join-continuity of a and the definition of $a(f_c)$ to get $\bigsqcup_{i\in\mathbb{N}} a(f_c\ x_{c,i}) \approx \bigsqcup_{i\in\mathbb{N}}(a(f_c)\ x_{a,i})$, which proves the desired statement.

Exact fixed-point transfer

We hone in on proving exact fixed-point transfer. The key step is lifting the Properties (P3) and (P4) from the definition of precision from the nonterminals to arbitrary HORS terms.

Recall that the semantics $\mathcal{M}[\![t]\!]$ of a HORS term t is a function with signature $(N \uplus V \to_p \mathcal{D}) \to \mathcal{D}$ that expects a valuation, an assignment of values to the nonterminals and the HORS variables that are free in t. We extend some of the notions that we have defined in this section to valuations in the expected way. A concrete valuation $v_c\colon N \uplus V \to_p \mathcal{D}_c$ is compatible if $v_c(x)$ is compatible for all x on which v_c is defined. For such a concrete valuation v_c, $a(v_c)\colon N \uplus V \to_p \mathcal{D}_a$ is the abstract valuation with $a(v_c)(x) = a(v_c(x))$ for all x on which v_c is defined.

We can now prove the proposition that shows that $\mathcal{M}_c[\![t]\!]v$ is compatible and that its a-image is $\approx$-equivalent to $\mathcal{M}_a[\![t]\!]a(v)$. We also prove that if we modify the valuation $a(v)$ by exchanging its entries for $\approx$-equivalent ones, we obtain an $\approx$-equivalent result.

18.4.10 Proposition
Assume a to be precise. Let t be a term and v_c be a compatible valuation.

a) $\mathcal{M}_c[\![t]\!]v_c$ is compatible.

b) $a(\mathcal{M}_c[\![t]\!]v_c) \approx \mathcal{M}_a[\![t]\!]a(v_c)$.

c) If an abstract valuation $v_a\colon N \uplus V \to_p \mathcal{D}_a$ satisfies $v_a \approx a(v_c)$, meaning $v_a(x) \approx a(v_c)(x)$ for all x, then $\mathcal{M}_a[\![t]\!]a(v_c) \approx \mathcal{M}_a[\![t]\!]v_a$.

Proof:
We proceed by induction on the structure of the term t and prove all statements simultaneously.

Base case $t = s$ for a terminal s. We have $\mathcal{M}_c[\![t]\!]v_c = s^{\mathcal{I}_a}$. For Part a), note that compatibility is simply Property (P3). For Part b), we have $a(\mathcal{M}_c[\![t]\!]v_c) = a(s^{\mathcal{I}_a}) \approx s^{\mathcal{I}_a} = \mathcal{M}_a[\![t]\!]a(v_c)$ using Property (P4) and the fact that the valuation does not play a role when evaluating a terminal. The latter fact also proves $\mathcal{M}_a[\![s]\!]a(v_c) \approx \mathcal{M}_a[\![s]\!]v_a$.

Base case $t = F$ for a nonterminal F. The value $\mathcal{M}_c[\![F]\!]v_c = v_c(F)$ is compatible since v_c is compatible. We have $a(\mathcal{M}_c[\![F]\!]v_c) = a(v_c(F)) = a(v_c)(F) = \mathcal{M}_a[\![F]\!]a(v_c)$ using the definition of $a(v_c)$. Finally, $\mathcal{M}_a[\![F]\!]a(v_c) = a(v_c)(F) \approx v_a(F) = \mathcal{M}_a[\![F]\!]v_a$ since $a(v_c) \approx v_a$ by assumption.

Base case $t = x$ for a HORS variable x. The reasoning is exactly the same as for $t = F$.

Inductive step, $t = \lambda x.t_1$. Observe that $\mathcal{M}_c[\![\lambda x.t_1]\!]v_c$ is a function that takes x_c and returns $\mathcal{M}_c[\![t_1]\!](v_c[x \mapsto x_c])$.

For Part a), we prove compatibility using its definition. Let x_c be compatible, then the value $\mathcal{M}_c[\![\lambda x.t_1]\!]v_c\ x_c$ equals $\mathcal{M}_c[\![t_1]\!](v_c[x \mapsto x_c])$. Since both v_c and x_c are compatible, $v_c[x \mapsto x_c]$ is a compatible valuation. We apply Part a) of the proposition using induction and obtain that $\mathcal{M}_c[\![t_1]\!](v_c[x \mapsto x_c])$ is compatible as required. Consider compatible x_c, x_c' with $a(x_c) \approx a(x_c')$. We have $a(\mathcal{M}_c[\![\lambda x.t_1]\!]v_c\ x_c) = a(\mathcal{M}_c[\![t_1]\!](v_c[x \mapsto x_c])) \approx \mathcal{M}_a[\![t_1]\!]a(v_c[x \mapsto x_c])$ by using Part b) of the induction hypothesis. Similarly, $a(\mathcal{M}_c[\![\lambda x.t_1]\!]v_c\ x_c') \approx \mathcal{M}_a[\![t_1]\!]a(v_c[x \mapsto x_c'])$. We have $a(v_c[x \mapsto x_c]) \approx a(v_c[x \mapsto x_c'])$ since $a(x_c) \approx a(x_c')$ so we can use Part c) of the induction hypothesis to conclude $\mathcal{M}_a[\![t_1]\!]a(v_c[x \mapsto x_c]) \approx \mathcal{M}_a[\![t_1]\!]a(v_c[x \mapsto x_c'])$ as desired.

For Part b), consider $a(\mathcal{M}_c[\![\lambda x.t_1]\!]v_c)$. We need to show that it is $\approx$-equivalent to $\mathcal{M}_a[\![\lambda x.t_1]\!]a(v_c)$. The latter is a function that takes x_a and returns $\mathcal{M}_a[\![t_1]\!]a(v_c)[x \mapsto x_a]$. We check the definition of $\approx$ for functions by considering x_a with $x_a \approx a(x_c)$ for some compatible x_c. We have

$$\begin{aligned} a(\mathcal{M}_c[\![\lambda x.t_1]\!]v_c)\ x_a &\approx a(\mathcal{M}_c[\![\lambda x.t_1]\!]v_c\ x_c) \\ &= a(\mathcal{M}_c[\![t_1]\!](v_c[x \mapsto x_c])) \\ &\approx \mathcal{M}_a[\![t_1]\!]a(v_c[x \mapsto x_c]) \\ &= \mathcal{M}_a[\![t_1]\!](a(v_c)[x \mapsto a(x_c)]) \end{aligned}$$

using the definition of a, the definition of the interpretation of lambda abstraction, and the induction hypothesis. The last step is simply evaluating $a(v_c[x \mapsto x_c])$. We now observe that $a(v_c)[x \mapsto a(x_c)] \approx a(v_c)[x \mapsto x_a]$ since $x_a \approx a(x_c)$. We use Part c) of the induction hypothesis and get

$$\mathcal{M}_a[\![t_1]\!](a(v_c)[x \mapsto a(x_c)]) \approx \mathcal{M}_a[\![t_1]\!](a(v_c)[x \mapsto x_a]) = \mathcal{M}_a[\![\lambda x.t_1]\!]a(v_c)\ x_a$$

as desired.

For Part c), we compare $\mathcal{M}_a[\![\lambda x.t_1]\!]a(v_c)$ and $\mathcal{M}_a[\![\lambda x.t_1]\!]v_a$. Consider x_a such that $x_a \approx a(x_c)$ for some compatible x_c. We have $\mathcal{M}_a[\![\lambda x.t_1]\!]v_a\ x_a = \mathcal{M}_a[\![t_1]\!](v_a[x \mapsto x_a])$ and similarly $\mathcal{M}_a[\![\lambda x.t_1]\!]a(v_c)\ x_a = \mathcal{M}_a[\![t_1]\!](a(v_c)[x \mapsto x_a])$. We have $a(v_c)[x \mapsto x_a] \approx a(v_c[x \mapsto x_c])$, so we apply the induction hypothesis to get $\mathcal{M}_a[\![t_1]\!](a(v_c)[x \mapsto x_a]) \approx \mathcal{M}_a[\![t_1]\!]a(v_c[x \mapsto x_c])$. We now use $a(v_c[x \mapsto x_c]) \approx v_a[x \mapsto x_a]$ and apply the induction hypothesis again to get $\mathcal{M}_a[\![t_1]\!]a(v_c[x \mapsto x_c]) \approx \mathcal{M}_a[\![t_1]\!](v_a[x \mapsto x_a])$ as desired.

Inductive step, $t = t_1\ t_2$. For Part a), we have

$$\mathcal{M}_c[\![t_1\ t_2]\!]v_c = \mathcal{M}_c[\![t_1]\!]v_c\ \mathcal{M}_c[\![t_2]\!]v_c$$

using the definition of the interpretation of function application. Both $\mathcal{M}_c[\![t_1]\!]v_c$ and $\mathcal{M}_c[\![t_2]\!]v_c$ are compatible by induction. The value $\mathcal{M}_c[\![t_1]\!]v_c$ is a function, and using the definition of compatibility for functions, the function value of a compatible function at a compatible argument is again compatible. We obtain that $\mathcal{M}_c[\![t_1]\!]v_c\ \mathcal{M}_c[\![t_2]\!]v_c$ is compatible.

For Part b), we get

$$\begin{aligned} a(\mathcal{M}_c[\![t_1\ t_2]\!]v_c) &= a(\mathcal{M}_c[\![t_1]\!]v_c\ \mathcal{M}_c[\![t_2]\!]v_c) \\ &\approx a(\mathcal{M}_c[\![t_1]\!]v_c)\ \mathcal{M}_a[\![t_2]\!]a(v_c) \end{aligned}$$

where the last transformation requires some explanation. By induction, we have $a(\mathcal{M}_c[\![t_2]\!]v_c) \approx \mathcal{M}_a[\![t_2]\!]a(v_c)$. Hence, $\mathcal{M}_c[\![t_2]\!]v_c$ is an a-preimage of $\mathcal{M}_a[\![t_2]\!]a(v_c)$ (modulo $\approx$) and we may use the definition of a for functions. We have $a(\mathcal{M}_c[\![t_1]\!]v_c) \approx \mathcal{M}_a[\![t_1]\!]a(v_c)$ by induction, so following the definition of $\approx$ for functions, we obtain

$$\begin{aligned} a(\mathcal{M}_c[\![t_1]\!]v_c)\ \mathcal{M}_a[\![t_2]\!]a(v_c) &\approx \mathcal{M}_a[\![t_1]\!]a(v_c)\ \mathcal{M}_a[\![t_2]\!]a(v_c) \\ &= \mathcal{M}_a[\![t_1\ t_2]\!]a(v_c)\,, \end{aligned}$$

proving the desired statement.

Finally, we consider Part c). We have

$$\mathcal{M}_a[\![t_1\ t_2]\!]a(v_c) \;=\; \mathcal{M}_a[\![t_1]\!]a(v_c)\ \mathcal{M}_a[\![t_2]\!]a(v_c)\,.$$

By induction $\mathcal{M}_a[\![t_1]\!]a(v_c) \approx \mathcal{M}_a[\![t_1]\!]v_a$ and $\mathcal{M}_a[\![t_2]\!]a(v_c) \approx \mathcal{M}_a[\![t_2]\!]v_a$ hold. Also by induction, we know that $\mathcal{M}_a[\![t_2]\!]a(v_c)$ is $\approx$ related to an element in the image of a (namely $\mathcal{M}_c[\![t_2]\!]v_c$), so we can use the definition of $\approx$ for functions and get

$$\mathcal{M}_a[\![t_1]\!]a(v_c)\ \mathcal{M}_a[\![t_2]\!]a(v_c) \;\approx\; \mathcal{M}_a[\![t_1]\!]v_a\ \mathcal{M}_a[\![t_2]\!]a(v_c)\,.$$

Have you noticed that we have not used the concept of permeability yet? The value $a(\mathcal{M}_c[\![t_1]\!]v_c)$ is permeable by Lemma 18.4.8. Since $a(\mathcal{M}_c[\![t_1]\!]v_c) \approx \mathcal{M}_a[\![t_1]\!]a(v_c) \approx \mathcal{M}_a[\![t_1]\!]v_a$, so is $\mathcal{M}_a[\![t_1]\!]v_a$ by Lemma 18.4.5. Similarly, $a(\mathcal{M}_c[\![t_2]\!]v_c)$ is permeable by Lemma 18.4.8, and both $\mathcal{M}_a[\![t_2]\!]a(v_c)$ and $\mathcal{M}_a[\![t_2]\!]v_a$ are permeable because they are $\approx$-equivalent to this value. We use Property (3) from the definition of permeability to conclude

$$\mathcal{M}_a[\![t_1]\!]v_a\ \mathcal{M}_a[\![t_2]\!]a(v_c) \;\approx\; \mathcal{M}_a[\![t_1]\!]v_a\ \mathcal{M}_a[\![t_2]\!]v_a \;=\; \mathcal{M}_a[\![t_1\ t_2]\!]v_a\,.$$

This completes the proof. ■

The previous proposition enables us to prove our main result, exact fixed-point transfer.

18.4.11 Theorem
If α is precise, then $\mathrm{sol}_a \approx \alpha(\mathrm{sol}_c)$.

Here, we use the notation for valuations, i.e. $\mathrm{sol}_a \approx \alpha(\mathrm{sol}_c)$ means that for every nonterminal F, we have $\mathrm{sol}_a(F) \approx \alpha(\mathrm{sol}_c)(F) = \alpha(\mathrm{sol}_c(F))$. This result also extends to arbitrary terms t: $\mathrm{sol}_a(t) \approx \alpha(\mathrm{sol}_c(t))$. If t is variable-closed and of kind o, we even get $\mathrm{sol}_a(t) = \alpha(\mathrm{sol}_c(t))$ because $\approx$ is simply equality on $\mathcal{D}_{a,o}$. We state this result as a corollary.

18.4.12 Corollary
If t is a variable-closed term of kind ground, then $\mathrm{sol}_a(t) = \alpha(\mathrm{sol}_c(t))$.

Note that the corollary in articular applies to the initial nonterminal S of the scheme, which we have required to be of kind o.

To finish this section, it remains to prove the theorem.

Proof of Theorem 18.4.11.:
Firstly, we show

$$\alpha\left(\mathrm{sol}_c^i\right) \approx \mathrm{sol}_a^i$$

for every $i \in \mathbb{N}$ using induction. We also argue that each sol_c^i is compatible.

In the base case, consider $i = 0$. For a nonterminal F of kind κ, $\mathrm{sol}_a^0(F) = \bot_{a,\kappa}$, as the 0^{th} approximant is the least element in every component. Similarly, $\alpha(\mathrm{sol}_c^0(F)) = \alpha(\bot_{c,\kappa}) = \bot_{a,\kappa}$ using Lemma 18.4.7. Also by Lemma 18.4.7, we get that sol_c^0 is compatible.

For the induction step, assume we have proven $\alpha(\mathrm{sol}_c^i) \approx \mathrm{sol}_a^i$ and consider $i+1$. Consider a nonterminal F of kind κ whose unique rule in the HORS is $F \to t$. By the definition of the $(i+1)^{\text{st}}$ approximant, we have $\mathrm{sol}_c^{i+1}(F) = \mathcal{M}_c[\![t]\!]\mathrm{sol}_c^i$, similar for sol_a. By induction, sol_c^i is compatible, then so is $\mathcal{M}_c[\![t]\!]\mathrm{sol}_c^i$ using Part a) of Proposition 18.4.10. We get

$$\begin{aligned}
\alpha\left(\mathrm{sol}_c^{i+1}(F)\right) &= \alpha\left(\mathcal{M}_c[\![t]\!]\mathrm{sol}_c^i\right) \\
&\approx \mathcal{M}_a[\![t]\!]\alpha\left(\mathrm{sol}_c^i\right) \\
&\approx \mathcal{M}_a[\![t]\!]\mathrm{sol}_a^i \\
&= \mathrm{sol}_a^{i+1}(F)\,.
\end{aligned}$$

Overview: Exact fixed-point transfer

Consider a deterministic HORS G, a concrete model $\mathcal{M}_c = (\mathcal{D}_c, \mathcal{I}_c)$, and an abstract model $\mathcal{M}_a = (\mathcal{D}_a, \mathcal{I}_a)$, both instantiations of the model template described in Section 18.2 (see Figure 18.2.a). Note that these models in particular provide domains $\mathcal{D}_{c,o}$ and $\mathcal{D}_{a,o}$ for kind o.
We get the existence of the least solution sol_c and sol_a of the interpreted system of equations associated to G for these models.

Precision
Assume we are given an abstraction function $\alpha\colon \mathcal{D}_{c,o} \to \mathcal{D}_{a,o}$ satisfying the following properties.

- α maps the least element $\bot_{c,o} \in \mathcal{D}_{c,o}$ to the least element $\bot_{a,o} \in \mathcal{D}_{a,o}$, $\alpha(\bot_{c,o}) = \bot_{a,o}$.
- $\alpha\colon \mathcal{D}_{c,o} \to \mathcal{D}_{a,o}$ is join-continuous.
- For each terminal symbol s of order one, say with arity k, consider $x_1, x_1', \ldots, x_k, x_k' \in \mathcal{D}_{c,o}$ so that $\alpha(x_i) = \alpha(x_i')$ for all $i \in [1,k]$.
  The concrete interpretation of s should satisfy
  $$\alpha\Big(s^{\mathcal{I}_c}\, x_1 \ldots x_k\Big) = \alpha\Big(s^{\mathcal{I}_c}\, x_1' \ldots x_k'\Big)\,.$$
- For each terminal symbol s, say with arity k, consider $x_1, \ldots, x_k \in \mathcal{D}_{c,o}$.
  The interpretations of s should satisfy
  $$\alpha\Big(s^{\mathcal{I}_c}\, x_1 \ldots x_k\Big) = s^{\mathcal{I}_a}\, \alpha(x_1) \ldots \alpha(x_k)\,.$$

Exact fixed-point transfer
Then for each variable-closed term t of kind ground, the abstraction of the value assigned to t by the concrete least solution is the value of t in the abstract least solution,

$$\alpha(\mathrm{sol}_c(t)) = \mathrm{sol}_a(t)\,.$$

Figure 18.4.a: Overview: Exact fixed-point transfer.

The first and the last equality are the above observation. The second transformation is Part b) of Proposition 18.4.10, which we can apply because sol_c^i is compatible by induction. The penultimate transformation is $\alpha(\mathrm{sol}_c^i) \approx \mathrm{sol}_a^i$ together with Part c) of Proposition 18.4.10.

Equipped with the knowledge that $\alpha(\mathrm{sol}_c^i) \approx \mathrm{sol}_a^i$ for all i, we turn to proving $\alpha(\mathrm{sol}_c) \approx \mathrm{sol}_a$. Kleene's theorem, Theorem 16.1.3, gives us

$$\mathrm{sol}_c = \bigsqcup_{i\in\mathbb{N}} \mathrm{sol}_c^i \qquad \text{and} \qquad \mathrm{sol}_a = \bigsqcup_{i\in\mathbb{N}} \mathrm{sol}_a^i \,,$$

see also Proposition 18.2.8. We get

$$\alpha(\mathrm{sol}_c) = \alpha\Big(\bigsqcup_{i\in\mathbb{N}} \mathrm{sol}_c^i\Big) \approx \bigsqcup_{i\in\mathbb{N}} \alpha\Big(\mathrm{sol}_c^i\Big) \approx \bigsqcup_{i\in\mathbb{N}} \mathrm{sol}_a^i = \mathrm{sol}_a \,.$$

The first and the last equality are Kleene's theorem. The second transformation is the join-continuity of α on chains of compatible elements, Lemma 18.4.9, together with our earlier observation that each sol_c^i is compatible. The penultimate transformation is the very first lemma in this section, Lemma 18.4.4, together with $\alpha(\mathrm{sol}_c^i) \approx \mathrm{sol}_a^i$ for all i. ■

We conclude the section with an *overview* in the form of Figure 18.4.a. It lists what a potential user of the framework has to show to guarantee that Corollary 18.4.12 holds. The contents of the overview are based on the basic definition from Definition 18.4.2, the observation of what these definitions mean for order one in Example 18.4.3, and the definition of precision in Definition 18.4.6.

18.5 Solving higher-order inclusion games

We finally discuss how to decide the regular inclusions games defined by higher-order recursion schemes. In Section 18.3, we have defined a concrete semantics for HORS games using the mode template that we have introduced in Section 18.2. In the form of Theorem 18.3.1 we have shown that the least solution to the system of equations interpreted using the concrete model provides the winner of the game. Unfortunately, this result is non-constructive. If we were able to compute the least solution, we could read off the winner, but since the domains used by the concrete semantics are infinite and do not satisfy the ascending chain condition, this is not possible.

In this section, we overcome this problem by using the framework for exact fixed-point transfer. We define an *abstract model* by again instantiating the model template. The abstract domains are finite, so it is possible to actually compute the least solution to the interpreted system of equations. Showing that the winner of the game can be read off from the least solution directly would be difficult. To this end, we use exact fixed-point transfer. We define a precise abstraction from the concrete to the abstract model and get that the abstraction of the concrete least solution is essentially the abstract least solution. This will tell us how to read off the winner of the game and prove that this approach is sound.

The abstract model

We start by defining the abstract model, using the model template from Section 18.2. We follow our overview on the model template, Figure 18.2.a. The template requires us to define a domain $\mathcal{D}_{a,o}$ for kind ground. We want to use positive Boolean formulas over a finite set of atoms. Recall that the winning condition of the game for the existential player is deriving a finite word that is in the regular language $\overline{\mathcal{L}(A)}$, the complement of the language of an NFA A. Let Q be the set of states of A. We define $\mathcal{D}_{a,o}$ to be the set of positive Boolean formulas over Q, factorized by logical equivalence and ordered by implication,

$$\mathcal{D}_{a,o} = (\mathsf{pBF}(Q)/_{\Longleftrightarrow}, \Longrightarrow) .$$

Similar to the domain that we have used in Chapter 17, this set is a CPPO. Its least element is the equivalence class of false. The join of an ascending chain of (equivalence classes of) formulas is the disjunction of the formulas. Since the set of atoms Q is finite, there are only finitely many equivalence classes. Hence, an infinite chain only consists of finitely many distinct elements and the disjunction of these elements is a well-defined finite formula. The model template now provides us with $\mathcal{D}_{a,\kappa_1\to\kappa_2} = \mathcal{D}_{a,\kappa_1} \to_{\sqcup} \mathcal{D}_{a,\kappa_1}$ for every kind $\kappa_1 \to \kappa_2$.

The idea behind using formulas over states is the following. We want to represent each word w by the set $\{q \mid q \xrightarrow{w} q_{\mathsf{final}} \in Q_{\mathsf{final}}\}$ of states from which w is accepted, i.e. the states q so that there is a run of A from q to a final state. Since our atoms are states and not sets of states, we

transform the above set into a conjunction. This means we represent w by $\bigwedge_{q \text{ s.t. } q \xrightarrow{w} q_{\text{final}} \in Q_{\text{final}}} q$. To improve readability, we write such a conjunction as $\bigwedge\{q \mid q \xrightarrow{w} q_{\text{final}} \in Q_{\text{final}}\}$ in the following.

The existential player needs to enforce the derivation of a finite word w in $\overline{\mathcal{L}(A)}$. This means that no run of A on w can be accepting. For each run of A on w, it must not be true that the run is a run from the initial to the final state. Using conjunction represents the fact that the existential player has to show that each run is non-accepting.

Following this intuition, we complete the instantiation of the model template by specifying the interpretation of the terminals. The word-end marker $\$: o$ is the representation of the empty word. Hence, it corresponds to the set of final states, which we see as conjunction,

$$\$^{\mathcal{I}_a} = \bigwedge_{q_{\text{final}} \in Q_{\text{final}}} q_{\text{final}} \in \mathcal{D}_{a,o} \,.$$

For each letter terminal $a : o \to o$ corresponding to a letter $a \in \Sigma$, we define the interpretation as

$$a^{\mathcal{I}_a} = \text{predec}_a \in \mathcal{D}_{a,o \to o} \,.$$

It takes a formula in $\mathcal{D}_{a,o}$ and distributes over conjunction and disjunction. When it reaches an atom q, it computes the a-predecessors of q in A and connects them using conjunction. Formally, the definition is

$$\begin{aligned} \text{predec}_a\big(H \overset{\wedge}{\vee} H'\big) &= \text{predec}_a\big(H \overset{\wedge}{\vee} H'\big) \\ \text{predec}_a(q) &= \bigwedge \left\{ q' \in Q \;\middle|\; q' \xrightarrow{a} q \right\} . \end{aligned}$$

It remains to define the interpretations of the terminals of the shape branch_F that are present in the determinization of the game HORS. Consider $\text{branch}_F : o \to \ldots \to o$ with arity k. If F is owned by the existential player, then the interpretation $\text{branch}_F^{\mathcal{I}_a}$ is k-ary disjunction. Otherwise, it is k-ary conjunction.

Before we can finalize the instantiation of the template, we need to show that the interpretations are join-continuous. Before doing so, we make the following observation.

18.5.1 Lemma
For each kind κ, $\mathcal{D}_{a,\kappa}$ and $(N \uplus V \to_p \mathcal{D}_a) \to_\sqcup \mathcal{D}_{a,\kappa}$ are finite.

Proof:
Since the set of states Q is finite, so is the set $\mathcal{D}_{a,o}$ of equivalence classes of formulas. The set of function from a finite domain to a finite target set is finite. The same is true for the subset of join-continuous functions. Hence, it is easy to show using induction that each $\mathcal{D}_{a,\kappa_1 \to \kappa_2} = \mathcal{D}_{a,\kappa_1} \to_\sqcup \mathcal{D}_{a,\kappa_2}$ is finite.

For $(N \uplus V \to_p \mathcal{D}_a) \to_\sqcup \mathcal{D}_{a,\kappa}$, note that we do not actually consider arbitrary valuations. We only consider valuations ν that assign to each HORS variable or nonterminal of kind κ a value from $\mathcal{D}_{a,\kappa}$. As we have just argued, each $\mathcal{D}_{a,\kappa}$ is finite. Since there are only finitely many HORS variables and nonterminals, the set of valuations that respect the kinds is finite. Hence, there are only finite many functions with signature $(N \uplus V \to_p \mathcal{D}_a) \to_\sqcup \mathcal{D}_{a,\kappa}$. ■

The finiteness of each $\mathcal{D}_{a,\kappa}$ allows us to use Remark 16.1.4. For a finite domain, join-continuity and monotonicity are equivalent. In order to show that the interpretations are values in their respective domain, we need to show that they are monotonic (and hence join-continuous) functions. The interpretation of the word-end marker is just a value and not a function, so there is nothing to do. The interpretation of each branch_F is conjunction or disjunction, and we have argued that these functions are monotonic in Section 17.2. It remains to show that predec_a is monotonic, which we prove in the form of the following lemma.

18.5.2 Lemma
If $H, H' \in \mathcal{D}_{a,o}$ with $H \implies H'$, then $\text{predec}_a(H) \implies \text{predec}_a(H')$.

Proof:
The proof is similar to the proof of Lemma 17.2.6 and uses the same observations on the equivalence of Boolean formulas.

We proceed by a nested induction. The outer induction is an induction on the structure of H. In its base case, $H = q$ is an atom. We proceed using an induction on the structure of H'. In the base case, $H' = q'$ is an atom, too. The implication $H \implies H'$ can only hold if $q = q'$. In this case $\text{predec}_a(H) = \text{predec}_a(H')$ holds and we get the desired implication.

In the induction step of the inner induction, consider $H' = H'_1 \wedge H'_2$. Since $H \implies H'$, we get both $H \implies H'_1$ and $H \implies H'_1$ using the 2nd Observation from the proof of Lemma 17.2.6. Using induction, we obtain $\text{predec}_a(H) \implies \text{predec}_a(H'_1)$ and $\text{predec}_a(H) \implies \text{predec}_a(H'_2)$. Using the observation again, we conclude $\text{predec}_a(H) \implies \text{predec}_a(H'_1) \wedge \text{predec}_a(H'_2) = \text{predec}_a(H'_2)$ as desired. If H' is a disjunction, the argumentation is similar.

In the induction step of the outer induction, consider $H = H_1 \wedge H_2$ and H' arbitrary. Using the 3rd Observation from the proof of Lemma 17.2.6, we obtain that $H \implies H'$ implies $H_1 \implies H'$ or $H_2 \implies H'$. With induction, we get $\text{predec}_a(H_1) \implies \text{predec}_a(H')$ or $\text{predec}_a(H_2) \implies \text{predec}_a(H')$. Applying the observation again gives us $\text{predec}_a(H) = \text{predec}_a(H_1) \wedge \text{predec}_a(H_2) \implies \text{predec}_a(H')$ as desired. If H is a disjunction, the argumentation is similar. ■

This completes the instantiation of the model template. We get the *abstract model* $\mathcal{M}_a = (\mathcal{D}_a, \mathcal{I}_a)$ and for each term t of kind κ the *abstract semantics*

$$\mathcal{M}_a[\![t]\!] : (N \uplus V \to_p \mathcal{D}_a) \to_\sqcup \mathcal{D}_{a,\kappa} .$$

By Proposition 18.2.8, the least *abstract* solution to the interpreted system of equation exists. It is obtained as the join of the abstract approximants,

$$\mathrm{sol}_a = \bigsqcup_{i \in \mathbb{N}} \mathrm{sol}_a^i .$$

In contrast to the concrete domains, the abstract domains satisfy the ascending chain condition as they are finite, see Lemma 18.5.1. Hence, the chain of the sol_a^i is stationary, and sol_a equals $\mathrm{sol}_a^{i_0}$ for some $i_0 \in \mathbb{N}$. This will later allow us to compute the winner of the game. We come back to the details when we discuss the complexity of the algorithm.

Exact fixed-point transfer

We want to prove that we can read off the winner of this game from the abstract least solution sol_a. Instead of reproving a result similar to Theorem 18.3.1, we want to apply exact fixed-point transfer. This will allow us to simply reuse Theorem 18.3.1.

We instantiate the framework from Section 18.4 by following our overview, Figure 18.4.a. The first step is defining an abstraction function $\alpha : \mathcal{D}_{c,o} \to \mathcal{D}_{a,o}$. The definition will follow the intuition behind the translation from words to formulas over states we have explained earlier. For the formal definition, we proceed as follows. For a set of formulas $\mathcal{H} \in \mathcal{D}_{c,o}$, we define

$$\alpha(\mathcal{H}) = \bigvee_{H \in \mathcal{H}} \alpha(H)$$

as the disjunction of the result of applying α to the elements of $\mathcal{H}$. Note that there are only finitely many distinct values in $\mathcal{D}_{a,o}$, so the disjunction on the right-hand side is a well-defined finite formula, even if $\mathcal{H}$ is an infinite set. For a formula in $\mathrm{pBF}(\Sigma^*)$, we define α as follows.

$$\begin{aligned} \alpha\big(H \mathbin{\overset{\wedge}{\vee}} H'\big) &= \alpha(H) \mathbin{\overset{\wedge}{\vee}} \alpha\big(H'\big) \\ \alpha(w) &= \bigwedge \left\{ q \;\middle|\; q \xrightarrow{w} q_{\mathrm{final}} \in Q_{\mathrm{final}} \right\} \\ \alpha(\mathrm{false}) &= \mathrm{false} . \end{aligned}$$

This means that α distributes over disjunctions and conjunctions. For a word w, it produces the conjunctions of the states q so that A has an accepting run on w from q.

Since formally, $\mathcal{D}_{c,o}$ and $\mathcal{D}_{a,o}$ are equivalence classes of (sets of) formulas, we should argue that α is well-defined. It is sufficient to argue that α is monotonic: $\mathcal{H} \implies \mathcal{H}'$ implies $\alpha(\mathcal{H}) \implies \alpha(\mathcal{H}')$. For individual formulas, it is easy to show that $H \implies H'$ implies $\alpha(H) \implies \alpha(H')$ with the same line of reasoning as in Lemma 18.5.2. For sets of formulas,

we observe that $\mathcal{H} \implies \mathcal{H}'$ means that for every $H \in \mathcal{H}$, there is some $H' \in \mathcal{H}'$ so that $H \implies H'$. This is because we have defined the evaluation semantics to treat sets of formulas as disjunctions. Hence, for every disjunct $a(H)$ of $a(\mathcal{H})$, there is a disjunct $a(H')$ of $a(\mathcal{H}')$ so that $a(H) \implies a(H')$. We obtain $a(\mathcal{H}) \implies a(\mathcal{H}')$ as desired.

In order to be able to use exact fixed-point transfer, we need to show that a is precise.

18.5.3 Lemma
The abstraction function a is precise.

Proof:
We use the definition of precision, Definition 18.4.6, and our outline, Figure 18.4.a.

Property (P1). The least element of $\mathcal{D}_{c,o}$ is the (equivalence class of the) set $\{\text{false}\}$. Its a-value is the disjunction with the unique disjunct $a(\text{false}) = \text{false}$, which is the least element of $\mathcal{D}_{a,o}$.

Property (P2). We consider join-continuity. Let $(\mathcal{H}_i)_{i\in\mathbb{N}}$ be an ascending chain of sets of formulas, and note that their join in $\mathcal{D}_{c,o}$ is the union $\bigcup_{i\in\mathbb{N}} \mathcal{H}_i$, corresponding to the disjunction of all formulas contained in some $\mathcal{H}_i$. We have that the join of the $a(\mathcal{H}_i)$ is the disjunction $\bigsqcup_{i\in\mathbb{N}} a(\mathcal{H}_i) = \bigvee_{H\in\mathcal{H}_i \text{ for some } i\in\mathbb{N}} a(H)$, which equals $a(\bigcup_{i\in\mathbb{N}} \mathcal{H}_i)$.

Property (P3). We prove that the concrete interpretations of the terminals are compatible. For the word-end marker, the interpretation is not a function and there is nothing to show. The abstraction function a distributing over disjunction and conjunction means the concrete interpretations of the terminals of the shape branch_F are compatible.

Let $a \in \Sigma$ be a letter and consider elements of $\mathcal{D}_{c,o}$ whose abstractions coincide. Here, it is important to take into account that the equality of equivalence classes means that their representatives are logically equivalent. Hence, consider $\mathcal{H}, \mathcal{H}'$ so that $a(\mathcal{H}) \iff a(\mathcal{H}')$. Using the definition of a, $a(\mathcal{H}) \iff a(\mathcal{H}')$ means $\bigvee_{H\in\mathcal{H}} a(H) \iff \bigvee_{H'\in\mathcal{H}'} a(H')$. As in the above proof that a is well-defined, this means that for every $H \in \mathcal{H}$, there is some $H' \in \mathcal{H}'$ so that $a(H) \implies a(H')$ and vice versa.

We need to show $a(\text{prepend}_a(\mathcal{H})) \iff a(\text{prepend}_a(\mathcal{H}'))$. Using the definition of the prepend function and of a, this is equivalent to showing

$$\bigvee_{H\in\mathcal{H}} a(\text{prepend}_a(H)) \iff \bigvee_{H'\in\mathcal{H}'} a\big(\text{prepend}_a(H')\big)\,.$$

We use the characterization of the equivalence of disjunctions again and get that we have to show that for every $H \in \mathcal{H}$, there is some $H' \in \mathcal{H}'$ so that $a(\text{prepend}_a(H)) \implies a(\text{prepend}_a(H'))$

and vice versa. If we show that $a(H) \implies a(H')$ implies $a(\text{prepend}_a(H)) \implies a(\text{prepend}_a(H'))$, we are done. We proceed by a nested induction on H and H'. In the base case of the inner induction, both $H = w$ and $H' = w'$ are atoms, words over Σ. We have $a(H) = \bigwedge Q_w$ with $Q_w = \{q \mid q \xrightarrow{w} q_{\text{final}} \in Q_{\text{final}}\}$, and $a(H') = \bigwedge Q_{w'}$ with $Q_{w'} = \{q \mid q \xrightarrow{w'} q_{\text{final}} \in Q_{\text{final}}\}$. Similarly, $a(\text{prepend}_a(H)) = a(aw) = \bigwedge Q_{aw}$ and $a(\text{prepend}_a(H')) = a(aw') = \bigwedge Q_{aw'}$ with $Q_{aw} = \{q \mid q \xrightarrow{aw} q_{\text{final}} \in Q_{\text{final}}\}$ and $Q_{aw'} = \{q \mid q \xrightarrow{aw'} q_{\text{final}} \in Q_{\text{final}}\}$. Observe that for two sets of states $Q', Q'' \subseteq Q$, we have $\bigwedge Q' \implies \bigwedge Q''$ if and only if $Q'' \subseteq Q'$. Since we assume that $a(H) \implies a(H')$, we get $Q_{w'} \subseteq Q_w$. Furthermore, we have that $Q_{aw} = \text{pre}_A(a, Q_w)$ is the set of a-predecessors of Q_w in A, similarly $Q_{aw'} = \text{pre}_A(a, Q'_w)$. The predecessor function $\text{pre}_A(a, -)$ is monotonic with respect to the set of states, so $Q_{w'} \subseteq Q_w$ implies $Q_{aw'} \subseteq Q_{aw}$. This means the desired implication $a(\text{prepend}_a(H)) = \bigwedge Q_{aw} \implies \bigwedge Q_{aw'} = a(\text{prepend}_a(H'))$ holds. This finishes the base case of the inner induction. The induction steps can be proven analogously to the proof of Lemma 18.5.2, using the fact that a distributes over conjunctions and disjunctions. We forgo giving the formal proof.

Property (P4). Finally, we need to prove that the abstractions of the concrete interpretations are $\approx$-equivalent to the abstract interpretations. For the word end marker, we have

$$a\left(\$^{\mathcal{I}_c}\right) = a(\{\varepsilon\}) = a(\varepsilon) = \bigwedge Q_{\text{final}} = \$^{\mathcal{I}_a} .$$

Since the terminals of the shape branch_F are interpreted as conjunctions resp. disjunctions in both domains, the desired property holds. It remains to consider a letter $a \in \Sigma$. Let $\mathcal{H} \in \mathcal{D}_{c,o}$. We need to show $a(a^{\mathcal{I}_c}\ \mathcal{H}) = a^{\mathcal{I}_a}\ a(\mathcal{H})$. We have

$$a\left(a^{\mathcal{I}_c}\ \mathcal{H}\right) = a(\text{prepend}_a(\mathcal{H})) = a\Big(\bigcup_{H \in \mathcal{H}} \text{prepend}_a(H)\Big) = \bigvee_{H \in \mathcal{H}} a(\text{prepend}_a(H))$$

and

$$a^{\mathcal{I}_a}\ a(\mathcal{H}) = \text{predec}_a(a(\mathcal{H})) = \text{predec}_a\Big(\bigvee_{H \in \mathcal{H}} a(H)\Big) = \bigvee_{H \in \mathcal{H}} \text{predec}_a(a(H))$$

using the definitions of prepend_a, predec_a, and a. If we show $a(\text{prepend}_a(H)) = \text{predec}_a(a(H))$, we are done. We proceed by induction on the structure. In the base case, $H = w$ and

$$\begin{aligned} a(\text{prepend}_a(w)) = a(aw) &= \bigwedge_{q : q \xrightarrow{aw} q_{\text{final}} \in Q_{\text{final}}} q = \bigwedge_{q' : q \xrightarrow{w} q_{\text{final}} \in Q_{\text{final}}} \bigwedge_{q : q' \xrightarrow{a} q} q \\ &= \text{predec}_a\Big(\bigwedge_{q' : q \xrightarrow{w} q_{\text{final}} \in Q_{\text{final}}} q'\Big) = \text{predec}_a(a(w)) . \end{aligned}$$

The induction step is again trivial because all involved functions distribute over conjunctions and disjunctions. This completes the proof. ■

The fact that α is precise allows us to use Theorem 18.4.11 resp. Corollary 18.4.12.

18.5.4 Corollary
We have $\alpha(\mathrm{sol}_c) \approx \mathrm{sol}_a$. In particular, for the initial symbol S of the HORS of kind o, we have $\alpha(\mathrm{sol}_c)(S) = \mathrm{sol}_a(S)$.

Determining the winner

We have proven in the form of Theorem 18.3.1 that the existential player wins the higher-order inclusion game iff $\mathrm{sol}_c(S)$ is satisfied by the language $\overline{\mathcal{L}(A)}$. It remains to see how this property can be translated to the abstract domain. The following lemma establishes the desired connection.

18.5.5 Lemma
For $\mathcal{H} \in \mathcal{D}_{c,o}$, $\mathcal{H}$ is satisfied by $\overline{\mathcal{L}(A)}$ if and only if $\alpha(\mathcal{H})$ is satisfied by $Q \setminus q_{\mathrm{init}}$, where q_{init} is the initial state of A.

Proof:
By definition, $\mathcal{H}$ is satisfied by $\overline{\mathcal{L}(A)}$ if and only if at least one formula $H \in \mathcal{H}$ is satisfied by $\overline{\mathcal{L}(A)}$. Similarly, $\alpha(\mathcal{H}) = \bigvee_{H \in \mathcal{H}} \alpha(H)$ is satisfied by $Q \setminus \{q_{\mathrm{init}}\}$ if and only if at least one $\alpha(H)$ is satisfied by $Q \setminus \{q_{\mathrm{init}}\}$. If we prove that every formula H is satisfied by $\overline{\mathcal{L}(A)}$ iff $\alpha(H)$ is satisfied by $Q \setminus \{q_{\mathrm{init}}\}$, we are done. We proceed by induction on the structure of H. In the base case, we have $H = w$. This formula is satisfied by $\overline{\mathcal{L}(A)}$ iff $w \in \overline{\mathcal{L}(A)}$, i.e. $w \notin \mathcal{L}(A)$. This means that A has no accepting run on w, no run from the initial state q_{init} to a final state. Every run of A that ends in a final state and processes word w must not start in the initial state.

Recall that $\alpha(w)$ is the conjunction of the states q so that $q \xrightarrow{w} q_{\mathrm{final}} \in Q_{\mathrm{final}}$. This conjunction is satisfied by $Q \setminus \{q_{\mathrm{init}}\}$ if and only if q_{init} is not one of these states, i.e. if $q_{\mathrm{init}} \xrightarrow{w} q_{\mathrm{final}} \in Q_{\mathrm{final}}$ does not hold. This proves the desired equivalence. Because α distributes over conjunctions and disjunctions, the induction step is again trivial. ■

With the previous lemma, we finally obtain a characterization of the winner of the higher-order inclusion game in terms of the abstract least solution.

18.5.6 Theorem
The existential player has a winning strategy for the higher-order inclusion game iff $\mathrm{sol}_a(S)$ is satisfied by $Q \setminus \{q_{\mathrm{init}}\}$.

Proof: Theorem 18.3.1, Corollary 18.5.4, and Lemma 18.5.5. ■

We have already argued that sol_a can be computed because the domains that are used in the fixed-point iteration are finite. Hence, we in particular get the decidability of higher-order inclusion games as a result.

18.5.7 Corollary
Higher-order inclusion games with a regular target language are decidable.

We state the algorithm that, given an instance (G, A) of a higher-order inclusion game, computes the winner, i.e. the player who has a winning strategy.

Given an instance (G, A), it works as follows.

1. Determinize the given game HORS G.
2. Construct the system of equations associated to the determinization of G as described in Section 18.2.
3. Solve the system of equations interpreted over the abstract model $\mathcal{M}_a$.
   - Initialize $\mathrm{sol}_a^0(F) = \bot_{a,\kappa}$ for each nonterminal F of kind κ.
   - Starting with $i = 0$, compute sol_a^{i+1} by evaluating the interpreted right-hand sides of the equations at sol_a^i. For each nonterminal F, we set

     $$\mathrm{sol}_a^{i+1}(F) = \mathcal{M}_a[\![t]\!]\mathrm{sol}_a^i\,,$$

     where $F \to t$ is the unique rule for F in the determinization of G.

     While $\mathrm{sol}^{i+1} \neq \mathrm{sol}^i$, increment i and repeat this step.
   - Let i_0 be the first index so that $\mathrm{sol}_a^{i_0} = \mathrm{sol}_a^{i_0+1}$.
4. The existential player has a winning strategy from S if and only if $\mathrm{sol}_a^{i_0}(S)$ evaluates to true under the assignment $Q \setminus \{q_{\mathrm{init}}\}$.

Note that the equality $\mathrm{sol}_a^{i_0} = \mathrm{sol}_a^{i_0+1}$ in Step 3 of the algorithm means that for each nonterminal F of κ, $\mathrm{sol}^{i_0}F = \mathrm{sol}^{i_0+1}F$, where the two values are from $\mathcal{D}_{a,\kappa}$. If F is of kind ground, this means the two values are equivalence classes of formulas in $\mathrm{pBF}(Q)$ and we need to check the implication $\mathrm{sol}^{i_0+1}F \implies \mathrm{sol}^{i_0}F$. (The implication in the other direction will always hold as the approximants sol_a^i form an ascending chain.) If F is of kind $\kappa_1 \to \kappa_2$, $\mathrm{sol}^{i_0}F$ and $\mathrm{sol}^{i_0+1}F$ are functions, and checking equality means checking for each value in $\mathcal{D}_{a,\kappa_1}$ whether the function values of the two functions coincide.

Computational complexity

We conclude this chapter by analyzing the complexity of solving higher-order inclusion games. This complexity of the problem is mostly determined by order k of G. Recall that the order of a HORS is the maximum order of any of its nonterminals, where the order of a term is the order of the associated kind. We will show that the problem is $(k+1)$EXP-complete, i.e. it is complete for the class of problems solvable in deterministic $(k+1)$-fold exponential time. For the upper bound, i.e. proving membership in $(k+1)$EXP, we will show the algorithm outlined above achieves this optimal time complexity.

Let us make these statements formal.

Solving higher-order inclusions games of order k	
Given:	Game HORS G of order k, NFA A.
Question:	Does the existential player have a winning strategy for (G, A) from the position S?

18.5.8 Theorem
Solving higher-order inclusions games of order k is $(k+1)$EXP-complete, and the above algorithm achieves this optimal time complexity.

Remark
The theorem also shows the following. If we do not restrict the order of the input game HORS, then solving higher-order inclusions games is primitive recursive, but non-ELEMENTARY. The time needed to solve the game is described by a tower of exponentials, where the height of the tower depends on a property of the input, namely the highest order of any nonterminal.

Upper bound / Membership

In order to show that higher-order inclusions games of order k can be solved in $(k+1)$EXP, we analyze the running time of the above algorithm. We start by considering various properties of the domain $\mathcal{D}_{a,\kappa}$ for kind κ.

When we say that a number is k-fold exponential in the following, we man that the number can be described as $exp_k(f(|Q|))$, where f is a polynomial.

18.5.9 Lemma
For each kind κ of order k, the size of $\mathcal{D}_{a,\kappa}$ is at most $(k+2)$-fold exponential, the height of $\mathcal{D}_{a,\kappa}$ is at most $(k+1)$-fold exponential, and objects from $\mathcal{D}_{a,\kappa}$ can be represented using $(k+1)$-fold exponential space.

Proof:
We proceed by induction on the order of the kind κ. In the base case, the order is zero, and the only kind of order zero is o. Hence, we analyze $\mathcal{D}_{a,o} = \mathrm{pBF}(Q)/\!\iff$. We may represent (equivalence classes of) formulas in $\mathcal{D}_{a,o}$ in conjunctive normal form, see Lemma 17.5.2. This allows us to identify $\mathcal{D}_{a,o}$ with $\mathcal{P}(\mathcal{P}(Q))$. The size of this domain is $2^{2^{|Q|}}$. Additionally, a formula can be represented as a conjunction of at most $2^{|Q|}$ different clauses of size at most $|Q|$, which is singly exponential. We can analyze the height of the domain using the same reasoning as in the proof of Proposition 17.5.3 and get that its height is $2^{|Q|}$ as desired.

Assume we have proven statement for all orders strictly less than k. Consider kind κ of order $k > 0$. In order to be able to apply the induction hypothesis, we need to conduct an inner induction on the arity of the kind. In the base case, the arity of κ is zero. However, this is only possible if κ is o, which violates the assumption that the order is $k > 0$. Hence, our desired statement trivially holds.

Consider $\kappa = \kappa_1 \to \kappa_2$ of order k. We have $\mathrm{order}(\kappa_1 \to \kappa_2) = \max(\mathrm{order}(\kappa_1) + 1, \mathrm{order}(\kappa_2))$. This means that κ_1 has order at most $k - 1$ and κ_2 has order at most k. We can apply the induction hypothesis of the outer induction to $\mathcal{D}_{a,\kappa_1}$. By the definition of the arity, $\mathrm{arity}(\kappa_1 \to \kappa_2) = \mathrm{arity}(\kappa_2) + 1$, the arity of κ_2 is strictly less than the arity of κ. We can apply the induction hypothesis of the outer induction to $\mathcal{D}_{a,\kappa_2}$. In summary, we get that the size of $\mathcal{D}_{a,\kappa_1}$ is $exp_{k+1}(f_1)$ for a suitable polynomial f_1. Similarly, the size of $\mathcal{D}_{a,\kappa_2}$ is $exp_{k+2}(f_2)$, the height is $exp_{k+1}(h_2)$, and the space needed to represent an element is $exp_{k+1}(g_2)$ for suitable polynomials f_2, h_2, g_2.

Let us analyze $\mathcal{D}_{a,\kappa_1\to\kappa_2}$, which is a subset of the set of functions from $\mathcal{D}_{a,\kappa_1}$ to $\mathcal{D}_{a,\kappa_2}$. Hence, the height and size of $\mathcal{D}_{a,\kappa_1} \to \mathcal{D}_{a,\kappa_2}$ bound the height and size of $\mathcal{D}_{a,\kappa_1\to\kappa_2}$. An element of $\mathcal{D}_{a,\kappa_1\to\kappa_2}$ can be represented as a map that assigns a function value from $\mathcal{D}_{a,\kappa_2}$ to each of the $|\mathcal{D}_{a,\kappa_1}|$ elements of $\mathcal{D}_{a,\kappa_1}$. Each function value can be represented using $exp_{k+1}(g_2)$ space, so the function itself can be represented using

$$\begin{aligned} exp_{k+1}(f_1) \cdot exp_{k+1}(g_2) &= 2^{exp_k(f_1)} \cdot 2^{exp_k(g_2)} = 2^{exp_k(f_1)+exp_k(g_2)} \\ &\leqslant 2^{exp_k(f_1 \cdot g_2)} = exp_{k+1}(f_1 \cdot g_2) \end{aligned}$$

space. The size of $\mathcal{D}_{a,\kappa_1} \to \mathcal{D}_{a,\kappa_2}$ is bounded by

$$\begin{aligned} \left|\mathcal{D}_{a,\kappa_2}\right|^{\left|\mathcal{D}_{a,\kappa_1}\right|} &= \left(exp_{k+2}(f_2)\right)^{exp_{k+1}(f_1)} \\ &= \left(2^{exp_{k+1}(f_2)}\right)^{exp_{k+1}(f_1)} = 2^{exp_{k+1}(f_2) \cdot exp_{k+1}(f_1)} \\ &\leqslant 2^{exp_{k+1}(f_2 \cdot f_1)} = exp_{k+2}(f_2 \cdot f_1)\,. \end{aligned}$$

Finally, we observe that a chain in $\mathcal{D}_{a,\kappa_1} \to \mathcal{D}_{a,\kappa_2}$ can be decomposed into at most $|\mathcal{D}_{a,\kappa_1}|$ chains in $\mathcal{D}_{a,\kappa_2}$, one chain for each value in the domain of the functions. The length of each such com-

ponent chain is bounded by the height of $\mathcal{D}_{a,\kappa_2}$. Altogether, the height of $\mathcal{D}_{a,\kappa_1} \to \mathcal{D}_{a,\kappa_2}$ is bounded by

$$exp_{k+1}(f_1) \cdot exp_{k+1}(h_2) \leqslant exp_{k+1}(f_1 \cdot h_2) \,.$$

This completes the induction step of the inner induction, which also finishes the induction step of the outer induction and the proof. ■

With the technical lemma at hand, we can prove one direction of Theorem 18.5.8.

18.5.10 Proposition
The above algorithm solves higher-order inclusions games of order k in $(k+1)$EXP.

Proof:
We analyze the time needed for each step of the algorithm. In contrast to the determinization of NFAs, the determinization of HORSes can be completed in polynomial time, and the result is a deterministic HORS of polynomial size. Constructing the associated system of equations can also be done in polynomial time.

Assume we had obtained $\mathrm{sol}_a^{i_0}(S)$, the least solution associated to the initial symbol S. Since S is of kind ground, $\mathrm{sol}_a^{i_0}(S)$ is a value of $\mathcal{D}_{a,o}$ whose size is at most exponential by Lemma 18.5.9. This value is at most $(k+1)$-fold exponential for all $k \in \mathbb{N}$. Evaluating this formula for the assignment $Q \setminus \{q_{\text{init}}\}$ can be done in time polynomial in the size of the formula.

It remains to analyze the cost of solving the interpreted system of equations. This cost is determined by two factors: the maximum number of iterations and the cost per iterations. The approximants sol_a^i form a chain in $N \to \mathcal{D}_a$, the domain of valuations from nonterminals to $\mathcal{D}_a$ Hence, the height of this domain bounds the number of iterations. The height of the domain of valuations is the product of the heights of the domains $\mathcal{D}_{a,\kappa}$ associated to the kind κ of each nonterminal. In the worst case, the order of each κ is the maximum order k, so $\mathcal{D}_{a,\kappa}$ has $(k+1)$-fold exponential height by Lemma 18.5.9. In this case, the total height is $|N|$ times a $(k+1)$-fold exponential number, which is a $(k+1)$-fold exponential number.

If we prove that each iteration of the loop needs at most $(k+1)$-fold exponential time, we are done. The number of operations needed for each iteration is polynomial in the size of the determinized scheme, which is polynomial in the size of the original HORS. We need to argue that each operation itself can be completed in $(k+1)$-fold exponential time. Here, we need to be careful. When evaluating e.g. $\mathcal{M}_a[\![t]\!]\mathrm{sol}_a^i$ for some term t, we will encounter subterms t' that are not variable free. For these subterms, the semantics is not simply a value from $\mathcal{D}_{a,\kappa'}$, where κ' is the kind of t'. Instead, it is a function $(V \to_p \mathcal{D}_a) \to \mathcal{D}_{a,\kappa'}$ that takes a valuation that assigns values to the free variables of t'. We will need to argue that each such function can be represented using space at most $(k+1)$-fold exponential.

We make the following observation. If k is the maximum order of any nonterminal in G, then $k-1$ is the maximum order of any variable that is used on the right-hand sides of the rules for G. Assume the contrary and consider a term $\lambda x.t$. Its kind is $\kappa_1 \to \kappa_2$, where κ_1 is the kind of x and κ_2 is the kind of t. The order of $\kappa_1 \to \kappa_2$ is $\max(\text{order}(\kappa_1)+1, \text{order}(\kappa_2))$. If variable x were of order k or higher, the term $\lambda x.t$ would be of order at least $k+1$. Since the kinds of the terms on the right-hand sides of HORS rules coincide with the kinds of the variables, and the highest order of any nonterminal is k, we are not able to use a variable of order more than $k-1$.

Using the observation, we get that each valuation assigns at most a value from $\mathcal{D}_{a,\kappa}$ to each variable, where κ is of order at most $k-1$. With the same argumentation as in the proof of Lemma 18.5.9, we obtain that the number of such valuations is at most $(k+1)$-fold exponential. Hence, each function with signature $(V \to_p \mathcal{D}_a) \to \mathcal{D}_{a,\kappa'}$ can be represented using $(k+1)$-fold exponential space. It remains to note that all operations that we need to compute, e.g. function applications, disjunctions and conjunctions, and predecessor computations, can be conducted in time polynomial in the size of the objects that are involved. As we have just argued, the size of each object involved is at most $(k+1)$-fold exponential, so each operation can be computed in $(k+1)$-fold exponential time. This completes the proof. ■

Hardness / Lower bound

To show that solving games define by a HORS of order k is $(k+1)$EXP complete, it remains to provide a matching lower bound, showing the hardness of the problem.

18.5.11 Proposition
Solving higher-order inclusions games of order k is $(k+1)$EXP-hard.

The author conjectures that one could prove this proposition similar to the proof of Theorem 17.5.4. This means one would use $(k+1)\text{EXP} = k\text{AEXPSPACE}$ and try to simulate an alternating Turing machine with k-fold exponential space consumption by a higher-order inclusion game. This would require accurately generating configurations of length $exp_k(n)$, where n is the size of the input of the machine. To this end, it should be possible to use the fact that HORSes can be used to generate k-fold exponential numbers [CW07].

However, this is not the approach to the proof that we will take in the following. Instead, we will give a reduction from another kind of automata model. Introducing this automata model in full detail would be beyond the scope of this thesis. We will give a sketch of the proof and refer to our publication [HMM17] resp. its full version [HMM17a] for the technical details.

Proof sketch for Proposition 18.5.11:
We reduce the word problem for the languages of alternating order-k pushdown automata with a work tape. At the end of Section 5.3, we have explained that there is a generalization of pushdown automata that is polynomially equivalent [KNU02] to a subclass of HORSes. An *order-k*

pushdown is a pushdown automaton that mains an order-k stack, where an order-one stack is a normal stack consisting of symbols, while an order-n stack for $n > 1$ is a stack of order-$(n-1)$ stacks. *Alternating automata* are defined, similar to alternating pushdown machines, by partitioning the control states into existential and universal control states. In contrast to Turing machines, the word that should be processed is not initially written on a tape. Instead, it is read letter-by-letter. If the automaton is in an existential control state and it reads letter a, there needs to be an a-labeled transition that can be continued, resulting in an accepting run. If the automaton is in a universal control state and it reads letter a, there needs to be an accepting run no matter which a-labeled transition is chosen. An *automaton with a work tape* has a tape of polynomial length that it can use to store information during its computation. Combining these three properties leads to the aforementioned model of *alternating order-k pushdown automata with a work tape*.

Engelfriet [Eng91] has shown that the word problem for this model, given a word w and an alternating order-k pushdown automata with a work tape B, decide whether $w \in \mathcal{L}(B)$ holds, is $(k+1)$EXP-complete. If we manage to encode this word problem into an instance of a HORS game (G, A) where G and A are of size polynomial in $|w| + |B|$ and G has order k, we obtain the desired result.

The first step is to get rid of the work tape. To this end, we first construct a DFA for the singleton language $\{w\}$. We then take the product of this DFA and B, and get an alternating order-k pushdown automaton with a work tape whose language is either empty if $w \notin \mathcal{L}(B)$, or it is equal to $\{w\}$ if $w \in \mathcal{L}(B)$. In this automaton, the language does not play a role anymore, it just matters whether a final state can be reached. Hence, we can redefine the automaton to not read word w, but instead read the sequence of worktape contents that occur during the computation. (Here, it is more intuitive to think of the pushdown automaton as an automaton that generates words rather than reading them.) If we drop the worktape of B, we are not actually able to output the precise worktape content. Instead, we will have to guess a worktape content. For the cell of the worktape that is the current head position, we can make sure that the output is consistent with the transition of B that was used. For the rest of the cells, we simply have to guess. The technical details can be found in the proof of Proposition 20 in [HMM17a]. Let B' be the alternating order-k pushdown automaton without a worktape that is the result of this operation.

An accepting computation of B' producing some word v corresponds to an accepting computation of B for word w if and only if v corresponds to a valid sequence of worktape contents. In order to verify the latter, we design an NFA A of polynomial size. It accepts v iff v is not a valid sequence of worktape contents. The idea behind this construction is similar to the construction of the NFA in the proof of Theorem 17.5.4. The NFA essentially detects if the head has been moved in an illegal way or if the tape content has been modified in an illegal way. Again, we omit the details of the construction. We get that $w \notin \mathcal{L}(B)$ if and only if B has no accepting computation for w, which is the case if and only if every output of B' is accepted by A. We have reduced the problem of checking $w \in \mathcal{L}(B)$ to deciding the inclusion $\mathcal{L}(B') \subseteq \mathcal{L}(A)$.

It remains to encode checking $\mathcal{L}(B') \subseteq \mathcal{L}(A)$ as a higher-order inclusion game. It is well-known that order-k pushdown automata can be translated into HORSes of order k of polynomial size [KNU02]. This construction can be extended so that it translates an alternating order-k pushdown automaton into a game HORS of order k. Basically, the players of the game HORS correspond to the alternation in the automaton. We apply this construction to B', obtaining a HORS G. For the details of the translation, we gain refer to [HMM17a].

Finally, we make use of the fact that we consider inclusion games where the winning condition for the existential player is non-membership in a given regular language. The existential player wins (G, A) if and only if she can enforce producing a word v not in A, which is the case if and only if the inclusion $\mathcal{L}(B') \subseteq \mathcal{L}(A)$ does not hold, which is the case if and only if $w \in \mathcal{L}(B)$. Hence, the reduction that takes as input w and B and returns G and A is as required and proves that solving higher-order inclusions games of order k is $(k + 1)$EXP-hard. ■

Together, Proposition 18.5.10 and Proposition 18.5.11 prove Theorem 18.5.8. We finish this chapter with some concluding remarks on the case that the HORS has order one or zero.

Game HORSes of order zero

Let us consider the case of a higher-order inclusion game (G, A) where G has order zero. This means that the nonterminals of G do not represent functions, but rather they are just values. In particular, the right-hand sides of the rules of G do not use any variables. Additionally, since the nonterminals represent values, they can only be used as the rightmost symbol in a term of the shape $a_1(\dots a_n(F)\dots)$. This means that HORSes of order zero are equivalent to right-linear grammars, a special type of context-free grammars. In a right-linear grammar, the right-hand side of each rule either contains no nonterminal, or it contains a single nonterminal that is the rightmost symbol. By translating nonterminals into states, it is easy to convert a right-linear grammar to a finite automaton; the class of languages of right-linear grammars (resp. word-generating HORSes of order zero) is the class of regular languages.

Theorem 18.5.8 states that higher-order inclusion games are EXP-complete when the HORS is of order zero. Let us inspect our proofs for that case. In the proof of the upper bound, Proposition 17.5.3, there is a small difficulty that we have swept under the carpet. Even though the HORS is of order zero, which means that all its nonterminals are of order zero, the right-hand sides of the rules can contain terminals of order one. Namely, they may contain terminals of the shape $a \in \Sigma$ or of the shape branch_F as introduced by the determinization of the HORS. This intricacy is unique to the case of HORSes of order zero. Luckily, we will never have to explicitly represent the interpretation of a terminal as a function. To be able to conduct Kleene iteration, it is sufficient to be able to apply these functions, i.e. to compute the predecessors and to compute disjunctions and conjunctions, in singly exponential time to formulas of exponential size. Our upper bound holds for order zero.

For the lower bound, we need to inspect the results we rely on. Firstly, note that an order-0 pushdown is simply a finite automaton. Engelfriet's results [Eng91] imply that the word problem for alternating finite automata with a work tape is EXP-complete. The rest of the proof of the lower bound, Proposition 17.5.3 also works: We can translate the word problem for alternating finite automata with a work tape into an inclusion problem whose left-hand side is given by an alternating finite automaton (without work-tape), and then translate this problem into a higher-order inclusion game of order zero.

It would also be possible to obtain a different proof for the EXP-hardness based on our proof of Theorem 17.5.4. In that proof, we have simulated an alternating Turing machine with exponential space consumption by constructing a context-free inclusion game that proceeds in two phases. The second phase of the game was used to be able to handle configurations of exponential length with a finite automaton of polynomial size. If we are only allowed to use a game grammar that is right-linear (or, equivalently, a game HORS of order zero), we cannot implement that second phase. However, this still allows us to simulate an alternating Turing machine with polynomial space consumption. Using APSPACE = EXP, we get the desired result.

Game HORSes of order one

Consider higher-order inclusion games (G, A) where the game HORS G is of order one. In this case, Theorem 18.5.8 states that solving these games is 2EXP-complete.

This should not be surprising. In Example 5.3.1, we have seen that a context-free grammar can be translated into a HORS of order one. If we apply this construction to a game grammar that specifies a context-free inclusion game, we end up with a game HORS of order one such that the two games are equivalent. For context-free inclusion games, we have proven 2EXP-completeness in Section 17.5.

It might be surprising that when solving context-free inclusion games in Chapter 17, we had to consider positive Boolean formulas over boxes. When considering higher-order games, we could get away with using formulas over states. This discrepancy is resolved by observing that we associate formulas over states to terms of order zero. When translating a CFG into a HORS, the nonterminals of the CFG are turned into symbols of order one. The abstract domain for the associated kind is the set of functions from formulas over states to formulas over states. The author conjectures that it is possible to convert such a function to a formula over boxes and vice versa.

19 Valence games

Contents

19.1 Valence systems over graph monoids 433
19.2 Valence games 441
19.3 Bounded context switching for valence systems 463

In the previous chapters, we have studied games on infinite game arenas that are defined by a finite syntax. We have seen that the finite description provided by the syntax is sufficient to ensure decidability in the case of games defined by context-free grammars and HORSes. In this chapter, our goal is to explore the frontier of decidability. We want to obtain a classification result that specifies exactly for which automata models we can hope to solve games on the induced infinite game arenas.

As the basis for this classification result, we use valence systems over graph monoids [Zet15b]. Valence systems are a general algebraic automaton model. Some of the models that we have considered in this thesis, including finite-state automata, pushdown automata, and Petri nets, can be seen as restricted valence systems. Our classification results will show that among all models that can be seen as valence systems, the only ones for which games on infinite arenas are decidable are essentially context-free models and strongly related ones.

Sources

The first section of this chapter presents material that is standard in the literature. We will give references in the main text. The second section is based on work by Roland Meyer, Georg Zetzsche, and the author of this thesis that has not been published. The final section gives a brief summary of our publication [MMZ18] (resp. its full version [MMZ18a]) without presenting all details. We will discuss the authors author's contributions to the material presented in this chapter in Chapter 20.

19.1 Valence systems over graph monoids

In this section, we will formally introduce the model of valence systems over graph monoids [Zet15b] that we will use in the rest of this chapter.

Valence systems

Recall that a *monoid* is a tuple $(\mathbb{M}, \cdot, \mathbb{1}_\mathbb{M})$ where $\mathbb{M}$ is a non-empty set and $\cdot : \mathbb{M} \times \mathbb{M} \to \mathbb{M}$ is a binary operation on $\mathbb{M}$ that satisfies *associativity*, i.e. $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ holds for all $x, y, z \in \mathbb{M}$. Furthermore, $\mathbb{1}_\mathbb{M} \in \mathbb{M}$ is neutral with respect to the operation, $x \cdot \mathbb{1}_\mathbb{M} = \mathbb{1}_\mathbb{M} \cdot x = x$ for all $x \in \mathbb{M}$. In the following, we will usually say that a set $\mathbb{M}$ is a monoid and assume that the operation and the neutral element are clear from the context.

A set of *generators* of a monoid $\mathbb{M}$ is a subset $G \subseteq \mathbb{M}$ such that all elements of $\mathbb{M}$ can be obtained by iteratively composing elements of G. A monoid is *finitely generated* if it has a finite set of generators. For example, consider the monoid $(\mathbb{N}, +, 0)$, the natural numbers with addition. It is generated by the set $\{1\}$: Zero is obtained as the empty composition (which always yields the neutral element), and every other number can be obtained by adding up 1 suitably often.

A *valence system* over monoid $\mathbb{M}$ is an automaton with finitely many control states that uses the elements of the monoid $\mathbb{M}$ as storage. Syntactically, it is a finite-state LTS with monoid elements as transition labels. Its semantics is defined as follows: A configuration is of the shape (q, m), consisting of a control state q and a monoid element $m \in \mathbb{M}$. A transition $q \xrightarrow{m'} q'$ that originates in control state q can be applied to this configuration, leading to the new configuration $(q, m \cdot m')$. This means that monoid elements represent both storage values and operations on the storage: The monoid element m' induces the operation $\cdot m'$ that composes the current storage value with m'. The neutral element represents the empty storage resp. the operation that does not modify the storage.

For the reachability problem for valence systems, we assume that a unique initial and a unique final control state, q_{init} and q_{final}, have been fixed. The goal is to reach the final state with empty storage starting from the initial state with empty storage.

Reachability for valence systems

Given:	Valence system over some monoid $\mathbb{M}$.
Question:	$(q_{\text{init}}, \mathbb{1}_\mathbb{M}) \to^* (q_{\text{final}}, \mathbb{1}_\mathbb{M})$?

The definition of the reachability problem justifies the right-invertibility restriction that we introduce in the following. A monoid element m is called *right invertible* if it has a right inverse, a monoid element m' such that $m \cdot m' = \mathbb{1}_\mathbb{M}$. It is clear that it is only possible to reach $(q_{\text{final}}, \mathbb{1}_\mathbb{M})$ from some configuration (q, m) if m is right invertible – right invertibility is a necessary condition. This allows us to forbid transitions that would lead to a storage value that is not right invertible without altering the answer to the reachability problem.

Altogether, we obtain the following formal definition of valence systems and their semantics.

19.1.1 Definition

Syntactically, a *valence system* is $S = (\mathbb{M}, Q, \delta, q_{\text{init}}, q_{\text{final}})$ where $\mathbb{M}$ is a monoid, Q is a finite set

of states, $q_{\text{init}}, q_{\text{final}} \in Q$ and $\delta \subseteq Q \times \mathbb{M} \times Q$. Its semantics is the induced transition system whose configurations are from the set $Q \times \mathbb{M}$. The transition relation δ induces a transition relation among configurations: $(q, m) \to (q'', m'')$ if there is a transition $q \xrightarrow{m'} q''$ in δ such that $m'' = m \cdot m'$ and m'' is right invertible.

It is straightforward to define valence automata as versions of valence system in which transitions are additionally labeled by letters from a finite alphabet. We can associate to the computations that are witnesses for reachability the finite words that occur as their labels, and hence obtain a definition for the language of a valence automaton.

Whether the reachability problem (and other problems for valence systems and automata) can be solved algorithmically depends on the underlying monoid. We will see later that there is a fixed monoid such that the class of valence system over that monoid is Turing-complete and hence the reachability problem is undecidable. The main goal of the research on valence systems is to classify the monoids for which certain problems are solvable.

However, the class of all monoids turns out to be too diverse to be able to obtain such classification results. There is a lack of criteria that are expressible on the level of general monoids that would be needed for a classification. There are two subclasses that come to mind as candidates on which one could base a classification. The first is the class of all finitely generated monoids. However, this class is still too diverse. In fact, since a valence system has a finite number of transitions, also the number of distinct transition labels is finite. The set of all reachable storage values is contained in the submonoid that is generated by these transition labels. In short, the restriction to finitely generated monoids is not a restriction at all. The second subclass that comes to mind is the class of all finite monoids. It is easy to see, however, that the class of valence systems over such monoids is just as expressive as the class of finite-state systems without any storage. The transition system induced by such a valence system would again be finite state.

Graph monoids

In the following, the class of graph monoids [Zet15b; Cha07] will serve as a basis for our classification results. It is a class of potentially infinite monoids, each of which can be described by a finite undirected graph.[1]

We consider graphs $G = (V, I)$ where V is a finite set of nodes, and the edges are given by $I \subseteq V \times V$, the *independence relation*. The latter is symmetric, i.e. $o_1 \; I \; o_2$ implies $o_2 \; I \; o_1$. It is neither necessarily reflexive nor necessarily anti-reflexive; $o_1 \; I \; o_1$ only holds if the graph has a self-loop at node o_1. We use infix notation and write $o_1 \; I \; o_2$ for $(o_1, o_2) \in I$ and $o_1 \; \neg I \; o_2$ for $(o_1, o_2) \notin I$.

[1] Note the analogy to automata theory, where we are interested in systems with a potentially infinite semantics that admit a finite syntactic representation.

Intuitively, the nodes of V are parts of the storage, and the independence relation specifies which parts of the storage are independent of each other. The graph monoid induced by such a graph consists of all sequences of storage operation, where we identify sequences that are equal but for the order of actions on independent parts of the storage. More formally, we associate to each node $o \in V$ two operations, a positive operation o^+ ("push o", "increment o") and a negative operation o^- ("pop o", "decrement o"). We call + resp. − the polarity of the operation. By $o^\pm$ we denote an arbitrary element from $\{o^+, o^-\}$. Let $\mathcal{O} = \{o^+, o^- \mid o \in V\}$ denote the set of all operations. We start by considering the free monoid $\mathcal{O}^*$ over $\mathcal{O}$, i.e. the set of all finite-length sequences over $\mathcal{O}$ with concatenation as the operation and the empty sequence ε as the neutral element. To obtain the *graph monoid* $\mathbb{M}_G = \mathcal{O}^*/_\cong$ *for graph* G, we factorize $\mathcal{O}^*$ by the smallest congruence $\cong$ (with respect to concatenation) that satisfies

$$o^+.o^- \cong \varepsilon \qquad \text{for all } o \in V, \text{ and} \tag{G1}$$

$$o_1{}^\pm.o_2{}^\pm \cong o_2{}^\pm.o_1{}^\pm \qquad \text{for all } o_1 \; I \; o_2\,. \tag{G2}$$

Intuitively, the first rule states that o^- is the right inverse of o^+ – an increment followed by a decrement leaves the storage unchanged. The second rule formalizes the above-mentioned intuition that sequences that differ only in the order of independent actions (actions $o_1{}^\pm, o_2{}^\pm$ with $o_1 \; I \; o_2$) should be identified.

Elements of the graph monoid are congruence classes of sequences over $\mathcal{O}$ wrt. $\cong$. The operation of the monoid can be applied by concatenating representatives of classes. The neutral element is the equivalence class of the empty word ε. It represents the empty storage, and concatenating it is the operation that leaves the storage unchanged. We will often use sequences to denote monoid elements and extend the corresponding notations and definitions whenever there is no risk of causing ambiguity.

19.1.2 Remark

We briefly discuss the implications of a node having or not having a self-loop in the graph.

If node o_1 does not have a self-loop, $o_1 \; \neg I \; o_1$, then the negative operation o^- is not the right inverse of the positive operation o^+. Rule (G1)only applies to the sequence $o^+.o^-$, not to $o^-.o^+$. In fact, o^- is not right invertible in this case.

If o_1 has a self-loop, $o_1 \; I \; o_1$, then o^- is the right inverse of o^+. We can first apply Rule (G2)to obtain $o^-.o^+ \cong o^+.o^-$, and then cancel the two operations using Rule (G1).

In the following, we will speak of a valence system over a graph and mean the valence system over the graph monoid defined by that graph. We will also call the underlying graph the *storage graph* of the system for obvious reasons.

Before giving some examples, we will need the notion of an induced subgraph.

i) pushdown systems. ii) multi-pushdown systems. iii) VASSes / Petri nets. iv) integer VASSes / integer Petri nets.

Figure 19.1.a: Graphs for which valence systems over the corresponding graph monoids are …

19.1.3 Definition

For a graph $G = (V, I)$ and a set of nodes $V' \subseteq V$, the *subgraph induced by* V' is the graph $(V', I \cap (V' \times V'))$, i.e. the graph that is obtained from G by discarding all nodes not in V' and all edges involving discarded nodes.

A graph is an *induced subgraph* of G if it occurs as the subgraph induced by a suitable set of nodes. Unlike normal subgraphs, induced subgraphs do not allow us to discard arbitrary edges. If a graph contains an edge, then any induced subgraph that contains the nodes connected by the edge will also contain that edge.

The expressiveness of valence systems is monotonic with respect to the subgraph order: The class of valence system over some graph G is at least as expressive as the class of valence system over any subgraph of G.

The class of graph monoids is a good basis for a classification of valence systems with respect to their algorithmic properties. On the one hand, graph-theoretic properties like connectedness are related to the decidability of certain problems for the valence systems over the corresponding graph monoids. On the other hand, valence systems over graph monoids generalize many well-known models of computation. This means that various types of automata will fit into the classification. We demonstrate this in the form of a few examples. We refer to Chapter 2.9 of [Zet15b] for more details.

19.1.4 Example

a) The empty graph induces the graph monoid with ε as the single element. Valence systems over this graph are finite-state systems, and any finite state system can be seen a valence system over this graph.

b) Consider the graph with two nodes a and b and no edge, i.e. the empty independence relation. It is depicted in Figure 19.1.a.i). As observed in Remark 19.1.2, neither a^- nor b^- are right invertible. In fact, any right-invertible element of the monoid can be represented by a sequence over $\{a^+, b^+\}$ that exclusively uses the positive operations, and any such sequence represents a unique element of the graph monoid. Any sequence involving a negative operation, e.g. a^-, can only represent a right-invertible monoid element if it contains an earlier occurrence of a^+ such that the two cancel out.

The right-invertible elements of the graph monoid over are the configurations of a LIFO stack (last in, first out) over the stack alphabet $\{a, b\}$. The operation $\cdot a^+$ pushes a onto the stack, the operation $\cdot a^-$ removes a from the top of the stack. The latter can only be applied if the current storage value can be represented by a sequence whose last element is a^+; otherwise, we end up with a value that is not right invertible.

With this reasoning, a valence system over this graph is a pushdown system with a binary stack alphabet as introduced in Section 5.1. A pushdown system with a k-letter stack alphabet can be seen as a valence system over a graph with k unconnected nodes that have no self-loops.

c) Consider the graph with nodes a_1, b_1, a_2, b_2 and the edges $a_1 \; I \; a_2$, $a_1 \; I \; b_2$, $b_1 \; I \; a_2$, $b_1 \; I \; b_2$ (and their symmetric versions). It is depicted in Figure 19.1.a.ii). Note that the subgraphs induced by both $\{a_1, b_1\}$ and $\{a_2, b_2\}$ are the graph that we considered in Part b). Furthermore, each node x_1 is connected to each node y_2 for $x, y \in \{a, b\}$.

The right-invertible elements of the corresponding graph monoid can be represented by sequences of the shape $w_1.w_2$ where w_1 exclusively contains the positive operations over $\{{a_1}^+, {b_1}^+\}$, similar for w_2 and $\{{a_2}^+, {b_2}^+\}$. Given an arbitrary representative, we first use Rule (G2) to reorder it into a prefix containing the operations corresponding to $\{a_1, b_1\}$ and a suffix. Then, we proceed as in Part b) of the example and use Rule (G2) exhaustively to remove occurrences of negative operations. If a negative operation cannot be removed, the sequence does not represent a right-invertible monoid element.

The right-invertible elements of the graph monoid represent configurations of two independent LIFO stacks, each over a binary stack alphabet. Hence, the corresponding valence systems are multi-pushdown systems that use such stacks as storage. It is well-known that multi-pushdown systems with at least two stacks and at least two symbols on each stack are Turing-complete: Intuitively, each of the stacks can store one half of the tape content of a Turing machine. Rice's theorem [Ric53] applies and all non-trivial semantic properties of valence systems over this graph monoid are undecidable.

d) Consider a 3-clique with no self-loops, i.e. a graph with the set of nodes $\{c_1, c_2, c_3\}$ in which any two distinct nodes are connected by an edge. It is depicted in Figure 19.1.a.iii).

Every right-invertible element of the corresponding graph monoid can be represented by a sequence of the shape $({c_1}^+)^{n_1}({c_2}^+)^{n_2}({c_2}^+)^{n_3}$. Hence, these elements represent tuples $(n_1, n_2, n_3) \in \mathbb{N}^3$ of counter values and the positive and negative operation for each c_i increment and decrement the corresponding counter value n_i, respectively. The decrement is blocking, it can only be applied if the corresponding component is non-zero; otherwise, we end up with a value that is not right invertible. One also says that the nodes of the storage graph are *partially blind counters*: Their value cannot be tested for being zero during runtime, but non-zeroness can be asserted by using the blocking decrement.

i) The graph P_4.

ii) The graph C_4.

iii) A PPN graph.

Figure 19.1.b: Graphs for which reachability in valence systems is undecidable, or not known to be decidable in the case of c).

Valence systems over this graph are 3-dimensional vector addition systems with states (VASSes), a model that we have mentioned in Section 6.1 as being equivalent to Petri nets with a corresponding number of unbounded places. VASSes of arbitrary dimension k can be seen as valence systems over a k-clique with no self-loops.

e) We add a self-loops to every node of the graph from Part d). The result is depicted in Figure 19.1.a.iv).

The self-loops mean that e.g. ${c_1}^+$ is now the right inverse of ${c_1}^-$ in the graph monoid: We can first use Rule (G2) to swap them, then use Rule (G1) to cancel them out. More precisely, every element of the graph monoid is right invertible. It represents a tuple of counters $(n_1, n_2, n_3) \in \mathbb{Z}^3$ that may attain negative values. Since we have no blocking decrement at hand to assert non-zeroness during runtime, we call these counters *blind*.

Valence systems over such graph monoids are integer vector addition systems with states, a model that corresponds to integer Petri nets or Petri nets that work on pseudo-markings, see Section 8.1.

In addition to these well-known types of automata, valence systems can also model various other interesting types of storage. This includes automata that use a stack of counters as storage, and automata that have access to both a stack and a set of partially blind counters. The latter model is sometimes called a *pushdown Petri net (PPN)*; it is famous for the fact that the decidability status of its reachability problem is a long-standing open question, see e.g. [Laz13]. One such *PPN graph* is depicted in Figure 19.1.b.iii). It consists of a stack with two stack symbols a, b and a partially blind counter c that is independent of both a and b.

Valence systems over graph monoids cannot model all types of automata models. For example, they can neither model higher-order systems as defined in Section 5.3, nor FIFO (first in, first out) queues. For a detailed explanation why modelling queues is impossible, we refer to Section 2 of our publication [MMZ18].

Georg Zetzsche has provided a classification of the decidability of the reachability problem for valence systems over graph monoids [Zet15c]. For the sake of simplicity, we restrict ourselves to graphs with no self-loops in the following. Let C_4 denote a cycle of four nodes, and let P_4

be a path of four nodes. These graphs are depicted in Figure 19.1.b.i) and Figure 19.1.b.ii), respectively. Note that the graph C_4 is equal to the graph from Part c) of Example 19.1.4 resp. Figure 19.1.a.ii) that corresponds to multi-pushdown systems. Zetzsche has shown that if a graph G contains C_4 or P_4 as an induced subgraph, then the reachability problem for valence systems over G is undecidable.

The converse result is true if one excludes graphs that do correspond to the aforementioned pushdown Petri nets: If a graph neither contains C_4, nor P_4, nor one of several so-called PPN-graphs as an induced subgraph, then the reachability problem for valence systems over that graph is decidable.

Other classification results for valence systems and automata over graph monoids have been obtained in the literature, including classifications of the eliminability of silent transitions [Zet13], the context-freeness and the semi-linearity of the Parikh image of their languages [BZ13], and the decidability of first-order logic with reachability [DMZ16].

19.2 Valence games

The goal of this chapter is to classify the decidability of several types of games on the configuration graphs of valence systems over graph monoids based on the underlying graph structure. We will tackle this challenge in this section. We consider games with reachability and coverability as the winning conditions. In both cases, we show that the only decidable cases are the context-free games that we have studied extensively in Chapter 17 and closely related games that can be reduced to context-free ones.

Reachability games on valence systems

We start by giving a formal definition of valence games. Consider a valence system in which the set of control states $Q = Q_\Box \uplus Q_\bigcirc$ is partitioned into the set $Q_\Box$ of states owned by the universal player and the set $Q_\bigcirc$ of states owned by the existential player. We usually write such a system as $(\mathbb{M}_G, Q_\Box \uplus Q_\bigcirc, \delta, q_{\text{init}}, q_{\text{final}})$ and call it a *game valence system* (over the graph monoid of G). From this definition, we obtain a game arena on the induced transition system as described in Chapter 15. The configurations owned by the existential player are all configurations (q, m) where the control stated is owned by the existential player, $q \in Q_\bigcirc$, independent of the storage value, similar for the universal player. Moves in the game are defined as before: $(q, m) \to (q', m'')$ if there is a transition $q \xrightarrow{m'} q'$ in δ so that $m'' = m \cdot m'$ and m'' is right invertible.

A game valence system fixes a game arena. To obtain a game, it remains to equip this game arena with a winning condition. For now, we focus on *valence reachability games* in which, starting from $(q_{\text{init}}, \varepsilon)$, the goal of the existential player is to reach $(q_{\text{final}}, \varepsilon)$. Here, ε denotes its equivalence class, the neutral element of the monoid. This means we start in the initial configuration consisting of the initial control state with empty storage and we want to reach the final state with empty storage. Note that reachability here should be understood as *configuration reachability*, i.e. we fix both the target control state and the target storage value, in contrast to control-state reachability games which we will consider later.

All reachability games are known to be positionally determined by the Borel determinacy theorem [Mar75]. Hence, we know that for our fixed initial position, exactly one of the players has a winning strategy. The key problem is computing this winner. As with context-free games and higher-order games, we are dealing with a situation in which we have to compute information about a game on a potentially infinite game arena based on the finite syntax generating it.

Formally, the problem of solving valence reachability games is defined as follows.

Solving valence reachability games

Given:	Game valence system $(\mathbb{M}_G, Q_\Box \uplus Q_\bigcirc, \delta, q_{\text{init}}, q_{\text{final}})$.
Question:	Can the existential player enforce visiting $(q_{\text{final}}, \varepsilon)$ from $(q_{\text{init}}, \varepsilon)$?

Remark

When defining the semantics of valence systems, we have introduced the right-invertibility restriction: We can only apply a transition $q \xrightarrow{m'} q'$ in configuration (q, m) if $m \cdot m'$ is right invertible. The justification for this restriction was the fact that if $m \cdot m'$ is not right invertible, then it will be impossible to reach $(q_{\text{final}}, \varepsilon)$ from $(q', m \cdot m')$. However, for the answer to the reachability problem, it is irrelevant whether we allow transitions to non-right-invertible storage values. We are only considering existential nondeterminism in this case, so the existence of additional transitions that are not helpful does not matter. This is different in the case of valence games. If the universal player would be allowed to take a transition leading to a non-right-invertible storage value, the result is a configuration from which the existential player cannot win. Transitions to non-right-invertible values typically correspond to illegal operations on the storage, e.g. a pop-operation that pops a symbol that is not the current top-of-stack. Hence, it is important to forbid these transitions to obtain the desired semantics for valence reachability games.

Valence reachability games can be seen as an extension of the reachability problem for valence systems; the latter corresponds to games in which all control-states are owned by the existential player. If the graph G contains C_4 from Figure 19.1.b.i) or P_4 from Figure 19.1.b.ii) as an induced subgraph, then reachability in a non-game setting is undecidable [Zet21]. Since reachability games are a generalization of the reachability problem for valence systems, we immediately obtain that valence reachability games are undecidable in this case. Hence, there is no hope for valence reachability games being solvable in general. As typical for valence systems, we study instead for which graphs G valence reachability games over that graph are decidable. We formalize this in the form of the following version of the decidability problem in which the underlying graph is fixed.

Solving valence reachability games over graph G

Given:	Game valence system $(\mathbb{M}_G, Q_\Box \uplus Q_\bigcirc, \delta, q_{\text{init}}, q_{\text{final}})$ where $\mathbb{M}_G$ is induced by G.
Question:	Can the existential player enforce visiting $(q_{\text{final}}, \varepsilon)$ from $(q_{\text{init}}, \varepsilon)$?

We already know some graphs G for which valence reachability games will be decidable. We have argued before that graphs that do not contain any edge correspond to pushdown systems, where the number of nodes in the graph corresponds to the number of stack symbols. The graph from Figure 19.1.a.i) is one such example. Valence reachability games for such graphs are essentially pushdown games, a type of context-free games. We have extensively discussed the decidability of such games Chapter 17, see in particular Section 17.6 for algorithms that can directly deal with pushdown games.

Our main result in this part of the section is that this is essentially the only case in which valence reachability games are decidable. To make this precise, we introduce some notation. We will see later that the existence of self-loops does not influence the decidability of reachability games.

We denote by G^- the *irreflexive version of* G, G with all self-loops removed. Formally, if $G = (V, I)$, then $G^- = (V, I \setminus \{(o, o) \mid o \in V\})$. Our classification will characterize decidability depending on G^- rather than on G itself.

We call a graph a *pushdown graph* if it contains no edges among distinct nodes. Graph G is a pushdown graph iff G^- contains no edge at all. Our main result states that pushdown graphs are exactly the graphs for which valence reachability games are decidable.

19.2.1 Theorem
Valence reachability games over graph G are decidable if and only if G is a pushdown graph.

The proof of the result is split into two parts. We first use the decidability of context-free games to show that valence reachability games over pushdown graphs are decidable. For the other direction, we show that if G^- contains an edge, then the class of valence reachability games over G are not decidable.

Valence reachability games – The decidable case

Let us treat the decidable case first. We start with considering a simple case. If G contains no edge at all, in particular if it contains no self-loops, then we have $G = G^-$. It is straightforward to see a valence reachability game over G as a context-free game defined by a pushdown automaton. We see transitions labeled by a^+ as transitions labeled by push a, similar for a^- and pop a. The goal in the game is to go from the initial state with empty stack to the final state with empty stack. The decidability of such games was first proven by Walukiewicz [Wal01]

To formally prove the result, it is more helpful to use the result by Cachat [Cac02]. We have given a detailed description in Section 17.6. Given a regular representation for the target set of a pushdown reachability game, Cachat's algorithm computes a regular representation of the winning region. In our case, the target set is the final state q_{final} together with the empty stack. We compute the winning region using the algorithm and then read off whether the initial state q_{init} together with the empty stack is contained in it. This proves the following result.

19.2.2 Lemma
If G contains no edge, then valence reachability games over graph G are decidable.

To prove one direction of Theorem 19.2.1, we need to extend the above result to graphs in which some nodes may have self-loops. Recall that a node a of the graph that has a self-loop corresponds to a blind counter, or a stack symbol that can be popped before it has been pushed. To deal with such nodes, we use a well-known trick that can be used to translate one-counter automata over $\mathbb{Z}$ into pushdown automata. For each node a that has a self-loop in G, $a\ I\ a$, we introduce two stacks symbols, a positive version $+a$ and a negative version $-a$. The operation a^+

of the valence system can be seen either as push $+a$ or as pop $-a$. Similarly, a^- is either push $-a$ or pop $+a$. Intuitively, we can decrement a below zero by pushing $-a$ symbols, which later can be removed by popping them on a^+ transitions.

In the case of nondeterministic automata, it would be sufficient to replace each a^+ transition by two transitions of the pushdown automaton as outlined above, similar for a^-. This is not true in the game setting: Assume that the top of stack in the pushdown game is a $-a$ symbol, corresponding to an earlier a^- transition in the valence game. When the universal player takes an a^+ transition in the valence game, this should cancel out the a^- symbol, i.e. she should pop $-a$ in the pushdown game. If we give her the free choice among popping $-a$ and pushing $+a$, she can decide to push a^+ instead. This leaves an infix $-a.+a$ on the stack, which corresponds to $a^+.a^-$ in the valence game. While $a^+.a^-$ cancels out in the graph monoid, it does not cancel out on the stack of the pushdown. The desired correspondence between plays of the valence game and plays of the pushdown game does not hold.

To overcome the problem, we modify the game so that the existential player can choose whether to use $+a$ or $-a$. We give the formal translation from valence reachability games to pushdown games and then prove its correctness.

Assume that $(\mathbb{M}_G, Q_\Box \uplus Q_\bigcirc, \delta, q_{\text{init}}, q_{\text{final}})$ is a valence reachability game over a pushdown graph G. Let $V = P \uplus S$ be a partitioning of the nodes of G into the nodes P that do not have self-loops and the nodes S that do have self-loops. We design a pushdown game with stack alphabet $\{o \mid o \in P\} \cup \{+a, -a \mid a \in S\}$. The set of control states is $Q_\Box \uplus Q_\bigcirc \uplus \hat{Q}$, where $Q_\Box$ and $Q_\bigcirc$ are as given, and owned by the corresponding player. The control states in $\hat{Q}$ are additional states that are owned by the existential player. If $q \xrightarrow{\varepsilon} p$ in the valence system, then $q \xrightarrow{\varepsilon} p$ in the pushdown system. If $q \xrightarrow{o^+} p$ in the valence system for some node $o \in P$ without self-loop, then $q \xrightarrow{\text{push } o} p$ in the pushdown system. Similarly, $q \xrightarrow{o^-} p$ translates to $q \xrightarrow{\text{pop } o} p$. If $q \xrightarrow{a^+} p$ in the valence system where $a \in S$ is a node with self-loop, then we take a fresh control state $\hat{q} \in \hat{Q}$ and add transitions $q \xrightarrow{\varepsilon} \hat{q}$, $q_\bigcirc \xrightarrow{\text{push } +a} p$, and $\hat{q} \xrightarrow{\text{pop } -a} p$. Similarly, $q \xrightarrow{a^-} p$ translates into the transitions $q \xrightarrow{\varepsilon} \hat{q}$, $\hat{q} \xrightarrow{\text{pop } +a} p$, and $q_\bigcirc \xrightarrow{\text{push } -a} p$. For each such transition, we use a fresh control state $\hat{q} \in \hat{Q}$ owned by the existential player. To improve readability, we do not reflect this fact in the notation.

The mechanism that we have described works as follows: Instead of executing transition $q \xrightarrow{a^-} p$, the owner of state q signals that she wants to take it by going to the intermediary state $\hat{q}$. In $\hat{q}$, it is the existential player's choice to either implement the transition by pushing $+a$ or by popping $+a$. We will show that by picking accordingly, she can enforce reaching the empty stack content at the end.

We prove the correctness of the construction in the form of the following proposition.

19.2.3 Proposition
Valence reachability games over pushdown graphs are decidable.

Proof:
Consider a given valence reachability game and construct the corresponding pushdown game as described above. The goal of the pushdown game is reaching control state q_{final} with empty stack from q_{init} with empty stack. We have argued before that such games are decidable, see Lemma 19.2.2. It remains to show that the winner of the valence game and the winner of the pushdown game coincide. Since both games are determined, it is sufficient to show that existential player wins the pushdown game if and only if she wins the valence game.

The key property that we will need is a translation of a play in the pushdown game into a play of the valence game. We contract each sequence of transitions $q \xrightarrow{\varepsilon} \hat{q} \xrightarrow{\text{push } +a} p$ in the pushdown game into a single transition $q \xrightarrow{a^+} p$ of the valence game, similar for all transitions corresponding to symbols a with $a \; I \; a$. All other transitions remain unchanged.

Assume the existential player wins the pushdown game, say with a winning strategy $s_{\bigcirc}^{\text{PDS}}$. We construct a strategy $s_{\bigcirc}^{\text{valence}}$ for the valence game that simply mimics $s_{\bigcirc}^{\text{PDS}}$ by picking the same moves on the states $Q_{\bigcirc}$. For every transition $q \xrightarrow{\varepsilon} \hat{q}$ of the pushdown, we pick the corresponding transition $q \xrightarrow{a^+} p$ resp. $q \xrightarrow{a^-} p$ of the valence system. Consider a play p^{valence} of the valence game that conforms to $s_{\bigcirc}^{\text{valence}}$. By considering the same moves of the universal player and following strategy $s_{\bigcirc}^{\text{PDS}}$, we obtain a corresponding play p^{PDS} of the pushdown game. Play p^{PDS} will visit q_{final} with the empty stack after finitely many moves. Hence, p^{valence} will visit (q_{final}, m). It remains to argue that $m \cong \varepsilon$. Because the graph contains no edges among distinct nodes, there is essentially no reordering in m. Any o^+ operation in m for some node o with $o \; \neg I \; o$ corresponds to a push o transition in p^{PDS}. Because p^{PDS} reaches q_{final} with the empty stack, p^{PDS} contains a corresponding pop o transition, which means that m contains a corresponding o^- operation with which o^+ cancels out. Consider an a^+ for some a with $a \; I \; a$. Either p^{PDS} contains a corresponding push $+a$ and, because we reach the empty stack, a later pop $+a$, or it contains pop $-a$ and an earlier push $-a$. Both pop $+a$ and push $-a$ correspond to a a^- transition in p^{valence} with which a^+ can cancel out. If a^- occurs before a^+, note that we can first swap them using Rule (G2) and then cancel them since we assume $a \; I \; a$. We have identified a negative operation for every positive operation in m such that the pair cancels out and conclude $m \cong \varepsilon$ as desired. The existential player wins p^{valence} and $s_{\bigcirc}^{\text{valence}}$ is a winning strategy.

Let us now assume that the existential player wins the valence game, say with strategy $s_{\bigcirc}^{\text{valence}}$. We construct a corresponding strategy of the pushdown game. For moves originating in states from $Q_{\bigcirc}$, we can simply mimic $s_{\bigcirc}^{\text{valence}}$. Assume that the play contains a move $q \xrightarrow{\varepsilon} \hat{q}$, corresponding to a transition $q \xrightarrow{a^+} p$ of the valence game. We now have to chose whether to use $\hat{q} \xrightarrow{\text{push } +a} p$ or $\hat{q} \xrightarrow{\text{pop } -a} p$. We let the strategy minimize the height of the stack. If the cur-

rent top-of-stack is $-a$, we use the pop transition; otherwise we push $+a$. Similarly, we prefer $\hat{q} \xrightarrow{\text{pop } +a} p$ in the pushdown game corresponding to $q \xrightarrow{a^-} p$ over $\hat{q} \xrightarrow{\text{push } -a} p$.

Assume a play that conforms to $s_\bigcirc^{\text{valence}}$ reaches configuration (q, m) in the valence game. The corresponding play that conforms to $s_\bigcirc^{\text{PDS}}$ will reach (q, Δ). The stack content Δ is a representative for m, assuming we replace every symbol o with $o \ \neg l \ o$ by o^+, every symbol $+a$ by a^+ and every $-a$ by a^-. Furthermore, our policy of minimizing the stack height will ensure that Δ is a representative that is minimal in the sense that it cannot be reduced using Rule (G1), even after potentially applying Rule (G2). Proving this property using induction is tedious but not fundamentally difficult; we will forgo giving a formal proof.

Since $s_\bigcirc^{\text{valence}}$ is winning, any play conforming to it will eventually reach $(q_{\text{final}}, \varepsilon)$. Hence, any play that conforms to $s_\bigcirc^{\text{PDS}}$ will eventually reach $(q_{\text{final}}, \Delta)$ where Δ is a minimal representative for the monoid element ε. This minimal representative has to be ε itself, meaning that Δ is the empty stack as desired. ■

Valence reachability games – The undecidable case

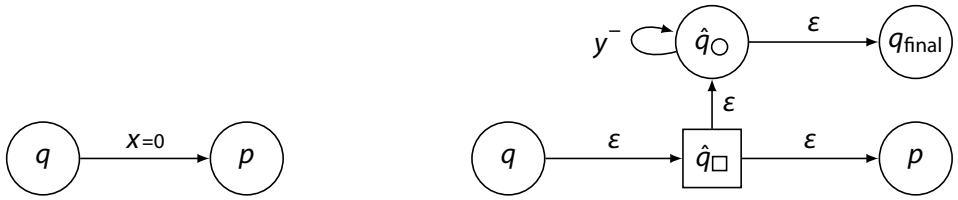
We have shown one direction of Theorem 19.2.1. It remains to show that if G^- contains an edge, which means that G contains an edge between distinct nodes, then valence reachability games over G are undecidable. The idea is to use a well-known result that shows games on systems with counters like Petri nets or VASSes to be undecidable [Jan95].

To be precise, we use that the reachability problem (in a non-game setting) for two-counter machines are undecidable [Min67]. Recall from Section 6.1 that a two-counter machine is an LTS with finitely many control states and labels from the set $\{\varepsilon, x{+}{+}, x{-}{-}, x{=}0, y{+}{+}, y{-}{-}, y{=}0\}$. (Here, we assume that we have replaced non-zero tests using blocking decrements.) Its semantics is defined in terms of configurations from $(q, n, m) \in Q \times \mathbb{N} \times \mathbb{N}$ that consist of a control state and of a value for each of the counters. The effect of the transitions is as expected.

Since two-counter machines are Turing-complete, in particular the following *(configuration) reachability problem* is undecidable.

Reachability problem for two-counter machines	
Given:	Two-counter machine M with initial state q_{init} and final state q_{final}.
Question:	Is there a computation $(q_{\text{init}}, 0, 0) \to^* (q_{\text{final}}, 0, 0)$?

19.2.4 Theorem (Minsky [Min67])
Reachability for two-counter machines is undecidable.



i) A zero-test transition in M. ii) Its translation in the reachability game.

Figure 19.2.a: The translation of a zero-test transition.

To show the undecidability of valence reachability games, our goal is to reduce this undecidable problem. Consider a graph with two distinct connected nodes. For now, we assume that these nodes do not have self-loops.

19.2.5 Proposition
If graph G contains two distinct nodes x, y with $x \; I \; y$ and neither has a self-loop, then valence reachability games over G are undecidable.

Proof:
Let M be a given two-counter machine. We may assume wlog. that the final state q_{final} of M has no outgoing transitions. Else, we create a copy of the final state for which we delete all outgoing transitions and designate this copy as the new final state. Every computation can now nondeterministically choose whether to enter the non-final version of the state and continue the computation or whether to use the final version in which the computation ends.

Our goal is to design a valence reachability game over G that exclusively uses the operations for x and y. The existential player will win the game if and only if M is a yes-instance of the reachability problem, i.e. if M has a computation from $(q_{\text{init}}, 0, 0)$ to $(q_{\text{final}}, 0, 0)$. Since this property is undecidable, valence reachability games over G have to be undecidable.

The valence system is defined to be $S = (\mathbb{M}_G, \hat{Q}_\square \uplus Q_\bigcirc, \delta, q_{\text{init}}, q_{\text{final}})$. Here $Q_\bigcirc = Q \uplus \hat{Q}_\bigcirc$ consists of the set Q of control states of M, including q_{init} and q_{final}. Additionally, there are some fresh control states $\hat{Q}_\bigcirc$ and $\hat{Q}_\square$. We translate transitions of M into transitions of S as follows. If $q \xrightarrow{\varepsilon} p$ in M, then $q \xrightarrow{\varepsilon} p$ in S. If $q \xrightarrow{x{+}{+}} p$ or $q \xrightarrow{x{-}{-}} p$ in M, then $q \xrightarrow{x^+} p$ or $q \xrightarrow{x^-} p$ in S, respectively, similar for y. We have argued before that the semantics of valence systems over graphs in which the nodes have no self-loops but are connected by edges is equal to the semantics of VASSes or Petri nets, see Part d) of Example 19.1.4. Our translation preserves the semantics of the transitions; in particular, the decrement is blocking. Furthermore, a tuple of counter values (n, m) for M will correspond to the monoid element $(x^+)^n(y^+)^m$ and vice versa. Note that the edge between x and y will allow us to bring any right-invertible monoid element into this shape.

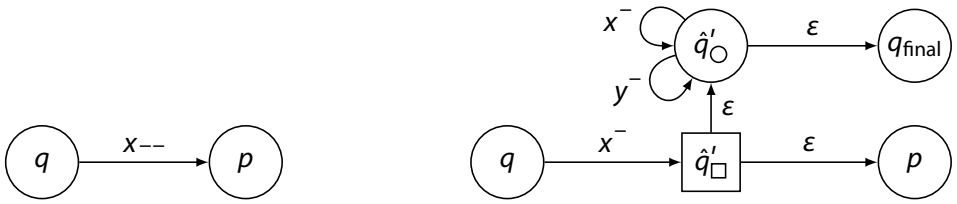
It remains to translate the zero-test transitions $q \xrightarrow{x=0} p$ that may be present in M but that are not available to us in the valence game. The translation is depicted in Figure 19.2.a. It uses a fresh control state $\hat{q}_\Box \in \hat{Q}_\Box$ owned by the universal player and a fresh control state $\hat{q}_\bigcirc \in \hat{Q}_\bigcirc$ owned by the existential player. If the existential player wants to take the transition $q \xrightarrow{x=0} p$ of M, she signals this by taking the transition $q \xrightarrow{\varepsilon} \hat{q}_\Box$. This state implements a challenge mechanism. The universal player can either trust that the current value of counter x is zero and use the transition $\hat{q}_\Box \xrightarrow{\varepsilon} p$ leading to control state p. She can also challenge the existential player to prove that the counter value is indeed zero by taking the transition $\hat{q}_\Box \xrightarrow{\varepsilon} \hat{q}_\bigcirc$. In $\hat{q}_\bigcirc$, the existential can decrement counter y using a y^--labeled loop arbitrarily often and then go to q_{final}.

We argue that the challenge mechanism enforces the desired behavior. If counter x is indeed zero, the universal player will lose if she issues a challenge by going to $\hat{q}_\bigcirc$. The existential player can bring the other counter to zero by using the y^- loop suitably often and then reach q_{final} with both counters being equal to zero (meaning that the monoid element for the current storage is ε). If the counter value is non-zero, the existential player loses if the universal player issues a challenge. She can manipulate the value of the other counter, but she will never be able to reach q_{final} with empty storage. The play will get stuck in q_{final} – recall that q_{final} has no outgoing transitions – with at least one counter that is not zero.

The valence reachability game is defined by translating all zero-tests transitions as explained above. We use fresh control states $\hat{q}_\Box$ and $\hat{q}_\bigcirc$ for each zero-test transition, but we do not reflect this in our notation to improve readability. In transitions labeled by $y{=}0$, we use a version of the gadget from Figure 19.2.a.ii) in which the roles of the counters are swapped.

We claim that the existential player wins the valence reachability game if and only if M has a computation from $(q_{\text{init}}, 0, 0)$ to $(q_{\text{final}}, 0, 0)$. If such a computation exists, we construct a strategy that follows the transitions used in the computation. Because the computation is valid, we only use zero-test transitions if the corresponding counter is indeed zero. If the universal player ever decides to challenge such a transition, she will lose as explained above. If she does not challenge any transition, the play enters q_{final} eventually, and since the counters in the computation of M are zero, the storage will be empty as required. The play is won by the existential player and her strategy is winning.

For the other direction, assume the existential player has a winning strategy for the reachability game. Consider the play in which the universal player does not challenge any transitions corresponding to zero tests. We claim that we can translate this play into a valid computation of M that reaches q_{final} with empty counters. To this end, we simply translate all transitions of S that do not correspond to zero tests back to the original transitions of M. We translate sequences of transitions $q \xrightarrow{\varepsilon} \hat{q}_\Box \xrightarrow{\varepsilon} p$ back to the zero-test transition $q \xrightarrow{x=0} p$ or $q \xrightarrow{y=0} p$ of M. Since the play is won by the existential player as it conforms to a winning strategy, it eventually reaches q_{final} with the empty storage. It remains to argue that this computation is valid, meaning that we only take zero-test transitions if the corresponding counter is indeed zero.



i) A decrement transition in M. ii) Its translation in the reachability game.

Figure 19.2.b: The translation of a decrement transition for a counter x with $x\ I\ x$ in G.

Assume that the computations has a prefix that leads to a configuration in which we take a zero-test transition $q \xrightarrow{x=0} p$. The corresponding prefix of the play ends with the move $q \xrightarrow{\varepsilon} \hat{q}_\Box$. In $\hat{q}_\Box$, the universal player could decide to challenge the zero test. Consider the play in which she does so, and note that this play also conforms to the winning strategy for the existential player. Hence, the existential player wins the challenge and the counter value for x is indeed zero. Applying this reasoning to all zero tests shows that the computation is valid, and M is a yes-instance of the reachability problem. ■

To complete the proof of Theorem 19.2.1,we need to show how to deal with the case that node x or y has a self-loop in the graph. Luckily, there is not much that we have to change in the above proof.

19.2.6 Proposition
If graph G is not a pushdown graph, meaning it contains distinct that are connected by an edge, valence reachability games over G are undecidable.

Proof:
Assume that x, y are distinct nodes in G so that $x\ I\ y$. Consider a two-counter machine M whose final state q_{final} has no outgoing transitions. We translate it into a valence reachability game, as in the proof of Proposition 19.2.5. For most types of transitions, the translation remains unchanged. Transitions $q \xrightarrow{\varepsilon} p$, $q \xrightarrow{x++} p$, and $q \xrightarrow{y++} p$ translate into $q \xrightarrow{\varepsilon} p$, $q \xrightarrow{x^+} p$, and $q \xrightarrow{y^+} p$, respectively.

For zero-test transitions, we use the same gadget as in Proposition 19.2.5. The gadget still functions under the presence of self-loops. The only difference is that a play may stay in the state $\hat{q}_\bigcirc$ infinitely long. This does not harm the correctness of the construction.

It remains to encode decrements $q \xrightarrow{x--} p$ of M. Note that we still assume that the configurations of M use non-negative counter values and the decrements are blocking. If x has no self-loop, we translate such a decrement as $q \xrightarrow{x^-} p$ as before. If x has a self-loop, we need to

be careful. The decrement $x{-}{-}$ of M is blocking, while the decrement x^- of the valence system is not. (Formally, x^- is right invertible if $x\ I\ x$ holds.)

We design a gadget that is similar to the gadget for zero tests from the proof of Proposition 19.2.5. The gadget is depicted in Figure 19.2.b. We replace a transition $q \xrightarrow{x--} p$ by a transition $q \xrightarrow{x^-} \hat{q}'_\Box$ to a fresh state owned by the universal player. The universal player can then either take the transition $\hat{q}'_\Box \xrightarrow{\varepsilon} p$ to the state p designated by the existential player. She can also challenge the decrement by using $\hat{q}'_\Box \xrightarrow{\varepsilon} \hat{q}'_\bigcirc$. In the fresh state $\hat{q}'_\bigcirc$, the existential player can decrement both counters arbitrarily often before going to the final state.

We argue that the gadget enforces the desired behavior. If we take the transition $q \xrightarrow{x--} p$ corresponding to $q \xrightarrow{x^-} \hat{q}'_\Box$ and the value of x was positive, the universal player loses if she challenges the decrement. If the value of x was positive, it is non-negative after using x^-. Furthermore, we assume that the rest of the game enforces that the value of counter y is also non-negative. Hence, the existential player can use the loops in $\hat{q}'_\bigcirc$ to bring both counters to zero (meaning that we obtain a monoid element equivalent to ε), then go to the final state with empty storage and win the play. If the counter value of x was zero, the universal player wins the play by challenging the decrement. If the value was zero before executing x^-, the value is negative after using x^-. The x^--labeled loop in $\hat{q}'_\bigcirc$ will not allow the existential player to get back to a non-negative counter value. Hence, she can either stay in $\hat{q}'_\bigcirc$ forever, or she can go to $\hat{q}'_\bigcirc$ with a non-empty storage because the counter value for x is still negative and then get stuck there. In both cases, she loses the play.

One can prove that a winning strategy for the existential player corresponds to a valid computation from $(q_{\text{init}}, 0, 0)$ to $(q_{\text{final}}, 0, 0)$ of M. The formal proof is very similar to the one of Proposition 19.2.5. ■

Together, Proposition 19.2.6 and Proposition 19.2.3 prove our classification result for the decidability of valence reachability games Theorem 19.2.1.

Coverability games

Our classification result for valence reachability games shows that they are decidable essentially only in the case of context-free games. This result is frustrating: The single decidable case is a model for which games have been known to be decidable for years. Furthermore, our result implies that more involved winning conditions like parity admit the same characterization. On the one hand, parity games can be seen as an extension of reachability games, so whenever reachability games are undecidable, so are parity games. On the other hand, context-free parity games are known to be decidable [Wal01]. Therefore, we will consider a weaker winning condition in the following in the hope to get decidability for a larger class of graphs.

Inspired by the research on Petri nets, where coverability is known to be easier to decide than reachability, we define *coverability* games in the following. Actually, we use the definition of

coverability for vector addition systems with states (VASSes) because the sequential nature of their computations means they are more closely related to valence systems. Petri nets and VASSes are equivalent models; while Petri net reachability corresponds to *configuration reachability* in VASSes, Petri net coverability corresponds to *control-state reachability*. In the control-state reachability problem, the goal is to reach a final control state with arbitrary counter values. Correspondingly, we define *valence coverability games* as reachability games on the game arena defined by a game valence system in which the goal is to reach the final control state q_{final} with arbitrary storage value. The formal definition of the decision problem is as follows.

Solving valence coverability games over graph G

Given: Game valence system $(\mathbb{M}_G, Q_\Box \uplus Q_\bigcirc, \delta, q_{\text{init}}, q_{\text{final}})$ where $\mathbb{M}_G$ is the graph of G.

Question: Can the existential player enforce visiting some (q_{final}, m) with arbitrary $m \in \mathbb{M}_G$ from $(q_{\text{init}}, \varepsilon)$?

Compared to valence reachability games, the game arena stays the same, only the winning condition has changed. The target set is now $\{(q_{\text{final}}, m) \mid m \in \mathbb{M}_G\}$ instead of the singleton set $\{(q_{\text{final}}, \varepsilon)\}$. Note that plays are still subject to the right-invertibility restriction: We can only reach configurations (q, m) for which m is right invertible. In fact, this restriction is what prevents us from taking an arbitrary path to the final control state in the valence system.

Our goal is to provide a classification result for valence coverability games, similar to the one for reachability games. Being able to state this result requires some notation. A *group* is a monoid in which every element has a right inverse. The following lemma gives a characterization for when the monoid associated to a graph is a group.

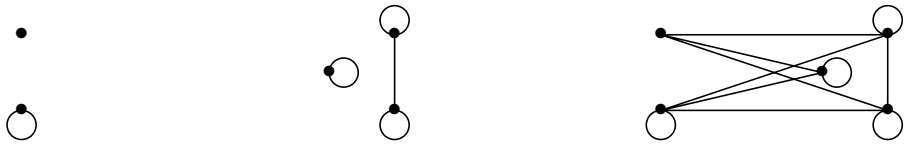
19.2.7 Lemma

The graph monoid associated to a graph G is a group if and only if the independence relation I is reflexive, i.e. if every node has a self-loop.

Proof:

If some node o has no self-loop, $o \neg I\ o$, then o^- has no right inverse, see Remark 19.1.2, and $\mathbb{M}_G$ is not a group.

If every node has a self-loop, consider a monoid element represented by some sequence $m \in \mathcal{O}^*$. We construct a right inverse by reversing m and swapping the polarity of all operations. The latter means replacing o^+ by o^- and vice versa. Let m^{-1} be the resulting sequence. We claim that $m.m^{-1} \cong \varepsilon$. Consider the last letter of m and the first letter of m^{-1}. Either the former is o^+ and the latter is o^-, in which case we can simply apply Rule (G1), or the former is o^- and the latter is o^-. Since $o\ I\ o$, we can use Rule (G2) to obtain $o^+.o^-$ and then apply Rule (G1). We iterate this process $|m|$ times to obtain a reduction that shows $m.m^{-1} \cong \varepsilon$. ■



i) A pushdown graph. ii) A group graph. iii) Their product.

Figure 19.2.c: The product of a pushdown graph and a group graph.

The lemma justifies the following definition. We call a graph a *group graph* if every node has a self-loop. For group graphs, coverability games are particularly simple.

19.2.8 Lemma
If G is a group graph, then valence coverability games over G are decidable.

Proof:
If G is a group graph, any monoid element is right invertible and the right-invertibility restriction does not apply. We can solve the valence coverability game as follows. Discard all transition labels and see the game valence system itself as a finite-state game in which the goal is to reach q_{final} from q_{init}. The valence coverability game and the finite-state game have the same winner, as a winning strategy for one game can be seen as a winning strategy for the other. Finite-state reachability games are decidable using the attractor construction as we have discussed in Chapter 15. ■

The lemma shows that group graphs essentially do not matter at all when it comes to coverability games. Basically, the absence of both the right-invertibility restriction and the obligation to reach a specific storage value means the storage operations can be ignored.

Our classification result will show that we can solve a coverability game if and only if the storage consists of two parts. One part is a group graph, and the associated storage operations can be ignored as in the proof of Lemma 19.2.8. The other part is a pushdown graph as defined in the context of Theorem 19.2.1, and this part of the graph has to be treated as in the case of reachability games.

Before finally stating the result, we need to make the notion of consisting of two parts precise. To this end, we define a product operation as follows. Let $G_1 = (V_1, I_1), G_2 = (V_2, I_2)$ be graphs whose sets of nodes are disjoint. Their product is the graph $G_1 \times G_2 = (V_1 \uplus V_2, I_1 \uplus I_2 \uplus V_1 \times V_2 \uplus V_2 \times V_1)$. Its set of nodes is the union of the nodes of G_1 and G_2. Its edges are the edges of G_1 among the nodes of G_1, the edges of G_2 among the nodes of G_2, and all edges that connect a node of G_1 to a node of G_2.

19.2.9 Example
a) Every graph can be seen as the product of itself and the empty graph.

b) The graph from Figure 19.1.a.ii) is a product of two copies of the graph from Figure 19.1.a.i). Indeed, the storage of a multi-pushdown system consists of several fully independent stacks.

c) The graph from Figure 19.1.a.iii) is a product of three copies of a graph consisting of a single node with no self-loop. Similarly, the graph from Figure 19.1.a.iv) is a product of three copies of a single node with a self-loop. Indeed, the storage of an (integer) VASS consists of several fully independent counters.

d) The graph from Figure 19.2.c.iii) is a product of the pushdown graph from Figure 19.2.c.i) and the group graph from Figure 19.2.c.ii).

Remark

One can also define the product operation on the level of monoids. The *(direct) product of two monoids* is a monoid whose elements are tuples consisting of an element of the first monoid and the second monoid. The operation on the product monoid is the component-wise application of the two monoid operations and the neutral element is the tuple of neutral elements.

We argue that the graph monoid associated to the product of two graphs is the product of the corresponding graph monoids. Let G_1, G_2 be graphs and let $G_\times = G_1 \times G_2$ be their product. Any element of $\mathbb{M}_{G_\times}$ can be represented by a sequence $m_1.m_2$ such that m_1 exclusively contains operations for the nodes of G_1, similar for m_2 and G_2. To this end, we use that the parts of the storage corresponding to G_1 and G_2 are fully independent, meaning $o_1 \; I \; o_2$ holds for any o from G_1 and o_2 from G_2. Hence, an element of $\mathbb{M}_{G_\times}$ can be seen as a tuple consisting of an element of $\mathbb{M}_{G_1}$ and an element of $\mathbb{M}_{G_2}$.

We call a graph G the *product of a pushdown and a group* if it can be written as $G_1 \times G_2$ where G_1 is a pushdown graph and G_2 is a group graph. Correspondingly, the associated graph monoid is the product of a graph monoid associated to a pushdown graph and a group. Unfolding the definitions, a graph G being the product of a pushdown and a group means that the set of nodes of G can be partitioned into $V = V_1 \uplus V_2$ such that (1) there are no edges among distinct nodes in V_1, (2) all nodes in V_2 have a self-loop, (3) for every pair $o_1 \in V_1, o_2 \in V_2$, there is an edge $o_1 \; I \; o_2$. The graph from Figure 19.2.c.iii) satisfies this property. Our classification result states that these graphs are exactly the ones for which valence coverability games are decidable.

19.2.10 Theorem

Valence coverability games over graph G are decidable if and only if G is the product of a pushdown and a group.

Before we prove the result, we discuss its implications. We have argued before that group graphs do not really matter when considering valence coverability games. Hence, our classi-

fication result essentially states that even with a much weaker winning condition, context-free games are still the only decidable case.

The proof of decidability is an easy extension of the proof for the decidability of reachability games, Proposition 19.2.3. The proof of undecidability, however, uses interesting techniques.

We show decidability by treating the pushdown part as before, while ignoring the part that belongs to the group. The latter is sound because elements of groups are always right invertible, and the corresponding part is independent of the rest. To show that all other cases are undecidable, we show that if the graph is not of the specified shape, we can reduce undecidable reachability games to coverability games. The rest of this section is dedicated to making both ideas precise.

Valence coverability games – The decidable case

Assume that graph $G = (V, I)$ is the product of a pushdown and a group, where $V = P \uplus S$ is the corresponding partitioning of the nodes. Let us denote by $\mathcal{O}_P$ and $\mathcal{O}_S$ the operations for nodes in P and S, respectively. We have argued before that any monoid element m of $\mathbb{M}_G$ can be represented using a sequence $m_P.m_S$ with $m_\in \mathcal{O}_P^*$ and $m_S \in \mathcal{O}_S^*$. We claim that m is right invertible if and only if m_P is. Indeed, the part m_S that corresponds to an element of a group always has a right inverse m_S^{-1} as discussed in the proof of Lemma 19.2.7. If m_P has a right inverse m_P^{-1}, then $m_P^{-1}.m_S^{-1} \cong m_S^{-1}.m_P^{-1}$ is a right inverse for $m_P.m_S$. Vice versa, a right inverse for $m_P.m_S$ can be written as $m_P^{-1}.m_S^{-1}$ where m_P^{-1} is a right inverse for m_P. Altogether, we obtain that the part of the graph that corresponds to the group graph can be simply ignored, as in the proof of Lemma 19.2.8. The formal construction is as follows.

19.2.11 Proposition
If G is the product of a pushdown and a group, valence coverability games over G are decidable.

Proof:
In graph G, discard all nodes S that correspond to the group graph. In the game valence system, replace all transitions labeled by $o^\pm$ with $o \in S$ by ε-labeled transitions. The result is a valence coverability game over a pushdown graph that is equivalent to the valence coverability game over G, because the operations corresponding to the group part do not matter.

To solve this game, we proceed as in the proof of Proposition 19.2.3. We first apply preprocessing to all nodes that have a self-loop. The result is a pushdown game in which the goal is to reach the final state with arbitrary stack content. This target set is regular and can be solved using Cachat's algorithm [Cac02], as discussed in the proof of Lemma 19.2.2. ■

This proves one direction of Theorem 19.2.10.

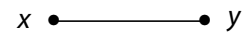


Figure 19.2.d: The 1st illegal graph

.

Valence coverability games – The undecidable case

Showing that valence coverability games over graph G are undecidable if G is not the product of a pushdown and a group is more involved. We proceed as follows: We identify three particular graphs for which we show undecidability. Then, we use the fact that decidability is monotonic with respect to the induced subgraph order: If graph G contains a graph for which the problem is undecidable as an induced subgraph, then the problem for G also has to be undecidable. Finally, we show that any graph that contains none of the three graphs as an induces subgraph is the product of a pushdown and a group.

We start by considering the first of the so-called *illegal graphs* for which we will show undecidability. It is depicted in Figure 19.2.d. It consists of two distinct nodes x, y that have no self-loops but are connected by an edge $x \; I \; y$.

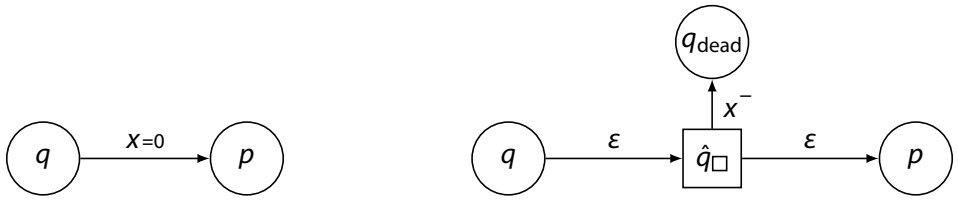
19.2.12 Lemma
Valence coverability games over the 1st illegal graph are undecidable.

The proof is similar to the proof of Proposition 19.2.5. However, we reduce from the *control-state reachability problem for two-counter machines*. This problem consists of checking whether there is a computation that starts in $(q_{\text{init}}, 0, 0)$ and reaches the final state q_{final} with arbitrary counter values. Formally, it is defined as follows.

Control-state reachability problem for two-counter machines	
Given:	Two-counter machine M with initial state q_{init} and final state q_{final}.
Question:	Is there a computation $(q_{\text{init}}, 0, 0) \rightarrow^* (q_{\text{final}}, n, m)$ for some $n, m \in \mathbb{N}$?

Since two-counter machines are a Turing-complete model, control-state reachability is just as undecidable as configuration reachability.

Proof of Lemma 19.2.12:
Assume we are given a two-counter machine M as an instance of the control-state reachability problem. We construct an equivalent valence coverability game over the 1st illegal graph. The construction is similar to the one in the proof of Proposition 19.2.5. We see all control states of M as control states of the game valence system owned by the existential player. We translate a transition $q \xrightarrow{\varepsilon} p$ in M into a transition $q \xrightarrow{\varepsilon} p$ in the valence game. We translate $q \xrightarrow{x++} p$ and



i) A zero-test transition in M. ii) Its translation in the coverability game.

Figure 19.2.e: The translation of a zero-test transition for coverability games.

$q \xrightarrow{x--} p$ as $q \xrightarrow{x^+} p$ and $q \xrightarrow{x^-} p$, respectively, similar for counter y. Note that x and y have no self-loops in the 1st illegal graph. Hence, the decrements are blocking and the semantics of all aforementioned operations in M and in valence coverability games coincide.

It remains to encode zero tests. We replace a zero-test transition $q \xrightarrow{x=0} p$ by a gadget consisting of several transitions and fresh control states. The translation is depicted in Figure 19.2.e. We first go to a fresh control state $\hat{q}_\Box$ owned by the universal player using an ε-labeled transition. In this state, the universal player can either accept that the counter value of x is zero and go to state p. If the counter value is not zero, the universal player can prove this by taking an x^--labeled transition to a deadlock state q_{dead}.

We claim that this gadget works as intended: The existential player can only take zero-test transitions if the current counter value is indeed zero. If the counter value is zero, the x^--labeled transition to q_{dead} is not enabled as the decrement is blocking. The universal player has no choice but to proceed to state p. If the counter value is non-zero, the universal player can move to q_{dead}. In this deadlock state, the game gets stuck and the existential player loses.

We use one fresh state $\hat{q}_\Box$ for every zero-test transition (but we do not reflect this in the notation to improve readability). We may use the same deadlock state q_{dead} for all zero-test transitions. For zero tests of the counter y, we use a version of the gadgets in which the transition to q_{dead} is labeled by y^-.

The formal proof that the existential player wins the coverability game if and only if M is a yes-instance of the control-state reachability problem is as in the proof of Proposition 19.2.5. ■

Our gadget from Figure 19.2.e encoding zero tests is much simpler than the one from Figure 19.2.a. However, the simple gadget relies on the nodes x, y not having self-loops and the decrements being blocking. Unlike the simple gadget, the more complex gadget in the proof of Proposition 19.2.5 generalizes to the case of the nodes having self-loops, so we were able to reuse it in the proof of Proposition 19.2.6.

We can now consider the 2nd illegal graph for which we show undecidability. It is depicted in Figure 19.2.f. Actually, we do not consider a single graph, but a set of graphs that are similar. Each of the graphs consists of three nodes x, y, and a such that a has no self-loop, x and y are connected by an edge, and a is connected to neither x nor y. Both x and y may or may not

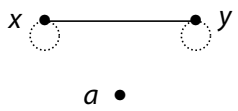


Figure 19.2.f: The 2nd illegal graph (or rather the set of such graphs).

have a self-loop; we obtain a set of four graphs in which none, one, or both of these nodes have a self-loop. We have depicted these self-loops that may or may not be present as dotted edges in Figure 19.2.f.

19.2.13 Lemma
Valence coverability games over (any of the) 2nd illegal graph(s) are undecidable.

This time, we will not need to give a reduction from two-counter machines, although it would not be too difficult to do so. Instead, we will be able to reduce from valence reachability games using the undecidability results that we have established in the first half of this section.

Proof:
Let G be one of the 2nd illegal graphs. Consider the subgraph G' induced by the set of nodes $\{x, y\}$. It is not a pushdown graph since $x \; I \; y$. By Theorem 19.2.10, reachability games over G' are undecidable. We show that if we could solve coverability games over G, then we could solve reachability games over the subgraph, a contradiction.

Consider a reachability game over the subgraph, say defined by the game valence system $(\mathbb{M}_{G'}, Q_\Box \uplus Q_\bigcirc, \delta, q_{\text{init}}, q_{\text{final}})$. We construct a new game valence system $(\mathbb{M}_G, \{\hat{q}_{\text{init}}, \hat{q}_{\text{final}}, \hat{q}_\bigcirc\} \uplus Q_\Box \uplus Q_\bigcirc, \hat{\delta}, \hat{q}_{\text{init}}, \hat{q}_{\text{final}})$ over G as follows. We keep all states and most of the transitions of the original game valence system. Furthermore, we add three new control states, all owned by the existential player, and some transitions.

Firstly, we add a new initial state $\hat{q}_{\text{init}}$ and a transition $\hat{q}_{\text{init}} \xrightarrow{a^+} q_{\text{init}}$ to the old one. Secondly, we replace every transition $q \xrightarrow{o} q_{\text{final}}$ going to the old final state by a transition $q \xrightarrow{o} \hat{q}_\bigcirc$ to $\hat{q}_\bigcirc$ with the same label. The state $\hat{q}_\bigcirc$ has two transitions: $\hat{q}_\bigcirc \xrightarrow{\varepsilon} q_{\text{final}}$, an ε-labeled transition to the old final state, and $\hat{q}_\bigcirc \xrightarrow{a^-} \hat{q}_{\text{final}}$, an a^--labeled transition to the new final state.

The effect of the first modification is obvious: It puts an a^+ into the storage that will be present for the rest of the play until it may be eventually canceled by the a^--labeled transition to $\hat{q}'_{\text{final}}$. (Note that the transitions of the given reachability game do not use transitions labeled by operations for a.) The second modification kicks in whenever a play would visit the old final state q_{final}. The existential player has the choice to either prove that the storage is empty but for the initial a^+ by going to q'_{final} with a^-. Alternatively, she can accept that this is currently not the case by going to q_{final}.

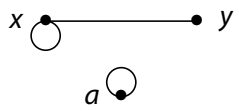


Figure 19.2.g: The 3rd illegal graph

.

Key to the correctness of the construction is that node a has no self-loop and that it is not connected by an edge to both x and y. Assume that the play has entered configuration $(q_{\bigcirc}, a^+.m)$, where m exclusively consists of operations for the nodes x and y. We have that $a^+.m.a^-$ is right invertible if and only if $m \cong \varepsilon$. Hence, the a^--labeled transition can only be taken if the storage is empty but for a^+.

With this observation, a winning strategy for the reachability can be turned into a winning strategy for the coverability game and vice versa. Assume that the existential player wins the reachability game. We construct a strategy for the coverability game. We start by taking the a^+-labeled transition to the old initial state. Then, we mimic the winning strategy for the reachability game. Whenever the play of the reachability game would visit q_{final} with non-empty storage, we take the sequence of transitions $q \xrightarrow{o} \hat{q}_{\bigcirc} \xrightarrow{\varepsilon} q_{\text{final}}$. When the play eventually visits q_{final} with empty storage, we take the sequence of transitions $q \xrightarrow{o} \hat{q}_{\bigcirc} \xrightarrow{a^-} \hat{q}_{\text{final}}$ to the new final state.

Vice versa, a winning strategy for the coverability game can be turned into a winning strategy for the reachability game by essentially just removing the a^+-labeled transition at the beginning and the a^--labeled transition at the end.

We have reduced the undecidable problem of solving reachability games over a graph that is not a pushdown graph to the problem of solving coverability games over the 2nd illegal graph. ■

Finally, we consider the 3rd illegal graph, depicted in Figure 19.2.g. It consists of three nodes x, y, and a. As in the 2nd illegal graph, x and y are connected by an edge. The nodes a and x have self-loops while y does not.

19.2.14 Lemma
Valence coverability games over the 3rd illegal graph are undecidable.

The proof is similar to the proof for the 2nd illegal graph. However, we have to extend the construction. In the proof of Lemma 19.2.13, we have used that $a^+.m.a^-$ is right invertible if and only if $m \cong \varepsilon$. This fact relies on node a not having a self-loop, which makes the a^- operation blocking. Since a has a self-loop in the 3rd illegal graph, using a^+ at the beginning of the play will be sufficient. Instead, we will use $y^+.a^+$ at the beginning and $a^-.y^-$ at the end.

Proof of Lemma 19.2.14:
Consider the subgraph of the 3rd illegal graph induced by the set of nodes $\{x, y\}$. Assume we are given a valence reachability game for this subgraph. Note that solving such games is undecidable by Theorem 19.2.1.

We construct a valence coverability game over the 3rd illegal graph as follows. We make sure that every play starts with the transitions $\xrightarrow{y^+}\xrightarrow{a^+}$ before going to the initial state of the given game To this end, we insert a new initial state and another fresh state as in the proof of Lemma 19.2.13.

Whenever there is a transition $q \xrightarrow{o} q_{\text{final}}$ entering the final state in the given game, we give the existential player the choice to either go to the old final state with operation o, or whether to go to the new final state using a sequence of transitions $\xrightarrow{o}\xrightarrow{a^-}\xrightarrow{y^-}$.

We argue that this construction is correct. Every play of the coverability game that reaches the new final state does so with a sequence of storage operations of the shape $y^+.a^+.m.a^-.y^-$, where m exclusively consists of operations over x and y. We claim that the final y^- transition is enabled if and only if $m \cong \varepsilon$. Obviously, $m \cong \varepsilon$ implies that $y^+.a^+.m.a^-.y^- \cong y^+.a^+.a^-.y^- \cong y^+.y^- \cong \varepsilon$ is right invertible.

For the other direction, assume that m is not equivalent to ε. In this case, $a^+.m.a^-$ is not equivalent to ε. Here, we use that a^+ and a^- cannot be swapped with any of the operations in m since $a \neg I\ x$ and $a \neg I\ y$. If $a^+.m.a^-$ is not equivalent to ε, then $y^+a^+.m.a^-$ is not equivalent to any sequence of operations that ends with y^+ Since $a \neg I\ y$ we can neither swap the y^+ from the beginning to the end, nor can we swap any y^+ that might be contained in m to the end. Consequently, $y^+.a^+.m.a^-.y^-$ is not right invertible. The final y^- cannot be canceled using an y^+ operation in the prefix $y^+.a^+.m.a^-$ as argued before. Furthermore, it can also not be canceled by a hypothetical later occurrence of an y^+ operation. Node y has no self-loop, so y^+ is not the right inverse of y^-.

One can now prove that the new coverability game is equivalent to the reachability game. The details are as in the proof of Lemma 19.2.13. ■

Before proving the result, we formally observe that decidability is monotonic with respect to the induced subgraph order.

19.2.15 Lemma
If graph G contains any of the three illegal graphs as an induced subgraph, valence coverability games over G are undecidable.

Proof:
Using the Lemmas 19.2.12, 19.2.13 and 19.2.14, we know that valence coverability games over the illegal graphs are undecidable. Assume G contains one of the illegal graphs as an induced

subgraph. Valence coverability games for that subgraph can be seen as valence coverability games over graph G that do not use the operations for any nodes not contained in the subgraph. If we could decide coverability games over graph G, we could also solve coverability games for the subgraph, a contradiction. ■

We can finally prove the remaining implication for Theorem 19.2.10.

19.2.16 Proposition
If graph G is not the product of a pushdown and a group, valence coverability games over G are undecidable.

Using Lemma 19.2.15, it is sufficient to show that any graph that is not the product of a pushdown and a group contains one of the illegal graphs as induced subgraph. We use contraposition and show that if a graph does not contain one of the illegal graphs, then it has to be the product of a pushdown and a group. We start with a general graph that may or may not contain any edge that is not a self-loop. Then, we iteratively show the existence or non-existence of some edges, until we obtain a graph that is the product of a pushdown and a group.

The most accessible version of this proof is via a sequence of pictures, Figure 19.2.h. For the sake of completeness, we also provide a formal write-up below.

Proof:
Using Lemma 19.2.15 and contraposition, it is sufficient to show that any graph that does not contain an illegal graph is the product of a pushdown and a group.

We proceed in several steps. The enumeration corresponds to the subfigures of Figure 19.2.h.

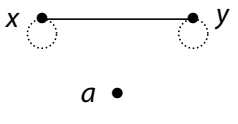
i) Consider an arbitrary graph (V, I). We may partition $V = E \uplus S$ so that each $s_j \in S$ has a self-loop, $s_j \; I \; s_j$, and each $e_i \in E$ does not, $e_i \; \neg I \; e_i$. All edges among distinct nodes may or may not exist.

   If the set E is empty, the graph is a group graph. Hence, it is the product of itself and an empty pushdown graph. It satisfies the requirements and we are done. In the following, we assume that there is at least one $e_i \in E$.

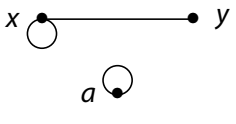
   If there are two distinct $e_i, e_{i'} \in E$, $e_i \neq e_{i'}$ that are connected by an edge, $e_i \; I \; e_{i'}$, the graph contains the 1^{st} illegal graph as induced subgraph. To see this, define $x = e_i$ and $y = e_{i'}$. Hence, we may restrict ourselves to graphs in which there is no edge among the nodes in E.

   If the set S is empty, we have shown that the graph is a pushdown graph. Hence, it is the product of itself and an empty group graph, and we are done. In the following, we will assume that there is at least one $s_j \in S$.

1st illegal graph.

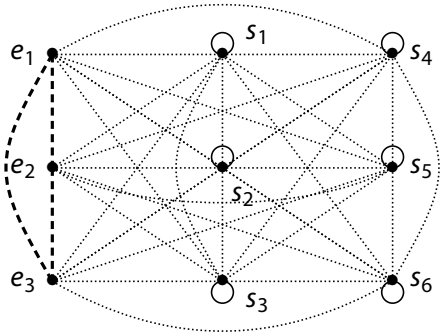


2nd illegal graph.

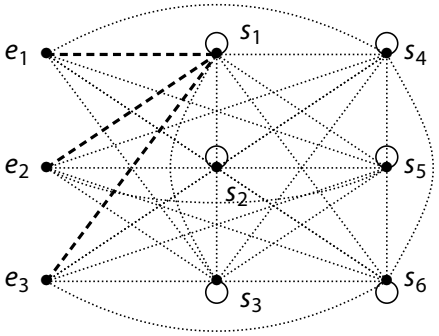


3rd illegal graph.

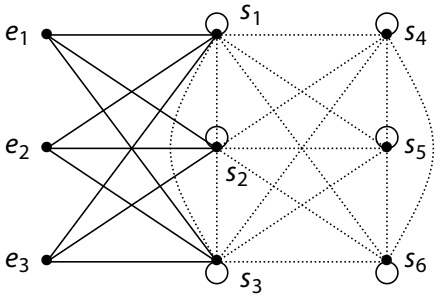
Recall: The three illegal graphs.



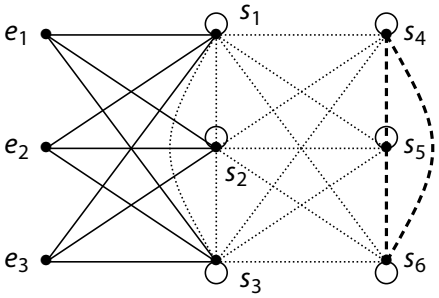
i) A general graph in which some nodes (the s_j) have self-loops while others (the e_i) do not. All dotted edges may or may not exist. If any of the thick dashed edges among the e_i exists, the graph contains the 1st illegal graph.



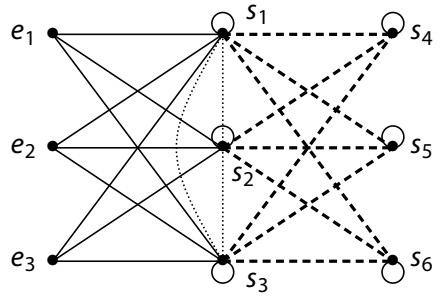
ii) If any of the thick dashed edges from the e_i to s_1 exists, all of them have to exist. Otherwise, the graph would contain the 2nd illegal graph. A similar reasoning holds for all s_j.



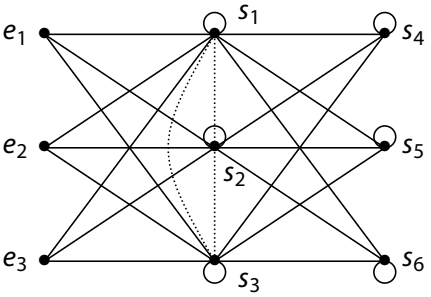
iii) Hence, some of the s_j ($S_g = \{s_1, s_2, s_3\}$) are adjacent to all e_i while the others ($S_p = \{s_4, s_5, s_6\}$) are not adjacent to any e_i.



iv) If any of the thick dashed edges among the nodes in $S_p = \{s_4, s_5, s_6\}$ exists, the graph contains the 2nd illegal graph



v) If any of the thick dashed edges from $S_g = \{s_1, s_2, s_3\}$ to $S_p = \{s_4, s_5, s_6\}$ is missing, the graph contains the 3rd illegal graph.



vi) The resulting graph is the product of a push-down with nodes $\{e_1, e_2, e_3, s_4, s_5, s_6\}$ and a group with nodes $\{s_1, s_2, s_3\}$. Dotted edges among $\{s_1, s_2, s_3\}$ may or may not exist.

Figure 19.2.h: The proof of Proposition 19.2.16 in pictures.

ii) If some $s_j \in S$ is adjacent to at least one $e_i \in E$, $e_i \; I \; s_j$, this s_j needs to be adjacent to all e_i. This property is trivially satisfied if there is just a single $E = \{e_i\}$. Otherwise, consider distinct elements $e_i, e_{i'} \in E$ and assume $e_{i'} \; \neg I \; s_j$. By defining $x = e_i$, $y = s_j$ and $a = e_{i'}$, we see that the graph would contain the 2^{nd} illegal graph as induced subgraph.

iii) The previously stated property holds for all elements of S. Hence, we may partition $S = S_g \uplus S_p$ such that the $s_g \in S_g$ are adjacent to all $e_i \in E$ while the $s_p \in S_p$ are not adjacent to any $e_i \in E$.

iv) If S_p contains distinct elements $s_p, s_{p'}$ that are connected by an edge, $s_p \; I \; s_{p'}$, the graph would contain the 2^{nd} illegal graph as induced subgraph. To see this, consider $e_i \in E$ and note that it is not adjacent to any element of S_p. We may set $a = e_i$, $x = s_p$ and $y = s_{p'}$ to identify the 2^{nd} illegal graph.

v) If the set S_g is empty, the graph is a pushdown graph and we are done. Indeed, there are no edges among the nodes in E, see Step i), no edges among the nodes in S_p, see Step iv), and no edges that connect an element of E to an element of S_p, see Step ii). Hence, assume that S_g is non-empty.

   If the set S_p is empty, the graph is a product of a group and a pushdown and we are done. The pushdown graph consists of the set of nodes E, while the group graph consists of the set of nodes S_g. Indeed, there are no edges among the elements of E, each node in S_g has a self-loop, and every $e_i \in E$ is adjacent to every $s_g \in S_g$. Hence, assume that S_p is non-empty.

   We claim that every $s_p \in S_p$ needs to be adjacent to every $s_g \in S_g$. Assume that $s_p \; \neg I \; s_g$ holds. Then we can identify the 3^{rd} illegal graph as an induced subgraph by defining $x = s_g$, $a = s_p$, and $y = e_i$ for some $e_i \in E$. Indeed, s_g is adjacent to e_i but s_p is not.

vi) We obtain a graph with set of nodes $E \uplus S_g \uplus S_p$. We claim this graph is the product of a pushdown and a group. The set of nodes belonging to the pushdown graph is $E \uplus S_p$ while the nodes in S_g belong to the group graph. Indeed, there are no edges among the nodes in $E \uplus S_p$, see Steps i), ii) and iv). Furthermore, each $s_g \in S_g$ is adjacent to any node is $E \uplus S_p$, see Steps iv) and v). The edges in $S_g \subseteq S$ have self-loops. Edges among the S_g may or may not exist. ■

Proposition 19.2.16 shows the missing direction for the proof of our classification result, Theorem 19.2.10. This completes our study of valence reachability and coverability games.

19.3 Bounded context switching for valence systems

In the last section, we have seen that reachability games over valence systems over graph monoids are decidable exclusively for graphs that correspond to pushdown systems. In this section, we present a restricted version of these reachability games and conjecture that they are decidable for any underlying graph. To justify the conjecture, we show that a similar restriction applied to the reachability problem leads to decidability in all cases.

Bounded context switching

The restriction that we will consider is *bounded context switching (BCS)* [QR05]. Consider a concurrent system. A context is a part of a computation in which only one component or thread is active. Bounding the number of context switches means that instead of each component becoming active arbitrarily often, there is a limit on how often the active component can change. We may see a system with a BCS restriction in place as an underapproximation of the real system. If the restricted system shows undesired behavior, e.g. an error location in the source code is reachability, the same is true for the unrestricted system. The BSC restriction is popular for two reasons. Firstly, it has been empirically shown that bugs in concurrent systems often occur within few context switches [MQ07; LPSZ08]. Secondly, the BCS restriction usually drastically improves the decidability and computational complexity of decision problems.

For example, consider multi-pushdown systems, systems that use several stacks as storage. We may think of such a system as a concurrent system consisting of one pushdown automaton per component. It is well known that this type of system is Turing-complete if we have at least two stacks with two stack symbols each. Hence, reachability is undecidable [Ram00]. In a context, only a single component is active. Hence, also just a single stack is used in a context. Hence, the BCS restrictions limits changing the active stack. Given a bound on the number of context switches encoded in unary, the reachability problem for the resulting model is not just decidable, but in fact NP-complete [QR05].

In the following, we want to define a BCS restriction for valence systems. In the case of valence systems, there is no natural notion of thread or component. Instead of introducing such a notion (which would then also force us to specify how the components communicate), we will base our notion of contexts exclusively on the storage. This is inspired by the definition of BCS for multi-pushdown systems, where context switches occur when the stack that is currently used changes.

In a valence system that consists of several components, we would expect each of the components to have a storage that is independent of the part of the storage used by every other component. Using the terminology introduced in the previous section, we would expect the storage graph to be the product of several graphs. Each graph that is a factor of the product corresponds to the storage used by one component.

A context switch should separate parts of the computation that use independent parts of the storage. For example, if $a \ I \ b$, then the monoid element a^+b^+ contains a context switch. A drawback of the storage-based definition is that the notion of context switch is not invariant under the congruence $\cong$. Hence, it is not well-defined on monoid elements, but rather on their representatives. For example, if $a \ I \ b$, then $a^+b^+b^-a^- \cong a^+a^- \cong \varepsilon$, where $a^+b^+b^-a^-$ contains two context switches, but a^+a^- and ε contain none. To circumvent the problems that arise from this, we redefine valence systems over graph monoids so that their configurations do not store monoid elements, but representatives. Formally, we consider configurations from $Q \times \mathcal{O}^*$ that consists of a control state and a sequence of operations that has been executed. In a configuration (q, m), we can use a transition $q \xrightarrow{m'} q'$ and move to $(q', m.m')$ if $m.m'$ represents a right-invertible monoid element. For the reachability problem, we consider the unique initial configuration $(q_{\text{init}}, \varepsilon)$. Solving the reachability problem is checking whether a configuration (q_{final}, m) with $m \cong \varepsilon$ is reachable. For a given finite syntactic representation of a valence system, this version of the reachability problem is equivalent to our definition from Section 19.1. We have just changed the representation of the (potentially infinite) transition system arising from the finite syntax.

We come back to the definition of context switching. For our earlier example, it seems reasonable to define a context switch to occur between two operations $a^\pm b^\pm$ if $a \ I \ b$. However, it turns out that this definition is not restrictive enough regarding distinct nodes $a \neq b$ and too restrictive if $a = b$. On the one hand, consider the case of a single node a that has a self-loop, $a \ I \ a$. Using operations for a successively would introduce one context switch per operation and heavily limit the usability of this part of the storage. Luckily, our definition of a context switch can avoid being so restrictive. On the other hand, consider the graph with three nodes a, b, c such that $a \ I \ b$ is the only edge. In the sequence $a^+c^+b^+$, no two adjacent operations are associated to nodes that are independent. However, the sequence contains a^+ and b^+ and uses independent parts of the storage. For the theory that we will develop in the rest of this section to work, we will need to see a^+c^+, b^+ as a context switch.

The above discussion entails the following formal definition.

19.3.1 Definition
A set $V' \subseteq V$ of nodes of the graph (V, I) is *dependent* if for any $o_1, o_2 \in V'$ with $o_1 \neq o_2$, $o_1 \neg I \ o_2$ holds. A set of operations is dependent if the underlying set of nodes is. A sequence $m \in \mathcal{O}^*$ is dependent if the set of operations used in that sequence is.

The *first context* of a sequence $m \in \mathcal{O}^*$ is its maximal dependent prefix. The decomposition into contexts is obtained inductively: A dependent sequence is a single context, a non-dependent sequences decomposes into its first context and the decomposition of the remaining suffix. The *number of context switches* of a sequence is the number of contexts minus one.

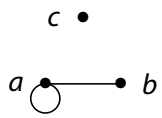


Figure 19.3.a: A graph for which we consider the BCS restriction.

We demonstrate the definition on an example. Consider a graph with the nodes $\{a, b, c\}$ and the edges $a \; I \; a$ and $a \; I \; b$. It is depicted in Figure 19.3.a. Note that this graph cannot be seen as the product of several non-empty graphs. Consider the sequence of operations $m = a^+a^+a^-c^+b^+b^-c^-b^+a^-b^-$ and note that $m \cong \varepsilon$. The sequence m has three context switches and consists of four contexts. Its maximal dependent prefix is $a^+a^+a^-c^+$. The next operation b^- starts a new context because $a \; I \; b$ are two distinct nodes connected by an edge. The remaining contexts are $b^+b^-c^-b^+$, a^-, and b^-.

For more examples, let us consider the graphs from Figure 19.1.a resp. Example 19.1.4. Graphs that represent pushdown systems do not contain edges, so every computation consists of a single context. Reachability and reachability under BCS coincide. For graphs that represent multi-pushdown systems like the graph C_4, our definition of BCS coincides with the well-known definition of BCS from the literature: A context is an infix of the computation in which just one stack is active. For graphs representing VASS or integer VASS, switching the context means using a different counter.

Reachability under bounded context switching

With the notion of context switches at hand, we can formally define the BCS restriction for the reachability problem.

Valence reachability under bounded context switching	
Given:	Valence system $(\mathbb{M}_G, Q, \delta, q_{\text{init}}, q_{\text{final}})$ over graph G, bound $k \in \mathbb{N}$ encoded in unary.
Question:	Is there a computation $(q_{\text{init}}, \varepsilon) \to^* (q_{\text{final}}, m)$ such that $m \cong \varepsilon$ and m has at most k context switches?

We will discuss this problem in detail later. For now, we want to focus on an extended version: valence reachability games under the BCS restriction.

Reachability games under bounded context switching

We can combine the definition of valence reachability games and valence reachability under BCS into a single definition of valence reachability games under bounded context switching. Assume we are given a game valence system over a graph monoid and a bound $k \in \mathbb{N}$ on the number of context switches. The goal of the existential player is, starting from $(q_{\text{init}}, \varepsilon)$ to reach (q_{final}, m) where $m \cong \varepsilon$ represents the empty storage and has at most k context switches.

Here, we use the modified semantics from earlier in which we do not just keep track of monoid elements, but instead track the sequence of operations.

When formally defining valence reachability games under BCS, we have a choice to make. Both players can influence the number of context switches in a play of the game. In one variant, we give the obligation of ensuring that the bound on the number of context switches is not exceeded to the existential player. She will lose all plays that contain too many context switches. If a play already contains the maximum number of allowed context switches, the universal player can enforce her win by choosing another context switch. The second option is to block all transitions that introduce more than the allowed number of context switches, no matter which player is active. The first option has the advantage that a winning strategy for the existential player in the restricted game is also a winning strategy in the game without the BCS restriction. This is because the BCS restriction only limits the moves of the existential player, while the universal player can still react using all moves in the unrestricted game.

In the following, we will consider the second variant. We claim that it is powerful enough to simulate the first variant. To this end, we modify the underlying valence system so that it keeps track of the number of context switches that have already occurred and the set of dependent nodes that have been used in the current context. The latter is needed to identify the next context switch. For the simulation, we simply replace every transition that would introduce the $(k+1)^{\text{st}}$ context switch by an ε-labeled transition that leads to a deadlock. If in the original game, the existential player is forced to introduce a $(k+1)^{\text{st}}$ context switch or if the universal player is able to do so, the modified game will move to this deadlock state. The final state has not been reached and the existential player loses. Note that the game is designed so that it is impossible for a computation to have more than k context switches, so applying the second restriction does not change the semantics.

We conjecture that for every graph monoid, valence reachability games over the associated graph monoid with the bounded-context-switching restriction are decidable.

19.3.2 Conjecture

Reachability games on valence systems over graph monoids with BCS are decidable.

While we have no proof for this statement, we have two reasons justifying the conjecture. The first one is the fact that decidability has been shown in the case of games over multi-pushdown systems. In fact, Seth [Set09] has a shown that multi-pushdown games are decidable both with a bounded-context-switching and a bounded-phase-switching restriction. Bounded phase switching is a weaker restriction than bounded-context switching (BCS) that allows more computations: In each phase, pushes can be executed on all stacks, but pops can only occur on a single stack. Our definition of BCS for valence systems over graph monoids is a generalization of BCS for multi-pushdown systems. One can hope that this also means that the decidability of reachability games generalizes from multi-pushdown games to valence games.

Note that Seth's algorithm is rather involved. It is a generalization of Walukiewicz's decision procedure [Wal01] for context-free games that we have discussed in Section 17.6. In fact, it can be seen as a k-fold application of Walukiewicz's procedure, where k is the bound on the number of phases or contexts. Hence, the procedure is k-fold exponential. It has been shown that the problem of solving such games is $(k-2)$EXP-complete if k is fixed [MW20] and non-ELEMENTARY [ABKS12; ABKS17] if k is part of the input.

The second justification for our conjecture is the following result that Meyer, Zetzsche, and the author of this thesis have shown in [MMZ18]. It proves that the aforementioned non-game version of reachability in valence systems over graph monoids is always decidable if we assume a bounded number of context switches. It even leads to an upper bound for the complexity.

19.3.3 Theorem
Valence reachability under bounded context switching is always decidable in NP.

The rest of this section is an outline of the proof of the above result. We do not give the technical details, and refer the reader to the full version of the paper [MMZ18a]. Afterwards, we comment on which techniques from the proof carry over to the game setting and which do not.

Proof sketch for Theorem 19.3.3

Consider a computation $(q_{\text{init}}, \varepsilon) \to^* (q_{\text{final}}, m)$, where $m \in \mathcal{O}^*$ consists of exactly k context switches, $m = m^{(1)} \dots m^{(k)}$. While the number of context switches is bounded, the length of each $m^{(i)}$ is not. To deal with this issue, our goal is to represent the part of the valence system that contributes to each context as a separate finite automaton. However, this turns out to be insufficient. There are two problems: Firstly, a single context may contain cancellations, i.e. a context may contain $a^+a^- \cong \varepsilon$. This cannot be captured by a finite automaton. Secondly, two operations in the same context may be canceled out by operations in different contexts. For example, we might have m as above with $m \cong \varepsilon$, where $m^{(i)} = a^+b^+$ for some i. The operation a^- that cancels out a^+ in $m^{(i)}$ may be contained in some context j, and the b^- associated to b^+ may be contained in some other context j' with $j \neq j'$.

To overcome the first problem, we restrict ourselves to irreducible contexts. Normally, an irreducible sequence of operations is one to which we cannot apply the cancellation rule, Rule (G1), even after potentially applying the swapping rule, Rule (G2). If we only consider contexts, the definition can be simplified. A context contains only operations that form a dependent set and the swapping rule can never be applied to operations for distinct nodes. Formally, a context is *irreducible* if it does not contain the infix $a^+.a^-$ for an arbitrary symbol or $a^-.a^+$ for a symbol a with $a \mathrel{I} a$.

To ensure that the restriction to irreducible contexts is valid, we apply a preprocessing step. We saturate the given valence system so that if there is a computation $(q_{\text{init}}, \varepsilon) \to^* (q_{\text{final}}, m)$ with $m \cong \varepsilon$ and at most k context switches, then there is also a computation $(q_{\text{init}}, \varepsilon) \to^* (q_{\text{final}}, m')$

where m' additionally satisfies that its contexts are irreducible. The saturation works by finding states q, p, p', q' such that $q \xrightarrow{a^+} p, p' \xrightarrow{a^-} q'$ and there is a sequence of ε-labeled transitions from p to p'. We then introduce a new ε-labeled transition from q to q'. Intuitively, the new ε-labeled transition allows the computation to skip the operations that would lead to a context being not irreducible. For symbols a with $a \mathrel{I} a$, we additionally at transitions that can be used to skip a sequence of transitions of the shape $\xrightarrow{a^-}\xrightarrow{\varepsilon}{}^*\xrightarrow{a^+}$. This process is repeated exhaustively, until no new transitions are added. Adding an ε-transition might lead to another transition being added in the next iteration, but since we only add transitions but no control states, the process terminates after at most a quadratic number of steps.

We come back to the first problem. The different operations in some fixed contexts might cancel out (or get canceled out by) operations in different contexts. To this end, we consider block decompositions and block-wise reductions. A block decomposition is decomposition of the irreducible contexts into infixes, e.g. $m^{(i)} = m^{(i,1)} \dots m^{(i,k)}$. A block-wise reduction is a proof of $m \cong \varepsilon$ that uses modified reduction rules. The rules for cancellation and swapping Rule (G1) and Rule (G2) work on individual operations in a sequence. The block-wise reduction rules work on blocks of operations. A block $m^{(i,j)}$ is canceled by $m^{(s,t)}$ if $m^{(s,t)}$ is a representative for the right inverse of $m^{(i,j)}$. A block $m^{(i,j)}$ can be swapped with $m^{(s,t)}$ if for every operation $o^{\pm}$ in $m^{(i,j)}$ and every operation $u^{\pm}$ in $m^{(s,t)}$, $o \mathrel{I} u$ holds.

Assume we are given a sequence m and a decomposition into blocks. Requiring that there is a block-wise reduction showing $m \cong \varepsilon$ is a property that is strictly stronger than just requiring $m \cong \varepsilon$. Consider $m = a^+b^+b^-a^-$ and the block decomposition a^+b^+, b^-, a^-. We have $m \cong \varepsilon$, but there is no block-wise reduction to ε since a^+b^+ is neither canceled by b^- nor by a^-. We have that if $m \cong \varepsilon$, then m admits a block-wise reduction to ε if we decompose it into blocks of length one that consists of single operations. In this case, the block reduction rules are equal to Rule (G1) and Rule (G2).

The crucial result that we need for our algorithm is the following. Consider a sequence m with $m \cong \varepsilon$ consisting of k irreducible contexts. Each of the k contexts can be decomposed into at most k blocks so that the resulting block decomposition of m admits a block-wise reduction to ε. To prove the result, we observe that for each pair of contexts $m^{(i)}$ and $m^{(s)}$, there is at most one infix of each block such that the two infixes cancel each other out using Rule (G1).

This result gives us a quadratic bound on the number of blocks in a block decomposition, but it does not give us a bound on the length of each block. To deal with the latter, we represent blocks by finite automata that generate these blocks as elements of their language. As a consequence, we will not execute a block-wise decomposition on concrete blocks[1], but on automata representing such blocks. Imitating the swapping rule is easy. The automata use the operations $o^{\pm}$ as the letters of their alphabet. Two automata can be swapped if for every oper-

[1] Pun intended.

ation $o^{\pm}$ in the alphabet of the first automaton and every operation $u^{\pm}$ in the alphabet of the second one, $o\ I\ u$ holds.

The cancellation rule is a bit more complicated. Firstly, we note that if a block of an irreducible context $m^{(i,j)}$ has a right inverse, then this right inverse can be constructed in a *syntactic* way, namely by reversing the order of $m^{(i,j)}$ and by swapping the popularity of the operations. The latter means we replace o^+ by o^- and vice versa. Two automata cancel each other if the first one generates a word $m^{(i,j)}$ and the second one generates the syntactic right inverse of that wort. To check this property, we apply the syntactic inverse operations to the second automaton, i.e. we reverse its order and swap the polarity of all transition labels. Then, we check intersection emptiness.

The algorithm

We have gathered all prerequisites to outline the procedure that decides valence reachability under bounded context switching in NP. Assume we are given a valence system and a bound $k-1$ on the number of context switches, i.e. we can assume that we have k contexts.

1. For each of the k contexts, guess a dependent set of nodes. We will assume that each context will just use operations from that set. Construct for each context a version of the valence system in which the transitions whose label is o^+ or o^- for a node o not in the corresponding set are discarded.

2. Apply saturation to these valence systems.

3. For each $j = 1, \ldots, k^2 - 1$ (each index of a block in the decomposition), guess a control-state q_j of the valence systems. Intuitively, the operations in the j^{th} block of the decomposition leads from control state q_j to q_{j+1}, with $q_0 = q_{\text{init}}$ and $q_{k^2} = q_{\text{final}}$. Guess for each block j a subset of the nodes of the context to which block j belongs. Construct a version of the valence system in which the operations for other nodes are removed as in Step 1.

4. For each $j = 0, \ldots, k^2 - 1$: See the valence system for the j^{th} block as a finite automaton A_j with q_j as the initial and q_{j+1} as the final state.

5. Consider the sequence of the automata $A_0, \ldots, A_{k^2-1}$ resulting from the previous step. Guess a block-wise reduction and verify that it reduces the sequence of automata to ε.

The first step is self-explanatory. The second step ensures that if we can reach the final state with empty storage, then we can do so with a sequence of operations with irreducible contexts.

It might seem weird that we first guess a subset of nodes for each context, and later guess another subset for each block. This is needed because when verifying the usage of swap rules in the block-wise decomposition, we need to make sure that for two blocks that we want to swap, all operations used in the first block commute with all operations in the second block. Hence, we need to discard all operations that might appear in the context, but not in that specific block.

In Step 4., each automaton represents a language that contains a block $m^{(i,j)}$ of the block decomposition. When guessing the decomposition, we guess in each step which kind of rule should be applied (cancel or swap), and to which two blocks $m^{(i,j)}$ and $m^{(s,t)}$ it should be applied. We then verify that it is valid and that it indeed reduces the sequence to ε. It is not hard to see that a single step of the block-wise reduction can be guessed and verified in polynomial time. To conclude that the whole procedure is in NP, note that the number of cancellation steps is exactly $\frac{k^2}{2}$, and that between two cancellation steps, the number of swapping steps is at most the number of remaining blocks squared.

This completes our proof sketch for Theorem 19.3.3. We refer to the full version of the paper [MMZ18a] for details.

In additional to the full proof, the paper [MMZ18; MMZ18a] also analyzes the computational complexity of solving valence reachability in more detail. We briefly outline the results. Recall that for a graph G, G^- is a version of G without any self-loops. We have identified classes of graphs for which the problem is solvable in P, i.e. deterministic polynomial time. If G is the graph under consideration and G^- contains the graph C_4 from Figure 19.1.b.i) as an induced subgraph, then the problem is NP-complete. Recall that C_4 is the graph corresponding to a multi-pushdown system with two stacks with a binary stack alphabet each. Hence, this NP-completeness result corresponds to the result that reachability in multi-pushdown systems under bounded context switching is NP-complete. In the case that G^- contains the graph P_4 from Figure 19.1.b.ii) as an induced subgraph, we were not able to determine the precise complexity of valence reachability under the BCS restriction. We could neither prove that the problem is NP-complete, which would imply that our algorithm is optimal, nor provide a better algorithm that solves the problem in deterministic polynomial time.

Another open problem is a generalization of the aforementioned *bounded phase switching* restriction to valence systems. A phase of the computation of a multi-pushdown system is an infix of the computation in which the system pushes only onto a single stack. However, the system can pop from all stacks. Correspondingly, it seems sensible to define a phase of the computation of a valence system as a sequence of operations in which only the positive operations have to be dependent. For multi-pushdown systems, reachability under bounded phase switching is NP-complete like reachability under BCS [TMP07], and games with the bounded-phase-switching restriction can be solved in (deterministic) k-EXP time for a fixed k [Set09]. We would expect that these results can be generalized to valence systems under a bounded-phase-switching restriction. However, it is not clear how the above proof, in particular the saturation step and block decompositions, would generalize to bounded phase switching. We leave this for future work.

In the time since the publication of our paper [MMZ18], Shetty, Krishna, and Zetzsche [SKZ21] have generalized our result to reachability under *scope boundedness*. This notion was originally introduced for multi-pushdown systems by Torre, Napoli, and Parlato [TNP20]. For a fixed

bound k, a computation of a multi-pushdown is k-scope-bounded if after pushing each symbol onto some stack, there are at most k contexts that use this stack until the symbol is popped again. This restriction is a generalization of bounded context switching: A computation that contains at most k contexts is necessarily k-scope-bounded. In contrast to bounded context switching, a computation of a multi-pushdown can be k-scope-bounded while still containing infinitely many contexts. Scope boundedness seems to be incomparable in expressive power to the aforementioned bounded-phase-switching restriction.

The first contribution of the paper [SKZ21] is to generalize the notion of scope boundedness from multi-pushdown systems to arbitrary valence systems. Then, the authors prove that reachability under scope boundedness can always be solved in PSPACE and they provide a classification result, showing PSPACE-completeness in many cases. This matches the PSPACE-completeness for multi-pushdown systems [TNP20]. The complexity-theoretic results show that the increased expressiveness of scope boundedness over bounded context switching comes with a penalty as the computational complexity rises from NP to PSPACE. The techniques used in [SKZ21] to prove the results include generalization of some of the concepts that we used in our study of bounded context switching, including block decompositions and block-wise reductions.

From systems to games

In the rest of this section, we explain the difficulties that arise when trying to lift the proof of decidability in valence systems under the BCS restriction to games. In the proof, we have implicitly used several times that the nondeterminism in the system is under the control of a single entity: Our algorithm guesses the set of operations that will be used in each block, and a witness of reachability can only be found if the guesses are correct. This means the entity controlling the nondeterminism in the system can be assumed to be the same entity that controls the nondeterminism in the algorithm for checking reachability. Additionally, when we saturate a valence system, we insert transitions and hence introduce nondeterministic choices. The transitions that are inserted by the saturation procedure do not hurt the correctness, since we can assume that the entity resolving the nondeterminism has perfect foresight: When taking a transition labeled by a^+, it already knows that it will then take a sequence of ε-labeled transitions followed by a a^- transition, and nobody else can interfere with that plan. Hence, it can directly take the ε-labeled transition introduced by saturation that skips this part of the computation.

In a game setting, both tricks do not work anymore. A nondeterministic algorithm cannot simply guess a set of operations for each block, and restrict a part of the game to that set of operations. Intuitively, the nondeterminism in the algorithm corresponds to one of the players. If the other player wants to use an operation that is not in this set, we cannot simply deny her this opportunity without affecting the semantics of the game. Similarly, the saturation does not carry over if we consider a set of states that are not owned by a single player: After taking an a^+-labeled transition, the other player may become active, so it is not clear to the player that chose a^+ whether a a^- transition will follow later. Allowing the player to take a transition that

was inserted by the saturation means that she could skip moves of the opponent, which affects the semantics of the game.

Walukiewicz's reduction [Wal01], originally designed for context-free games (see Section 17.6), but also used by Seth [Set09] for multi-pushdown games with a BCS restriction, offers a way to resolve these issues. For example, instead of letting the algorithm guess sets of nodes, this mechanism could be incorporated into the game using a negotiation mechanism: The existential player proposes a prediction, then the other player can either accept it – which means she loses if she violates the prediction – or she can challenge it, in which case she wins if and only if she can enforce a violation of the prediction.

Even with this technique at hand, trying to obtain a decision procedure for valence reachability games has turned out to be very involved. Firstly, our result that bounds the length of a block decomposition requires all contexts of the computation to be irreducible. It is not clear whether the saturation procedure can be lifted to the game setting. If not, it might be possible to design a negotiation mechanism à la Walukiewicz to ensure irreducible contexts, but this seems to be quite intricate.

Assume we could successfully overcome these problems. In Step 4. of the algorithm, instead of a sequence of automata, we would now have to deal with a sequence of games. It is not clear how a block-wise reduction could be conducted on the level of games. We leave solving these problems for future work.

Part VI.

Conclusion

Contents

20 Contributions 475

21 Future work 479

20 Contributions

The purpose of this chapter is twofold. We will summarize the contributions to the field of automata theory that we have presented in the thesis. Additionally, we will also list for each of the results the publications that it stems from. While doing the latter, we will highlight the contributions of the author of this thesis to these publications. We proceed by considering each of the parts of the thesis separately.

Part II. Models of computation

This part consists of the preliminaries, so we forgo recapitulating its content. We have almost exclusively presented the work of others, for which we have given the appropriate references. There are two exceptions: The first is a novel and elegant way of defining ω-context-free languages in Section 5.2. This definition and the accompanying results are taken from the non-peer-reviewed publication [MMN17]. The second is a proof for the word problem of BPP languages being NP-complete, Proposition 6.4.8. The upper bound has been obtained by adapting our proof that BPP-SREDC, the problem of checking whether the language of a simple regular expression is contained in the downward closure of a BPP net language, is in NP. The latter result, presented as Theorem 9.4.2 in this thesis, is taken from the publication [AMMS17]. We will discuss both of these publications in more detail later.

Part III. Closures of Petri net languages

In this part of the thesis, we have studied the computation of the upward and downward closures of Petri net languages. Both of these closures are regular overapproximations that are useful in the context of verification. After providing some basic definition, we have turned to considering the upward closure of Petri net languages in Chapter 8. We have established that an automaton of doubly exponential size representing the upward closure can be computed in doubly exponential time. Restricting the problem leads to lower complexities: The upward closure of a BPP net language has exponential state complexity, and the problem of checking whether a simple regular expression is contained in the upward closure is EXPSPACE- and NP-complete for general Petri nets and BPP nets, respectively. We have obtained a similar collection of results for the downward closures in Chapter 9. For the downward closure of general Petri net languages, the literature [HMW10] already provides a non-primitive recursive upper bound. We have shown that the restricted versions share the complexities with their counterparts for the upward closure, e.g. the downward closure of a BPP net language has exponential state-complexity. For all the aforementioned upper bounds on the state complexity of the closures, we have provided matching lower bounds. We have concluded the part in Chapter 10 by showing that the regular containment problem for Petri net coverability languages is decidable, extending an earlier result for trace languages from the literature. From this, we have deduced that it is decidable whether the language of a Petri net equals its upward or downward closure.

The content of this part of the thesis is taken from the peer-reviewed conference publication [AMMS17] by Mohamed Faouzi Atig, Roland Meyer, Prakash Saivasan, and the author of this thesis. The full version [AMMS17a] including the proofs of all results is available on *arXiv*. The author has majorly contributed to the proofs of the upper bounds for computing the upward closure of both general Petri nets and BPP nets, the proof for the upper bound for computing the downward closure of a BPP net, and the proof of regular containment being decidable. This thesis extends the paper by also considering the case of Petri nets whose size is measured in terms of their unary encoding, providing new bounds wherever they are needed.

Part IV. Separability

In this part of the thesis, we have considered the regular separability of WSTS languages, a class that includes the Petri net coverability languages. A separator is a certificate for the disjointness of two languages – in general, disjointness is necessary, but not sufficient for separability, i.e. the existence of a separator. After giving an introduction in the first chapter of the part, we have established some results on the relations between various classes of WSTS languages, defined by restricting the WSTSes that generate them, in Chapter 12. In the big picture, these results allow us to extend our results on separability, e.g. from deterministic WSTS languages to the more general ω^2-WSTS languages and finitely branching WSTS languages. The main result, presented in Chapter 13, shows that under mild conditions, any two disjoint WSTS languages are regularly separable – disjointness is both necessary and sufficient to guarantee the existence of a regular separator. In order to prove the result, we have shown how to obtain a separator under the assumption that we start with a certain type of invariant in the state space of the WSTS. In a second step, we have used ideals to prove that any two language-disjoint WSTSes can be transformed to enforce the existence of such an invariant. We have completed our studies of regular separability by giving an explicit construction for obtaining a separator in the case of disjoint Petri net coverability languages. This has resulted in a triply exponential upper bound on the state complexity of the separator, and we have shown a doubly exponential lower bound.

The content of this part of the thesis is taken from the peer-reviewed conference publication [CLMMKS18] by Wojciech Czerwiński, Sławomir Lasota, Roland Meyer, K. Narayan Kumar, Prakash Saivasan, and the author of this thesis. The full version [CLMMKS18a] including the proofs of all results is available on *arXiv*. The author has majorly contributed to the proofs of various of the results in Chapter 12, to the proof that the ideal decomposition of an invariant is again an invariant, Proposition 13.3.8, as well as to the bounds presented in Chapter 14.

Part V. Games

In the last of the three main parts of the thesis, we have discussed solving games with perfect information. After giving some preliminary definitions, we have laid out the framework of effective denotational semantics that we use as a vehicle for solving games. Before turning to games, we have demonstrated its usage by applying it to the (ω-)regular inclusion problem for (ω-)context-free languages. Chapter 17 discusses an approach to context-free games that is

also based on the framework. We have shown how a context-free grammar defining a game can be turned into a system of equations that, after choosing a suitable domain, can be solved via fixed-point iteration. The least solution to the system allows us to read off the winner of the game. This leads to a procedure that solves context-free games within doubly exponential time, which is optimal. We have extensively compared our result to other works on context-free games from the literature, provided a prototype implementation, and extended the theory towards games with an ω-regular winning condition.

In Chapter 18, we have considered higher-order games, a generalization of context-free games. Our goal was to again apply effective denotational semantics by associating a system of equations to a higher-order recursion scheme. We have established a framework for the transfer of properties of the fixed-point solution with respect to one domain to the fixed-point solution with respect to another domain. We have instantiated this framework to prove the correctness of our approach. It provides a $(k + 1)$-fold exponential procedure to solve higher-order games of order k, which is optimal.

Finally, we have considered the boundaries of the decidability of games in Chapter 19. To this end, we employ the model of valence systems over graph monoids, which subsumes various popular automata models, including context-free systems and Petri nets. We obtain a complete classification of the decidability of reachability and coverability games on valence systems, based on the structure of the underlying graph monoid. We essentially show that context-free games and games that can be reduced to context-free games are the only decidable cases. For games on valence systems over graph monoids that are not representing context-free systems or minor extensions thereof, undecidability holds. To mitigate this undecidability, we propose employing a bounded-context-switching restriction. While we cannot prove that this restriction makes games on valence systems decidable, we demonstrate that a similar result holds in the less general case of valence reachability: Under a bound on the number of context switches, reachability in valence systems is always solvable in NP (and NP-complete in many cases).

Let us now discuss the publications that are related to this part of the thesis. The contents of Chapter 15 and Chapter 16 serve as preliminaries for the part. We have mostly presented the work of others and given the appropriate references. In particular, this includes our extensive discussion of the regular inclusion problem for context-free languages as an introductory example, which is taken from the paper [HM15] by Meyer and Holík. The content of Section 16.3 contains work by the author and is taken from the paper [MMN17]. We will come back to this publication in a moment.

The content of Chapter 17 is mostly taken from the peer-reviewed conference publication [HMM16] by Roland Meyer, Lukáš Holík, and the author of this thesis. The full version [HMM16a] including the proofs of all results is available on *arXiv*. Except for the algorithmic considerations that go beyond what is featured in our prototype implementation, presented in Section 17.7 in this thesis, the author has majorly contributed to all parts of the paper and the

implementation. The final two sections of the chapter, Sections 17.8 and 17.9, that are concerned with extending our results to the case of games with an ω-regular winning condition, are taken from the paper [MMN17] by Roland Meyer, Elisabeth Neumann, and the author of this thesis. We have mentioned this paper before when talking about the Sections 5.2 and 16.3. The paper has not been published in a peer-reviewed form, but it is available on *arXiv*.

The chapter on higher-order games, Chapter 18, is an extension of the peer-reviewed conference publication [HMM17] by Matthew Hague, Roland Meyer, and the author of this thesis. The full version [HMM17a] including the proofs of all results is available on *arXiv*. The author has majorly contributed to the parts of the publication that are concerned with modeling a higher-order game as fixed-point system, proving the correctness of this approach, as well as the complexity-theoretic considerations for the upper bound. The presentation of the material in this thesis extends the paper by giving a version of the framework for fixed-point transfer in Section 18.4 that is easier to apply as it requires fewer preconditions.

Our work on valence games in Chapter 19 stems from unpublished work by Roland Meyer, Georg Zetzsche, and the author of this thesis. The author has contributed to some proofs of the decidability resp. undecidability of various types of reachability and coverability games (presented in Section 19.2). The final section on reachability in valence systems under bounded context switching presents the contents of the peer-reviewed conference publication [MMZ18] by the three aforementioned authors. The full version [MMZ18a] including the proofs of all results is available on *arXiv*. The author of this thesis has majorly contributed to the proof that reachability under BCS is in NP.

21 Future work

We conclude by giving a brief overview of future directions for extending the research presented in this thesis.

Applications

In the main part of the thesis, we have almost exclusively considered the theoretical aspects of the problems under consideration. Embedding our proposed solutions for these problems into frameworks and tools that solve verification problems in practice is a challenge that we have not tackled. This process would consist of at least three steps. The first is designing a tool that translates an actual instance of a verification problem, e.g. the source code of a program (in some fixed programming language) and a specification (in some fixed specification language) into an instance of an automaton model. We have briefly explained the challenges associated to this step at the beginning of Chapter 1. Secondly, one needs to implement the algorithms proposed in this thesis. We have presented a prototype implementation only for solving context-free games in Section 17.7. Extending the implementation to also deal with ω-games or even incorporating it into a general framework for effective denotation semantics (as explained at the beginning of Chapter 16) would be desirable. Finally, one would need to optimize the implementations so that they are well-behaved on the instances that are of practical interest. It is well-known that a naive implementation of an algorithm often does not perform well in practice, even if its worst-case running time matches a complexity-theoretic lower bound. The instances that are relevant in practice often exhibit a special structure that can be exploited if the algorithm is designed and optimized accordingly, e.g. by using suitable data structures and heuristics. For solving context-free games, we have reported on some early work in this direction in Section 17.7.

In addition to these considerations on applying our results, there are a few open questions on the theoretical side. We have discussed most of them extensively in the main parts of the thesis. Therefore, it should be sufficient to conclude the thesis by giving a brief summary.

Part III. Closures of Petri net languages

All the upper and lower bounds that we have presented for the sizes of closures of Petri net coverability languages match. The same holds true for the variations of these problems, e.g. the case of BPP nets and the containment of SREs in the closures. We have not investigated the complexity of regular containment and the question of whether a given Petri net coverability language is upward or downward closed in Chapter 10, neither for general Petri nets nor for BPP nets. Finding an algorithm for regular containment whose termination does not rely on properties of WQOs and hence allows an estimation of the running time would be desirable.

Additionally, we want to highlight that the class of languages of BPP nets is not widely studied yet. We have established a few results in this thesis, e.g. the word problem being NP-complete, but there are algorithmic problems for which, to the best of the author's knowledge, decidability and/or computational complexity are open questions. For example, the proof that inclusion and equality of Petri net coverability languages are undecidable [Jan95; Wim08] does not carry over to BPP net languages. It is not clear whether these properties are decidable and if so, what the computational complexity is.

Part IV. Separability

We have encountered a handful of open problems related to the separability of WSTS languages. Arguably, the most interesting one is the question on whether the inclusions among language classes in Theorem 12.2.1 are strict. This boils down to the question of whether there is an infinitely branching non-ω^2 WSTS whose language is not the language of a finitely branching WSTS. As argued at the end of Section 12.2, this problem is of little practical interest, but solving it would provide insights on the expressive power of non-ω^2 WSTSes and whether finite words are sufficient to distinguish them from ω^2-WSTSes. If it turns out that the aforementioned classes are equal, we also obtain a generalization of our results in Section 13.1. For example, if the classes are equal, we immediately obtain that any two disjoint WSTS languages are regularly separable. If the two classes were proven to be different, separability would remain an open question. Hopefully, the proof for classes not being equal would provide additional insight.

In Chapter 14 we have presented non-matching upper and lower bounds on the size of the separator in the case of Petri net coverability languages. We have already conjectured that the upper bound can be improved by a construction for determinization that is more clever.

Finally, it should be mentioned that the regular separability of Petri net reachability languages is at the time of this writing the most important open problem in the area of separability. Note that Petri net reachability languages are not WSTS languages (at least not with reaching an upward-closed set of configurations as the acceptance condition). In contrast to WSTS languages, there are pairs of Petri net reachability languages that are disjoint but not regularly separable. Hence, the goal here is to understand whether regular separability is a decidable property.

Part V. Games

We have extensively studied context-free games producing finite words and compared our approach to other works. We have discussed that our approach avoids an upfront determinization of the problem input which would be needed in order to apply various algorithms from the literature (e.g. the ones by Cachat [Cac02] and Walukiewicz [Wal01]). When it comes to games that produce infinite words, however, we have employed a determinization in Section 17.9. It is unclear whether this step in our construction can be avoided. We refer to the end of the aforementioned section for a discussion of the topic and related work that hints at the construction being potentially unavoidable. In the case of higher-order games, we have not yet attempted at all to generalize our approach to the case of infinite words.

Finally, when considering valence games in the last chapter of the part, we have conjectured in Section 19.3 that all reachability games become decidable once we restrict them by bounding the number of context switches. We have justified this conjecture by proving a weaker result for valence reachability under bounded context switching and discussed the problems that prevent us from adapting that proof to the case of games in a straightforward manner. We hope that these problems can be overcome in the future in order to obtain a proof for the conjecture.

Bibliography

[ABDFG+15] R. Alur et al. *Syntax-guided synthesis*. In: Dependable Software Systems Engineering. Volume 40. 2015, pages 1–25.

[ABKS12] M. F. Atig, A. Bouajjani, K. N. Kumar, and P. Saivasan. *Model checking branching-time properties of multi-pushdown systems is hard*. CoRR abs/1205.6928 (2012). arXiv: 1205.6928. URL: https://arxiv.org/abs/1205.6928.

[ABKS17] M. F. Atig, A. Bouajjani, K. N. Kumar, and P. Saivasan. *Parity games on bounded phase multi-pushdown systems*. In: NETYS. Volume 10299. Lecture Notes in Computer Science. 2017, pages 272–287.

[ACBJ04] P. A. Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. *Using forward reachability analysis for verification of lossy channel systems*. FMSD 25.1 (2004).

[ACCHH+10] P. A. Abdulla, Y. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. *Simulation subsumption in Ramsey-based Büchi automata universality and inclusion testing*. In: CAV. Volume 6174. LNCS. 2010, pages 132–147.

[ACCHH+11] P. A. Abdulla, Y. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. *Advanced Ramsey-based Büchi automata inclusion testing*. In: CONCUR. Volume 6901. LNCS. 2011, pages 187–202.

[ACHKSZ16] M. F. Atig, D. Chistikov, P. Hofman, K. N. Kumar, P. Saivasan, and G. Zetzsche. *Complexity of regular abstractions of one-counter languages*. In: LICS. 2016, pages 207–216.

[ACHMV10] P. A. Abdulla, Y. Chen, L. Holík, R. Mayr, and T. Vojnar. *When simulation meets antichains*. In: TACAS. Volume 6015. LNCS. 2010, pages 158–174.

[ACJT96] P. A. Abdulla, K. Cerans, B. Jonsson, and Y. Tsay. *General decidability theorems for infinite-state systems*. In: LICS. 1996, pages 313–321.

[Ack28] W. Ackermann. *Zum Hilbertschen Aufbau der reellen Zahlen*. Mathematische Annalen 99 (1928), pages 118–133.

[Aeh07] K. Aehlig. *A finite semantics of simply-typed lambda terms for infinite runs of automata*. Log. Methods Comput. Sci. 3.3 (2007).

[AJ93] P. A. Abdulla and B. Jonsson. *Verifying programs with unreliable channels*. In: LICS. 1993, pages 160–170.

[Ale01] A. Alexandrescu. *Modern C++ design: Generic programming and design patterns applied*. Addison-Wesley Longman, 2001.

[AM04] R. Alur and P. Madhusudan. *Visibly pushdown languages*. In: STOC. 2004, pages 202–211.

[AMMS17] M. F. Atig, R. Meyer, S. Muskalla, and P. Saivasan. *On the upward/downward closures of Petri nets*. In: MFCS. Volume 83. LIPIcs. 2017, pages 49:1–49:14.

[AMMS17a] M. F. Atig, R. Meyer, S. Muskalla, and P. Saivasan. *On the upward/downward closures of Petri nets*. CoRR abs/1701.02927 (2017). arXiv: 1701.02927. URL: https://arxiv.org/abs/1701.02927.

[Ang87] D. Angluin. *Learning regular sets from queries and counterexamples*. Inf. Comput. 75.2 (1987), pages 87–106.

[BCHS12] C. Broadbent, A. Carayol, M. Hague, and O. Serre. *A saturation method for collapsible pushdown systems*. In: ICALP. Volume 7392. LNCS. 2012, pages 165–176.

[Bei90] B. Beizer. *Software testing techniques (2. ed.)* Van Nostrand Reinhold, 1990.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. *Reachability analysis of pushdown automata: Application to model-checking*. In: CONCUR. Volume 1243. LNCS. 1997, pages 135–150.

[Ber79] J. Berstel. *Transductions and context-free languages*. Teubner, 1979.

[BFGHM15] M. Blondin, A. Finkel, S. Göller, C. Haase, and P. McKenzie. *Reachability in two-dimensional vector addition systems with states is PSPACE-complete*. In: LICS. 2015, pages 32–43.

[BFM17] M. Blondin, A. Finkel, and P. McKenzie. *Well behaved transition systems*. LMCS 13.3 (2017).

[BG11] L. Bozzelli and P. Ganty. *Complexity analysis of the backward coverability algorithm for VASS*. In: RP. 2011, pages 96–109.

[Bir92] J. Birget. *Intersection and union of regular languages and state complexity*. Inf. Process. Lett. 43.4 (1992), pages 185–190.

[Bir93] J. Birget. *Partial orders on words, minimal elements of regular languages and state complexity*. Theor. Comput. Sci. 119.2 (1993), pages 267–291.

[BLS15] G. Bachmeier, M. Luttenberger, and M. Schlund. *Finite automata for the sub- and superword closure of CFLs: Descriptional and computational complexity*. In: LATA. LNCS. 2015.

[BM04] A. Bouajjani and A. Meyer. *Symbolic reachability analysis of higher-order context-free processes*. In: FSTTCS. Volume 3328. LNCS. 2004, pages 135–147.

[Bör97] E. Börger. *Ten years of Gurevich's abstract state machines*. J. Univers. Comput. Sci. 3.4 (1997), pages 230–232.

[BPS61] Y. Bar-Hillel, M. Perles, and E. Shamir. *On formal properties of simple phrase structure grammars*. Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung 14 (1961), pages 143–172.

[BT76] I. Borosh and L. B. Treybig. *Bounds on positive integral solutions of linear diophantine equations*. Proc. AMS 55.2 (1976), pages 299–304.

[Büc62] J. R. Büchi. *On a decision method in restricted second-order arithmetic*. In: CLMPST. 1962, pages 1–11.

[BV93] E. Bernstein and U. V. Vazirani. *Quantum complexity theory*. In: STOC. 1993, pages 11–20.

[BW18] J. C. Bradfield and I. Walukiewicz. *The mu-calculus and model checking*. In: Handbook of Model Checking. 2018, pages 871–919.

[BZ13] P. Buckheister and G. Zetzsche. *Semilinearity and context-freeness of languages accepted by valence automata*. In: MFCS. 2013, pages 231–242.

[Cac02] T. Cachat. *Symbolic strategy synthesis for games on pushdown graphs*. In: ICALP. Volume 2380. LNCS. 2002, pages 704–715.

[Cac03] T. Cachat. *Higher order pushdown automata, the Caucal hierarchy of graphs and parity games*. In: ICALP. Volume 2719. LNCS. 2003, pages 556–569.

[Cau02] D. Caucal. *On infinite terms having a decidable monadic theory*. In: MFCS. Volume 2420. LNCS. 2002, pages 165–176.

[CC77] P. Cousot and R. Cousot. *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: POPL. 1977, pages 238–252.

[CCLP17a] L. Clemente, W. Czerwiński, S. Lasota, and C. Paperman. *Regular separability of Parikh automata*. In: ICALP. 2017, pages 117:1–117:13.

[CCLP17b] L. Clemente, W. Czerwiński, S. Lasota, and C. Paperman. *Separability of reachability sets of vector addition systems*. In: STACS 2017. 2017, pages 24:1–24:14.

[CFM11] M. Cadilhac, A. Finkel, and P. McKenzie. *On the expressiveness of Parikh automata and related models*. In: NCMA. 2011, pages 103–119.

[CG77] R. S. Cohen and A. Y. Gold. *Theory of ω-languages (I and II)*. JCSS 15.2 (1977), pages 169–208.

[CGJLV00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. *Counterexample-guided abstraction refinement*. In: CAV. 2000, pages 154–169.

[CGP03] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. *Learning assumptions for compositional verification*. In: TACAS. Volume 2619. LNCS. 2003, pages 331–346.

[Cha07] R. Charney. *An introduction to right-angled Artin groups*. Geometriae Dedicata 125 (2007), pages 141–158.

[Chr93] S. Christensen. *Decidability and decomposition in process algebras*. PhD thesis. Edinburgh University, 1993.

[Chu36] A. Church. *An unsolvable problem of elementary number theory*. American Journal of Mathematics 58.2 (1936), pages 345–363.

[Chu63] A. Church. *Application of recursive arithmetic to the problem of circuit synthesis*. Journal of Symbolic Logic 28.4 (1963), pages 289–290.

[CJKLS17] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. *Deciding parity games in quasipolynomial time*. In: STOC. 2017, pages 252–263.

[CKS81] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. *Alternation*. J. ACM 28.1 (1981), pages 114–133.

[CL17] W. Czerwiński and S. Lasota. *Regular separability of one counter automata*. In: LICS. 2017, pages 1–12.

[CLLLM19] W. Czerwiński, S. Lasota, R. Lazic, J. Leroux, and F. Mazowiecki. *The reachability problem for Petri nets is not elementary*. In: STOC. 2019, pages 24–33.

[CLMMKS18] W. Czerwiński, S. Lasota, R. Meyer, S. Muskalla, K. N. Kumar, and P. Saivasan. *Regular separability of well-structured transition systems*. In: CONCUR. Volume 118. LIPIcs. 2018, pages 35:1–35:18.

[CLMMKS18a] W. Czerwiński, S. Lasota, R. Meyer, S. Muskalla, K. N. Kumar, and P. Saivasan. *Regular separability of well structured transition systems*. CoRR abs/1702.05334 (2018). arXiv: 1702.05334. URL: https://arxiv.org/abs/1702.05334.

[CMM13] W. Czerwiński, W. Martens, and T. Masopust. *Efficient separability of regular languages by subsequences and suffixes*. In: ICALP. 2013, pages 150–161.

[CN78] B. Courcelle and M. Nivat. *The algebraic semantics of recursive program schemes*. In: MFCS. Volume 64. LNCS. 1978, pages 16–30.

[CO21] W. Czerwiński and L. Orlikowski. *Reachability in vector addition systems is Ackermann-complete*. In: FOCS. 2021, pages 1229–1240.

[Coo00] S. Cook. *The P versus NP problem*. (Accessed 2022-05-19). 2000. URL: https://www.claymath.org/sites/default/files/pvsnp.pdf.

[Coo71] S. A. Cook. *The complexity of theorem-proving procedures*. In: STOC. 1971, pages 151–158.

[Cou91] B. Courcelle. *On constructing obstruction sets of words*. Bulletin of the EATCS (1991).

[CPSW16] L. Clemente, P. Parys, S. Salvati, and I. Walukiewicz. *The diagonal problem for higher-order recursion schemes is decidable*. In: LICS. 2016, pages 96–105.

[CR72] S. A. Cook and R. A. Reckhow. *Time-bounded random access machines*. In: STOC. 1972, pages 73–80.

[CS99] S. Cook and M. Soltys. *Boolean programs and quantified propositional proof systems*. Bulletin of the Section of Logic (1999).

[CSS19] C. Coester, T. Schwentick, and M. Schuster. *Winning strategies for streaming rewriting games*. In: FCT. Volume 11651. LNCS. 2019, pages 49–63.

[CW07] T. Cachat and I. Walukiewicz. *The complexity of games on higher order pushdown automata*. CoRR abs/0705.0262 (2007). arXiv: 0705.0262. URL: https://arxiv.org/abs/0705.0262.

[Dam82] W. Damm. *The IO- and OI-hierarchies*. Theor. Comp. Sci. 20 (1982), pages 95–207.

[DDHR06] M. De Wulf, L. Doyen, T. A. Henzinger, and J. Raskin. *Antichains: A new algorithm for checking universality of finite automata*. In: CAV. Volume 4144. LNCS. 2006, pages 17–30.

[Dem13] S. Demri. *On selective unboundedness of VASS*. JCSS 79.5 (2013).

[DFS98] C. Dufourd, A. Finkel, and P. Schnoebelen. *Reset nets between decidability and undecidability*. In: ICALP. Volume 1443. LNCS. 1998, pages 103–115.

[DG86] W. Damm and A. Goerdt. *An automata-theoretical characterization of the OI-hierarchy*. Information and Control 71.1/2 (1986), pages 1–32.

[DHW91] D. L. Dill, A. J. Hu, and H. Wong-Toi. *Checking for language inclusion using simulation preorders*. In: CAV. Volume 575. LNCS. 1991, pages 255–265.

[Dic13] L. E. Dickson. *Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors*. American Journal of Mathematics 35.4 (1913), pages 413–422.

[Dij18] T. van Dijk. *Oink: An implementation and evaluation of modern parity game solvers*. In: TACAS. Volume 10805. LNCS. 2018, pages 291–308.

[DJW97] S. Dziembowski, M. Jurdzinski, and I. Walukiewicz. *How much memory is needed to win infinite games?* In: LICS. 1997, pages 99–110.

[DMZ16] E. D'Osualdo, R. Meyer, and G. Zetzsche. *First-order logic with reachability for infinite-state systems*. In: LICS. 2016, pages 457–466.

[EH08] J. Esparza and K. Heljanko. *Unfoldings – A partial-order approach to model checking*. Monographs in Theoretical Computer Science. Springer, 2008.

[EJ91] E. A. Emerson and C. S. Jutla. *Tree automata, mu-calculus and determinacy (Extended abstract)*. In: FOCS. 1991, pages 368–377.

[EJS01] E. A. Emerson, C. S. Jutla, and A. P. Sistla. *On model checking for the μ-calculus and its fragments*. Theor. Comput. Sci. 258.1-2 (2001), pages 491–522.

[End72] H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[Eng91] J. Engelfriet. *Iterated stack automata and complexity classes*. Information and Computation 95.1 (1991), pages 21–75.

[Esp97] J. Esparza. *Petri Nets, commutative context-free grammars, and basic parallel processes*. Fundam. Inf. 31.1 (1997).

[Esp98] J. Esparza. *Decidability and complexity of Petri net problems – An introduction*. In: Lectures on Petri Nets I: Basic Models. Volume 1491. LNCS. 1998, pages 374–428.

[FG09] A. Finkel and J. Goubault-Larrecq. *Forward analysis for WSTS, part I: Completions*. In: STACS. 2009, pages 433–444.

[FG12] A. Finkel and J. Goubault-Larrecq. *Forward analysis for WSTS, part II: Complete WSTS*. Logical Methods in Computer Science 8.3 (2012).

[Fin16] A. Finkel. *The ideal theory for WSTS*. In: RP. 2016, pages 1–22.

[Fin87] A. Finkel. *A generalization of the procedure of Karp and Miller to well structured transition systems*. In: ICALP. 1987, pages 499–508.

[Fin90] A. Finkel. *Reduction and covering of infinite reachability trees*. Inf. Comput. 89.2 (1990), pages 144–179.

[For13] L. Fortnow. *The golden ticket: P, NP and the search for the impossible*. Princeton University Press, 2013.

[FS01] A. Finkel and P. Schnoebelen. *Well-structured transition systems everywhere!* Theoretical Computer Science 256.1–2 (2001), pages 63–92.

[FV10] S. Fogarty and M. Y. Vardi. *Efficient Büchi universality checking*. In: TACAS. Volume 6015. LNCS. 2010, pages 205–220.

[GH15] H. Gruber and M. Holzer. *From finite automata to regular expressions and back — A summary on descriptional complexity*. International Journal of Foundations of Computer Science 26.8 (2015), pages 1009–1040.

[GH82] Y. Gurevich and L. Harrington. *Trees, automata, and games*. In: STOC. 1982, pages 60–65.

[GHK07] H. Gruber, M. Holzer, and M. Kutrib. *More on the size of Higman-Haines sets: Effective constructions*. In: MCU. LNCS. 2007.

[GNP18] D. Giannakopoulou, K. S. Namjoshi, and C. S. Păsăreanu. *Compositional reasoning*. In: Handbook of Model Checking. 2018, pages 345–383.

[Göd31] K. Gödel. *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme*. Monatshefte für Mathematik und Physik 38.1 (1931), pages 173–198.

[GRB06] G. Geeraerts, J.-F. Raskin, and L. V. Begin. *Expand, enlarge and check: New algorithms for the coverability problem of WSTS*. Journal of Computer and System Sciences 72.1 (2006), pages 180–203.

[GRV07] G. Geeraerts, J.-F. Raskin, and L. Van Begin. *Well-structured languages*. Acta Informatica 44.3-4 (2007), pages 249–288.

[GS97] S. Graf and H. Saïdi. *Construction of abstract state graphs with PVS*. In: CAV. Volume 1254. LNCS. 1997, pages 72–83.

[GTW02] E. Grädel, W. Thomas, and T. Wilke (Editors). *Automata, logics, and infinite games: A guide to current research*. Volume 2500. LNCS. Springer, 2002.

[Gur00] Y. Gurevich. *Sequential abstract-state machines capture sequential algorithms*. ACM Trans. Comput. Log. 1.1 (2000), pages 77–111.

[Haa14] C. Haase. *Subclasses of Presburger arithmetic and the weak EXP hierarchy*. In: CSL-LICS. 2014, pages 47:1–47:10.

[Had12] A. Haddad. *IO vs OI in higher-order recursion schemes*. In: FICS. Volume 77. EPTCS. 2012, pages 23–30.

[Hai17] F. Haifani. *Antichain optimizations using simulation relations for context-free games*. (Accessed 2022-05-23). Master's thesis. TU Kaiserslautern, 2017. URL: https://www.tcs.cs.tu-bs.de/documents/fajar_haifani_masters_thesis.pdf.

[Hai69] L. H. Haines. *On free monoids partially ordered by embedding*. Journal of Combinatorial Theory 6.1 (1969).

[HCDGN+17] M. Heizmann et al. *Ultimate Automizer with an on-demand construction of Floyd-Hoare automata (Competition contribution)*. In: TACAS. Volume 10206. LNCS. 2017, pages 394–398.

[HHP10] M. Heizmann, J. Hoenicke, and A. Podelski. *Nested interpolants*. In: POPL. 2010, pages 471–482.

[Hig52] G. Higman. *Ordering by divisibility in abstract algebras*. Proc. London Math. Soc. 3 2.7 (1952).

[HIJSB18] M. Heule, M. Iser, M. Jarvisalo, M. Suda, and T. Balyo. *SAT competition 2022 – Certified UNSAT*. (Accessed 2022-05-19). 2018. URL: https://satcompetition.github.io/2022/certificates.html.

[HK71] J. E. Hopcroft and R. M. Karp. *A linear algorithm for testing equivalence of finite automata*. Technical report. Cornell University, 1971.

[HKO16] M. Hague, J. Kochems, and C.-H. L. Ong. *Unboundedness and downward closures of higher-order pushdown automata*. In: POPL. 2016.

[HM15] L. Holík and R. Meyer. *Antichains for the verification of recursive programs*. In: NETYS. Volume 9466. LNCS. 2015, pages 322–336.

[HMM16] L. Holík, R. Meyer, and S. Muskalla. *Summaries for context-free games*. In: FSTTCS. Volume 65. LIPIcs. 2016, pages 41:1–41:16.

[HMM16a] L. Holík, R. Meyer, and S. Muskalla. *Summaries for context-free games*. CoRR abs/1603.07256 (2016). arXiv: 1603.07256. URL: https://arxiv.org/abs/1603.07256.

[HMM17] M. Hague, R. Meyer, and S. Muskalla. *Domains for higher-order games*. In: MFCS. Volume 83. LIPIcs. 2017, pages 59:1–59:15.

[HMM17a] M. Hague, R. Meyer, and S. Muskalla. *Domains for higher-order games*. CoRR abs/1705.00355 (2017). arXiv: 1705.00355. URL: https://arxiv.org/abs/1705.00355.

[HMOS08] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. *Collapsible pushdown automata and recursion schemes*. In: LICS. 2008, pages 452–461.

[HMOS17] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. *Collapsible pushdown automata and recursion schemes*. ACM Trans. Comput. Log. 18.3 (2017), pages 25:1–25:42.

[HMW10] P. Habermehl, R. Meyer, and H. Wimmel. *The downward-closure of Petri net languages*. In: ICALP. LNCS. 2010.

[HO07] M. Hague and C.-H. L. Ong. *Symbolic backwards-reachability analysis for higher-order pushdown systems*. In: FoSSaCS. Volume 4423. LNCS. 2007, pages 213–227.

[HO09] M. Hague and C.-H. L. Ong. *Winning regions of pushdown parity games: A saturation method*. In: CONCUR. Volume 5710. LNCS. 2009, pages 384–398.

[Hoa69] C. A. R. Hoare. *An axiomatic basis for computer programming*. Commun. ACM 12.10 (Oct. 1969), pages 576–580.

[HP06] T. A. Henzinger and N. Piterman. *Solving games without determinization*. In: CSL. Volume 4207. LNCS. 2006, pages 395–410.

[HP79] J. Hopcroft and J.-J. Pansiot. *On the reachability problem for 5-dimensional vector addition systems*. Theoretical Computer Science 8.2 (1979), pages 135–159.

[HU79] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[Imm88] N. Immerman. *Nondeterministic space is closed under complementation*. SIAM Journal on Computing 17 (1988), pages 935–938.

[Imm98] N. Immerman. *Descriptive complexity*. Texts in Computer Science. Springer, 1998.

[ISO90] ISO. *ISO/IEC 9899:1990 Programming languages – C*. 1990.

[Jan87] M. Jantzen. *Language theory of Petri nets*. In: Petri Nets: Central Models and Their Properties. 1987, pages 397–412.

[Jan95] P. Jančar. *Undecidability of bisimilarity for Petri nets and some related problems*. TCS 148.2 (1995), pages 281–301.

[Jan99] P. Jancar. *A note on well quasi-orderings for powersets*. Inf. Process. Lett. 72.5-6 (1999), pages 155–160.

[JEM99] P. Jančar, J. Esparza, and F. Moller. *Petri nets and regular processes*. J. Comput. Syst. Sci. 59.3 (1999), pages 476–503.

[Jon83] C. B. Jones. *Tentative steps toward a development method for interfering programs*. ACM Trans. Program. Lang. Syst. 5.4 (1983), pages 596–619.

[JP21] S. Jacobs and G. A. Pérez (Organizers). *The reactive synthesis competition*. (Accessed on 2022-05-30). 2021. URL: http://www.syntcomp.org.

[JR93] T. Jiang and B. Ravikumar. *Minimal NFA problems are hard*. SIAM Journal on Computing 22.6 (1993), pages 1117–1141.

[Jur98] M. Jurdzinski. *Deciding the winner in parity games is in UP ∩ co-UP*. Inf. Process. Lett. 68.3 (1998), pages 119–124.

[Kar72] R. M. Karp. *Reducibility among combinatorial problems*. In: COCO. 1972, pages 85–103.

[Kle43] S. C. Kleene. *Recursive predicates and quantifiers*. Trans. Amer. Math. Soc. 53 (1943), pages 41–73.

[Kle52] S. C. Kleene. *Introduction to metamathematics*. Van Nostrand, 1952.

[KM69] R. M. Karp and R. E. Miller. *Parallel program schemata*. JCSS 3.2 (1969), pages 147–195.

[KML07] P. Kaye, M. Mosca, and R. Laflamme. *An introduction to quantum computing*. Oxford University Press, 2007.

[KN01] B. Khoussainov and A. Nerode. *Automata theory and its applications*. Birkhäuser, 2001.

[Kna28] B. Knaster. *Un théorème sur les fonctions d'ensembles*. Ann. Soc. Polon. Math. 6 (1928), pages 133–134.

[KNU02] T. Knapik, D. Niwinski, and P. Urzyczyn. *Higher-order pushdown trees are easy*. In: FOSSACS. Volume 2303. LNCS. 2002, pages 205–222.

[KNUW05] T. Knapik, D. Niwiński, P. Urzyczyn, and I. Walukiewicz. *Unsafe grammars and panic automata*. In: ICALP. Volume 3580. LNCS. 2005, pages 1450–1461.

[KO09] N. Kobayashi and C.-H. L. Ong. *A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes*. In: LICS. 2009, pages 179–188.

[Kob09] N. Kobayashi. *Types and higher-order recursion schemes for verification of higher-order programs*. In: POPL. 2009, pages 416–428.

[Kőn27] D. Kőnig. *Über eine Schlussweise aus dem Endlichen ins Unendliche*. Acta Sci. Math. (Szeged) 3 (1927), pages 121–130.

[Kop16] E. Kopczynski. *Invisible pushdown languages*. In: LICS. 2016, pages 867–872.

[Kos82] S. R. Kosaraju. *Decidability of reachability in vector addition systems (Preliminary version)*. In: STOC. 1982, pages 267–281.

[Koz06] D. C. Kozen. *Theory of computation*. Texts in Computer Science. Springer, 2006.

[Koz97] D. C. Kozen. *Automata and computability*. Springer, 1997.

[KP92] M. Kabil and M. Pouzet. *Une extension d'un théorème de P. Jullien sur les âges de mots*. ITA 26 (1992), pages 449–484.

[KS15] D. Kuperberg and M. Skrzypczak. *On determinisation of good-for-games automata*. In: ICALP. Volume 9135. LNCS. 2015, pages 299–310.

[KV00] O. Kupferman and M. Y. Vardi. *An automata-theoretic approach to reasoning about infinite-state systems*. In: CAV. Volume 1855. 2000, pages 36–52.

[Lam77] L. Lamport. *Proving the correctness of multiprocess programs*. IEEE Trans. Software Eng. 3.2 (1977), pages 125–143.

[Lam92] J. Lambert. *A structure to decide reachability in Petri nets*. Theoretical Computer Science 99.1 (1992), pages 79–104.

[Laz13] R. Lazic. *The reachability problem for vector addition systems with a stack is not elementary*. CoRR abs/1310.1767 (2013). arXiv: 1310.1767.

[LCMM12] Z. Long, G. Calin, R. Majumdar, and R. Meyer. *Language-theoretic abstraction refinement*. In: FASE. Volume 7212. LNCS. 2012, pages 362–376.

[Lee78] J. van Leeuwen. *Effective constructions in well-partially-ordered free monoids*. Discrete Mathematics 21.3 (1978).

[Ler11] J. Leroux. *Vector addition system reachability problem: A short self-contained proof*. In: POPL. 2011, pages 307–316.

[Ler21] J. Leroux. *The reachability problem for petri nets is not primitive recursive*. In: FOCS. 2021, pages 1241–1252.

[Lev73] L. A. Levin. *Универсальные задачи перебора*. Problems of Information Transmission 9.3 (1973), pages 115–116.

[LFLMLL18] C.-M. Li, X. Fan, M. Luo, F. Manya, Z. Lu, and Y. Li. *Polynomial multiplication*. In: SAT Competition 2018: Solver and Benchmark Descriptions. 2018, pages 61–62. URL: https://helda.helsinki.fi/handle/10138/237063.

[Lin76] M. Linna. *On ω-sets associated with context-free languages*. Inf. Cont. 31.3 (1976), pages 272–293.

[Lip76] R. J. Lipton. *The reachability problem requires exponential space*. Technical report. Yale University, Department of Computer Science, 1976.

[LL78] P. Lorenzen and K. Lorenz. *Dialogische Logik*. Darmstadt - Wissenschaftliche Buchgesellschaft, 1978.

[LNS82] J.-L. Lassez, V. Nguyen, and E. Sonenberg. *Fixed point theorems and semantics: A folk tale*. Information Processing Letters 14.3 (1982), pages 112–116.

[LPS13] J. Leroux, M. Praveen, and G. Sutre. *A relational trace logic for vector addition systems with application to context-freeness*. In: CONCUR. 2013, pages 137–151.

[LPSZ08] S. Lu, S. Park, E. Seo, and Y. Zhou. *Learning from mistakes: A comprehensive study on real world concurrency bug characteristics*. In: ASPLOS. 2008, pages 329–339.

[LR94] C. LaCasse and D. Ross. *The microeconomic interpretation of games*. PSA 1994.1 (1994), pages 379–387.

[LS15a] R. Lazic and S. Schmitz. *The ideal view on Rackoff's coverability technique*. In: RP. 2015, pages 76–88.

[LS15b] J. Leroux and S. Schmitz. *Demystifying reachability in vector addition systems*. In: LICS. 2015, pages 56–67.

[LS19] J. Leroux and S. Schmitz. *Reachability in vector addition systems is primitive-recursive in fixed dimension*. In: LICS. 2019, pages 1–13.

[Mar75] D. A. Martin. *Borel determinacy*. Annals of Mathematics 102.2 (1975), pages 363–371.

[Mar94] A. Marcone. *Foundations of BQO theory*. Transactions of the American Mathematical Society 345.2 (1994), pages 641–660.

[May03] R. Mayr. *Undecidable problems in unreliable computations*. TCS 1-3.297 (2003).

[May81] E. W. Mayr. *An algorithm for the general Petri net reachability problem*. In: STOC. 1981, pages 238–246.

[MC81] J. Misra and K. M. Chandy. *Proofs of networks of processes*. IEEE Trans. Software Eng. 7.4 (1981), pages 417–426.

[McN93] R. McNaughton. *Infinite games played on finite graphs*. Ann. Pure Appl. Logic 65.2 (1993), pages 149–184.

[Min67] M. L. Minsky. *Computation: Finite and infinite machines*. Prentice-Hall, 1967.

[MKRS98a] M. Mukund, K. N. Kumar, J. Radhakrishnan, and M. A. Sohoni. *Robust asynchronous protocols are finite-state*. In: ICALP. 1998, pages 188–199.

[MKRS98b] M. Mukund, K. N. Kumar, J. Radhakrishnan, and M. A. Sohoni. *Towards a characterisation of finite-state message-passing systems*. In: ASIAN. 1998, pages 282–299.

[MM81] E. W. Mayr and A. R. Meyer. *The complexity of the finite containment problem for Petri nets*. JACM 28.3 (1981).

[MMN17] R. Meyer, S. Muskalla, and E. Neumann. *Liveness verification and synthesis: New algorithms for recursive programs*. CoRR abs/1701.02947 (2017). arXiv: 1701.02947. URL: https://arxiv.org/abs/1701.02947.

[MMZ18] R. Meyer, S. Muskalla, and G. Zetzsche. *Bounded context switching for valence systems*. In: CONCUR. Volume 118. LIPIcs. 2018, pages 12:1–12:18.

[MMZ18a] R. Meyer, S. Muskalla, and G. Zetzsche. *Bounded context switching for valence systems*. CoRR abs/1803.09703 (2018). arXiv: 1803.09703. URL: https://arxiv.org/abs/1803.09703.

[Mos84] A. W. Mostowski. *Regular expressions for infinite trees and a standard form of automata*. In: Computation Theory. Volume 208. LNCS. 1984, pages 157–168.

[Mos91] A. W. Mostowski. *Games with forbidden positions*. Technical report. Uniwersytet Gdański, Instytut Matematyki, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - Safety*. Springer, 1995.

[MQ07] M. Musuvathi and S. Qadeer. *Iterative context bounding for systematic testing of multithreaded programs*. In: PLDI. 2007, pages 446–455.

[MS62] J. Mycielski and H. Steinhaus. *A mathematical axiom contradicting the axiom of choice*. Bulletin de l'Académie Polonaise des Sciences. Série des Sciences Mathématiques, Astronomiques et Physiques 10 (1962), pages 1–3.

[MS72] A. R. Meyer and L. J. Stockmeyer. *The equivalence problem for regular expressions with squaring requires exponential space*. In: Switching and Automata Theory, 1972, pages 125–129.

[MSB12] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. (3rd ed). John Wiley & Sons, 2012.

[MSS06] A. Muscholl, T. Schwentick, and L. Segoufin. *Active context-free games*. Theory Comput. Syst. 39.1 (2006), pages 237–276.

[Mul63] D. E. Muller. *Infinite sequences and finite machines*. In: Switching Circuit Theory and Logical Design. 1963, pages 3–16.

[Mus16] S. Muskalla. *RIGG – Regular Inclusion Grammar Games*. (Accessed on 2022-05-24). 2016. URL: https://github.com/SebastianMuskalla/RIGG.

[MW20] R. Meyer and S. van der Wall. *On the complexity of multi-pushdown games*. In: FSTTCS. Volume 182. LIPIcs. 2020, pages 52:1–52:35.

[MW80] Z. Manna and R. J. Waldinger. *A deductive approach to program synthesis*. ACM Trans. Program. Lang. Syst. 2.1 (1980), pages 90–121.

[MY60] R. McNaughton and H. Yamada. *Regular expressions and state graphs for automata*. IRE Transactions on Electronic Computers EC-9.1 (1960), pages 39–47.

[Nas68] C. S. J. A. Nash-Williams. *On better-quasi-ordering transfinite sequences*. Mathematical Proceedings of the Cambridge Philosophical Society 64.2 (1968), pages 273–290.

[Ner58] A. Nerode. *Linear automaton transformations*. Proceedings of the American Mathematical Society 9.4 (1958), pages 541–544.

[Neu17] E. Neumann. *Algorithms for context-free games: A comparison of saturation, guess & check and summarization*. (Accessed 2022-05-23). Master's thesis. TU Kaiserslautern, 2017. URL: https://www.tcs.cs.tu-bs.de/documents/elisabeth_neumann_masters_thesis.pdf.

[Niv72] M. Nivat. *On the interpretation of recursive program schemes*. Symposia Matematica (1972).

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[OG76] S. S. Owicki and D. Gries. *Verifying properties of parallel programs: An axiomatic approach*. Commun. ACM 19.5 (1976), pages 279–285.

[Ong06] C.-H. L. Ong. *On model-checking trees generated by higher-order recursion schemes*. In: LICS. 2006, pages 81–90.

[Pad21] S. Padhi et al. *SyGuS*. (Accessed 2022-08-01). 2021. URL: https://sygus.org/.

[Par66] R. J. Parikh. *On context-free languages*. J. ACM 13.4 (Oct. 1966), pages 570–581.

[Pét35] R. Péter. *Konstruktion nichtrekursiver Funktionen*. Mathematische Annalen 111 (1935), pages 42–60.

[Pey03] S. Peyton Jones (Editor). *Haskell 98 language and libraries – The revised report*. Cambridge University Press, 2003.

[PGBCB08] C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. *Learning to divide and conquer: Applying the L* algorithm to automate assume-guarantee reasoning*. Formal Methods Syst. Des. 32.3 (2008), pages 175–205.

[Pnu77] A. Pnueli. *The temporal logic of programs*. In: FOCS. 1977, pages 46–57.

[Pnu84] A. Pnueli. *In transition from global to modular temporal reasoning about programs*. In: Logics and Models of Concurrent Systems. Volume 13. NATO ASI Series. 1984, pages 123–144.

[Pos81] J. Postel. *Transmission Control Protocol*. (Accessed on 2022-05-30). 1981. URL: https://www.rfc-editor.org/info/rfc793.

[Pou92] W. Poundstone. *Prisoner's dilemma*. Doubleday, 1992.

[PP04] D. Perrin and J. Pin. *Infinite words - Automata, semigroups, logic and games*. Volume 141. Pure and applied mathematics series. Elsevier, 2004.

[Pre31] M. Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*. 1931.

[PRZ13a] T. Place, L. van Rooijen, and M. Zeitoun. *Separating regular languages by locally testable and locally threshold testable languages*. In: FSTTCS. 2013, pages 363–375.

[PRZ13b] T. Place, L. van Rooijen, and M. Zeitoun. *Separating regular languages by piecewise testable and unambiguous languages*. In: MFCS. 2013, pages 729–740.

[PZ14] T. Place and M. Zeitoun. *Going higher in the first-order quantifier alternation hierarchy on words*. In: ICALP. 2014, pages 342–353.

[PZ16] T. Place and M. Zeitoun. *Separating regular languages with first-order logic*. Logical Methods in Computer Science 12.1 (2016).

[QR05] S. Qadeer and J. Rehof. *Context-bounded model checking of concurrent software*. In: TACAS. Volume 3440. LNCS. 2005, pages 93–107.

[Rab68] M. O. Rabin. *Decidability of second-order theories and automata on infinite trees*. Bull. Amer. Math. Soc. 74.5 (Sept. 1968), pages 1025–1029.

[Rac78] C. Rackoff. *The covering and boundedness problems for vector addition systems*. TCS 6.2 (1978).

[Rad54] R. Rado. *Partial well-ordering of sets of vectors*. Mathematika 1.2 (1954), pages 89–95.

[Rag94] T. Raghavan. *Chapter 20: Zero-sum two-person games*. In: Handbook of Game Theory with Economic Applications. Volume 2. 1994, pages 735–768.

[Ram00] G. Ramalingam. *Context-sensitive synchronization-sensitive analysis is undecidable*. ACM Trans. Program. Lang. Syst. 22.2 (2000), pages 416–430.

[Ram30] F. P. Ramsey. *On a problem of formal logic*. Proceedings of the London Mathematical Society 1 (1930), pages 264–286.

[Rei85] W. Reisig. *Petri nets: An introduction*. Monographs in Theoretical Computer Science. Springer, 1985.

[Ric53] H. G. Rice. *Classes of recursively enumerable sets and their decision problems*. Trans. Amer. Math. Soc. 74 (1953), pages 358–366.

[Ros12] J. Rosenberger. *P vs. NP poll results*. Communications of the ACM 55.5 (May 2012), page 10.

[RS59] M. O. Rabin and D. Scott. *Finite automata and their decision problems*. IBM Journal of Research and Development 3 (2 1959), pages 114–125.

[Rus03] B. Russell. *Principles of mathematics*. Cambridge University Press, 1903.

[Saf88] S. Safra. *On the complexity of omega-automata*. In: SFCS. 1988, pages 319–327.

[Sav70] W. J. Savitch. *Relationships between nondeterministic and deterministic tape complexities*. Journal of Computer and System Sciences 4.2 (1970), pages 177–192.

[Sca84] B. Scarpellini. *Complexity of subcases of Presburger arithmetic*. Transactions of the AMS 284.1 (1984).

[Sch16] S. Schmitz. *Complexity hierarchies beyond elementary*. TOCT 8.1 (2016), pages 3:1–3:36.

[Ser03] O. Serre. *Note on winning positions on pushdown games with omega-regular conditions*. Inf. Process. Lett. 85.6 (2003), pages 285–291.

[Set09] A. Seth. *Games on multi-stack pushdown systems*. In: LFCS. Volume 5407. LNCS. 2009, pages 395–408.

[SKZ21] A. K. Shetty, S. N. Krishna, and G. Zetzsche. *Scope-bounded reachability in valence systems*. In: CONCUR. Volume 203. LIPIcs. 2021, pages 29:1–29:19.

[SM73] L. J. Stockmeyer and A. R. Meyer. *Word problems requiring exponential time: Preliminary report*. In: STOC. 1973, pages 1–9.

[SS15] M. Schuster and T. Schwentick. *Games for active XML revisited*. In: ICDT. Volume 31. LIPIcs. 2015, pages 60–75.

[SS71] D. Scott and C. Strachey. *Towards a mathematical semantics for computer languages*. Proceedings of the Symposium on Computers and Automata 21 (1971).

[SS78] W. J. Sakoda and M. Sipser. *Nondeterminism and the size of two way finite automata*. In: STOC. 1978, pages 275–286.

[ST77] G. S. Sacerdote and R. L. Tenney. *The decidability of the reachability problem for vector addition systems (Preliminary version)*. In: STOC. 1977, pages 61–76.

[Str81] R. S. Streett. *Propositional dynamic logic of looping and converse*. In: STOC. 1981, pages 375–383.

[Stu17] F. Stutz. *Operations on a symbolic domain for synthesis*. (Accessed 2022-05-23). Bachelor's thesis. TU Kaiserslautern, 2017. URL: https://www.tcs.cs.tu-bs.de/documents/felix_stutz_bachelors_thesis.pdf.

[Sum77] P. D. Summers. *A methodology for LISP program construction from examples*. J. ACM 24.1 (1977), pages 161–175.

[SVW87] A. P. Sistla, M. Y. Vardi, and P. Wolper. *The complementation problem for Büchi automata with appplications to temporal logic*. Theor. Comput. Sci. 49 (1987), pages 217–237.

[SW15a] S. Salvati and I. Walukiewicz. *A model for behavioural properties of higher-order programs*. In: CSL. Volume 41. LIPIcs. 2015, pages 229–243.

[SW15b] S. Salvati and I. Walukiewicz. *Using models to model-check recursive schemes*. Log. Methods Comput. Sci. 11.2 (2015).

[SW76] T. G. Szymanski and J. H. Williams. *Noncanonical extensions of bottom-up parsing techniques*. SIAM Journal on Computing 5.2 (1976), pages 231–250.

[SWH12] H. Seidl, R. Wilhelm, and S. Hack. *Compiler design: Analysis and transformation*. Springer, 2012.

[Sze87] R. Szelepcsényi. *The method of forcing for nondeterministic automata*. Bulletin of the EATCS 33 (1987), pages 96–100.

[Tar49] A. Tarski. *A fixed point theorem for lattices and its applications (Preliminary version)*. Bull. A.M.S. 55 (1949), pages 1051–1052.

[Tar55] A. Tarski. *A lattice theoretical fixed point theorem and its applications*. Pacific J. Math. 5 (1955), pages 285–309.

[Tho90] W. Thomas. *Automata on infinite objects*. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. 1990, pages 133–191.

[TMP07] S. L. Torre, P. Madhusudan, and G. Parlato. *A robust class of context-sensitive languages*. In: LICS. 2007, pages 161–170.

[TNP20] S. L. Torre, M. Napoli, and G. Parlato. *Reachability of scope-bounded multistack pushdown systems*. Inf. Comput. 275 (2020), page 104588.

[Tse68] G. Tseytin. *On the complexity of derivation in propositional calculus*. Zapiski Nauchnykh Seminarov LOMI 8 (1968), pages 234–259.

[Tur36] A. M. Turing. *On computable numbers, with an application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society 2.42 (1936), pages 230–265.

[TV05] D. Tabakov and M. Y. Vardi. *Experimental evaluation of classical automata constructions*. In: LPAR. Volume 3835. LNCS. 2005, pages 396–411.

[TZ19] R. S. Thinniyam and G. Zetzsche. *Regular separability and intersection emptiness are independent problems*. In: FSTTCS. 2019, pages 51:1–51:15.

[Val75] L. G. Valiant. *Regularity and related problems for deterministic pushdown automata*. J. ACM 22.1 (1975), pages 1–10.

[Var18] M. Y. Vardi. *The siren song of temporal synthesis (Invited talk)*. In: CONCUR. Volume 118. LIPIcs. 2018, pages 1:1–1:1.

[Var98] M. Y. Vardi. *Reasoning about the past with two-way automata*. In: ICALP. Volume 1443. LNCS. 1998, pages 628–641.

[VSS05] K. N. Verma, H. Seidl, and T. Schwentick. *On the complexity of equational Horn clauses*. In: CADE. 2005, pages 337–352.

[Wal01] I. Walukiewicz. *Pushdown processes: Games and model-checking*. Inf. Comput. 164.2 (2001), pages 234–263.

[Wal02] I. Walukiewicz. *Monadic second-order logic on tree-like structures*. Theor. Comput. Sci. 275.1-2 (2002), pages 311–346.

[Wim08] H. Wimmel. *Entscheidbarkeit bei Petri Netzen: Überblick und Kompendium*. eXamen.press. Springer, 2008.

[Zet13] G. Zetzsche. *Silent transitions in automata with storage*. In: ICALP. 2013, pages 434–445.

[Zet15a] G. Zetzsche. *An approach to computing downward closures*. In: ICALP. LNCS. 2015.

[Zet15b] G. Zetzsche. *Monoids as storage mechanisms*. PhD thesis. TU Kaiserslautern, 2015.

[Zet15c] G. Zetzsche. *The emptiness problem for valence automata or: Another decidable extension of Petri nets*. In: RP. 2015, pages 166–178.

[Zet21] G. Zetzsche. *The emptiness problem for valence automata over graph monoids*. Inf. Comput. 277 (2021), page 104583.

[Zie98] W. Zielonka. *Infinite games on finitely coloured graphs with applications to automata on infinite trees*. Theor. Comput. Sci. 200.1-2 (1998), pages 135–183.