Context-free Games on Strings and Nested Words

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Martin Schuster

Dortmund

2017

Tag der mündlichen Prüfung:	11. Oktober 2017
Dekan:	Gernot A. Fink
Gutachter:	Thomas Schwentick
	Christof Löding

Preface

When Thomas Schwentick first introduced me to context-free games, they were originally just supposed to be a little side project for me while I was looking for a suitable topic for my dissertation. The idea, back then, was for me to take some results from Joscha Kulbatzki's diploma thesis, expand on them and make a paper out of it as a way to get used to doing research. At that time, I believed that most of the work on context-free games had already been done, and there wasn't much more left to research.

By the time this side project turned into my first publication, however, my views had changed. Thomas and I had uncovered a lot more potential for research into context-free games by extending them from strings into nested words, and I found the topic just too interesting to give up on, so we decided that context-free games should indeed be the subject of my dissertation.

Now, four years and two rather voluminous publications later, that dissertation is finally complete, and I must say that getting here was quite the eventful journey – often difficult, sometimes downright frustrating in stretches with little progress, but always fascinating and, in retrospect, definitely worth the effort.

With this dissertation, I hope to share with its readers some of my fascination for formal games and automata in general and context-free games in particular, and perhaps raise some interest for the very natural, yet surprisingly little-known model that is context-free games.

I am deeply obliged to all the people without whom this dissertation would not have been possible. First of all, I thank Thomas Schwentick for being a committed advisor, a considerate employer, and a shining example in research and teaching; most of what I know about good research, scientific writing, and presentation, I learned from him.

Along with Thomas, I also thank my current and former colleagues in our working group and at our chair for providing fruitful discussion and an all-around pleasant work environment that made me enjoy my job even in stressful times.

I am grateful to teachers and professors too numerous to list in Dortmund, Tübingen, Balingen and Baden-Baden. Everyone who had a part in sparking and fanning my curiosity towards science, mathematics, computer science, and the world in general, deserves my heart-felt thanks.

On a more personal note, I thank Peter, Till, Niclas and Christian for providing me with a home away from home and a refuge from science in shared fantasy, and, along with Dave, I thank them for excellent food, enjoyable company and a welcome chance to polish my English skills every now and then. I thank Olav for brightening up my day whenever I needed it most (and lots of other times besides). I am grateful to all the lovely people (too many, again, to name each one on their own) in my extended circle of friends who supported me and helped give me strength to finish this dissertation.

Finally, I thank my parents from the bottom of my heart, for their unconditional support and unwavering belief in me throughout my entire life. Of all the people who helped make this dissertation happen, I owe them the most.

Dortmund, May 2017

Martin Schuster

Contents

1. Introduction

Formal two-player games have been playing an important role in several areas of theoretical computer science, at the very least since Chandra, Kozen and Stockmeyer's seminal work on alternation [CKS81], if not before that. Perhaps one of the most interesting uses of two-player games (and certainly the most relevant for this dissertation) lies in modelling scenarios where an algorithm has to deal with an uncertain environment that can be roughly constrained to certain types of behaviour, but not predicted accurately. Modelling a scenario of this kind by a two-player game usually follows a "games against nature" approach, which casts the algorithm and the environment as opposing players trying to win against each other, and thus makes the worst-case assumption that the environment is actively working against the algorithm, trying to keep the algorithm from reaching its goal as best as possible. Some more concrete examples of this are given below, after a brief sketch of the intuition behind two-player games.

In a nutshell, these games are played by two players who take turns performing moves in some given arena, with each player trying to reach some given objective while at the same time keeping the other player from reaching theirs. The central algorithmic questions for games of this sort usually concern the existence of *strategies* with certain properties for some player, i.e. instructions for the player on how to move in any given game position with a given history. An important example of such strategies are *winning strategies*, which allow a player following the winning strategy to always win the game, regardless of how the other player moves. Some (more or less practical) applications for two-player games are as follows.

In complexity theory and logic, two-player games can be used to define semantics or alternate characterisations for alternating computation models or logics (see, e.g. [Chl86; GKR16]), with the first player being responsible for existential choices, the second player for universal choices, and the existence of a winning strategy for the first player corresponding to acceptance of an input or fulfilment of a logical formula.

More algorithmically oriented fields such as controller synthesis or competitive analysis of on-line algorithms (see e.g. [MPS95; BLS92]) sometimes utilise two-player games in the "games against nature" approach sketched above. In these applications, the goal is generally to find a strategy for the first player that is optimal in some sense, which corresponds to designing a controller or algorithm that deals with the environment in the most secure or efficient way possible.

The *context-free games* examined in this dissertation were originally formalised by Muscholl, Schwentick and Segoufin [MSS06] in the vein of this "games against nature" approach to pinpoint the complexity of the schema rewriting problem for Active XML (described in some more detail in Section 1.2); more recently, an application of context-free games to synthesis of recursive programs has also been investigated [HMM16]. Seeing

as context-free games are a very natural generalisation of a well-known problem, it is likely that further applications exist.

1.1. Context-free games: Basic intuition

In computational complexity, many two-player game arise in a natural manner by taking a "puzzle" problem that is easily solved using nondeterminism and turning this puzzle into a game. For instance, the Tiling problem asks, given two natural numbers ℓ, m and a set of tiles that may only be placed adjacent to each other according to certain restrictions, whether it is possible to tile an $\ell \times m$ rectangle while adhering to the given restrictions. In the two-player version of this problem, players take turns sequentially placing tiles in accordance with the given restriction, and the first player wins the game if and only if this process ends with a valid tiling.[1] While Tiling is a natural complete problem for nondeterministic polynomial time, the problem 2-Player Tiling of determining whether the first player has a winning strategy in the corresponding two-player game is complete for alternating polynomial time.

In their most simple form, context-free games can be derived by a similar "gamification" process from a slight relaxation of the word problem for context-free grammars: Given a context-free grammar G and sentential forms w, w' (i.e. strings that may contain both terminal and non-terminal symbols), can w' be derived from w according to the rules of G? This problem has a very natural solution algorithm using nondeterminism to choose which non-terminal symbols should be replaced, and what right-hand side from the corresponding rule should be used to replace them.

In the two-player version, it is the first player (called Juliet throughout the literature on context-free games) who gets to choose which non-terminal to *call* (i.e. select for replacement), while the choice of the right-hand side to replace the chosen symbol is up to the second player (called Romeo). The question whether w' can be derived from w then naturally generalises to the question whether Juliet has a winning strategy in the corresponding game that *enforces* rewriting w into w'. Due to the practical motivation (detailed in Section 1.2), and because the target string w' does not have to consist solely of terminal symbols, from now on, we refer to non-terminals as *function symbols* instead, as the term "non-terminal" is somewhat inappropriate for symbols that may be allowed (or even desired) in the final result of a game.

This very simple model of context-free games can be extended in several ways, most of which have a direct motivation from the practical application of context-free games in Active XML. The first work on context-free games [MSS06] already considered a more general winning condition for Juliet, where her objective is not to reach a specific string w', but an *arbitrary* string from a given regular *target language*. A similar generalisation can be applied to the replacement rules – [MSS06] also examined games in which the replacement mechanism is basically an *extended* context-free grammar, i.e. Romeo gets to choose replacement strings from regular (instead of just finite) *replacement languages*.

[1] More information on tiling problems, as well as formal definitions, can be found in Section 2.5.

Example 1.1. *Consider the extended context-free grammar over the alphabet* $\{a, f, g\}$ *with function symbols* f, g *and replacement rules* $f \to g(f + g^*)g$, $g \to af + f$, *and a target language given by the regular expression* $a^*ff^*gg^*$. *On the initial string* $w = agfg$, JULIET *can enforce a successful rewriting into the target language by selecting the leftmost* g *for replacement.* ROMEO *can replace this* g *either by* af *or by* f, *yielding one of the strings* $aaffg$ *or* $affg$, *both of which are contained in the target language. In other words,* JULIET *has a winning strategy on* $agfg$ *in this context-free game.*

Context-free games can be further generalised from strings to more intricate structures. In [SS15], this was done for *nested words*, an XML-style linearisation of trees into strings. Since the structure of nested words (or, equivalently, trees) is somewhat more complex than that of strings, basically allowing JULIET to request the replacement of entire substrings (or subtrees, respectively), this opens up another avenue of variation in how ROMEO is allowed to respond to this request, depending on the substring (subtree) to be replaced. In the most simple case, ROMEO'S response may only depend on the root label of the substring (subtree) to be replaced; in more involved cases, more complex dependencies between the input and output of replacements can be considered as well, such as the *validation games* of [SS15] and games with transducer-based replacement [Sch16]. Further variation is possible by requiring the originally called substring (subtree) to be kept and appended to, instead of replaced.

Parallel to these generalisations, it is also important to consider restrictions on context-free games, in order to reduce the complexity of deciding whether JULIET has a winning strategy, or to avoid undecidability of this problem. One way of imposing such restrictions is by only allowing certain classes or representations of replacement rules and target languages; the other main class of restrictions constrains the manner in which JULIET may select function symbols for replacement. One central example of such a restriction is the *left-to-right (L2R)* restriction, which prohibits JULIET from calling function symbols that are strictly to the left of a function symbol she has previously called; this restriction will be assumed throughout most of this dissertation, as the winning problem quickly gets undecidable in unconstrained games (i.e. games without this restriction). Another important restriction is a limit to the *Call depth*, i.e. the recursion depth of function calls, constraining the extent to which JULIET may call function symbols that were previously returned by ROMEO inside results of prior function calls.

Generalising all the variants discussed, context-free games could be defined as games played on some (logical) structure by two players, JULIET and ROMEO, in which players take turns with JULIET choosing some sub-structure of the current structure (in accordance with some restriction on the order in which she may choose sub-structures) and ROMEO replacing the sub-structure chosen by JULIET (in accordance with some replacement relation), where JULIET wins a play if, at some point, a structure from some given target set of structures is reached.[2]

[2] Formalising this general definition and instantiating it to each of the classes of games examined in this dissertation would be a rather technical and somewhat unintuitive task. For this reason, separate definitions are given throughout this dissertation for each of the major classes of context-free games studied here, even though this creates some redundancy and repetition in the concrete definitions.

Therefore, a class of context-free games can be specified by the following parameters:

- The class of structures being played on. This dissertation uses only two comparatively simple classes of structures (strings and nested words), but examining more general structures (graphs, data words/trees, arbitrary logical structures, etc.) might be of future interest as well.
- The restrictions for Juliet on selecting sub-structures. Most of this dissertation assumes the left-to-right restriction discussed above, since the winning problem for games without this restriction becomes undecidable quite easily (see also the discussion in Chapter 3). In most cases, we also examine the impact of replay on the complexity of the winning problem for Juliet, and some chapters introduce additional restrictions such as bounded *Call width* and *write-once games* in Chapter 9.
- The replacement relation according to which Romeo can choose replacement sub-structures depending on the sub-structures chosen by Juliet. A major part of this dissertation focuses on the case where the replacement relation gives a regular language for each function symbol and Romeo may choose a replacement from the language corresponding to the central function symbol in the sub-structure chosen by Juliet (i.e. the symbol chosen in a string, or the root of a chosen nested word), but more complex replacement mechanisms (validation-based replacement in Chapter 7 and transducer-based replacement for various forms of transducers in Chapter 9) are also studied.
- The class of target sets of structures. Throughout this dissertation, target sets will be regular languages (of strings or nested words) and subclasses thereof, whose representation (deterministic automata of various kinds, or explicit enumeration for finite languages) may influence the complexity of the winning problem. Other representations (such as non-deterministic automata, grammars, or logic-based representations) as well as different classes of target sets might also be of interest for future research.

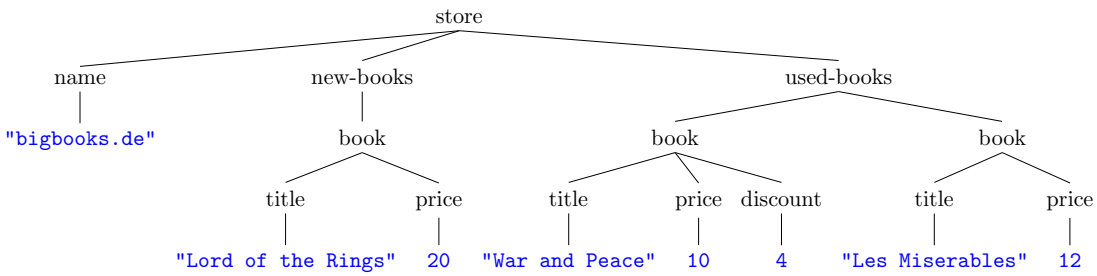
1.2. An application: Schema rewriting for Active XML

As mentioned above, the initial research into context-free games was originally motivated by an application in database theory, the schema rewriting problem for Active XML. While Active XML is not a focus of this dissertation, many of the results and restrictions presented here can be related to this original application; therefore, a short primer on (Active) XML and schema rewriting may be in order to convey the motivation for this research. This section focuses on intuitive descriptions and examples, with all required formalism delegated to later chapters.

XML. XML documents are widely used for storing and exchanging semi-structured data, particularly on the internet. As an example, Figure 1.1a shows a simplified excerpt

```
<store>
   <name> "bigbooks.de" </name>
   <new-books>
      <book>
         <title> "Lord of the Rings" </title>
         <price> 20 </price>
      </book>
   </new-books>
   <used-books>
      <book>
         <title> "War and Peace" </title>
         <price> 10 </price>
         <discount> 4 </discount>
      </book>
      <book>
         <title> "Les Miserables" </title>
         <price> 12 </price>
      </book>
   </used-books>
</store>
```

(a) Example XML document



(b) Tree representation of the XML document in Figure 1.1a.

Figure 1.1.: Example XML document from Section 1.2 (Fig. 1.1a) along with its tree representation (Fig. 1.1b).

of an XML document that might be used to store information about an online book store, giving the name of the store ("bigbooks.de") as well as a selection of books on sale, with their names, prices, and potentially a discount on the price.[3]

XML documents can be interpreted as node-labelled trees in the standard fashion by interpreting each matched pair of an opening $\langle a \rangle$ and a closing $\langle /a \rangle$ tag (for some label a) as a node labelled with a whose child forest consists of the nodes corresponding to the part of the document nested between $\langle a \rangle$ and $\langle /a \rangle$; as an example, the tree corresponding to the document from Figure 1.1a is shown in Figure 1.1b. As something of a special case, infinite-domain *data values* such as texts or numbers (like the leaves marked in `blue true-type` text in the tree of Figure 1.1b) are not tag labels and have to be handled separately; this dissertation, like much of the literature on the theory of XML, abstracts away from data values and assumes that XML documents generally correspond to node-labelled trees over some finite label alphabet.

Schemas for XML. Formal validation and classification tasks often require *schemas* as formalisms for describing sets of XML documents. In the standard abstraction of XML documents as node-labelled unranked trees, a schema is simply some formalism for describing a language of such trees. From the perspective of automata theory, the most simple class of tree languages are *regular* tree languages, which are described by *finite tree automata*[4]. However, for practical applications of XML, regular tree languages are often unsuited since they offer more expressive power than is generally required, at the expense of increased complexity for some decision problems (cf. [MNSB06; MNS09]). Instead, the theory of XML generally focusses on two more restricted classes of schema languages: those given by *Document Type Definitions (DTDs)* and *XML Schema*.

DTDs, from a language-theoretic perspective, are just *extended context-free grammars*, i.e. context-free grammars with productions $a \to r_a$ mapping each alphabet symbol a to a regular expression r_a, called the *content model* of a.[5] The tree language defined by a DTD is simply the set of its derivation trees, i.e. a tree *matches* a DTD if, for each of its nodes labelled with some symbol a, the string obtained by concatenating the node's children's labels from left to right matches the content model for a. For instance, XML trees for online book stores like the one in Figure 1.1b (without `data values`) could be described by a DTD with starting symbol `store` and the following rules:

`store`	$\to$	`name new-books used-books`
`new-books`	$\to$	`book`*
`used-books`	$\to$	`book`*
`book`	$\to$	`title price` (`discount` $+ \epsilon$)

XML Schema somewhat extends DTDs. In a nutshell, an XML Schema may be viewed as an *extended DTD* [PV00; BPV04; MNSB06], i.e. a DTD over an alphabet Δ of *types*, along with a *labelling function* mapping types to symbols from some label alphabet

[3] This example is inspired by a similar one given in [MNSB06].

[4] See e.g. [Sch07] for a survey on automata models for XML.

[5] More specifically, regular expressions used in DTDs and XML Schema are required to be *deterministic* in order to allow for polynomial-time construction of an equivalent deterministic finite-state automaton. For more discussion, see e.g. [BW98].

Σ with a *single-type* restriction requiring that no two types appearing together in any content model are mapped to the same label from Σ. A tree t then *matches* an XML Schema if there is some tree t' with labels from Δ that matches the base DTD such that applying the labelling function to each of the node labels of t' yields t. As an example, the following DTD along with a labelling function that maps both $\texttt{book}_1$ and $\texttt{book}_2$ to `book` (and all other labels to themselves) yields a somewhat more accurate model for trees like the one from Figure 1.1b, assuming that only used books can be subject to a discount[6]:

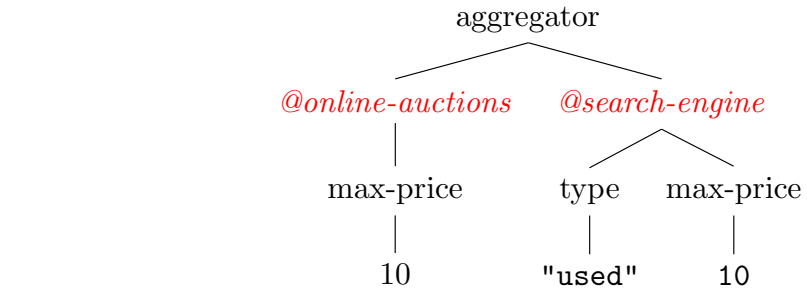
$$
\begin{array}{lcl}
\texttt{store} & \rightarrow & \texttt{name new-books used-books} \\
\texttt{new-books} & \rightarrow & \texttt{book}_1^* \\
\texttt{used-books} & \rightarrow & \texttt{book}_2^* \\
\texttt{book}_1 & \rightarrow & \texttt{title price} \\
\texttt{book}_2 & \rightarrow & \texttt{title price (discount} + \epsilon\texttt{)}
\end{array}
$$

Formally, XML Schemas (respectively DTDs) may be modelled as *single-type* (resp. *local*) *tree grammars* defined in detail in Chapter 2.

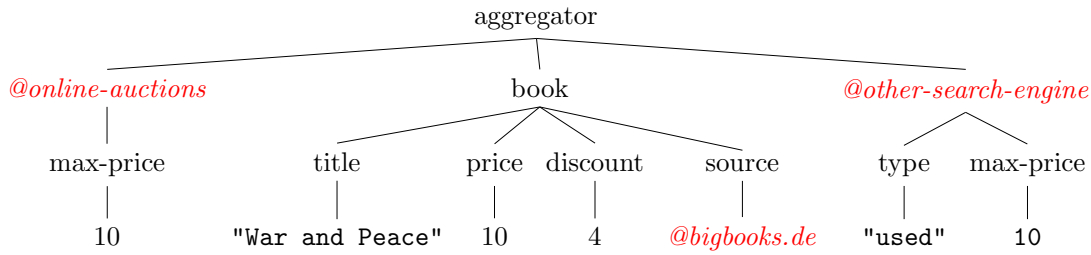
Active XML. *Active XML* (or *AXML* for short) is an XML-based formalism for modelling dynamic data exchange. At its core, AXML describes *intensional* XML documents in which data need not be given explicitly as part of the document itself but may instead be referenced by way of *function nodes* referring to external services or sources. These function nodes can then be *called* at any time, querying the corresponding external service for up-to-date data to be *materialised* in the document. In the standard AXML semantics, when a function call occurs, the subtree rooted at the function node that was called is passed to the corresponding external service as a call parameter; the external service then returns a forest as a function call result, which replaces the called function node (and the subtree rooted at it) in the AXML document. Alternative semantics (where for instance the function call result is inserted to the right of the called function node instead of replacing it) exist; for a survey, see e.g. [ABM08]. In general, external services come with schemas specifying the allowed inputs and possible outputs for function calls; these are specified in the standard manner for XML, as DTDs or XML Schema.

As a simple example, consider the AXML document in Figure 1.2a, which might be part of an aggregator collecting offers for books from various sources; in this small example, the aggregator might query some search engine or some online auction site, represented by the function nodes printed in *red italics*. The subtree rooted at the @search-engine function node, for instance, contains parameters telling the external service to return used books with a maximum price of 10. After calling the @search-engine function node, the rewritten AXML document might look like the one in Figure 1.2b; here, the subtree rooted at the @search-engine node has been replaced by a subforest containing two trees, of which the first represents a search result (the book "War and Peace" from bigbooks.de), and the second represents an option to search further using a different

[6] This assumption is actually somewhat realistic, as several countries in the EU (among them Germany) have fixed price book laws mandating that new books may generally not be discounted.



(a) Example AXML document before rewriting



(b) Document from Figure 1.2a after calling the @search-engine function node.

Figure 1.2.: Example of Active XML rewriting from Section 1.2.

search engine. Note that the result of this function call inserts additional function nodes into the document in this example.

The schema rewriting problem. The process of performing function calls and materialising data yields a way of rewriting AXML documents, which leads to the very natural questions if, and how, a given document can be rewritten into a desired form, regardless of the precise returns of function calls. This problem was first investigated in the form of *schema rewriting* in [MAABN05]: Given an initial AXML document, specifications for the input and output schemas of external services, and a *target schema*, is it always possible to rewrite the initial document into a document conforming to the target schema by calling functions and materialising call results?

In the example of Figure 1.2, the target schema might for instance require that in the rewritten document, there is a certain minimum or maximum number of books, or that if the rewritten document contains results from online auctions, all other search results have to consist solely of used books. Since it can be seen in the example that the results of calls to function nodes may contain additional function nodes, rewriting a given document in the target schema may require additional function calls *within* the results of prior calls.

Obviously, assuming that the results of function calls are not known in advance, if and when a function node should be called has to depend on the results of prior function calls; what we are interested in, therefore, is (the existence of) a *safe rewriting strategy*, i.e. a set of instructions specifying which function node to call in which situation, in

order to ensure that, at some point, a document matching the target schema is reached.

Connection to context-free games. The schema rewriting problem for AXML can be modelled using context-free games in a very natural way, again following the "games against nature" approach described in the previous section – JULIET'S choice of function symbols to call corresponds to a rewriting strategy's choice of function nodes, and ROMEO'S choice of replacements corresponds to the uncertainty about the exact results of function calls, with the game's target set of structures being constructed from the target schema in the schema rewriting problem.

The original paper by Milo et al. [MAABN05] examined the schema rewriting problem where all schemas are given by DTDs. Due to the relatively simple nature of DTDs along with some additional restrictions, in their setting, the schema rewriting problem can essentially be reduced to the winning problem for JULIET in context-free games on strings using regular target and replacement languages. More expressive schema languages such as XML Schema and regular tree languages then led to context-free games on nested words being investigated in [SS15]. In this setting, some care has to be taken as to how function parameters are handled; the simplest case where parameters are simply ignored and overwritten is generally much less complex than the case where validity of parameter subtrees with respect to input schemas has to be taken into account. Actual semantic dependencies between input parameters and outputs of function calls require a more involved replacement relation still; for transducer-based replacement, this was studied in [Sch16], and research is still ongoing.

1.3. Methods and techniques

While this dissertation characterises the complexity of several different variants of context-free games (see Section 1.1 for a rough overview), some basic proof ideas and techniques keep re-occurring throughout. This section offers a brief intuitive introduction of those techniques; a more detailed and formal discussion is given in later chapters when these techniques are first used (most notably in Chapter 3 with concrete examples for context-free games on strings).

Alternation. The connection between two-player games and alternation is a well-known and long-standing one [Chl86], so the most direct approach for solving context-free games consists of translating a game into its "trivial" alternating algorithm which guesses existentially moves for JULIET and guesses universally moves for ROMEO. While this approach is feasible in some cases (most notably for several simpler variants of games with transducer-based replacement), it suffers from two main problems in most variants of context-free games considered here. First off, there is often no sufficiently strict bound on the time games take to terminate when replay is allowed; in fact, with unbounded replay, games may not be guaranteed to terminate at all. Second, if the replacement relation is not finite, this allows for an unbounded number of potential choices of replacement for ROMEO; in this case, again, there is no immediate upper bound on the time and

space used by the trivial alternating algorithm, as replacement substructures may grow arbitrarily large. This dissertation uses two main techniques to deal with these two problems: Size bounding and abstract representation of (sub-)games.

Size bounding. In settings with bounded replay, an upper bound on the termination time for the trivial alternating algorithm may be derived from the depth of replay and the maximum size of replacement substructures. Therefore, if the size of replacement substructures can be bounded in these settings, both problems of the trivial alternating algorithm can be solved at once. In the case of finite replacement relations, this size bound is immediately obvious, while for more general replacement relations, some additional arguments can sometimes be used to show that ROMEO can be restricted to choosing from a finite sub-relation without loss of generality.

Abstract representation of (sub-)games. In settings where the size of replacement substructures cannot easily be bounded, the standard approach of this dissertation is to instead prove a bound on the set of possible impacts that replacements may have on the (representation of the) target set. The most common way to do this is the *effect technique* used in context-free games on strings or nested words with regular replacement languages and target languages given as deterministic automata. In a nutshell, instead of considering concrete replacement strings (or nested words, respectively), one considers states reachable in the target automaton by these strings (nested words) and summarises strategies for JULIET by the set of states she can enforce against various counter-strategies of ROMEO. This approach generalises to several large classes of games, with the added advantage that it allows the trivial alternating algorithm to be formulated as an alternating automaton, thus also proving regularity for the set of initial strings on which JULIET has a winning strategy. Some slightly less general variants of abstract representation are also used for validation games in Chapter 7.

Games for lower bounds and the protest technique. The most direct way of proving matching lower bounds on the complexity of the winning problem for JULIET is by reduction from problems inherently containing alternation, simulating existential choice by choices of JULIET and universal choice by choices of ROMEO. The main obstacle in these kinds of reduction is that standard alternation is symmetric, with universal and existential choice basically being handled the same way, while context-free games contain an asymmetry in the roles of JULIET and ROMEO – while ROMEO may choose from a given set of replacement substructures, JULIET'S choice is (at least with the standard left-to-right restriction) essentially, for each sub-structure, just a binary choice of whether to have that substructure replaced or not. Most of the work in these kinds of reduction therefore goes into simulating more expressive styles of existential choice by strategy decisions of JULIET.

The second main type of lower bound proof leverages this asymmetry by reducing from existence problems of the form "given some input and desired property, does there exist a witness proving the desired property of the input?". In these kinds of reduction, ROMEO

is usually tasked with providing a (not necessarily correct) witness, after which JULIET has to give evidence that this witness is incorrect by flagging down errors in the witness. A more elaborate variant of this type of reduction uses the *protest technique* introduced in [MSS06]; there, the supposed errors pointed out by JULIET may not be correct, either, so ROMEO may in turn "protest" by marking errors in JULIET'S evidence, to which JULIET may in turn raise a counter-protest, possibly followed by further iterations of protest and counter-protest. A somewhat intricate example of this technique can be seen in the lower bound proofs of Chapter 4.

1.4. Related work

Beyond the works already cited ([MSS06; BSSK13; SS15; Sch16]), little previous work on context-free games exists; [AMB05] examine *one-pass* strategies, which impose a "streaming-like" restriction on left-to-right strategies, thus giving rise to context-free games with imperfect information. In their application of context-free games to program synthesis, [HMM16] reproduce a result from [MSS06] using a technique based on Boolean formulas over transition monoids similar to the effect technique presented here. This was further extended to games on infinite strings [MMN17], but the corresponding work is (as of this writing) not yet formally published.

A survey on Active XML (including more context on the schema rewriting problem from [MAABN05]) is presented in [ABM08]; further information can be found on the Active XML web site [AXML].

Even though at first glance, there seems to be some similarity between context-free games and term rewriting (cf. e.g. [BN98]), to the best of the author's knowledge there is no directly related work. A form of 2-player rewrite games more closely related to term rewriting, where players take turns applying rewriting rules until a normal form is reached are studied in [Wal02]. In a logical setting, more abstract *structure rewriting games* using rewriting rules that generalise term and graph rewriting are defined in [Kai09].

In addition to these general sources of related work, each chapter of this dissertation contains more specific references pertaining to its subject matter.

1.5. Overview

Aside from basic notation, definitions and general prior results, which are discussed in Chapter 2, this dissertation may roughly be divided into three parts dealing with context-free games of increasing complexity.

- The first part, consisting of Chapters 3 and 4, considers context-free games on strings. Chapter 3 gives basic definitions for these kinds of games and explains in detail most of the methods for upper and lower bound proofs sketched in Section 1.3. It uses these methods in the proof of a complete complexity-theoretic classification of the winning problem for JULIET, with complexities ranging from

PTIME-complete to EXPTIME-complete, where tractable cases require no replay or finite replacement languages.

Chapter 4 then relates games with the standard left-to-right restriction to the more general unconstrained games without this restriction and gives proof that (contrary to a claim in [AMB05]) it is decidable whether any winning strings are lost by introducing the left-to-right restriction to a game. For proof techniques, the chapter also introduces the concept of *dual* games and effects, which is interesting in its own right.

- The second part, consisting of Chapters 5 and 6, deals with the most simple setting for context-free games on nested words where there is no dependence between the input and output of function calls. To that end, Chapter 5 introduces variants of (alternating) *nested word automata* and gives basic closure properties and complexity results for these automata models. In light of the practical motivation, these variants include *simple* (alternating) nested word automata that are very close in expressiveness to XML Schema and whose relevant decision problems have complexities similar to those of finite-state automata.

  Chapter 6 then gives complexity results for context-free games based on these automata, mostly generalising the methods introduced in Chapter 3. With general nested word automata, the winning problem is intractable, its complexity increasing greatly compared to context-free games on strings, whereas simple nested word automata yield games with a winning problem whose complexity equals that for games on strings.

- The third part, consisting of Chapters 7-9, examines context-free games with parameter dependencies. A simple kind of dependencies, where replacements only depend on the parameter subtree being contained in some given validation language, is the subject of Chapter 7. There, it is shown that, even for games without replay, the complexity of the winning problem increases compared to games without validation. In particular, the winning problem is only tractable when the number of function symbols is bounded by a constant and all relevant languages are represented by DTDs.

  Chapters 8 and 9 go into the more complicated field of transducer-based replacement. Of these, Chapter 8 introduces *nested word transducers*, which require some careful design of precise capabilities and restrictions in order to yield desirable closure and complexity properties. As a technical tool, the chapter also introduces *ϵ-transitions* into the nested word automata from Chapter 5 and shows that the resulting model is equivalent to standard nested word automata.

  Chapter 9 then examines the complexity of solving context-free games whose replacement mechanism is based on the (various classes of) transducers introduced in Chapter 8. It is shown that, again, the winning problem becomes much more difficult compared to the games examined in Chapter 6, easily becoming undecidable or of non-elementary complexity if replay is allowed. Tractable cases, on the other

hand, are shown to require some very strong restrictions to both the expressive power of transducers used and the class of strategies allowed to JULIET.

Finally, Chapter 10 closes the dissertation with a brief summary, as well as open questions and possible directions for further research.

As a general note, while the main topic of this dissertation is context-free games and not Active XML, some chapters and sections are written with more of a focus towards the application in Active XML than others. This is due to both historic reasons (as the respective parts were originally created with the application in mind) and the fact that some parts have more bearing on the practical application than others. However, no deeper understanding of AXML than that provided in Section 1.2 is required, and the respective parts can still be read and considered with only the game-theoretic background in mind.

2. Preliminaries

This chapter collects some basic notation, general definitions and prior results. Although most results cited here were already proven in other sources, some proof sketches are also given where specific proof details are relevant to later chapters. This chapter is mainly intended for reference and may be skipped by readers familiar with the subject matter.

In addition to the basics given here, this dissertation uses standard definitions and notations of computational complexity theory as presented e.g. in [AB09], with Turing machines assumed to have a tape that is infinite to the right but has a left end.

2.1. Numbers, sets and formulas

As general notation, let $\mathbb{N}$ denote the set of natural numbers. For any $n \in \mathbb{N}$, let $[n]$ denote the interval of natural numbers up to n, i.e. $[n] = \{1, \ldots, n\}$. By $\mathrm{poly}(n)$, we denote a function of the form n^c for some $c \geqslant 1$. By $\mathrm{Exp}(k, n)$, we denote the k-fold exponential tower function in n, recursively defined by $\mathrm{Exp}(0, n) = n$ and $\mathrm{Exp}(k, n) = 2^{\mathrm{Exp}(k-1,n)}$ for all integers $k > 0$ and $n \geqslant 0$.

For a set S, let $|S|$ denote its cardinality, and let $\mathscr{P}(S)$ denote the powerset of S, i.e. the set of all subsets of S. For sets U, V, let $U \cup V$, $U \cap V$ and $U \backslash V$ denote the union, intersection and set difference of U and V, respectively, and let $U \uplus V$ denote the disjoint union of U and V (i.e. the union $U \cup V$ with the assertion that $U \cap V = \varnothing$).

Normalised sets. For a finite set $\mathcal{D}$ of finite sets, denote by $[\mathcal{D}]_{\min}$ the set constructed by removing from $\mathcal{D}$ all sets that are non-minimal with regard to inclusion, i.e. $[\mathcal{D}]_{\min} = \{D \in \mathcal{D} \mid \nexists D' \in \mathcal{D} : D' \subsetneq D\}$. We call a set $\mathcal{D}$ of sets *normalised* if $\mathcal{D} = [\mathcal{D}]_{\min}$, that is, if it contains no two sets X, Y such that $X \subsetneq Y$. For two sets of sets $\mathcal{D}_1, \mathcal{D}_2$, we write $\mathcal{D}_1 \sqsupseteq \mathcal{D}_2$ (or, equivalently, $\mathcal{D}_2 \sqsubseteq \mathcal{D}_1$) if and only if every $X \in \mathcal{D}_1$ has a subset in $\mathcal{D}_2$.

Lemma 2.1. *Let $\mathcal{D}_1, \mathcal{D}_2$ be two normalised sets of sets. If $\mathcal{D}_1 \sqsupseteq \mathcal{D}_2$ and $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$, then $\mathcal{D}_1 = \mathcal{D}_2$.*

Proof. We prove only $\mathcal{D}_1 \subseteq \mathcal{D}_2$; inclusion in the other direction then follows by symmetry. Let $X_1 \in \mathcal{D}_1$, and let $X_2 \in \mathcal{D}_2$ with $X_2 \subseteq X_1$, as guaranteed by $\mathcal{D}_1 \sqsupseteq \mathcal{D}_2$. Thanks to $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$, there also exists $X_1' \in \mathcal{D}_1$ with $X_1' \subseteq X_2$, and therefore $X_1' \subseteq X_2 \subseteq X_1$. Since both X_1 and X_1' are in $\mathcal{D}_1$, and $\mathcal{D}_1$ is normalised by assumption, this inclusion cannot be proper, and it follows that $X_1' = X_2 = X_1$, and therefore $X_1 = X_2$ and $X_1 \in \mathcal{D}_2$. □

Boolean formulas. For any set P of propositions, $\mathcal{B}^+(P)$ denotes the set of all positive boolean combinations over elements of $P \cup \{\mathtt{true}, \mathtt{false}\}$ using the binary operators

$\wedge$ and $\vee$. Such formulas are evaluated on *truth assignments* $\alpha : P \to \{0,1\}$ mapping propositions to 0 or 1 (interpreted as "true" or "false", respectively) in the standard way. If a formula φ evaluates to 1 under a truth assignment α, we say that α *satisfies* φ (written as $\alpha \models \varphi$). We often identify a truth assignment α with the set $P_\alpha = \{p \in P \mid \alpha(p) = 1\}$ of propositions satisfied by α and write $P_\alpha \models \varphi$ instead of $\alpha \models \varphi$.

2.2. Strings, trees, languages and automata

For an alphabet Σ, we let Σ^* denote the set of all strings over Σ and let ϵ denote the empty string. For $w \in \Sigma^*$, the *length of* w is denoted by $|w|$. For strings $u, w \in \Sigma^*$, u is called a *prefix* (resp. *suffix*) of w if there exists $v \in \Sigma^*$ with $w = uv$ (resp. $w = vu$), and a *proper prefix* (resp. *proper suffix*) if, additionally, $v \neq \epsilon$. A *language* is a set $L \subseteq \Sigma^*$ of strings.

Trees. A *tree domain* is a language $D \subseteq \mathbb{N}^*$ such that, whenever $wi \in D$ for some $w \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then also $w \in D$ and $wj \in D$ for each $j \leqslant i$. Strings in a tree domain are interpreted as node addresses for ordered trees in the standard way: ϵ addresses the root, and if $w \in D$ addresses some node with k children, then $w1, \ldots, wk \in D$ address those children. Formally, for some alphabet Σ, an (unranked) Σ-*labelled tree* is a tuple (D, λ), where D is a tree domain and $\lambda : D \to \Sigma$ is a *labelling function* mapping node addresses to node labels. A tree is called *finite* if its tree domain is a finite language.

Regular expressions. For any finite alphabet Σ, *regular expressions (REs) over* Σ are defined inductively as follows: ϵ, $\varnothing$ and each $a \in \Sigma$ are regular expressions; and if α, β are REs, then so are $(\alpha + \beta)$, $(\alpha \cdot \beta)$ and (α^*). Each RE α describes a language $L(\alpha)$ over Σ, with the semantics of REs being defined as usual (see e.g. [HMU01]). For legibility, the $\cdot$ for concatenation is usually omitted, as are brackets, assuming the standard precedence of operators (* before $\cdot$ before $+$). Some practical applications such as schema languages for XML require REs to be *deterministic*. Intuitively, a RE is deterministic, if each of its positions can be matched uniquely with a symbol of the RE, without lookahead. Formally let, for a RE α, $D(\alpha)$ be the expression, in which the i-th symbol σ of α is replaced by (σ, i), e.g. $D((a + b)^*a) = ((a,1) + (b,2))^*(a,3)$. We call α *deterministic* (or a *DRE* for short), if there do not exist strings w, v, v', symbol σ and numbers i, j such that $w(\sigma, i)v \in L(D(\alpha))$, $w(\sigma, j)v' \in L(D(\alpha))$ and $i \neq j$. A language $L \subseteq \Sigma^*$ is called a *(deterministic) regular language* if $L = L(\alpha)$ for some (deterministic) RE α. It is not hard to see that all finite languages are deterministic, but not every regular language is deterministic (see, e.g., [BW92]).

We use the following shorthand notations in regular expressions.

- $\alpha^?$ for $\alpha + \epsilon$; α^n for the n-fold concatenation of α with itself; and α^{n+} for $\alpha^n\alpha^*$;
- For a set $S = \{a_1, \ldots, a_n\}$ of symbols, S stands as a shorthand for the expression $a_1 + \cdots + a_n$;

- If I is an index set and α_i is a regular expression for every $i \in I$, then $\bigoplus_{i \in I} \alpha_i$ denotes the disjunction of all α_i, $i \in I$.

Finite automata. A *nondeterministic finite automaton (NFA)* $A = (Q, \Sigma, \delta, q_0, F)$ consists of a *state set* Q, a finite alphabet Σ, a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$, an *initial state* $q_0 \in Q$ and a set of *accepting states* $F \subseteq Q$. We usually interpret δ as a function from $Q \times \Sigma$ to $\mathscr{P}(Q)$ and write $q' \in \delta(q, a)$ instead of $(q, a, q') \in \delta$. A *run* of A on a string $w = w_1 \cdots w_n$ from $q \in Q$ to $q' \in Q$ is a sequence $\rho = p_0 p_1 \cdots p_n \in Q^{n+1}$ of states with $p_0 = q$ and $p_n = q'$, where for each $i \in [n]$ it holds that $p_i \in \delta(p_{i-1}, w_i)$. If there exists a run of A on w from q to q', we also write $q \overset{w}{\rightsquigarrow}_A q'$. We say that A *accepts* w if $q_0 \overset{w}{\rightsquigarrow}_A q_f$ for some $q_f \in F$, and we call the corresponding run *accepting*. The language $L(A) \subseteq \Sigma^*$ is defined as the set of all strings accepted by A. An NFA is called a *deterministic finite automaton* (or a DFA) if $|\delta(q, a)| = 1$ for all $p \in Q$ and $a \in \Sigma$. In this case, we simply write $\delta(q, a) = q'$ instead of $\delta(q, a) = \{q'\}$, and $\delta^*(q, w) = q'$ if q' is the unique state, for which $q \overset{w}{\rightsquigarrow}_A q'$. It is well-known (see e.g. [BS86]) that REs, DFAs and NFAs are equivalent with regard to expressiveness; the transformations from NFAs to DFAs and from DFAs to REs generally come with the cost of an exponential size increase. DREs, on the other hand, can be transformed into DFAs in polynomial time [BW98].

Tree grammars. We use *single-type tree grammars* and *local tree grammars* as well-established abstractions of XML Schema and DTDs, respectively (see, e.g., [MLMK05]). For simplicity, we sometimes refer to grammars of these types as *XML Schemas* and *DTDs*, respectively.

A *(regular) tree grammar* is a tuple $T = (\Sigma, \Delta, S, P, \lambda)$, where

- Σ is a finite alphabet of *labels*,
- Δ is a finite alphabet of *types*,
- $S \in \Delta$ is the *root* or *starting type*,
- P is a set of *rules* or *productions* of the form $X \to r_X$ mapping each type $X \in \Delta$ to a deterministic regular expression r_X over Δ, called the *content model* of X, and
- $\lambda : \Delta \to \Sigma$ is a *labelling function* assigning a label from Σ to each type in Δ.

T is *single-type* if for each $X \in \Delta$, the content model r_X contains no competing types, i.e. if r_X contains no two types $Y \neq Z$ with $\lambda(Y) = \lambda(Z)$. T is *local*, if it has exactly one type for every label. A given tree t *matches* the tree grammar T, if it can be obtained by applying the labelling function λ to all labels of some derivation tree of T (interpreted as an extended context-free grammar with nonterminals in Δ and no terminals); in this manner, T induces a language $L(T)$ of unranked Σ-labelled trees in the natural way.

2.3. Formal games and strategies

Arenas and reachability games. An *arena* is a tuple $A = (V, E)$ where $E \subseteq V \times V$ with $V = V_0 \uplus V_1$, i.e. a directed graph whose vertices (also called *positions*) are partitioned into sets V_0 of *0-positions* and V_1 of *1-positions*. Intuitively, each position $p_0 \in V$ induces a two-player game between Players 0 and 1 in the following manner: Initially, a pebble marking the *current position* is placed on position p_0; if the current position p is a 0-position, Player 0 chooses a *successor position* p' with $(p, p') \in E$ to move the pebble to (and thus become the new current position); if p is a 1-position, Player 1 chooses a successor position analogously. Formally, a *play from* p_0 *in* A is either a finite sequence $\Pi = p_0 p_1 \dots p_n$ with $(p_{i-1}, p_i) \in E$ for all $i \in [n]$ such that there is no $p' \in V$ with $(p_n, p') \in E$ (called a *finite play*) or an infinite sequence $\Pi = p_0 p_1 \dots$ with $(p_{i-1}, p_i) \in E$ for each $i \in \mathbb{N}$ (called an *infinite play*). A *prefix* of a play is defined in the obvious manner. A *reachability game* $G = (A, W)$ consists of an arena $A = (V, E)$ and a set $W \subseteq V$ of *0-winning positions*. A play Π from $p_0 \in V$ in A is called *0-winning (in G)* if it contains at least one position from W, otherwise it is called *1-winning (in G)*. The intuition behind this is that it is the goal of Player 0 to reach a 0-winning position in a play, while Player 1 aims to prevent that from happening.

Strategies. A *strategy* for Player i ($i \in \{0, 1\}$) in a reachability game $G = (A, W)$ is a function $\sigma_i : V^* V_i \to V$ mapping prefixes of plays ending in an i-position to positions such that $(p, \sigma_i(\Pi)) \in E$ for each prefix $\Pi = p_0 \dots p$ of a play from p_0 with $p \in V_i$. A strategy σ_i is *memoryless* if it only depends on the last position in a prefix, i.e. if $\sigma_i(\Pi p) = \sigma_i(\Pi' p)$ for all $\Pi, \Pi' \in V^*$ and $p \in V$; in this case, we simply assume σ_i to be a function from V_i to V. For each initial position $p_0 \in V$, a pair of strategies σ_0, σ_1 for Players 0 and 1 uniquely determines a play from $p_0 \in A$ in the natural way (by applying σ_0 in 0-positions and σ_1 in 1-positions), denoted by $\Pi(p_0, \sigma_0, \sigma_1)$. A strategy σ_0 for Player 0 (resp. σ_1 for Player 1) is a *winning strategy in G from* p_0 for Player 0 (resp. Player 1) if the play $\Pi(p_0, \sigma_0, \sigma_1)$ is 0-winning (resp. 1-winning) in G for every strategy σ_1 for Player 1 (resp. σ_0 for player 0). The following classical result (see, e.g., [GTW02]) will be of use throughout this dissertation, as it applies to all games examined here.

Proposition 2.2. *Let G be a reachability game and p_0 a position of G. Then either Player 0 or Player 1 has a winning strategy in G from p_0, and that winning strategy is memoryless.*

Example 2.3 (Evaluation of positive boolean formulas)**.** *The problem of evaluating a formula $\varphi \in \mathcal{B}^+(P)$ on some truth assignment $\alpha : P \to \{0, 1\}$ can be viewed as a reachability game $G_{\varphi,\alpha} = (A_\varphi, W_\alpha)$ in the following way. The arena $A_\varphi = (V_\varphi, E_\varphi)$ has as positions all subformulas of φ. 0-positions in A_φ are all formulas in V_φ of the form $\psi_1 \vee \psi_2$ or $p \in P$, and 1-positions are all formulas of the form $\psi_1 \wedge \psi_2$. Each position corresponding to a composite formula $\psi_1 \vee \psi_1$ (or $\psi_1 \wedge \psi_2$) has as its two successors the positions ψ_1 and ψ_2, while single propositions have no successors, i.e. $E_\varphi = \{(\psi, \psi_1), (\psi, \psi_2) \mid \psi = \psi_1 \vee \psi_2 \text{ or } \psi = \psi_1 \wedge \psi_2\}$. The set W_α of winning positions for Player 0 is just the set of all propositions satisfied by α, i.e. $W_\alpha = \{p \in P \mid \alpha(p) = 1\}$.*

Since A_φ is a binary tree, it is clear that all plays on A_φ are finite and terminate at positions labelled with single propositions from P. An easy induction argument shows that Player 0 has a winning strategy in $G_{\varphi,\alpha}$ from position φ if and only if φ evaluates to 1 under the assignment α.

2.4. Alternating finite automata

An *alternating finite automaton (AFA)* is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ consisting of a *state set* Q, a finite alphabet Σ, a *transition function* $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$, an *initial state* $q_0 \in Q$ and a set of *accepting states* $F \subseteq Q$. As before, $\mathcal{B}^+(Q)$ denotes the set of all positive boolean combinations over elements of Q. Similar to NFAs, the semantics of AFAs is defined via runs. A *run* $\rho = (D, \lambda)$ *of* A *on* $w = w_1 \cdots w_n \in \Sigma^*$ *from* q is a tree of depth n with tree domain D and labelling function $\lambda : D \to Q$ such that $\lambda(\epsilon) = q$ and for every node address x of length i with ℓ children, it holds that $\{\lambda(x \cdot 1), \ldots, \lambda(x \cdot \ell)\} \models \delta(\lambda(x), w_{i+1})$. A run of A on w is *minimal* if no proper subtree of it is a run of A on w; a run is *accepting* if all of its leaves are labelled with accepting states, and an AFA A *accepts a string* $w \in \Sigma^*$ if there is an accepting run of A on w from q_0. Intuitively, when reading a symbol $a \in \Sigma$ in state $q \in Q$, an AFA guesses nondeterministically a set $P \subseteq Q$ of states with $P \models \delta(q, a)$, branches universally into all states in P and accepts a string if all the branches end up in accepting states.

AFAs are efficiently closed under all boolean operations [FJY90]; the following proof sketch for closure under complement will be of use in Chapter 4.

Lemma 2.4. *Let $A = (Q, \Sigma, \delta, q_0, F)$ be an AFA. There is an AFA $\hat{A} = (Q, \Sigma, \hat{\delta}, q_0, \hat{F})$ with $L(\hat{A}) = \Sigma^* \backslash L(A)$ which can be computed from A in polynomial time.*

Proof. For a formula $\varphi \in \mathcal{B}^+(Q)$, we define its *dual formula* $\hat{\varphi}$ inductively by $\hat{q} = q$ for each $q \in Q$, $\widehat{\varphi \wedge \psi} = \hat{\varphi} \vee \hat{\psi}$ and $\widehat{\varphi \vee \psi} = \hat{\varphi} \wedge \hat{\psi}$. By an easy induction argument, it is possible to show that an assignment $\alpha : Q \to \{0, 1\}$ satisfies $\hat{\varphi}$ if and only if its *complementary assignment* $\overline{\alpha}$ defined by $\overline{\alpha}(q) = 1 - \alpha(q)$ for each $q \in Q$ does *not* satisfy φ.

We define $\hat{\delta}(q, a) = \widehat{\delta(q, a)}$ for each $q \in Q$ and $a \in \Sigma$, and $\hat{F} = Q \backslash F$. An easy induction shows that the automaton $\hat{A}$ thus defined has a run on some string $w \in \Sigma^*$ whose leaves are all in some set $X \subseteq Q$ if and only if A does *not* have a run whose leaves are all in $Q \backslash X$. Setting $X = Q \backslash F$ then yields the correctness of this construction. The complementation and dualisations needed for the construction are obviously feasible in polynomial time. □

It is well-known that AFA can be transformed into NFA with an exponential blow-up in state size; the construction given here is somewhat non-standard, but will also be of use in Chapter 4.

Lemma 2.5. *Let $A = (Q, \Sigma, \delta, q_0, F)$ be an AFA. There is a NFA A' of size exponential in $|Q|$ with $L(A') = L(A)$ that can be computed from A in exponential time.*

Proof. Let $A' = (\mathscr{P}(Q), \Sigma, \delta', \{q_0\}, \mathscr{P}(F))$, where, for each $X = \{q_1, \ldots, q_n\} \in \mathscr{P}(Q)$ and $a \in \Sigma$,

$$\delta'(X, a) = \left[\{X_1 \cup \cdots \cup X_n \mid X_i \models \delta(q_i, a), i \in [n]\}\right]_{\min}.$$

An easy induction argument shows that, on any string $w \in \Sigma^*$, A' has an accepting run on w if and only if A has a minimal accepting run on w, which in turn is the case if any only if A has *any* accepting run on w. Each $\delta'(X, a)$ can be computed from δ by an iteration over all subsets of Q, which is feasible in exponential time as claimed. □

The complexity of standard decision problems for AFAs are as follows [HK11].

Theorem 2.6. *(a) The membership problem for AFA is* PTIME*-complete with respect to logspace-reductions.*

(b) The emptiness problem for AFA is PSPACE*-complete.*

In parts of this work, the upper bounds in Theorem 2.6 will prove to be somewhat problematic, as we will sometimes have to deal with AFAs whose transition function is of exponential size in the size of their state sets. Theorem 2.6 would simply yield an exponential time (resp. space) upper bound; however, the following statements show that, in this case, the membership and emptiness problem can still be solved using space at most polynomial in the state set size. The proof sketches for these results will be of further use in Chapter 5.

Proposition 2.7. *There is an algorithm deciding, for each AFA $A = (Q, \Sigma, \delta, q_0, F)$ and $w \in \Sigma^*$, whether $w \in L(A)$ holds, with a runtime in $\mathcal{O}(|w| \cdot |Q| \cdot poly(|\delta|))$ and using space $\mathcal{O}(\log(|w|) + |Q| + \log(|\delta|))$.*

Proof. The basic idea behind this algorithm is to read w *backwards*, while storing a set S of states from which there is an accepting run on the suffix of w that has been read so far. The set S is initialised with $S = F$, and each time some symbol a of w is read, the algorithm tests, for all states $q \in Q$, whether $S \models \delta(q, a)$ and, if so, adds q to a new set S'. Once all states have been tested, the algorithm sets $S = S'$, resets S' to the empty set, and continues with the next symbol of w. After w has been completely read, the algorithm accepts if and only if $q_0 \in S$ holds. The correctness of this algorithm is easily proven by a simple induction argument showing that, each time the set S is updated, it holds that $q \in S$ if and only if there is an accepting run of A from q on the suffix of w that has been read so far.

The time and space upper bounds follow by a direct analysis of this algorithm, since it simply tests, for each symbol a of the input word and each state q of A, whether a given truth assignment satisfies the formula $\delta(q, a)$. The polynomial time and logarithmic space complexity in δ of this test follows from the fact that evaluating a boolean formula is in NC^1 [BCGR92] and therefore decidable in logarithmic space. □

Proposition 2.8. *There is an algorithm deciding, for each AFA $A = (Q, \Sigma, \delta, q_0, F)$, whether $L(A) \neq \varnothing$ holds, using space $\mathcal{O}(\log(|\Sigma|) + |Q| + \log(|\delta|))$.*

Proof. This algorithm follows a similar idea of "backwards iteration" to that of Proposition 2.7 to determine the set S of all states $q \in S$ such that there exists some string w for which A has an accepting run starting at q; once this set S has been computed, checking for non-emptiness of $L(A)$ is simply a matter of testing whether $q_0 \in S$ holds.

The algorithm initialises the set S to the set F of accepting states. Then, it performs the following iteration updating S until a fixpoint is reached: For each $q \in Q \backslash S$ and $a \in \Sigma$, test whether $S \models \delta(q, a)$; if this is the case, add q to S. Clearly, a state q is added to S in this iteration if there exists some symbol a such that there is a run of A starting at q and ending in states inside S; by induction, it then follows that, once a fixpoint is reached, S is indeed the set of states from which there is an accepting run of A on some string.

Storing the value of S after each iteration requires space $\mathcal{O}(|Q|)$; furthermore, since each iteration consists of at most $|\Sigma| \cdot |Q|$ tests whether a boolean formula of size $\mathcal{O}(|\delta|)$ is satisfied by a given assignment, each iteration can be performed using space $\mathcal{O}(\log(|\Sigma|) + \log(|Q|) + \log(|\delta|))$, with the last term again using boolean formula evaluation in NC^1 [BCGR92]. This proves the desired space bound. □

2.5. Tilings

Several of the lower bound proofs in this work use algorithmic problems involving tilings, as most of the complexity classes appearing here have characteristic complete tiling problems.

A *tiling of height m and width ℓ* (or *$m \times \ell$-tiling*) over a *tile set U* with *vertical constraints* $V \subseteq U \times U$, *horizontal constraints* $H \subseteq U \times U$, *initial tile* $u_i \in U$ and *final tile* $u_f \in U$ is a mapping $t : [m] \times [\ell] \to U$ such that

- $(t(i, j), t(i + 1, j)) \in V$ for each $i \in [m - 1]$ and $j \in [\ell]$,
- $(t(i, j), t(i, j + 1)) \in H$ for each $i \in [m]$ and $j \in [\ell - 1]$,
- $t(1, 1) = u_i$, and
- $t(m, \ell) = u_f$.

Intuitively, a tiling arranges $m \cdot \ell$ tiles from U in m rows and ℓ columns such that the first row starts with the initial tile, the last row ends with the final tile and horizontally or vertically adjacent tiles match (as per the horizontal and vertical constraints). Accordingly, for a tiling t of width ℓ and height m, we refer to the string $t(i, 1)t(i, 2) \cdots t(i, \ell) \in U^*$ as the *i-th row* and to $t(1, j)t(2, j) \cdots t(m, j) \in U^*$ as the *j-th column* of t.

We will often encode an $m \times \ell$-tiling t as a string w_t of the form $(U^\ell \#)^m$ over the alphabet $U \uplus \{\#\}$ for a special *line divider symbol* $\# \notin U$, with the interpretation that $t(i, j)$ is the $(i - 1) \cdot (\ell + 1) + m$-th symbol of w_t.

The k-Exponential Tiling problem (simply called Tiling for $k = 0$ and Exponential Tiling for $k = 1$) asks, given a set U of tiles, horizontal and vertical constraints $H, V \subseteq U \times U$, initial and final tiles $u_i, u_f \in U$ and numbers m, ℓ (given in unary),

whether there is a $\mathrm{Exp}(k,m) \times \mathrm{Exp}(k,\ell)$-tiling for the given tile set, constraints, initial and final tile. For the k-Exponential Corridor Tiling problem (respectively Corridor Tiling for $k = 0$ and Exponential Corridor Tiling for $k = 1$), no width parameter is given and the question is whether there is a $\mathrm{Exp}(k,m) \times \ell$-tiling with the given parameters for *any* ℓ.

The two-player variants of these problems consider a *tiling game* where Player 1 and Player 2 place tiles from U in an alternating fashion (starting with u_i for Player 1); Player 1 wins the game if Player 2 places a tile violating some vertical or horizontal constraint, or some player places a tile completing a valid tiling the dimensions specified in the input instance, while Player 2 wins if Player 1 places an invalid tile or if the game does not terminate with a valid tiling of the given dimensions[1]. The two-player variant of k-Exponential Tiling (resp. k-Exponential Corridor Tiling), called 2-Player k-Exponential Tiling (resp. 2-Player k-Exponential Corridor Tiling) asks whether Player 1 has a winning strategy in the tiling game on a given instance.

Since Turing machine computations can be encoded as tilings in a very natural way, the following results are quite straightforward and well-known [Chl86; EB97].

Theorem 2.9. *The following statements hold for all* $k \geqslant 1$*:*

(a) Tiling *is* NP*-complete.*

(b) Corridor Tiling *and* 2-Player Tiling *are* PSPACE*-complete*

(c) 2-Player Corridor Tiling *is* EXPTIME*-complete.*

(d) k-Exponential Tiling *is complete for* k-NEXPTIME*.*

(e) k-Exponential Corridor Tiling *is complete for* k-EXPSPACE*.*

(f) 2-Player k-Exponential Tiling *is complete for* k-EXPSPACE*.*

(g) 2-Player k-Exponential Corridor Tiling *is complete for* $(k+1)$-EXPTIME*.*

[1] Note that this victory condition for Player 2 includes infinite plays in the case of corridor tiling games

Part I.

Context-free games on strings

3. Winning problems for games on strings

Following the general intuition from Section 1.1, we now examine context-free games on strings as the most simple class of structures. As was already mentioned in Chapter 1, the central question we are interested in for context-free games is the decidability and complexity of the winning problem for JULIET: Given a game G and an initial string w, does JULIET have a winning strategy on w in G? This chapter gives an extensive classification of the complexity of this problem for various classes of context-free games on strings in Section 3.2; for games with finite replacement languages, we cite complexity results from the seminal paper on context-free games [MSS06], and for games with arbitrary regular replacement, we prove upper and lower bounds in Sections 3.4 and 3.3, respectively. For reference, the central complexity results of this chapter are summarised in Table 3.1.

In giving detailed upper and lower bound proofs, this chapter also serves a secondary purpose besides presenting the relevant complexity results. While context-free games on strings are somewhat simpler in nature and easier to handle than games on more complex structures, many of the tools and techniques used here carry over to more general settings. For this reason, we use context-free games on strings as an exemplary setting to examine in detail the proof techniques sketched intuitively in Section 1.3 that will be used extensively throughout this dissertation.

3.1. Definitions

In this section, we define context-free games on strings. Following the intuition from Section 1.1, a context-free game (or cfG for short) is played on a string containing *function symbols*; the first player, JULIET, decides which function symbols should be *called* and replaced, and the second player, ROMEO, decides what a called function symbol should be replaced with, in accordance with some replacement formalism. Throughout this chapter (and the rest of this dissertation), it will generally be assumed, unless otherwise noted, that JULIET can only call function symbols in a *left-to-right* order, i.e. one can imagine JULIET moving a focus through the input string from left to right and deciding, for each function symbol she encounters, whether that function symbol should be called (a *Call move*) or not (a *Read move*). The motivation for this restriction will become clear later, when we examine *unconstrained* cfGs without this restriction and see that unconstrained games have an undecidable winning problem.

Definition 3.1. A *context-free game on strings (cfG)* $G = (\Sigma, \Gamma, R, T)$ consists of a finite alphabet Σ, a set $\Gamma \subseteq \Sigma$ of *function symbols*, a *(replacement) rule set* $R \subseteq \Gamma \times \Sigma^*$

	No replay	Bounded	Unbounded
Regular replacement	PTIME	PSPACE	EXPTIME
Finite replacement	PTIME	PTIME	EXPTIME

Table 3.1.: Summary of main complexity results for cfGs on strings. All results are completeness results.

and a *target language* $T \subseteq \Sigma^*$. We only consider the case where T and, for each symbol $a \in \Gamma$, the set $R_a \stackrel{\text{def}}{=} \{u \mid (a,u) \in R\}$ is a non-empty regular language.

Towards a formal definition of cfGs as reachability games, a *configuration* is a tuple $\kappa = (p,u,v) \in \{\mathrm{J},\mathrm{R}\} \times \Sigma^* \times \Sigma^*$ where p is the player to move, $uv \in \Sigma^*$ is the *current string*, and, if $v \neq \epsilon$, the first symbol of v is *the current symbol*. We call $|u|+1$ the *current position* of κ. A configuration (p,u,v) with $v \neq \epsilon$ is called *playable*, otherwise it is *final*.

The configuration $\kappa' = (p',u',v')$ is a *successor configuration* of the playable configuration $\kappa = (p,u,v)$ (Notation: $\kappa \to \kappa'$) if one of the following holds:

(1) $p' = p = \mathrm{J}$, $u' = ua$, and $av' = v$ for some $a \in \Sigma$ (Juliet plays *Read*);

(2) $p = \mathrm{J}$, $p' = \mathrm{R}$, $u = u'$, $v = v' = az$ for $z \in \Sigma^*$, $a \in \Gamma$, (Juliet plays *Call*);

(3) $p = \mathrm{R}$, $p' = \mathrm{J}$, $u' = u$, $v = az$ for $z \in \Sigma^*$, $a \in \Gamma$, and $v' = y'z$ for some $y' \in R_a$ (Romeo plays y').

With these definitions, each context-free game G induces a reachability game (as defined in Section 2.3 whose arena consists of all configurations as defined above, with 0-positions (respectively 1-positions) corresponding to positions of Juliet (resp. Romeo) and an edge relation given by successor configurations. The set of winning positions for Juliet is the set of all configurations of the form (J,v,ϵ) with $v \in T$. The definitions of plays and (winning) strategies then carry over from Section 2.3; for the sake of easier presentation, we slightly abuse notation to say that strategies σ for Juliet map configurations κ to moves $\sigma(\kappa) \in \{\mathit{Read}, \mathit{Call}\}$ and strategies τ for Romeo map configurations κ to strings $\tau(\kappa) \in \Sigma^*$.[1] Strategies for Juliet will usually be denoted by $\sigma, \sigma', \sigma_1, \ldots$ and strategies for Romeo by $\tau, \tau', \tau_1, \ldots$.

For a cfG G, strategies σ, τ for Juliet and Romeo, and a string $w \in \Sigma^*$, the *play* $\Pi(w,\sigma,\tau)$ *on* w *in* G is defined as the play according to strategies σ and τ in the reachability game induced by G starting from position (J,ϵ,w). If $\Pi(w,\sigma,\tau)$ is finite, we denote the word represented by its final configuration by $\mathrm{word}_G(w,\sigma,\tau)$.

A strategy σ for Juliet is *finite* on string w if the play $\Pi(w,\sigma,\tau)$ is finite for every strategy τ of Romeo. We only consider finite strategies for Juliet, since Juliet's winning condition requires reaching a final position, and thus a finite play.

We denote the set of all finite strategies for Juliet in the game G by $\mathrm{STRAT}_{\mathrm{J}}(G)$, and the set of all strategies for Romeo by $\mathrm{STRAT}_{\mathrm{R}}(G)$.

[1] Note that this notation assumes strategies to be memoryless; this assumption can be made without loss of generality by Proposition 2.2.

The *Call depth* of a finite play Π is the maximum nesting depth of *Call* moves in Π. That is, the *Call* depth of a play is zero, if no *Call* is played at all, and one, if no *Call* is played inside a string yielded by a replacement move. More formally, we associate with each symbol in the current string of each position a *Call* depth; symbols from the input string in the initial position have depth 0, and each time JULIET plays *Call* of a symbol of depth i, all symbols in the string returned by ROMEO have depth $i + 1$. The *Call* depth of a finite play Π is then the maximum depth of any symbol in its final string.

For a strategy σ of JULIET and a string $w \in \Sigma^*$, the *Call depth* $\mathrm{Depth}^G(\sigma, w)$ of σ on w is the maximum *Call* depth in any play $\Pi(w, \sigma, \tau)$. A strategy σ has *k-bounded Call depth* if $\mathrm{Depth}^G(\sigma, w) \leqslant k$ for all $w \in \Sigma^*$. We denote by $\mathrm{STRAT}^k_{\mathrm{J}}(G)$ the set of all strategies with k-bounded *Call* depth for JULIET on G. As a more intuitive formulation, we use the concept of *replay*, which is defined as *Call* depth (if it exists) minus one: Strategies for JULIET of *Call* depth one are called *replay-free*, and strategies of k-bounded *Call* depth, for any k, have *bounded replay*.

By $\mathrm{JWin}(G)$ we denote the set of all words for which JULIET has a winning strategy in $\mathrm{STRAT}_{\mathrm{J}}(G)$ (likewise for $\mathrm{JWin}^k(G)$ and $\mathrm{STRAT}^k_{\mathrm{J}}(G)$).

Throughout this dissertation, we study the following algorithmic problem $\mathrm{JWIN}(\mathcal{G})$, for various classes $\mathcal{G}$ of context-free games.

$\mathrm{JWIN}(\mathcal{G})$	
Given:	A context-free game $G \in \mathcal{G}$ and a string w.
Question:	Is $w \in \mathrm{JWin}(G)$?

For the purposes of this chapter, a class $\mathcal{G}$ of context-free games on strings in $\mathrm{JWIN}(\mathcal{G})$ comes with three parameters:

- the representation of the target language T,
- the representation of the replacement languages R_a, and
- to which extent replay is restricted.

It is a fair assumption that the representations of the target language and the replacement languages are of the same kind, but we will usually discuss the impact of the replacement language representations separately. Throughout this dissertation, we generally assume the target language to be represented by a deterministic automaton (i.e. a DFA in this chapter); this is primarily due to the complexity of $\mathrm{JWIN}(\mathcal{G})$ becoming quite high, even for classes $\mathcal{G}$ of games with deterministic target automata.[2]

In this chapter, we consider both regular and finite replacement languages and show that the precise representation of regular replacement languages is actually irrelevant,

[2] Note, however, that upper bounds on the complexity of games with deterministic target automata directly yield trivial upper bounds for games with nondeterministic target automata that determinise in exponential time (like all automata considered in this dissertation); the results of [MSS06] concerning games with NFA-represented target languages suggest that most of these trivial upper bounds may actually be tight.

as we prove lower bounds for DREs and upper bounds for NFAs. Some care has to be taken in the case of finite replacement languages, however, as DFA-represented finite languages can be exponentially more succinct than finite languages whose strings are explicitly enumerated as part of the input. Here, and in following chapters, we generally assume that finite replacement languages are explicitly enumerated, but some of the upper bounds in later chapters also hold for finite languages represented by automata.

In each setting, we consider the cases of unrestricted replay, bounded replay (*Call* depth k, for some k), and no replay (*Call* depth 1). [3]

Since in most cases, the class $\mathcal{G}$ of games will be clear from the context, we usually just write JWin instead of JWin($\mathcal{G}$) to improve legibility.

We usually denote the automata representing the target and replacement languages by $A(T) = (Q, \Sigma, \delta, q_0, F)$ and $A(R_a) = (Q_a, \Sigma, \delta_a, q_{0,a}, F_a)$ (for $a \in \Gamma$), respectively. We denote by $|R|$ the combined size of all $A(R_a)$, $a \in \Gamma$, and by $|G|$ the size of (a sensible representation of) G, i.e. $|G| = |\Sigma| + |R| + |A(T)|$.

As already mentioned, the semantics of context-free games assumes a left-to-right restriction on the order of function calls. One way of lifting this restriction is by allowing *left steps* in the definition of cfGs.

Definition 3.2. An *unconstrained context-free game on strings* $G = (\Sigma, \Gamma, R, T)$ is defined like a cfG on strings in Definition 3.1, with the exception that its semantics allows for an additional *left step (LS)* move.

To this end, a configuration $\kappa' = (p', u', v')$ is a *successor configuration* of configuration $\kappa = (p, u, v)$ if one of the following holds:

(1) $p' = p =$ J, $u' = ua$, and $av' = v$ for some $a \in \Sigma$ (Juliet plays *Read*);

(2) $p =$ J, $p' =$ R, $u = u'$, $v = v' = az$ for $z \in \Sigma^*$, $a \in \Gamma$, (Juliet plays *Call*);

(3) $p =$ R, $p' =$ J, $u' = u$, $v = az$ for $z \in \Sigma^*$, $a \in \Gamma$, and $v' = y'z$ for some $y' \in R_a$ (Romeo plays y');

(4) $p =$ J, $p' =$ J, $v = \epsilon$, $u' = \epsilon$, $v' = u$ (Juliet plays LS).

All other definitions (plays, strategies, etc.) carry over from Definition 3.1. We denote by JWin$^{\leftarrow}(G)$ the set of all strings on which Juliet has a winning strategy in the unconstrained game G.

Note that a left step move can only be played once Juliet's focus has reached the end of the current string and always puts the focus at the very start of the string; this, however, is not truly a restriction, because Juliet can basically jump from any position inside the string to any other position to its left by just playing *Read* moves before and after the left step to reach her desired position. In this way, left steps make it possible to simulate the "intuitive" form of context-free games where Juliet simply chooses a function symbol at an arbitrary position in the current string as each of her moves.

[3] We note that replay depth is formally not an actual game parameter, but the algorithmic problem can be restricted to strategies of Juliet of the stated kind. Viewing replay as a game parameter in this way serves to simplify the presentation of results in this and subsequent chapters.

3.2. Complexity results

The first systematic investigation into context-free games on strings was conducted in [MSS06]. This section gives an overview over the relevant results of that paper, as well as some new results proven as joint work with Thomas Schwentick [SS15; SS16].

As already mentioned, the main reason for generally defining the semantics of context-free games to include the left-to-right restriction in this dissertation was the following result:

Theorem 3.3 ([MSS06], Theorem 4.2)**.** *For the class of unconstrained cfGs with finite replacement languages,* JWin *is undecidable.*

Due to this result, the left-to-right restriction has become the *de facto* standard setting investigated in theory [AMB05; SS15] and practice [MAABN05].

Most of the work in [MSS06] focussed on games with finite replacement languages and restrictions thereof. While, technically, replay of strategies was not a parameter in [MSS06], it is easy to see that (for finite replacement languages), we can simulate a cfG with *Call* depth bounded by some constant d through a d-bounded left-to-right cfG in the sense of [MSS06], i.e. a cfG whose rule set, when interpreted as a context-free grammar, only allows for derivation trees of depth at most d. This simulation is done by annotating each alphabet symbol with a "generation number" ranging from 0 to d, with symbols in the input string having generation number 0 and function symbols of generation number k always being replaced by strings of symbols of generation number $d + 1$. Conversely, d-bounded left-to-right cfGs admit only strategies of *Call* depth bounded by d.

Theorem 3.4 ([MSS06],Theorems 4.6, 5.8)**.** *For the class of cfGs with finite replacement languages, the following complexity results hold for the* JWin *problem:*

(a) with unbounded replay, it is EXPTIME*-complete, and*

(b) with bounded or without replay, it is PTIME*-complete (under logspace reductions).*

While the special case of cfGs with finite replacement languages is of some interest (and will help with reducing complexity in some places), our main interest lies in the case where replacement languages are arbitrary regular languages, as motivated by the XML-based application.[4]

Theorem 3.5. *For the class of cfGs with regular replacement languages, the following complexity results hold for the* JWin *problem:*

(a) with unbounded replay, it is EXPTIME*-complete,*

(b) with bounded replay, it is PSPACE*-complete, and*

[4] Technically, this means that we should actually focus on *deterministic* regular languages, but as their main characteristic is a polynomial-time transformation from RE to DFA and results presented in later chapters show that the representation of replacement language usually does not impact complexity, we will mostly stick with arbitrary regular languages.

(c) without replay, it is PTIME*-complete (under logspace reductions).*

All lower bounds hold already if target and replacement languages are given as deterministic regular expressions.

The lower bounds in Theorem 3.5 will be proven in Section 3.3 (Propositions 3.7, 3.8 and 3.6), while the upper bound proofs are the subject of Section 3.4 (Theorem 3.9). Since this chapter also serves to explain and motivate techniques used several times throughout this dissertation (and already sketched in Section 1.3), discussion of proofs and proof ideas in this chapter will be somewhat more extensive than in later chapters.

3.3. Techniques for lower bound proofs

In this section, we prove the lower bounds in Theorem 3.5. We also use this opportunity to examine in-depth some of the standard techniques used in lower bounds on the complexity of JWin for various classes of games, as already sketched in Section 1.3.

Wherever possible, lower bound reductions are from problems containing alternation in some form. Transforming input instances for these problems into input instances for JWin is usually done in a quite straightforward fashion – we set up the game in such a way that Juliet makes choices for existential quantification or disjunction, Romeo makes choices for universal quantification or conjunction, and the target language verifies whether the transformed input instance, with the choices made by Juliet and Romeo, is a positive one.

In the following Proposition 3.6, for instance, the reduction is from the monotonic circuit value problem (given a Boolean circuit without negations, does this circuit evaluate to 1?). The basic idea in this reduction is that Juliet and Romeo determine in an alternating fashion a path from the output gate to some input gate, with Juliet choosing the in-gate of OR-gates and Romeo choosing the in-gate of AND-gates. The target language then has to verify whether the path chosen by Juliet and Romeo ends in an input gate with input 1.

However, the proof of Proposition 3.6 also exemplifies that, while the basic idea behind such reductions is usually quite straightforward, its actual execution may get somewhat more complicated. The main reason for this is that, in the course of a play, players may attempt to "cheat" by making choices that do not correspond to correct choices in the original problem's alternation but would still allow the cheating player to win in a naïve construction of the reduction.[5] Large parts of a reduction will therefore usually be concerned with *cheating prevention*, modifying the target and replacement languages from the naïve construction in such a way that any player who attempts to cheat will, by this attempt, end up in a position where he or she irrevocably loses the game.

The most basic example of this can be seen in the proof of Proposition 3.6: For Romeo to make a choice, Juliet first has to pass control to him by playing *Call* on some symbol

[5] Technically, since "cheating" implies that players actually break the rules of the game instead of just going against the "unspoken rules" intended in the construction, a term like "rules lawyering" might be more appropriate, but we will stick with "cheating" as it is catchier and more concise.

where ROMEO is to make a universal choice (here: the follow-up gate for an AND gate). In situations like these, JULIET could imaginably try to cheat by simply refusing to pass control and playing *Read* instead. To deal with this sort of cheating attempts, we usually modify the target language in such a way that it recognises uncalled function symbols in places where a "mandatory" *Call* should have been played and rejects any string containing such a function symbol. Throughout this dissertation, we will see some much more intricate examples of this technique and of cheating prevention in general; however, the basic pattern of first constructing a naïve reduction and then extending the construction to prevent cheating is the same in most lower bound proofs.

Proposition 3.6. *For the class of replay-free cfGs with replacement and target languages given as deterministic regular expressions,* JWIN *is* PTIME*-hard (under logspace reductions).*

Proof. The proof is by reduction from the monotone circuit value problem: Given a boolean circuit C with binary AND-gates, OR-gates and two constant (input) gates with value 0 and 1 does C evaluate to 1? This problem is known to be PTIME-complete [Gol77].

The basic idea behind this reduction is to use a context-free game to decide whether an input valued 1 can be reached from the output gate in an alternating fashion; given some gate g (starting at the output gate), if g is an AND gate, ROMEO chooses some in-gate of g, and if g is an OR gate, JULIET chooses an in-gate of g; play then proceeds on the chosen in-gate and ends as soon as the 0-gate or the 1-gate is reached, and JULIET wins if the latter happens.

Our main task in this proof is to formalise this intuition by a context-free game G and input string w such that JULIET has a replay-free winning strategy on w in G if and only if C evaluates to 1.

We assume without loss of generality that the gates of C are inverse topologically sorted and numbered by $1, \ldots, m, m+1$, i.e. the output gate is numbered with 1, the two inputs of each gate numbered i have numbers strictly greater than i, gate m is the 1-gate and gate $m+1$ is the 0-gate.

We represent each non-source gate i with $i \in [m-1]$ by a string $w(i)$ as follows:

$$w(i) = \begin{cases} g_i x_j x_k, & \text{if } i \text{ is an OR gate with in-gates } j \text{ and } k \\ g_i y_i, & \text{if } i \text{ is an AND gate.} \end{cases}$$

The input string for the game G is $w = \hat{1} \cdot w(1) \cdot w(2) \cdots w(m-1) \cdot m$.

Play should proceed on the input string as follows: The gate i such that $\hat{i}$ is the right-most symbol from $\{\hat{j} \mid j \in [m]\}$ is supposed to be the current *active* gate, i.e. the gate next to be dealt with by JULIET and ROMEO. If gate i is active JULIET is supposed to first play *Call* on g_i, replacing it by i and confirming that JULIET has indeed picked the substring $w(i)$ corresponding to the active gate. If i is an OR gate, JULIET next plays *Call* on one of the two symbols $x_j x_k$ immediately to its right, replacing x_j by $\hat{j}$ (or x_k by $\hat{k}$) and thus activating one of the in-gates of gate i. If, on the other hand, i is an AND gate, JULIET has to play *Call* on y_i next, allowing ROMEO to activate, as a replacement,

one of the two in-gates of i in a similar fashion. If, at any point during the game, gate m becomes active, Juliet wins the game.

These considerations yield the game G with the alphabet

$$\Sigma = \Gamma \cup [m+1] \cup \{\hat{j} \mid j \in [m+1]\}$$

and function symbols

$$\Gamma = \{g_j, x_j, y_j \mid j \in [m+1]\}.$$

The replacement languages for each $i \in [m+1]$ are as follows:

- $R_{g_i} = \{i\}$,
- $R_{x_i} = \{\hat{i}\}$, and
- $R_{y_i} = \{\hat{j}, \hat{k} \mid j, k \text{ are in-gates of gate } i\}$.

The target language is given by the DRE R^*, where

$$R = \bigoplus_{i \in [m]} \big(\hat{i}\Gamma^* i(\Gamma + \epsilon))\big).$$

Clearly, G can be constructed from C in logarithmic space.

To prove correctness of the reduction it can be shown, by induction on the depth of gates, that Juliet has a winning sub-strategy on a gate i that has just become active if and only if gate i evaluates to 1 in C. Clearly, applying this equivalence to the output gate yields the desired result. For the input gates the equivalence is obvious. The inductive step can be shown with the help of the following observations.

- Once gate i becomes active, Juliet is forced to call g_i next, as this is the only way to produce the i symbol required by the target language; she may not call any other function symbols in between, or she loses the game immediately.
- Once Juliet has called g_i, she needs to call either the y_i or one of the $x_j x_k$ immediately to its right, as this is the only way to obtain another symbol from $\{\hat{j} \mid j \in [m]\}$ at distance at most two to the right of i. If Juliet fails to do so, she loses the game.
- After the activation of gate i, Juliet can win the game in this fashion if either i is an OR gate and Juliet can win the game by activating one of its in-gates, or if i is an AND gate and Juliet has a winning strategy for the case that either of its in-gates becomes active.
- If Juliet adheres to this order of calling function symbols, once gate m becomes active, she immediately wins the game.

□

As we saw in the previous proof, our standard method for letting JULIET make existential choices in reductions from alternating problems is preparing a string in which all of the possible existential choices are spelled out, and letting JULIET decide on one option by playing *Call* on it. This is due to the asymmetry in the definition of context-free games – JULIET may make only binary choices (*Read* or *Call*), while the set of options for ROMEO is usually much larger (any string from a possibly infinite regular language). For this reason, things get a bit more complicated when we reduce from alternating problems where there is more than a constant number of options for each existential choice.

The standard way of dealing with this requires replay and uses the replacement languages to encode options for existential choices. Whenever an existential choice is required in the problem instance, the input string constructed from it contains a function symbol that, when called, gets replaced by (representations of) all possible options for the existential choice. From these, JULIET may then select one by playing *Call* on it, as in the case of a constant number of options. In a way, this technique employs ROMEO to "spell out" the options for JULIET, with the replacement language for existential choices usually being a singleton language and thus not offering ROMEO any choice of his own. When using this method, cheating prevention tends to become a bit more involved, as the proof of the following result shows.

Proposition 3.7. *For the class of cfGs with unbounded replay and with replacement and target languages given as deterministic regular expressions,* JWIN *is* EXPTIME*-hard.*

Proof. The proof is by reduction from the reachability problem for alternating pushdown systems, which is known to be EXPTIME-complete [Wal01].

An *alternating pushdown system (APDS)* $\mathcal{P} = (S = S_E \cup S_A, \Delta, \delta, F)$ consists of a set S of *states* (partitioned into a set S_E of *existential* states and a set S_A of *universal* states), a *pushdown alphabet* Δ, a *transition relation* $\delta \subseteq S \times \Delta \times S \times \Delta^*$ and a set $F \subseteq S$ of *final states*. A *position* of $\mathcal{P}$ is an element $[q, \alpha]$ of $S \times \Delta^*$; it is *universal* (or *existential*, or *final*, respectively) if $q \in S_A$ (or $q \in S_E$, or $q \in F$, respectively). A position $[r, \beta\alpha]$ of $\mathcal{P}$ is a *successor position* of $[q, A\alpha]$ if there is a transition $(q, A, r, \beta) \in \delta$. The set of *accepting* positions is the least set for which position $[q, \alpha]$ of $\mathcal{P}$ is accepting, if

- $[q, \alpha]$ is final, or
- $[q, \alpha]$ is existential and there is an accepting successor position of $[q, \alpha]$, or
- $[q, \alpha]$ is universal and all successor positions of $[q, \alpha]$ are accepting.

The reachability problem for APDS is to determine, given an APDS $\mathcal{P}$, a state s, and a stack symbol A, whether $[s, A]$ is accepting. Intuitively, this is the same as deciding whether an alternating pushdown automaton with empty input has an accepting run starting at s with symbol A on its pushdown.

We denote by *the transitions (in $\mathcal{P}$) from $q \in S$ with $A \in \Delta$* the set $\delta(q, A) \stackrel{\text{def}}{=} \delta \cap (\{q\} \times \{A\} \times S \times \Delta^*)$.

Without loss of generality and for ease of presentation, we restrict ourselves to APDS with the following properties.

- For each transition $(q, A, r, \beta) \in \delta$, either $\beta = A_1 A_2$ for some $A_1, A_2 \in \Delta$ or $\beta = \epsilon$. We denote the former type of transitions as *push-transitions* and the latter as *pop-transitions*.
- There is a fixed number k such that for each $q \in S$, $A \in \Delta$, there are exactly k transitions from q with A.
- The set $S = \{q_1, \ldots, s_{|S|}\}$ of states and, for each q and A, the set $\delta(q, A) = \{t_1, \ldots, t_k\}$ are canonically ordered.

It is easy to see that any APDS can be transformed into such an APDS with at most polynomial blow-up.

We show how to construct a game G and input string w from $\mathcal{P}, s, A$ such that $[s, A]$ is a winning position of $\mathcal{P}$ if and only if Juliet has a winning strategy in G starting at w. The game has the alphabet

$$\Sigma = \Gamma \cup S \cup \hat{S} \cup \{\$, \#\},$$

where

- $\Gamma = \{\langle q, A\rangle \mid q \in S, A \in \Delta\} \cup \{\langle q, A, i\rangle \mid q \in S, A \in \Delta, i \in [k]\}$ is the set of function symbols, and
- $\hat{S} = \{\hat{q} \mid q \in S\}$.

We write $\Gamma_\#$ for $\Gamma \cup \{\#\}$.

The general idea behind the reduction is to use the choices made by Juliet and Romeo to simulate alternation in $\mathcal{P}$, with Juliet choosing successors of existential positions, Romeo choosing successors of universal positions, and Juliet winning if a final position is reached in this manner. However, since a cfG is not symmetric with respect to Juliet and Romeo, existential and universal moves are handled differently. To this end, the game is designed so that Juliet can select a transition by choosing a position in a so-called choice-string. The choice string $w(q, A)$ for an existential state q and stack symbol A is $\langle q, A, 1\rangle \cdots \langle q, A, k\rangle$. To indicate the choice of the i-th transition for an existential state q and a stack symbol A, Juliet plays *Call* on the position of $\langle q, A, i\rangle$ in $w(q, A)$. Although choice strings are only needed to deal with existential states, we define them also for universal states and lift them to the level of stack symbols, as follows.

The *choice string for* $q \in S$ *and* $A \in \Delta$ is defined as

$$w(q, A) = \begin{cases} \langle q, A\rangle, & \text{if } q \in S_A \\ \langle q, A, 1\rangle \cdots \langle q, A, k\rangle, & \text{if } q \in S_E \end{cases}$$

The *choice string for* $A \in \Delta$ is then $w(A) = w(q_1, A)w(q_2, A) \cdots w(q_{|S|}, A)$.

The game is designed to maintain the following correspondence between positions of $\mathcal{P}$ and configurations of G: a position $(p, AB_1 \cdots B_m)$ of $\mathcal{P}$ corresponds to a configuration (u, pv) of G, where

(1) u is a (possibly empty) concatenation of strings of the form $qx\hat{q}$, in which x is of the form $\Gamma^*_\#\$\Gamma^*$, starting (if u is nonempty) with a string of the form $sx\hat{s}$, and

(2) v is of the form $\Gamma^*_\# \$w(A)\#\Gamma^*_\# \$w(B_1)\#\Gamma^*_\# \$ \cdots \$w(B_k)\#\Gamma^*_\#$.

The sequence of states occurring in the string u reflects the computation of $\mathcal{P}$ that led to position $(p, AB_1 \cdots B_m)$. In v, mainly the choice strings $w(A)$ and $w(B_1), \ldots, w(B_m)$ are relevant, all intermediate symbols from Γ are unselected positions of choice strings in which Juliet already has selected a position. We call the former (choice strings and their positions) *enabled*, and the latter positions *disabled*. It is easy to observe that all enabled positions occur between some symbol \$ and the subsequent #, whereas disabled positions are between some # and the subsequent \$.

The replacement languages are as follows:

- for each $q \in S_A$ and $A \in \Delta$,

$$R_{\langle q,A\rangle} = \{\hat{q}r \mid (q, A, r, \epsilon) \in \delta\} \cup \{\hat{q}r\$w(A_1)\#\$w(A_2)\# \mid (q, A, r, A_1A_2) \in \delta\},$$

- and for each $q \in S_E$, $A \in \Delta$ and $i \in [k]$ with $\delta(q, A) = \{t_1, \ldots, t_k\}$,

$$R_{\langle q,A,i\rangle} = \begin{cases} \{\hat{q}r\} & \text{if } t_i = (q, A, r, \epsilon), \text{ and} \\ \{\hat{q}r\$w(A_1)\#\$w(A_2)\#\} & \text{if } t_i = (q, A, r, A_1A_2). \end{cases}$$

Since these languages are finite they can indeed be described by DREs.

The simulation of a transition in $\mathcal{P}$ by G from a position $(q, AB_1 \cdots B_m)$ is supposed to proceed as follows. Juliet has to skip all positions until and including the first symbol \$ in v and to select a position in the following string $w(A)$. To maintain condition (1) for x she needs to select a position which yields $\hat{q}$, therefore she has to select a position in $w(q, A)$. If q is universal, she just selects $\langle q, A\rangle$, for which Romeo answers by choosing the string from $R_{\langle q,A\rangle}$ that represents his desired transition. If q is existential, she directly selects the symbol $\langle q, A, i\rangle$ that represents her desired transition. In either case, the game then continues by simulating the next transition.

It is easy to see that Juliet can maintain in this way the invariance conditions (1) and (2). Indeed, by selecting a position in $w(q, A)$ the required symbol $\hat{q}$ is generated and, since $w(A)$ is immediately after the first occurrence of \$ in v, it occurs between \$ and the subsequent occurrence of #. Thus, (1) is maintained. By the replacement, the selected $\langle q, A\rangle$ or $\langle q, A, i\rangle$ is replaced either by a string $\hat{q}r$ or a by a string $\hat{q}r\$w(A_1)\#\$w(A_2)\#$. In the former case, all symbols between r and $\$w(B_1)$ are from $\Gamma^*_\#$ and therefore the new remaining string has the correct form. In the latter case, the string between r and the subsequent \$ is empty, again yielding a string of the correct form with respect to (2).

It remains to be justified that Juliet can not improve her winning chances by deviating from the intended pattern.

- If she skips the next enabled choice string $w(A)$, the resulting string has two occurrences of \$ without an intermediate symbol from $\hat{S}$, thus u does not obey (1).
- If she selects a position in $w(A)$ that corresponds to some other state than q, the resulting string is in conflict with (1), too.
- If she selects more than one positions from some choice string $w(A)$, then the second of them is disabled, since the first selection yielded the symbol # and all further replacements before the second selected position in $w(A)$ take place on the left of this symbol #. This, too, would yield a string violating (1).

It remains to give a DRE for the target language T. Following the above considerations, T needs to ensure that (1) holds in the resulting string w and that the last state in w is from F. To this end, we define

$$T = \Big(\bigoplus_{q \in S \setminus F} q\Gamma_{\#}^{*}\$\Gamma^{*}\hat{q} \Big)^{*} F\Sigma^{*}.$$

It is easy to see that T is indeed a DRE and that G can be constructed in polynomial time. The invariants (1) and (2) can be shown by induction on the number of transition steps, along the lines indicated above. Note that T (and invariant (1)) do not enforce the *presence* of # symbols in order to allow for correct handling of pop-transitions (which do not introduce # symbols into the current string).

Finally, it follows from invariants (1) and (2), as well as the construction of T, that Juliet has a winning strategy in the game on string $s\$w(A)$ if and only if $[s, A]$ is an accepting position for $\mathcal{P}$. □

The second standard type of lower bound proof for context-free games uses a reduction from (the complement of) some existence problem, such as the Corridor Tiling problem used in the proof of Proposition 3.8 (given a tiling instance, is there a valid tiling of given width and arbitrary height?). In the game constructed in such a reduction, it is usually Romeo's task to give a witness for the existence of an object of the desired form (here: a tiling), and Juliet then has to show that the witness given by Romeo is incorrect (here: that the tiling contains some error and is thus not valid). The target language should then accept if and only if Juliet has indeed disproven the correctness of Romeo's witness (or if Romeo has tried to cheat). In this way, Juliet should have a winning strategy in the game constructed from a problem instance if and only if the instance is a negative one.

We call this proof technique (first used in [MSS06]) the *protest technique*, as Juliet will usually highlight errors in the witness by playing *Call* on selected symbols of the witness string that make up an error and forcing Romeo to put some sort of flag on those symbols (i.e. *protesting* on the chosen symbols). Cheating prevention for reductions with the protest technique generally has to take into account that Juliet may try to protest on symbols that do not constitute an error. For errors that are not easily verified by polynomial-sized regular conditions, it may even be necessary to allow Romeo to *counter-protest* and prove that Juliet's protest was in fact not justified. A rather

involved example of protest and counter-protest can be seen in the lower bound proof for Theorem 4.1(a); for this section, though, we simply examine a relatively basic and straightforward use of the protest technique.

Proposition 3.8. *For the class of cfGs with bounded replay and with replacement and target languages given as deterministic regular expressions,* JWin *is* PSPACE*-hard.*

Proof. The proof is a reduction from the complement of the PSPACE-complete problem Corridor Tiling, which was defined in Section 2.5. Since PSPACE is closed under complement, the complement of Corridor Tiling is complete for PSPACE as well.

The reduction from the complement of Corridor Tiling to JWin is constructed as follows. Given an instance $\mathcal{I} = (U, V, H, u_i, u_f, n)$ for Corridor Tiling, the reduction constructs a game $G = (\Sigma, \Gamma, R, T)$ and a symbol s from Γ such that Juliet has a winning strategy on the string s if and only if $\mathcal{I}$ does *not* have a valid corridor tiling. The basic idea is that, after Juliet's first *Call* move on s, Romeo answers by an encoding w of a valid corridor tiling, if one exists. With her *Call* moves of depth 2, Juliet may then try to flag inconsistencies (i.e. constraint violations) in the tiling given by Romeo; finally, the target automaton should accept a tiling if Juliet did indeed point out an actual inconsistency.

We assume, without loss of generality, that there are no pairs (u_f, v) in H or V, that is, u_f can only appear as the final tile and nowhere else in a valid tiling.

The game G is over an alphabet $\Sigma = U \cup \hat{U} \cup \{\#\}$ where $\hat{U} = \{\hat{u} \mid u \in U\}$ is disjoint from U. The replacement and target languages are described next.

A *tiling candidate* (for $\mathcal{I}$) is a string of the form $(U^n\#)^*$, as defined in Section 2.5. The replacement language R_s consists of all tiling candidates v such that u_i is the first symbol of v. It is easy to see that R_s can be described by a DRE of polynomial size in $|\mathcal{I}|$. The other replacement languages are very simple: $R_u = \{\hat{u}\}$ for each $u \in U$.

There can be the following three kinds of inconsistencies in a tiling candidate.

- There are two horizontally adjacent tiles u, v, but $(u, v) \notin H$;
- There are two vertically adjacent tiles u, v, but $(u, v) \notin V$;
- The final tile is different from u_f.

The target language is constructed such that Juliet can win in the first two cases by selecting the two positions of u and w and in the latter case by selecting the last U-position u of the string. In all cases, Romeo inevitably replaces the selected positions by $\hat{u}$ and $\hat{v}$, respectively. The target language thus consists of all tiling candidates in which one or two symbols have been replaced by symbols from $\hat{U}$ such that one of the above situations arises. It can be expressed by the following DRE.

$$(U + \#)^*\Big(\hat{u}_f\Sigma\Sigma\Sigma^* + \bigoplus_{u \in U \setminus \{u_f\}} \hat{u}(\alpha_u + \beta_u)\Big),$$

where

- $\alpha_u = \bigoplus_{(u,v)\notin H} \hat{v}\Sigma^*$ and
- $\beta_u = U\big((U+\#)^{n-1} \bigoplus_{(u,v)\notin V} \hat{v}\Sigma^*\big) + \#\big(\epsilon + (U^{n-1} \bigoplus_{(u,v)\notin V} \hat{v}\Sigma^*)\big)$.

We note that α_u accounts for horizontal mistakes whereas β_u accounts for vertical mistakes *and* for the wrong final tile. The disjunct $\hat{u_f}\Sigma\Sigma\Sigma^*$ makes use of our assumption regarding u_f: If u_f occurs at a non-final position, Juliet wins instantly and no particular condition needs to be tested. However, this frees us from the need to distinguish the case of u_f in the remainder of the expression. □

3.4. Techniques for upper bound proofs

When proving upper bounds for two-player games of complete information, a straightforward approach is trying to translate the game into an alternating algorithm that simply guesses moves for both players; for context-free games, this approach would correspond to guessing existentially a strategy for Juliet and universally branching for moves of Romeo.

In the setting of context-free games with regular replacement languages, there are the following obstacles to this approach: (1) Romeo can, in general, choose from an infinite number of (and thus arbitrarily long) replacement strings, and (2) it is not a priori clear that such an alternating algorithm terminates on all branches. Whereas the latter obstacle is not too serious (if Juliet has a winning strategy, termination on all branches is guaranteed), the former requires a more refined approach.

We can usually deal with it in two ways: in some cases, it is possible to show that it does not help Romeo to choose strings of length beyond some bound; in other cases, the algorithms use abstracted moves instead of the actual replacement moves of the game. While the former approach is quite straightforward, the latter deserves some additional discussion.

The generic algorithm for this approach works in two main stages for a given cfG $G = (\Sigma, \Gamma, R, T)$ and word w. It first analyses the game G and aggregates all necessary information in a so-called *call effect* $\mathcal{C}[G]$, which only depends on G. Then it uses $\mathcal{C}[G]$ to decide whether Juliet has a winning strategy in the game G on w.

In a nutshell, the call effect contains, for every function symbol f and every state q of the target automaton $A(T)$, all possible impacts that the subgame beginning with a *Call* move of Juliet on symbol f may have on the automaton $A(T)$ starting at state q. It summarises which sets of states Juliet can enforce by some strategy σ, taking into account arbitrary counter strategies of Romeo against σ.

At the end of the first stage, the algorithm computes an alternating automaton $A_{\mathcal{C}[G]}$ from $\mathcal{C}[G]$ that decides the set $\mathrm{JWin}(G)$. In the second stage, $A_{\mathcal{C}[G]}$ is then evaluated on w.

The described basic framework for the generic algorithm is summarised above as Algorithm 1. The details of call effects, how they can be computed and how they can be translated into automata are specific to each setting. In this section, we examine these

Algorithm 1 Generic algorithm for JWin

1: Input: cfG G, string $w \in \Sigma^*$
2: Compute from G the call effect $\mathcal{C}[G]$
3: Compute from $\mathcal{C}[G]$ an alternating automaton $A_{\mathcal{C}[G]}$ deciding JWin(G)
4: Run $A_{\mathcal{C}[G]}$ on input w
5: If $A_{\mathcal{C}[G]}$ accepts w, accept; otherwise reject.

specifics for the comparatively simple setting of cfGs with regular replacement languages to prove the following upper bounds:

Theorem 3.9. *For the class of cfGs with regular replacement languages, the following complexity results hold for the* JWin *problem:*

(a) with unbounded replay, it is in EXPTIME*,*

(b) with bounded replay, it is in PSPACE*, and*

(c) without replay, it is in PTIME*.*

Throughout this chapter, recall that $A(T) = (Q, \Sigma, \delta, q_0, F)$ denotes the target language DFA.

Example 3.10 (Running example)**.** *Throughout this section, we again consider the cfG $G = (\Sigma, \Gamma, R, T)$ already introduced in Example 1.1, where $\Sigma = \{a, f, g\}$, $\Gamma = \{f, g\}$, $R_f = L(g(f + g^*)g)$, $R_g = \{af, f\}$, and T is the language of all strings of the form $a^*ff^*gg^*$, represented by the DFA $A(T) = (\{q_a, q_f, q_g, q_\perp\}, \Sigma, \delta, q_a, \{q_g\})$ with $\delta(q_a, a) = q_a$, $\delta(q_a, f) = \delta(q_f, f) = q_f$, $\delta(q_f, g) = \delta(q_g, g) = q_g$, and $\delta(q, b) = q_\perp$ for all other combinations of $q \in \{q_a, q_f, q_g, q_\perp\}$, $b \in \Sigma$. A graphical representation of $A(T)$ is shown in Figure 3.1.*

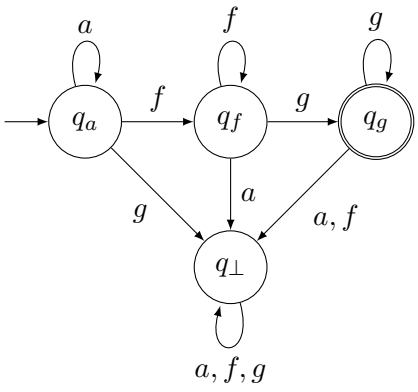


Figure 3.1.: Target language DFA $A(T)$ deciding $T = a^*ff^*gg^*$ in the game from Example 3.10.

3.4.1. Call effects and word effects

Our abstraction from actual cfGs, and the very notion of call effects, is based on the simple observation that instead of knowing the final word $\mathrm{word}_G(w, \sigma, \tau)$ that is reached in a play $\Pi(\sigma, \tau, w)$, it suffices to know whether $\delta^*(q_0, \mathrm{word}_G(w, \sigma, \tau)) \in F$ to determine the winner. If we fix a strategy σ of Juliet in a game on w, the possible outcomes of the game (for the different strategies of Romeo) are the words in

$$\mathrm{words}_G(w, \sigma) \stackrel{\text{def}}{=} \{\mathrm{word}_G(w, \sigma, \tau) \mid \tau \in \mathrm{STRAT}_{\mathrm{R}}(G)\},$$

which (by the above observation) can be summarised by

$$\mathrm{states}_G(q_0, w, \sigma) \stackrel{\text{def}}{=} \{\delta^*(q_0, v) \mid v \in \mathrm{words}_G(w, \sigma)\}.$$

To simplify notation, the subscript G will often be omitted if the game G is clear from the context.

In the following, we sometimes consider *subgames* on a certain part of a string and talk about strategies for subgames. From a configuration (J, u, vw), Juliet can use a strategy σ on the subgame on v. This means that she follows σ until a configuration (uv', w) is reached. It will be particularly useful to study the (abstractions of) possible outcomes of subgames that start from a *Call* move on some symbol $a \in \Gamma$ until the focus moves to the symbol after a.

Definition 3.11. For a cfG $G = (\Sigma, \Gamma, R, T)$ with a target DFA $A(T) = (Q, \Sigma, \delta, q_0, F)$, the *call effect* $\mathcal{C}[G] : \Gamma \times Q \to \mathscr{P}(\mathscr{P}(Q))$ is defined, for every $a \in \Gamma$, $q \in Q$, by

$$\mathcal{C}[G](a, q) \stackrel{\text{def}}{=} \left[\{\mathrm{states}_G(q, a, \sigma) \mid \sigma \in \mathrm{STRAT}_{\mathrm{J},\mathit{Call}}(G)\}\right]_{\min},$$

where $\mathrm{STRAT}_{\mathrm{J},\mathit{Call}}(G)$ contains all strategies of Juliet that start by playing *Call* on a, and the operator $[\cdot]_{\min}$ (as defined in Chapter 2) removes all sets that are non-minimal with respect to inclusion from a set of sets.

The intuition behind call effects is the following abstraction into a single-round game of the subgame on some function symbol a where Juliet starts by playing *Call* on a: Juliet first chooses a strategy $\sigma \in \mathrm{STRAT}_{\mathrm{J},\mathit{Call}}(G)$, then Romeo chooses a strategy τ; the outcome of the game on a is uniquely determined by σ and τ. In terms of effects, this corresponds to Juliet picking a set $X = \mathrm{states}_G(q_0, a, \sigma) \in \mathcal{C}[G](a, q_0)$ and Romeo then choosing a final state $q = \delta^*(q_0, \mathrm{word}_G(a, \sigma, \tau)) \in X$. This intuition also explains the use of the $[\cdot]_{\min}$ operator, as it makes no sense for Juliet to offer Romeo a choice from a set $X \subseteq Q$ if she can instead offer him the more limited options in some $X' \subsetneq X$.

We now lift this abstraction from call effects of subgames on single function symbols, as defined above, to effects of arbitrary strings. Formally, a *(word) effect* maps states q of $A(T)$ to sets of sets of states of $A(T)$. The effect of a game G on a word w relative to state q is basically the set of all state sets X, for which Juliet has a strategy that guarantees that every play on w yields some word v with $\delta^*(q, v) \in X$.

Definition 3.12. For a cfG $G = (\Sigma, \Gamma, R, T)$ with a target DFA $A(T) = (Q, \Sigma, \delta, q_0, F)$ and a string $w \in \Sigma^*$, the *word effect,* $\mathcal{E}[G, w] : Q \to \mathscr{P}(\mathscr{P}(Q))$*, of* G *on* w is the function defined by

$$\mathcal{E}[G, w](q) \stackrel{\text{def}}{=} [\{\text{states}_G(q, w, \sigma) \mid \sigma \in \text{STRAT}_\text{J}(G)\}]_{\min},$$

for every $q \in Q$, where the operator $[\cdot]_{\min}$ removes all non-minimal sets from a set of sets as before.

The intuition behind word effects, similar to call effects, is abstracting cfGs into single-round games, where on an input string w, Juliet chooses a strategy σ, then Romeo chooses a strategy τ. As described for call effects, this also corresponds to Juliet first picking a set $X = \text{states}_G(q_0, w, \sigma) \in \mathcal{E}[G, w](q_0)$, and Romeo then choosing a final state $q = \delta^*(q_0, \text{word}_G(w, \sigma, \tau)) \in X$ corresponding to the final string induced by σ and τ.

Note that there is a slight difference between the call effect $\mathcal{C}[G](a, q)$ and word effect $\mathcal{E}[G, a](q)$ – the former only considers strategies for Juliet which play *Call* on the initial a, while the latter considers *all* strategies, including the one that plays *Read*. This distinction may seen somewhat artificial (in fact, the original proof in [BSSK13] exclusively considered word effects), but it generalises much more directly to the nested word setting of Chapter 6.

Example 3.13. *For any cfG* $G = (\Sigma, \Gamma, R, T)$ *with target language DFA* $A(T) = (Q, \Sigma, \delta, q_0, F)$*, any string* $w \in (\Sigma \backslash \Gamma)^*$ *containing no function symbols and any strategy* $\sigma \in STRAT_\text{J}(G)$*, it holds that* $words_G(w, \sigma) = \{w\}$*, and therefore, for any* $q \in Q$*,* $states_G(q, w, \sigma) = \{\delta^*(q, w)\}$ *and* $\mathcal{E}[G, w](q) = \{\{\delta^*(q, w)\}\}$*. In particular, it holds that* $\mathcal{E}[G, \epsilon](q) = \{\{q\}\}$*.*

Example 3.14 (Running example)**.** *In the cfG* G *from Example 3.10, for the strategy* $\sigma_\perp$ *of* Juliet *on the string* $w = f$ *that plays Call on* f *and Read on any string returned by* Romeo *as a call result, it holds that* $words_G(f, \sigma_\perp) = L(g(f + g^*)g)$*,* $states_G(q_a, f, \sigma_\perp) = \{q_\perp\}$*, and therefore* $\{q_\perp\} \in \mathcal{E}[G, f](q_a)$*. Since* $\sigma_\perp \in STRAT_{\text{J},Call}(G)$*, it also holds that* $\{q_\perp\} \in \mathcal{C}[G](f, q_a)$*.*

On the other hand, for the strategy $\sigma_{g,\perp} \in STRAT_{\text{J},Call}(G)$ *that plays Call on* f*, then plays Call on the leftmost* g *if* Romeo *returns* gfg *and Read on all symbols otherwise, it holds that* $words_G(f, \sigma_{g,\perp}) = \{affg, ffg\} \cup L(gg^*g)$ *and* $states_G(q_a, f, \sigma_{g,\perp}) = \{q_g, q_\perp\}$*. However,* $\{q_g, q_\perp\} \in \mathcal{C}[G](f, q_a)$ *does* not *hold, because* $\mathcal{C}[G](f, q_a)$ *already contains the subset* $\{q_\perp\}$ *of* $\{q_g, q_\perp\}$*.*

It is easy to see that Juliet has a winning strategy in G on w if and only if there is some $X \in \mathcal{E}[G, w](q_0)$ such that $X \subseteq F$; to determine whether Juliet has a winning strategy it therefore suffices to compute $\mathcal{E}[G, w]$. This is the basic idea behind the AFA $A_{\mathcal{C}[G]}$ in Step 4 of the generic algorithm – on input $w \in \Sigma^*$, $A_{\mathcal{C}[G]}$ uses the call effect $\mathcal{C}[G]$ to compute in an alternating fashion some set $X \in \mathcal{E}[G, w](q_0)$ and then checks whether $X \subseteq F$ holds. However, variants of $A_{\mathcal{C}[G]}$ are also used in the computation of $\mathcal{C}[G]$ from G (Step 2 of Algorithm 1), so we first examine how to compute $A_{\mathcal{C}[G]}$ from $\mathcal{C}[G]$ (Step 3 of Algorithm 1).

3.4.2. Transforming call effects into AFAs

The main focus of this subsection is the proof of the following proposition.

Proposition 3.15. *There is an algorithm that computes from the call effect $\mathcal{C}[G]$ of a game G in polynomial time in $|\mathcal{C}[G]|$ and $|G|$ an AFA $A_{\mathcal{C}[G]}$ such that $L(A_{\mathcal{C}[G]}) = JWin(G)$.*

We note that the algorithm proposed in this result only runs in polynomial time in $|G|$ if $|\mathcal{C}[G]|$ is polynomial in $|G|$; on the other hand, since $\mathcal{C}[G]$ basically stores a polynomial number of (minimised) sets of sets of states, the algorithm's runtime is always at most exponential in $|G|$.

As mentioned in the previous subsection, the basic idea behind the construction used by the algorithm is that $A_{\mathcal{C}[G]}$ uses the call effect $\mathcal{C}[G]$ to compute word effects $\mathcal{E}[G, w]$ for input strings w. To this end, we first examine how word effects can be computed from call effects.

It is natural to reason inductively about word effects. As the basis for this induction, the definitions of $\mathcal{E}[G, a]$ and $\mathcal{C}[G]$ directly yield the following result on single-symbol word effects.

Lemma 3.16. *For every cfG $G = (\Sigma, \Gamma, R, T)$, $q \in Q$ and $a \in \Gamma$ it holds that*

$$\mathcal{E}[G, a](q) = \mathcal{C}[G](a, q) \cup \{\{\delta(q, a)\}\}.$$

For composition of strings, we first consider the intuitive perspective on the (abstracted) game. From Juliet's point of view, the game on a string uv (with $u, v \in \Sigma^*$) from a state q on proceeds as follows. Juliet fixes a strategy σ on u. The set of states that Romeo can choose from at the end of the subgame on u is just $\text{states}_G(q, u, \sigma)$. For each state $p \in \text{states}_G(q, u, \sigma)$, Juliet can choose a strategy σ_p for v and the result set is then the union of all sets that can be reached by Romeo against any σ_p on v. To express the set of all combinations of outcomes for the second part, we use the following operator.

Definition 3.17. Let $\mathcal{D} = \{D_1, \ldots, D_n\}$ be a set of sets of sets. Then $\textsc{Mix}(\mathcal{D})$ is the set

$$[\{d_1 \cup \cdots \cup d_n \mid d_1 \in D_1 \wedge \cdots \wedge d_n \in D_n\}]_{\min}.$$

In other words, the Mix operation yields every way of taking the union of one element from each of $D_1, \ldots, D_n$ and then removes non-minimal sets.

Let E_1, E_2 be mappings from Q into $\mathscr{P}(\mathscr{P}(Q))$. Then the *composition of E_1 and E_2* is defined as the mapping $E_1 \circ E_2 : Q \to \mathscr{P}(\mathscr{P}(Q))$ with

$$(E_1 \circ E_2)(q) \stackrel{\text{def}}{=} \left[\bigcup_{X \in E_1(q)} \textsc{Mix}(\{E_2(q') | q' \in X\}) \right]_{\min}.$$

Example 3.18. *Let $E_1(q) = \{\{q_1, q_2\}, \{q_3\}\}$, $E_2(q_1) = \{\{q_4, q_5\}, \{q_6\}\}$, $E_2(q_2) = \{\{q_7\}\}$ and $E_2(q_3) = \{\{q_5\}\}$. Then, it holds that*

$$(E_1 \circ E_2)(q) = [\{\{q_4, q_5, q_7\}, \{q_6, q_7\}\} \cup \{\{q_5\}\}]_{min} = \{\{q_6, q_7\}, \{q_5\}\}.$$

Not surprisingly, effect composition corresponds to word concatenation.

Lemma 3.19. *For every cfG $G = (\Sigma, \Gamma, R, T)$ and $u, v \in \Sigma^*$ it holds that*

$$\mathcal{E}[G, uv] = \mathcal{E}[G, u] \circ \mathcal{E}[G, v].$$

Proof. Let $q \in Q$. This proof uses Lemma 2.1 to prove the equality of the two normalised sets $\mathcal{E}[G, uv](q)$ and $(\mathcal{E}[G, u] \circ \mathcal{E}[G, v])(q)$.

($\sqsupseteq$): Let $X \in \mathcal{E}[G, uv](q)$. Then there exists some strategy σ_{uv} for Juliet such that $X = \text{states}_G(q, uv, \sigma_{uv})$. Let σ_u be the restriction of σ_{uv} to the subgame on u, let $X_u \in \mathcal{E}[G, u](q)$ with $X_u = \{p_1, \ldots, p_k\} \subseteq \text{states}_G(q, u, \sigma_u)$. For each $i \in [k]$, let σ_v^i be a restriction of σ_{uv} to the subgame on v, in case Romeo chooses a strategy τ with $\text{state}_G(q, u, \sigma_u, \tau) = p_i$, and let $X_v^i \in \mathcal{E}[G, v](p_i)$ such that $X_v^i \subseteq \text{states}_G(p_i, v, \sigma_v^i)$ for all $i \in [k]$. Let $X' = X_v^1 \cup \ldots X_v^k$.

By definition of $\circ$, and because of normalisation, there exists some $X'' \in (\mathcal{E}[G, u] \circ \mathcal{E}[G, v])(q)$ with $X'' \subseteq X'$. So, to show the desired inclusion, it suffices to prove that $X' \subseteq X$.

Let $p' \in X'$. Then, $p' \in X_v^i$ and therefore $p' \in \text{states}_G(p_i, v, \sigma_v^i)$ for some $i \in [k]$. Also, $p_i \in X_u \subseteq \text{states}_G(q, u, \sigma_u)$, i.e. $p_i = \text{state}_G(q, u, \sigma_u, \tau)$ for some $\tau \in \text{STRAT}_\text{R}(G)$. By the definition of σ_u and σ_v^i, this implies that $p' \in \text{states}_G(q, uv, \sigma_{uv}) = X$.

($\sqsubseteq$): Let $X \in (\mathcal{E}[G, u] \circ \mathcal{E}[G, v])(q)$. By definition of $\circ$, there are sets $X_u \in \mathcal{E}[G, u](q)$ with $X_u = \{q_1, \ldots, q_k\}$ (for some k) and $X_v^i \in \mathcal{E}[G, v](q_i)$ for each $i \in [k]$ such that $X \subseteq X_v^1 \cup \ldots \cup X_v^k$. By definition of $\mathcal{E}[G, \cdot]$, there are strategies $\sigma_u, \sigma_v^1, \ldots, \sigma_v^k \in \text{STRAT}_\text{J}(G)$ with $X_u = \text{states}_G(q, u, \sigma_u)$ and $X_v^i = \text{states}_G(q_i, v, \sigma_v^i)$.

Define a strategy σ_{uv} on uv as follows. On u, Juliet plays according to σ_u; if this play yields some string $u_i \in \text{words}_G(u, \sigma_u)$ with $\delta^*(q, u_i) = q_i$, Juliet then plays according to σ_v^i on v.

Denote $\text{states}_G(q, uv, \sigma_{uv})$ by X' for short. Due to normalisation, there exists some $X'' \in \mathcal{E}[G, uv]$ with $X'' \subseteq X'$. What needs to be shown is therefore only that $X' \subseteq X$.

Let $q' \in X'$. Then there exists a strategy τ for Romeo as well as strings $u', v' \in \Sigma^*$ such that $u'v' = \text{word}_G(uv, \sigma_{uv}, \tau)$, $u' = \text{word}_G(u, \sigma_{uv}, \tau)$ and $\delta^*(q, u'v') = q'$. Let $q_u = \delta^*(q, u')$; then, it holds that $q' = \delta^*(q_u, v')$. By the definition of σ_{uv}, it follows that $q_u \in X_u$ and $q' \in X_v^i$ for some i, so $q' \in X$, which concludes the proof. □

It follows directly from Lemma 3.19 that the sequential composition of effects is associative.

We can now define the AFA $A_{\mathcal{C}[G]}$ from Proposition 3.15. The intuition behind it is that $A_{\mathcal{C}[G]}$ uses alternation to guess strategy choices for Juliet and Romeo in the above abstraction of G on w using call effects and tracks a current state q in the target language DFA $A(T)$. On non-function symbols, as well as on function symbols for which $A_{\mathcal{C}[G]}$ existentially guesses Juliet's move to be *Read*, $A_{\mathcal{C}[G]}$ simply simulates $A(T)$; on function symbols a where $A_{\mathcal{C}[G]}$ decides for Juliet to play *Call*, $A_{\mathcal{C}[G]}$ then chooses existentially a set $X \in \mathcal{C}[G](a, q)$ (corresponding to a substrategy for Juliet after the *Call* on a) and branches universally into all states $q' \in X$ (corresponding to Romeo's choice of a counter-strategy and a corresponding resulting state).

Formally, $A_{\mathcal{C}[G]} = (Q, \Sigma, \delta_C, q_0, F)$ is an AFA where δ_C is defined as follows. (Recall that $A(T) = (Q, \Sigma, \delta, q_0, F)$ is the target language DFA.)

- For $a \in \Sigma \backslash \Gamma, q \in Q$:
$$\delta_C(q, a) \stackrel{\text{def}}{=} \delta(q, a).$$
- For $a \in \Gamma, q \in Q$:
$$\delta_C(q, a) \stackrel{\text{def}}{=} \delta(q, a) \vee \bigvee_{X \in \mathcal{C}[G](a,p)} \bigwedge_{r \in X} r.$$

Note that the state space of $A_{\mathcal{C}[G]}$ is the same as that of $A(T)$, but (the representation of) its transition function is generally exponential in the size of $A(T)$.

We go on to prove the correctness of $A_{\mathcal{C}[G]}$. Recall that we call a run ρ of an AFA A on a string w *minimal* if no proper subtree of ρ is a run of A on w (i.e. if each set of states chosen to follow up some state on reading some symbol is inclusion-minimal among the sets of states fulfilling the corresponding transition formula).

Lemma 3.20. *Let $q \in Q$, $w \in \Sigma^*$ and $X \subseteq Q$. Then, $X \in \mathcal{E}[G, w](q)$ if and only if there is a minimal run of $A_{\mathcal{C}[G]}$ on w starting at q and ending in states from X.*

Proof. Let $q \in Q$, $X \subseteq Q$ and $w \in \Sigma^*$. The proof is by induction on the structure of w.

For $w = \epsilon$, the claim is trivially fulfilled, since $\mathcal{E}[G, \epsilon](q) = \{\{q\}\}$ by the definition of string effects. Similarly, for $a \in \Sigma \backslash \Gamma$, $\mathcal{E}[G, a](q) = \{\{\delta(q, a)\}\}$.

For $w = a \in \Gamma$, a minimal run of $A_{\mathcal{C}[G]}$ on a from q consists of a q-labelled root whose children are leaves and have as labels either just $\delta(q, a)$ or exactly the states in some set $X \in \mathcal{C}[G](a, p)$. With this observation, the desired equivalence follows directly from Lemma 3.16.

Let $w = uv$ for $u, v \in \mathrm{WF}(\Sigma)$. For the "only if" direction, it follows from Lemma 3.19 that there are sets $X_u = \{q_i, \ldots, q_k\} \in \mathcal{E}[G, u](q)$ and $X_v^1, \ldots, X_v^k$ with $X_v^i \in \mathcal{E}[G, v](q_i)$ for each $i \in [k]$ and $X = X_v^1 \cup \ldots \cup X_v^k$. By induction, there exist a minimal run ρ_u of $A_{\mathcal{C}[G]}$ starting at q and ending inside X_u and for each $i \in [k]$ a minimal run ρ_v^i on v starting at q_i and ending inside X_v^i. From these, we can construct a run ρ of $A_{\mathcal{C}[G]}$ on w by replacing each leaf labelled q_i in ρ_u with the entire run ρ_v^i rooted at q_i. Obviously, ρ is a run of $A_{\mathcal{C}[G]}$ starting at q and ending inside X, and ρ is minimal because ρ_u and all ρ_v^i are. The "if" direction is proven analogously. □

Now we are in the position to prove Proposition 3.15:

Proposition 3.15 ***(restated).*** *There is an algorithm that computes from the call effect $\mathcal{C}[G]$ of a game G in polynomial time in $|\mathcal{C}[G]|$ and $|G|$ an AFA $A_{\mathcal{C}[G]}$ such that $L(A_{\mathcal{C}[G]}) = JWin(G)$.*

Proof. The statement follows from Lemma 3.20, as $A_{\mathcal{C}[G]}$ has an accepting run on any string $w \in \mathrm{WF}(\Sigma)$ if and only if it has a minimal such run. Obviously, $A_{\mathcal{C}[G]}$ is of polynomial size in the size of G and $\mathcal{C}[G]$ and can be constructed from these in polynomial time. □

3.4.3. Computing call effects from games

We next describe how to compute $\mathcal{C}[G]$ from a given cfG G, that is, how step 2 of Algorithm 1 can be carried out. This step of the algorithm follows a fixpoint-based approach. It computes inductively, for $k = 1, 2, \ldots$ the call effect of the restricted game of maximum *Call* depth k.

To this end, let, for every cfG G, $a \in \Sigma$, $q \in Q$, and $k \geqslant 1$,

$$\mathcal{C}^k[G](a,q) \stackrel{\text{def}}{=} \Big[\{\text{states}_G(q,a,\sigma) \mid \sigma \in \text{STRAT}^k_{\text{J},Call}(G)\}\Big]_{\min},$$

where $\text{STRAT}^k_{\text{J},Call}(G) = \text{STRAT}_{\text{J},Call}(G) \cap \text{STRAT}^k_{\text{J}}(G)$ is the set of all strategies for Juliet of *Call* depth bounded by k that start out with a *Call* as their first move.

As an important special case, the call effect of replay-free games – the basis for the inductive computation – consists of only one set.

Lemma 3.21. *For every $q \in Q$ and $a \in \Sigma$, it holds that*

$$\mathcal{C}^1[G](a,q) = \{\{\delta^*(q,v) \mid v \in R_a\}\}.$$

In particular, $\mathcal{C}^1[G]$ can be computed from G in polynomial time.

Proof. This just follows from the definitions, since Romeo can choose any string from R_a. The computation of $\mathcal{C}^1[G](a,q)$ (for each $a \in \Gamma$, $q \in Q$) simply requires checking for each $q' \in Q$ whether there is a $v \in R_a$ such that $q' = \delta^*(q,v)$, which can be done with a standard product construction in polynomial time. □

We next describe how each $\mathcal{C}^{k+1}[G]$ can be computed from $\mathcal{C}^k[G]$. Afterwards, we show that the fixpoint reached by this process is the actual call effect $\mathcal{C}[G]$.

Lemma 3.22. *Given a state $q \in Q$, a function symbol $a \in \Gamma$, and $\mathcal{C}^k[G]$, for some $k \geqslant 1$, the call effect $\mathcal{C}^{k+1}[G](a,q)$ can be computed in polynomial space in $|G|$.*

Proof. Let $a \in \Gamma$, $q \in Q$, $X \subseteq Q$, and $k \geqslant 1$. We show that, given $\mathcal{C}^k[G]$ and a set $X \subseteq Q$, it can be decided in polynomial space in $|G|$ and polynomial time in $|\mathcal{C}^k[G](a,q)|$ whether a subset of X is in $\mathcal{C}^{k+1}[G](a,q)$.

First off, if X has a subset in $\mathcal{C}^k[G](a,q)$, it also has a subset in $\mathcal{C}^{k+1}[G](a,q)$, since $\text{STRAT}^k_{\text{J},Call}(G) \subseteq \text{STRAT}^{k+1}_{\text{J},Call}(G)$. Checking whether X has a subset in $\mathcal{C}^k[G](a,q)$ is obviously feasible in polynomial time in $|\mathcal{C}^k[G]|$ by iterating over all sets $X' \in \mathcal{C}^k[G](a,q)$ and checking whether $X' \subseteq X$ holds. Since this test requires at most logarithmic space in $|\mathcal{C}^k[G](a,q)|$, and since $\mathcal{C}^k[G](a,q) \subseteq \mathscr{P}(Q)$, this is also possible in polynomial space in $|G|$. Assume therefore for the rest of this proof that X does not have a subset in $\mathcal{C}^k[G](a,q)$.

Let $A_{\mathcal{C}^k[G]}$ be as defined for the proof of Lemma 3.20 with $\mathcal{C}^k[G]$ as its basic *Call* effect, and let A be its modification with initial state q and set X of accepting states. As per Lemma 3.20, A accepts all strings w on which there exists a strategy σ for Juliet of *Call* depth at most k such that $\text{states}_G(q,w,\sigma) \subseteq X$.

Let, for each $a \in \Gamma$, $A_a = (Q_a, \Sigma_a, \delta_a, q_{0,a}, F_a)$ be a NFA for R_a.

By definition, X has a subset in $\mathcal{C}^{k+1}[G](a,q)$, if Juliet has a strategy σ of call depth $k+1$ on a that plays *Call* on a with $\text{states}_G(q,a,\sigma) \subseteq X$. Such a strategy σ for Juliet exists if and only if *for every word* $w \in R_a$ there is a strategy σ_w of call depth k for Juliet on w with $\text{states}_G(q,w,\sigma_w) \subseteq X$, thus if and only if $R_a \subseteq L(A)$, equivalently $R_a \cap \overline{L(A)} = \varnothing$.

By using a standard product construction and a complementation of an AFA, the test boils down to a non-emptiness test for an AFA with a state set of polynomial size in $|G|$ and can thus be done in polynomial space thanks to Theorem 2.6[6] □

By Lemmas 3.21 and 3.22, one can compute $\mathcal{C}^k[G]$ inductively, for every $k \geqslant 1$. By definition it holds, for every q and a, that $\mathcal{C}^k[G](a,q)$ is contained in the closure of $\mathcal{C}^{k+1}[G](a,q)$ under supersets. As there are at most $2^{|Q|}$ sets in each $\mathcal{C}^k[G](a,q)$ (for $a \in \Gamma, q \in Q$), the computation reaches a fixed point after at most exponentially many iterations. We denote this fixed point by $\mathcal{C}^*[G]$, that is, we define, for every $a \in \Sigma$, $q \in Q$:

$$\mathcal{C}^*[G](a,q) \stackrel{\text{def}}{=} \left[\bigcup_{k=1}^{\infty} \mathcal{C}^k[G](a,q)\right]_{\min} .$$

In particular, for each game G, there is a number $\ell \leqslant |\Gamma| \times |Q| \times 2^{|Q|}$ such that $\mathcal{C}^*[G] = \mathcal{C}^\ell[G]$ and $\mathcal{C}^m[G] = \mathcal{C}^\ell[G]$, for every $m \geqslant \ell$. However, it is not self evident that this process really constructs $\mathcal{C}[G]$, i.e., that $\mathcal{C}^*[G] = \mathcal{C}[G]$. The following result shows that this is actually the case.

Proposition 3.23. *For every cfG G it holds: $\mathcal{C}^*[G] = \mathcal{C}[G]$.*

The following lemma will be used in the proof of Proposition 3.23.

Lemma 3.24. *For a cfG $G = (\Sigma, \Gamma, R, T)$ with a target DFA $A(T) = (Q, \Sigma, \delta, q_0, F)$, it holds, for every $a \in \Gamma$ and $q \in Q$:*

$$\mathcal{C}[G](a,q) = \text{Mix}(\{\mathcal{E}[G,w](q) \mid w \in R_a\}).$$

Proof. Since both sides of the claimed equation are minimal sets, it suffices by Lemma 2.1 to show that each element of a set on one side of the equation has a subset on the other side. The proof itself is technical, but straightforward.

($\sqsupseteq$): Let $a \in \Gamma$, $q \in Q$ and $X \in \mathcal{C}[G](a,q)$. By definition of $\mathcal{C}[G]$, there exists a strategy $\sigma \in \text{STRAT}_{\text{J},Call}$ such that $X = \text{states}_G(q,a,\sigma)$. Again by definition, Juliet plays *Call* on a according to σ.

For every choice $w \in R_a$ with which Romeo might respond to Juliet's initial *Call* move on a, there is a sub-strategy σ_w of σ on w. For each $w \in R_a$, let $X_w = \text{states}_G(q,w,\sigma)$. Obviously, each X_w has a subset in $\mathcal{E}[G,w](q)$, and therefore the set $X' = \bigcup_{w \in R_a} X_w$ has a subset in $\text{Mix}(\{\mathcal{E}[G,w](q) \mid w \in R_a\})$. It only remains to be proven that $X' \subseteq X$, so let $q' \in X'$. Then, by the definition of X', there is some $w \in R_a$,

[6] Note that the transition formulas of this AFA may be of exponential size in $|G|$; however, Proposition 2.8 still yields only a polynomial space complexity in $|G|$ here.

strategy $\tau_w \in \mathrm{STRAT_R}$ and $w' \in \Sigma^*$ such that $w' = \mathrm{word}_G(w, \sigma_w, \tau_w)$ and $\delta^*(q, w') = q'$. From the way σ_w was defined from σ, it follows that $w' \in \mathrm{words}_G(a, \sigma)$, and therefore $q' \in X$.

($\sqsubseteq$): Let $a \in \Gamma$, $q \in Q$ and $X \in \textsc{Mix}(\{\mathcal{E}[G, w](q) \mid w \in R_a\})$. Then, for each $w \in R_a$ there exists some set $X_w \in \mathcal{E}[G, w](q)$ such that $X = \bigcup_{w \in R_a} X_w$. By the definition of $\mathcal{E}[G, w]$, this means that for every $w \in R_a$ there is some strategy $\sigma_w \in \mathrm{STRAT_J}$ such that $\mathrm{states}_G(q, w, \sigma_w) = X_w$. Let $\sigma \in \mathrm{STRAT}_{\mathrm{J},\mathit{Call}}$ be the strategy on a where Juliet plays *Call* on a and then, if Romeo picks $w \in R_a$ as a replacement, keeps playing according to σ_w on w. By definition of $\mathcal{C}[G]$, the set $X' = \mathrm{states}_G(q, a, \sigma)$ has a subset in $\mathcal{C}[G](a, q)$, and it only remains to be proven that $X' \subseteq X$. Let therefore $q' \in X'$ Then, there is some strategy $\tau \in \mathrm{STRAT_R}$ and string $w' \in \Sigma^*$ such that $w' = \mathrm{word}_G(a, \sigma, \tau)$. Since Juliet's first move according to σ is a *Call* on a, there is some string $w \in R_a$ which Romeo chooses as a replacement according to τ; by definition of σ, it then holds that $q' \in \mathrm{states}_G(q, w, \sigma_w) = X_w \subseteq X$ as was to be proven. □

Now we are prepared to give the proof of Proposition 3.23.

Proof of Proposition 3.23. For the proof we construct from an initial cfG $G = (\Sigma, \Gamma, R, T)$ a game $G' = (\Sigma, \Gamma, R', T)$, where R' consists of particular *finite* sublanguages $R'_a \subseteq R_a$, for every $a \in \Gamma$. Then we show

(a) $\mathcal{C}^*[G] = \mathcal{C}^*[G']$,

(b) $\mathcal{C}^*[G'] = \mathcal{C}[G']$, and finally

(c) $\mathcal{C}[G'] = \mathcal{C}[G]$.

To construct G', we first restrict the replacement languages of G to finite sets. To that end, for each $a \in \Gamma$, $k \geqslant 1$ and $w \in R_a$, let $v(a, w, k)$ be a string of minimum length such that $\mathcal{E}^k[G, w] = \mathcal{E}^k[G, v(a, w, k)]$ and $v(a, w, k) \in R_a$, and let similarly $v(a, w)$ be a string of minimum length such that $\mathcal{E}[G, w] = \mathcal{E}[G, v(a, w)]$ and $v(a, w) \in R_a$. Furthermore, let $V_a = \{v(a, w) \mid w \in \Sigma^*\}$ and let $W_a = \{v(a, w, k) \mid w \in \Sigma^*, k \geqslant 1\}$. Since there are only finitely many different string effects, all sets V_a and W_a for $a \in \Sigma$ must be finite.

The replacement rules R' for G' are now constructed as follows: For each $a \in \Gamma$, let $R'_a \stackrel{\text{def}}{=} W_a \cup V_a$. By construction, it holds that R'_a is a finite subset of R_a, and an easy induction argument shows that $\mathcal{C}^k[G'] = \mathcal{C}^k[G]$ for each $k \geqslant 1$. Along with the definition of $\mathcal{C}^*[\cdot]$, this proves (a).

For (b) it is sufficient to show that each finite strategy $\sigma \in \mathrm{STRAT_J}[G']$ on a word w has bounded *Call* depth. This can be easily established with the help of Kőnig's Lemma. To this end, we consider the strategy tree $\mathit{Tree}_{G,w}(\sigma)$ for σ on w where each node is a game position of the form (p, u, v) with a player index $p \in \{\mathrm{J}, \mathrm{R}\}$ and strings $u, v \in \Sigma^*$, each node corresponding to a game position κ of Romeo has as children the possible follow-up positions κ' with $\kappa \to \kappa'$, and each node corresponding to a game position κ of Juliet has as its only child the follow-up configuration $\sigma(\kappa)$ according to σ. Each node of this tree has a finite number of children – nodes corresponding to positions belonging to Juliet have only a single child each (as σ is fixed), and positions in which Romeo is to replace some $a \in \Sigma$ have one child for each string in R'_a. Thus, the *Call* depth of nodes

is bounded, as otherwise $\mathit{Tree}_{G,w}(\sigma)$ would be a finitely branching tree with branches of arbitrary length, which by Kőnig's Lemma would yield that T has an infinite branch, contradicting the finiteness assumption for σ.

Towards (c), we prove the slightly stronger claim that for all $q \in Q$ and $w \in \Sigma^*$, it holds $\mathcal{E}[G, w](q) = \mathcal{E}[G', w](q)$. Lemma 3.24 then implies (c). To this end, we prove that each set in $\mathcal{E}[G', w](q)$ has a subset in $\mathcal{E}[G, w](q)$ and vice versa, which proves the desired equality by Lemma 2.1.

One of these directions is almost trivial, as Romeo simply has no more possible moves in G' than in G. Thus, any strategy $\sigma \in \mathrm{STRAT}_{\mathrm{J},\mathit{Call}}(G)$ induces a sub-strategy $\sigma' \in \mathrm{STRAT}_{\mathrm{J},\mathit{Call}}(G')$ with $\mathrm{words}_{G'}(w, \sigma') \subseteq \mathrm{words}_G(w, \sigma)$; for this strategy, it therefore also holds that $\mathrm{states}_{G'}(q, w, \sigma') \subseteq \mathrm{states}_G(q, w, \sigma)$.

For the other direction, let $q \in Q$, $w \in \Sigma^*$ and let $\sigma' \in \mathrm{STRAT}_{\mathrm{J},\mathit{Call}}(G')$ with $X = \mathrm{states}_{G'}(q, w, \sigma') \in \mathcal{E}[G', w](q)$. Let $d \stackrel{\text{def}}{=} \mathrm{Depth}^{G'}(\sigma', w)$. This is well-defined as σ' is finite. We prove by nested induction over d and the structure of w that there exists a strategy σ in G with $\mathrm{states}_G(q, w, \sigma) \subseteq X$, which implies that X has a subset in $\mathcal{E}[G, w](q)$.

If $d = 0$, then Juliet only plays *Read* on the entirety of w; obviously, this strategy is feasible in G as well and yields the same result.

If $d > 0$, Juliet must play *Call* on w at some point, and therefore it holds that $w \neq \epsilon$.

For $w = a \in \Sigma$, if Juliet plays *Read* on a according to σ', we set Σ to be the strategy in G that also plays *Read* on a. Then, clearly, $\mathrm{states}_G(q, a, \sigma) = \{\delta(q, a)\} = \mathrm{states}_{G'}(q, a, \sigma')$. Assume therefore that Juliet plays *Call* on a. We define the strategy σ on a in G to also play *Call*, and we define sub-strategies σ_z for each response $z \in R_a$ by Romeo to this *Call*. For any $z \in R_a$, it holds that $v(a, z) \in R'_a$, so let σ'_z be the sub-strategy of σ' on $v(a, z)$. Since $\mathrm{Depth}^{G'}(\sigma'_z, v(a, z)) < d$, by induction there is a strategy $\sigma_{v(a,z)}$ on $v(a, z)$ in G with $\mathrm{states}_G(q, v(a, z), \sigma_{v(a,z)}) \subseteq \mathrm{states}_{G'}(q, v(a, z), \sigma'_z)$. Furthermore, by definition of $v(a, z)$, it holds that $\mathcal{E}[G, z] = \mathcal{E}[G, v(a, z)]$, and therefore there is a strategy σ_z on z with $\mathrm{states}_G(q, z, \sigma_z) = \mathrm{states}_G(q, v(a, z), \sigma_{v(a,z)})$. By the above inclusions, it follows that $\mathrm{states}_G(q, a, \sigma) \subseteq X$.

For $w = uv$ with $u, v \in \Sigma^*$ and $u, v \neq \epsilon$, let σ'_u be the sub-strategy of σ' on u, and let $\{q_1, \ldots, q_k\} = \mathrm{states}_{G'}(q, u, \sigma'_u)$. For each $i \in [k]$, let further $\sigma'_{v,i}$ be a sub-strategy of σ' on v in case the play on u yields some string u' with $\delta^*(q, u') = q_i$. By induction, there exist strategies σ_u on u and $\sigma_{v,i}$ on v in G such that $\mathrm{states}_G(q, u, \sigma_u) \subseteq \{q_1, \ldots, q_k\}$ and $\mathrm{states}_G(q_i, v, \sigma_{v,i}) \subseteq \mathrm{states}_{G'}(q_i, v, \sigma'_{v,i})$. Let σ be the strategy on uv in G where Juliet plays according to σ_u on u and according to $\sigma_{v,i}$ if the play on u yielded a string u' with $\delta^*(q, u') = q_i$. Then, it holds that $\mathrm{states}_G(q, w, \sigma) \subseteq \bigcup_{i \in [k]} \mathrm{states}_{G'}(q_i, v, \sigma'_{v,i}) \subseteq X$. □

Now we can give a (high-level) proof for the upper bounds in Theorem 3.9, thus completing the proof of Theorem 3.5.

Proof of Theorem 3.9. Let G be a cfG and w a word. By Lemma 3.21, $\mathcal{C}^1[G]$ can be computed in polynomial time from G. For the replay-free case (c), we can immediately construct an AFA $A_{\mathcal{C}^1[G]}$ for $\mathrm{JWin}^1(G)$ and evaluate it on w, yielding a PTIME upper bound by Proposition 2.6.

By Lemma 3.22, each $\mathcal{C}^{k+1}[G]$ can be computed from $\mathcal{C}^k[G]$ in polynomial space; the idea behind the polynomial-space upper bound in (b) is simply to iteratively com-

pute effects up to some given maximum *Call* depth k^*, compute the corresponding AFA $A_{\mathcal{C}^{k^*}[G]}$ from $\mathcal{C}^{k^*}[G]$ and then evaluate it on w. However, some care has to be taken, as the (representation of the) intermediate automata and the resulting automaton $A_{\mathcal{C}^{k^*}[G]}$ can be of more than polynomial (but at most exponential) size. However, as usual for space bounded computations, the information about $A_{\mathcal{C}^{k^*}[G]}$ and the intermediate automata can be recomputed whenever it is needed. The composition of these constantly many polynomial space computations then yields an overall polynomial space bound. It is crucial here that, as observed in Propositions 2.7 and 2.8, evaluation of $A_{\mathcal{C}^{k^*}[G]}$ and nonemptiness tests for all intermediate automata are possible in polynomial space in $|w|$ and the number of states of $A_{\mathcal{C}^{k^*}[G]}$.

For the exponential-time upper bound in (a), since the fixed-point computation of $\mathcal{C}[G]$ terminates after at most exponentially many iterations of computing successive effects, $\mathcal{C}[G]$ can be computed from G in exponential time. Therefore, by Proposition 3.15, the AFA $A_{\mathcal{C}[G]}$ can be computed in time polynomial in $|G|$ and $|\mathcal{C}[G]|$, which is exponential in $|G|$; whether $w \in L(A_{\mathcal{C}[G]})$ holds can then be tested in polynomial time in $|A_{\mathcal{C}[G]}|$ and $|w|$, that is, in at most exponential time in $|G|$ and $|w|$. □

Example 3.25 (Running example)**.** *In the cfG G from Example 3.10, the fixpoint algorithm for computing $\mathcal{C}[G]$ reaches its fixpoint in its third iteration; the intermediate values for $\mathcal{C}^k[G]$ can be seen in the following tables.*

$\mathcal{C}^1[G]$:

	f	g
q_a	$\{\{4\}\}$	$\{\{2\}\}$
q_f	$\{\{3,4\}\}$	$\{\{2,4\}\}$
q_g	$\{\{3,4\}\}$	$\{\{4\}\}$
$q_\perp$	$\{\{4\}\}$	$\{\{4\}\}$

$\mathcal{C}^2[G]$:

	f	g
q_a	$\{\{4\},\{3\}\}$	$\{\{2\},\{4\}\}$
q_f	$\{\{4\}\}$	$\{\{2,4\},\{3,4\}\}$
q_g	$\{\{4\}\}$	$\{\{4\}\}$
$q_\perp$	$\{\{4\}\}$	$\{\{4\}\}$

$\mathcal{C}^3[G] = \mathcal{C}[G]$:

	f	g
q_a	$\{\{4\},\{3\}\}$	$\{\{2\},\{3\},\{4\}\}$
q_f	$\{\{4\}\}$	$\{\{4\}\}$
q_g	$\{\{4\}\}$	$\{\{4\}\}$
$q_\perp$	$\{\{4\}\}$	$\{\{4\}\}$

Since all sets in $\mathcal{C}[G](\cdot,\cdot)$ are singletons, the resulting AFA $A_{\mathcal{C}[G]}$ can be viewed as a NFA, which, after removing the non-accepting sink state $q_\perp$ and all transitions leading into it, looks like the one depicted in Figure 3.2.

As can easily be seen, this NFA decides the language described by the regular expression $a^*(f+g)(\epsilon+f^*g)g^*$; in other words, winning strings need only have a single f or *g (as opposed to at least one f* and *g required by the target language), because* Juliet *can enforce rewriting of either f or g into a string of the form $a^*ff^*gg^*$. Similarly, a single g appearing before all fs can be rewritten into af or f. However,* Juliet *can not enforce a successful rewriting on an initial string having an f to the right of more than one g, as any rewriting of such a string can be enforced by* Romeo *to have an f to the left of an a or a g to the right of an f.*

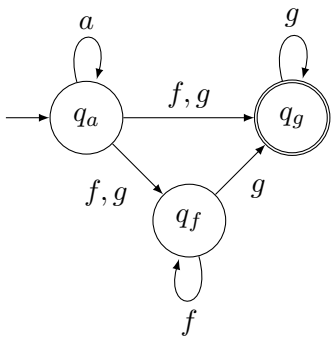


Figure 3.2.: NFA constructed from the AFA $A_{\mathcal{C}[G]}$ from Example 3.25

3.5. Outlook and bibliographical remarks

This chapter gave an overview over complexity results for solving context-free games on strings. It provided a summary of prior work for games with finite replacement languages as well as a complete classification of the complexity of the winning problem in games with regular replacement languages. The interesting difference between the two settings of finite and regular replacement is in the bounded-replay case, where finite replacement leads to a tractable winning problem, while the winning problem with regular replacement languages is complete for polynomial space. For no replay and unbounded replay, the complexities in the finite and regular replacement settings coincide.

Additionally, this chapter introduced the basic proof methods for context-free games: For upper bounds, the abstraction of (sub-)games into *effects*, the transformation of effects into alternating automata and the incremental computation of effects using these automata; and for lower bounds, the idea of reduction from existence problems where Romeo provides a "witness" and Juliet checks that "witness", along with the protest technique. All of these proof methods, most notably the generic algorithm for solving context-free games, will be of further use for games on nested words in Chapter 6.

Bibliographic remarks. The EXPTIME-completeness result for games with unbounded replay and regular replacement languages in Theorem 3.5 was originally proven as Theorem 7.3 in [MSS06]; however, their upper bound proof uses a reduction to parity pushdown games. An effect-based upper bound proof was first given in [BSSK13], and the results for bounded and no replay in Theorem 3.5, as well as matching lower bounds with deterministic regular replacement languages (based closely on those in [MSS06]), were proven in [SS15]; the proofs can also be found in the latter paper's journal version [SS16]. The presentation of upper bound proofs in this chapter, using the generic effect-based algorithm, follows the presentation of the corresponding proofs for context-free games on nested words in [SS15]. A small error in our original version of the PSPACE lower bound (Proposition 3.8) was found and fixed by Krystian Kensy as part of his Master's thesis [Ken14].

The basic idea behind effect-based proofs stems from the diploma thesis of Joscha

Kulbatzki [Kul10], based upon which the concept of effects was developed by the author in collaboration with Henrik Björklund and Thomas Schwentick.

The concept of effects bears some similarity to that of powers [Ben03] or effectivity functions [PP03] for game logics but was developed independently of those sources. Minimisation of effects roughly corresponds to the monotonicity of powers or effectivity functions. However, using, as we do, an inclusion-minimal "basis" instead of a monotonic "upward closure" allows for a more succinct representation and lower complexity in some places.

The more restricted class of *one-pass* games on strings, in which Juliet has to choose her moves in a streaming fashion, without knowing the input string beyond the part she has already read, were first investigated in [AMB05]. This setting introduces incomplete information to context-free games and results for these kinds of games have turned out to be rather elusive. Some progress was made by Christian Cöster in his Master's thesis [Cös15] and, later on, in collaboration with Thomas Schwentick, but as of this writing, no further results have been published.

4. Universality of left-to-right strategies on strings

As the previous chapter showed, in context-free games on strings there is a quite big gap between strategies with and without the left-to-right restriction on the order in which Juliet may choose function symbols to be replaced – solving context-free games is undecidable for unconstrained games, and EXPTIME-complete with the standard left-to-right restriction. The restriction to games with bounded or no replay (and most notably the tractable restrictions) and the regularity of the set JWin of winning strings for Juliet shown in that chapter also assume the left-to-right restriction.

From the practical standpoint of AXML document rewriting, it is generally in our interest to design specifications (i.e. games) in such a manner that we can validate whether all initial document allowed by some input schema can be rewritten into a given target schema. If left steps are not allowed, this simply boils down to checking the inclusion of the input schema in the set of safely rewritable input strings, i.e. to an inclusion of regular languages.

It would be desirable to also be able to find out in this manner whether all initial documents admitted by the input schema can be rewritten into the target schema by *any* rewriting strategy, i.e. whether any safe rewriting capabilities are lost by the restriction to left-to-right strategies. To this end, we now study the question whether, for a given game G, strategies without left steps are *universal*, that is, the following algorithmic problem.

L2RAll	
Given:	A context-free game G.
Question:	Is $\mathrm{JWin}^{\leftarrow}(G) \subseteq \mathrm{JWin}(G)$?

In this section, we prove the following result on the complexity of L2RAll :

Theorem 4.1.

(a) L2RAll *is* EXPSPACE*-complete.*

(b) If all replacement languages are finite and explicitly given in the input, L2RAll *is* EXPTIME*-complete.*

The upper and lower bounds are proven separately as Propositions 4.11 and 4.12 in Section 4.1 for the upper bounds and Propositions 4.13 and 4.16 in Section 4.2 for the lower bounds. The upper bound proofs introduce the concept of *dual effects* as an alternative way to summarise games (similar to the word effects introduced in Chapter

3), while the lower bounds use a particularly intricate example of the protest technique introduced in Section 3.3.

Throughout this chapter, to stress the distinction from strategies with left steps in unconstrained games, we often refer to strategies without left steps as *left-to-right (L2R) strategies.*[1]

4.1. Upper bounds

In this section, we prove the upper bounds in Theorem 4.1. We start by stating some general auxiliary results and introducing the concept of dual effects before giving the upper bound proof for the general case; finally, we examine how the proof for the general case can be adapted to yield (presumably) lower complexity if replacement languages are explicitly enumerated.

The first obvious difficulty in deciding L2RAll is finding out whether there exist a string in $\mathrm{JWin}^{\leftarrow}(G)\backslash\mathrm{JWin}(G)$ *without* having to decide whether a given string is in $\mathrm{JWin}^{\leftarrow}(G)$, as the latter problem is undecidable by Theorem 3.3. Fortunately, it turns out that we can restrict our search to a much smaller class of strategies which also has a decidable winning problem.

Definition 4.2. A strategy σ of Juliet is an *extended left-to-right* (or *L2R*$^+$)-strategy if for every string w and every strategy τ of Romeo, Juliet plays LS at most once and plays at most one *Call* before the LS-move.

We denote the set of strings on which Juliet has a winning L2R$^+$strategy in a game G by $\mathrm{JWin}^{(\leftarrow)}(G)$.

Lemma 4.3. *Let G be a context-free game. Then $JWin^{\leftarrow}(G) = JWin(G)$ if and only if $JWin^{(\leftarrow)}(G) = JWin(G)$.*

The interesting direction of the proof of Lemma 4.3 is the "if" direction, which seems rather obvious when stated as its contrapositive – intuitively, if there exists a string w on which Juliet can win by a strategy σ with arbitrarily many left steps, we can just follow σ on w until we reach some position in which Juliet just has a single *Call* and LS move left before a final left-to-right pass. The main difficulty in the proof is the fact that isolating such a position is not entirely trivial, as both the number of LS moves in a strategy with left steps and the number of replacement strings for Romeo (and thus both the depth and branching factor of the strategy tree) may be unbounded. Consequently, dealing with these problems makes up the main part of the proof.

Proof. If $\mathrm{JWin}^{\leftarrow}(G) = \mathrm{JWin}(G)$, then $\mathrm{JWin}^{(\leftarrow)}(G) = \mathrm{JWin}(G)$ by definition.

Assume that $\mathrm{JWin}^{\leftarrow}(G) \neq \mathrm{JWin}(G)$ and let w be a string in $\mathrm{JWin}^{\leftarrow}(G)\backslash\mathrm{JWin}(G)$. Let σ be a winning strategy for Juliet on w, i.e., starting from the configuration $(\mathrm{J}, w, \epsilon)$.

[1] We note that, technically, left steps are not a feature of strategies but of cfG semantics. However, we can simply interpret all cfGs as unconstrained cfGs and restrict Juliet to strategies without left steps for standard cfGs, which will simplify the presentation of proofs in this chapter (similar to viewing replay as a game parameter, cf. footnote 3 on page 28).

We again consider the strategy tree $\mathit{Tree}_{G,w}(\sigma)$ for σ on w where each node is a game position of the form (p, u, v) with a player index $p \in \{\mathrm{J}, \mathrm{R}\}$ and strings $u, v \in \Sigma^*$, each node corresponding to a game position κ of Romeo has as children the possible follow-up positions κ' with $\kappa \to \kappa'$, and each node corresponding to a game position κ of Juliet has as its only child the follow-up configuration $\sigma(\kappa)$ according to σ. We note that, while nodes corresponding to positions of Juliet have only a single child each, nodes corresponding to positions of Romeo may have an infinite number of children, so $\mathit{Tree}_{G,w}(\sigma)$ is not necessarily a finite tree.

In addition to the configuration labels, we mark each node n in this tree with a value $\mathrm{LS}(n)$, where $\mathrm{LS}(n)$ is the lowest upper bound on the number of LS moves, on any branch of the subtree rooted in n. Since the tree has infinite branching, the value $\mathrm{LS}(n)$ can, in general, be unbounded, i.e., $\mathrm{LS}(n) = \infty$. Since σ is a winning strategy, however, the tree has no infinite branches.

Nodes n with $\mathrm{LS}(n) \neq \infty$ and $\mathrm{LS}(n) > 0$ are also marked by $S^G_{\mathrm{call}}(n)$, the lowest upper bound on the number of *Call* moves that occur before the first LS step, on any branch of the subtree rooted in n. We note that $S^G_{\mathrm{call}}(n)$ might be ∞.

In the following, we call, for nodes n with $\mathrm{LS}(n) \neq \infty$, the pair $(\mathrm{LS}(n), S^G_{\mathrm{call}}(n))$ the *marking* of n and we denote by $\leqslant$ the lexicographic order on markings.

Without loss of generality, we may assume that σ is *optimally efficient* in the following sense. We assume that for every node n of the strategy tree, labelled with a configuration (p, u, v), such that $\mathrm{LS}(n) \neq \infty$, there is no other winning strategy σ' on w, such that the strategy tree for σ' and w has a node n' labelled with the same configuration but having a lexicographically smaller marking. Such an optimally efficient strategy can be constructed for every configuration (p, u, v) by nested induction on the minimal value of $(\mathrm{LS}(n), S^G_{\mathrm{call}}(n))$ that nodes n representing (p, u, v) can assume in winning strategies for (p, u, v).

As $\mathrm{JWin}^{\leftarrow}(G) \neq \mathrm{JWin}(G)$, there must be a node n in $\mathit{Tree}_{G,w}(\sigma)$ with $\mathrm{LS}(n) > 0$.

We first show that $\mathit{Tree}_{G,w}(\sigma)$ must contain nodes n with $\mathrm{LS}(n) > 0$, $\mathrm{LS}(n) \neq \infty$ and with a marking different from $(1, 0)$, i.e. configurations in which Juliet actually has to make at least one more *Call* before her last LS move.

If $\mathit{Tree}_{G,w}(\sigma)$ has nodes with LS-value ∞, it also has a node n', where $\mathrm{LS}(n') = \infty$, but $\mathrm{LS}(n) \neq \infty$, for every child node n of n'. Otherwise, $\mathit{Tree}_{G,w}(\sigma)$ would have infinite branches, contradicting the fact that σ is a winning strategy. There must be arbitrarily large LS-values among the children of n' as otherwise $\mathrm{LS}(n') \neq \infty$. In particular, n' must have a Juliet-grandchild n with $\mathrm{LS}(n) > 1$ and therefore a marking differing from $(1, 0)$.

If $\mathit{Tree}_{G,w}(\sigma)$ has no nodes with LS-value ∞, then for the root r of $\mathit{Tree}_{G,w}(\sigma)$ it holds that $\mathrm{LS}(r) \neq \infty$, and thus $\mathrm{LS}(r) \geqslant 1$ (as otherwise $w \in \mathrm{JWin}(G)$) and $S^G_{\mathrm{call}}(r) > 0$ (as otherwise one LS-step less would suffice — at the root the current position is 1!).

Thus, there must be a Juliet-node n_1 with $\mathrm{LS}(n_1) \geqslant 1$, $\mathrm{LS}(n_1) \neq \infty$ and with a marking different from $(1, 0)$.

Let n be any node with $\mathrm{LS}(n) > 0$, $\mathrm{LS}(n) \neq \infty$ and with a marking $(i, j) \neq (1, 0)$. For the markings of the children and grandchildren of n there are the following possibilities.

(i) Juliet plays *Read* on n and for the unique child n' of n the marking is (i, j).

(ii) JULIET plays *Call* on n, $j = \infty$, and there is a grandchild n' of n with marking (i, ∞).

(iii) JULIET plays *Call* on n, $j = \infty$, there are grandchildren n'' with $\mathrm{LS}(n'') = i$ and markings of the form (i, j') with $j' \neq \infty$ for all such grandchildren. In particular, there is a grandchild n' with marking (i, j'), for some $j' > 0$.

(iv) JULIET plays *Call* on n, $j \neq \infty$, and all grandchildren have markings that are strictly smaller than (i, j), including one child n' with marking $(i, j-1)$.

(v) JULIET plays LS on n, $j = 0$ and the child n' of n has a configuration of the form $(\mathrm{J}, u, \epsilon)$ and marking $(i-1, j')$ with $j' > 0$.

We can construct a sequence $n_1, n_2, \ldots$ of nodes by choosing, in all cases (i)-(v), $n_{i+1} = n'_i$, for $i \geqslant 1$. As this sequence follows a branch of the tree and n_1 is a winning node for σ, the sequence can not be infinite. Furthermore, each leaf has marking $(1, 0)$. Therefore, the sequence must contain a JULIET-node n_ℓ with marking $(1, 1)$. Let (J, x, y) be the configuration of n_ℓ. We claim that $xy \in \mathrm{JWin}^{(\leftarrow)}(G) \backslash \mathrm{JWin}(G)$.

First, $xy \notin \mathrm{JWin}(G)$, as otherwise the marking of n_ℓ would be at most $(1, 0)$ (no *Call* move needed before the LS-step).

On the other hand, as the marking of n_ℓ is $(1, 1)$, starting from $(\mathrm{J}, xy, \epsilon)$, JULIET can play *Read* on x and can win with one *Call* before the one and only LS move, therefore $xy \in \mathrm{JWin}^{(\leftarrow)}(G)$.

Thus, $\mathrm{JWin}^{(\leftarrow)}(G) \neq \mathrm{JWin}(G)$, completing the proof. □

The basic idea behind an algorithm for the upper bound proofs is now quite simple: Guess strings $u, w \in \Sigma^*$ and a function symbol f and then verify that $ufw \notin \mathrm{JWin}(G)$ and that $uvw \in \mathrm{JWin}(G)$ for each $v \in R_f$; clearly, these two conditions hold if and only if $ufw \in \mathrm{JWin}^{(\leftarrow)}(G) \backslash \mathrm{JWin}(G)$. The problem with this approach, however is that an appropriate string u, in particular, may be of doubly exponential length and therefore can only be guessed one symbol at a time in a streaming fashion. We therefore need some way of tracking all relevant information regarding winning strategies for both JULIET and ROMEO on u; to be precise, the proof utilises NFAs for both $\mathrm{JWin}(G)$ and its complement. For JULIET, we can utilise effects (introduced in Chapter 3), whereas for ROMEO, we now examine the concept of *dual effects*.

4.1.1. Dual games and effects

Chapter 3 introduced effects as a means of summarising subgames on substrings. Recall that for a cfG $G = (\Sigma, \Gamma, R, T)$ with a target DFA $A(T) = (Q, \Sigma, \delta, q_0, F)$ and a string $w \in \Sigma^*$, the word effect of G on w was defined as a function $\mathcal{E}[G, w] : Q \to \mathscr{P}(\mathscr{P}(Q))$ mapping each state $q \in Q$ to the set

$$\mathcal{E}[G, w](q) = \left[\{\{\delta^*(q, \mathrm{word}_G(w, \sigma, \tau)) \mid \tau \in \mathrm{STRAT}_{\mathrm{R}}(G)\} \mid \sigma \in \mathrm{STRAT}_{\mathrm{J}}(G)\}\right]_{\min}.$$

Essentially, examining the word effect of G on w makes it possible to abstract any play in G into a single-round game where JULIET first chooses a strategy $\sigma \in \mathrm{STRAT}_{\mathrm{J}}(G)$

(and thus a corresponding set $X \in \mathcal{E}[G,w](q_0)$ of states), then Romeo chooses a counter-strategy $\tau \in \mathrm{STRAT}_{\mathrm{R}}(G)$ (and thus a state $q \in X$), with Juliet winning the game if and only if she wins the game on w with her chosen strategy against Romeo's chosen counter-strategy (i.e. if and only if $q \in F$ holds). This was used in Chapter 3 to determine whether Juliet has a winning strategy on w in G by essentially checking whether there is some $X \in \mathcal{E}[G,w](q_0)$ with $X \subseteq F$.

Since context-free games can be interpreted as reachability games, by Proposition 2.2 it holds that for each starting string exactly one of the players has a winning strategy; this result directly carries over to the abstracted single-round games described above. This, in turn, implies that it does not matter which player chooses their strategy first in the abstraction – the fact that effects are defined in a way that makes Juliet choose her strategy first is merely for convenience in computing effects. In just the same way, we can define word effects in the *dual* abstracted single-round game where it is Romeo who chooses his strategy first.

Definition 4.4. For a cfG $G = (\Sigma, \Gamma, R, T)$ with a target DFA $A(T) = (Q, \Sigma, \delta, q_0, F)$ and a string $w \in \Sigma^*$, the *dual word effect,* $\hat{\mathcal{E}}[G,w] : Q \to \mathscr{P}(\mathscr{P}(Q))$*, of* G *on* w is the function defined by

$$\hat{\mathcal{E}}[G,w](q) \stackrel{\text{def}}{=} [\{\{\delta(q, \mathrm{word}_G(w,\sigma,\tau)) \mid \sigma \in \mathrm{STRAT}_{\mathrm{J}}(G)\} \mid \tau \in \mathrm{STRAT}_{\mathrm{R}}(G)\}]_{\min},$$

for every $q \in Q$.

To stress the distinction from dual effects, we usually refer to non-dual effects as *primal* effects. The dual effect of a string can be obtained from its primal effect via a simple operation, SMix, very similar to the Mix operation defined in Section 3.4. Let $\mathcal{D} = \{D_1, \ldots, D_n\}$ be a set of sets. Then

$$\mathrm{SMix}(\mathcal{D}) = \mathrm{Norm}(\{\{d_1, \ldots, d_n\} \mid d_1 \in D_1 \wedge \cdots \wedge d_n \in D_n\}).$$

In other words, SMix contains all sets that can be formed by **s**electing one element from each of the elements of $\mathcal{D}$. Notice that, while Mix takes a set of sets of sets and returns a set of sets, SMix takes a set of sets and returns a set of sets.

Lemma 4.5. *Let* w *be a string and* $q \in Q$ *a state of* $A(T)$*. Then* $\hat{\mathcal{E}}[G,w](q) = \mathrm{SMix}(\mathcal{E}[G,w](q))$.

Proof. As both sets are normal it suffices, thanks to Lemma 2.1, to show that for every $\hat{X} \in \hat{\mathcal{E}}[G,u](q)$ there is some $X \in \mathrm{SMix}(\mathcal{E}[G,w](q))$ such that $X \subseteq \hat{X}$, and vice versa.

Let $\hat{X} \in \hat{\mathcal{E}}[G,w](q)$. By definition, there is a strategy $\tau \in \mathrm{STRAT}_{\mathrm{R}}(G)$ of Romeo such that $\hat{X} = \{\delta(q, \mathrm{word}_G(w,\sigma,\tau)) \mid \sigma \in \mathrm{STRAT}_{\mathrm{J}}(G)\}$. This means that for every $\sigma \in \mathrm{STRAT}_{\mathrm{J}}$ there is a a state in $\mathrm{states}_G(q,w,\sigma)$ that also belongs to $\hat{X}$. In particular, there is an element X in $\mathrm{SMix}(\mathcal{E}[G,w](q))$ such that $X \subseteq \hat{X}$.

For the other direction, we use the concept of *game trees*. The game tree $\mathit{Tree}_{G,w}$ on w is a (possibly infinite) tree where each node is a game position of the form (p,u,v) (as defined in Definition 3.1), and each node corresponding to a game position κ has as

children the possible follow-up positions κ' such that $\kappa \to \kappa'$. The *strategy tree* $\mathit{Tree}_{G,w}(\sigma)$ of a strategy σ for Juliet (resp. $\mathit{Tree}_{G,w}(\tau)$ of a strategy τ for Romeo) is the restriction of $\mathit{Tree}_{G,w}$ where all nodes corresponding to a position κ for Juliet (resp. Romeo) have as their only child the node corresponding to position $\sigma(\kappa)$ (resp. $\tau(\kappa)$).

Let $X \in \mathrm{SMix}(\mathcal{E}[G,w](q))$. By definition of $\mathcal{E}[G,w](q)$ and SMix, for every finite strategy σ of Juliet there is a strategy τ of Romeo such that $\delta^*(q,v) \in X$, where $v = \mathrm{word}_G(w,\sigma,\tau)$.

Let $t = \mathit{Tree}_{G,w}$ be the (full, i.e. possibly infinite) game tree for w. Let L_X be the set of leaves of t that are labelled by configurations $(\mathrm{J}, v, \epsilon)$ with $\delta^*(q,v) \in X$. Let S_X be the set of nodes n of t such that for every (not necessarily finite) strategy σ of Juliet, the subtree of the strategy tree $\mathit{Tree}_{G,w}(\sigma)$ rooted in n has an infinite branch or a leaf in L_X.

The root of t must belong to S_X. Otherwise, Juliet would have a finite strategy σ such that no strategy of Romeo yields a state in X, contradicting the above statement about X. Furthermore, if a node in t belongs to S_X and is labelled by a configuration where Juliet is to move, then all its children belong to S_X. If a node in t belongs to S_X and Romeo is to move, then at least one of its children belongs to S_X. We can define a strategy τ for Romeo that from a node in S_X always selects a child node in S_X. In the strategy tree $\mathit{Tree}_{G,w}(\tau)$, every node belongs to S_X. This immediately implies that $\{\delta^*(q, \mathrm{word}_G(w,\sigma,\tau)) \mid \sigma \in \mathrm{STRAT}_{\mathrm{J}}(G)\} \subseteq X$. □

From the definition of dual effects, it is easy to see that (similar to primal effects) Romeo has a winning strategy on w in G if and only if there is some $X \in \hat{\mathcal{E}}[G,w](q_0)$ with $X \subseteq Q \backslash F$. This similarity can be generalised to the following relationship between primal and dual effects.

Lemma 4.6. *For any $w \in \Sigma^*$, $q \in Q$ and $X \subseteq Q$, it holds that X has a subset in $\mathcal{E}[G,w](q)$ if and only if $Q \backslash X$ does not have a subset in $\hat{\mathcal{E}}[G,w](q)$.*

Proof. For the "only if" direction, let $X' \subseteq X \subseteq Q$ with $X' \in \mathcal{E}[G,w](q)$. Using the fact that $\hat{\mathcal{E}}[G,w](q) = \mathrm{SMix}(\mathcal{E}[w](q))$ (Lemma 4.5), by the definition of the SMix operator, each set in $\hat{E}[w](q)$ contains at least one element from X', and therefore from X. Thus, no set in $\hat{\mathcal{E}}[G,w](q)$ can be contained in the complement of X, which proves this direction of the claim.

For the contraposition of the "if" direction, assume that X does not have a subset in $\mathcal{E}[G,w](q)$. Therefore, each set in $\mathcal{E}[G,w](q)$ contains at least one element from $Q \backslash X$. Again using Lemma 4.5 and the definition of SMix, this implies that there is a set in $\mathrm{SMix}(\mathcal{E}[w](q)) = \hat{\mathcal{E}}[G,w](q)$ composed solely of elements from $Q \backslash X$, i.e. that $Q \backslash X$ has a subset in $\hat{\mathcal{E}}[G,w](q)$, as was to be proven. □

If we are solely interested in deciding whether Romeo has a winning strategy on some string w in G (against strategies for Juliet without left steps), Algorithm 1 already yields an exponential-time decision algorithm – compute the call effect $\mathcal{C}[G]$ from G, compute the AFA $A_{\mathcal{C}[G]}$ from $\mathcal{C}[G]$ and accept if and only if $A_{\mathcal{C}[G]}$ does not accept in input w. However, the connection between $A_{\mathcal{C}[G]}$ and dual effects of G goes even deeper than that, as the following analogue of Lemma 3.20 shows.

Lemma 4.7. *Let G be a game, let $A_{\mathcal{C}[G]} = (Q, \Sigma, \delta_C, q_0, F)$ be the AFA deciding JWin(G) as defined in Section 3.4.2, and let $\hat{A}_{\mathcal{C}[G]}$ be the AFA obtained from $A_{\mathcal{C}[G]}$ by the complement construction (Lemma 2.4). Let further $q \in Q$, $w \in \Sigma^*$ and $X \subseteq Q$. Then, $X \in \hat{\mathcal{E}}[G, w](q)$ if and only if there is a minimal run of $\hat{A}_{\mathcal{C}[G]}$ on w starting at q and ending in states from X.*

Proof. By Lemma 4.6, X has a subset in $\hat{\mathcal{E}}[G, w](q)$ if and only if $Q \backslash X$ has no subset in $\mathcal{E}[G, w](q)$. By Lemma 3.20, this is the case if and only if there is no run of $A_{\mathcal{C}[G]}$ on w starting at q and ending inside $Q \backslash X$, which, by the proof of Lemma 2.4, is equivalent to the existence of a run of $\hat{A}_{\mathcal{C}[G]}$ on w starting at q and ending inside X. Minimality of runs and minimisation of effects then yield the claim. □

Using the technique for transforming AFAs into equivalent NFAs from Lemma 2.5, we obtain the following result.[2]

Corollary 4.8. *Let $G = (\Sigma, \Gamma, R, T)$ be a context-free game with target language DFA $A(T) = (Q, \Sigma, \delta, q_0, F)$. Then, there exist NFAs A_{J} and A_{R} with state set $\mathscr{P}(Q)$ such that for each $w \in \Sigma^*$, A_{J} (respectively A_{R}) has a run on from $\{q\}$ to $X \subseteq Q$ if and only if $X \in \mathcal{E}[G, w](q)$ (respectively $X \in \hat{\mathcal{E}}[G, w](q)$).*

4.1.2. Algorithms for L2RAll

We first prove the upper bound for Theorem 4.1(a). As mentioned above, the basic proof idea is giving an algorithm for the complement of L2RAll that guesses nondeterministically strings $u, w \in \Sigma^*$ and a function symbol f such that $ufw \notin \mathrm{JWin}(G)$, but $uvw \in \mathrm{JWin}(G)$ for each $v \in R_f$. The main difficulty in the proof lies in the fact that we need *all* relevant strategy information for Juliet about the string u, because her *Call* move to f in a L2R$^+$strategy occurs *before* the play on u, and Juliet's strategy on u may therefore depend on the call result $v \in R_f$. This poses a problem because the minimal length of a string u yielding a desired effect $\mathcal{E}[G, u](q_0)$ may be doubly exponential in $|Q|$ and the string u can therefore not be explicitly guessed and stored in exponential space.

To solve this problem, we guess u in a streaming fashion, one symbol at a time, and use the NFA A_{J} from Corollary 4.8 to compute $\mathcal{E}[G, u](q_0)$ as we go along, similar to the standard implicit powerset construction used for the word problem for DFAs.

The strings v and w, on the other hand, will be handled much more easily using two auxiliary results that we prove before giving the full upper bound proof.

For any string w, let $F(w) = \{q \in Q \mid \mathcal{E}[G, w](q) \cap \mathscr{P}(F) \neq \varnothing\}$ be the set of states from which Juliet has a winning strategy (without left steps) on w.

For a state q and a set S of states let $A_{\mathrm{J}}^{q,S}$ denote the automaton that is obtained from A_{J} by choosing $\{q\}$ as initial state and $\mathscr{P}(S)$ as set of accepting states.

[2] Technically, the transformation to NFA is not required for the upper bound proofs in Section 4.1.2, but it somewhat simplifies the presentation of the proofs.

Lemma 4.9. *For every state q and $w \in \Sigma^*$ the automaton $A_{\mathrm{J}}^{q,F(w)}$ accepts exactly the strings v such that there is a winning strategy for* Juliet *on vw starting at state q in $A(T)$.*

Lemma 4.9 follows immediately from Corollary 4.8.

For a state $q \in Q$ let G_q denote the game obtained from G by choosing the state q as initial state of the target automaton; furthermore, denote the state set of the replacement language NFA for R_f by Q_f.

Lemma 4.10. *Let $q \in Q$, $w \in \Sigma^*$ and $f \in \Gamma$. If there is a string $v \in R_f$ such that $vw \in JWin(G_q)$ then there is a string v' of length at most $|Q_f| \cdot 2^{|Q|}$ such that $v'w \in JWin(G_q)$.*

Proof. This follows from Lemma 4.9 and a standard pumping argument for the product automaton B combining A_{J} and $A(R_f)$: For any two states $(X_1, p_1), (X_2, p_2) \in \mathscr{P}(Q) \times Q_f$ there is a string v with $\delta_B((X_1, p_1), v) = (X_2, p_2)$ if and only if there is such a string v' of length at most $|\mathscr{P}(Q) \times Q_f| = |Q_f| \cdot 2^{|Q|}$. □

We are now ready to prove the upper bound of Theorem 4.1(a) in full.

Proposition 4.11. L2RAll $\in$ EXPSPACE.

Proof. We give a nondeterministic exponential-space algorithm $\mathcal{A}$ deciding $\overline{\text{L2RAll}}$, the complement of L2RAll . This yields the result since EXPSPACE is closed under complement and NEXPSPACE = EXPSPACE thanks to Savitch's Theorem [Sav70].

The idea is that $\mathcal{A}$ guesses a symbol $f \in \Gamma$ and strings u, w such that $ufw \in \mathrm{JWin}^{(\leftarrow)}(G) \backslash \mathrm{JWin}(G)$ is a witness string on which Juliet plays *Call* on f in the first pass on ufw. Thanks to Lemma 4.10, $\mathcal{A}$ only needs to verify that, for all replacement strings $v \in R_f$ of length at most $|Q_f| \cdot 2^{|Q|}$, it holds that $uvw \in \mathrm{JWin}(G)$. A short summary of $\mathcal{A}$ is given as Algorithm 2.

The main challenge is that the string u may in general be of doubly exponential length and therefore cannot be stored.

Therefore, to compute the set $\mathcal{U} = \mathcal{E}[G, u](q_0) = \{U_1, U_2, \ldots, U_n\}$ as used in Algorithm 2, $\mathcal{A}$ guesses u in a streaming fashion, one symbol at a time. It simulates A_{J} on u and computes $\mathcal{E}[G, u](q_0)$ online. This can be done in exponential space by storing the set $\mathcal{E}[G, u](q_0) \in \mathscr{P}(\mathscr{P}(Q))$. At the same time, having guessed a set $\hat{X}_{uf}$, it guesses a run of A_{R} on uf, effectively verifying that there is a strategy for Romeo allowing him to enforce one of the states from $\hat{X}_{uf}$ in the sub-play on uf (against a strategy of Juliet without left steps).

Afterwards, to compute $F(w) \in \mathscr{P}(Q)$, $\mathcal{A}$ guesses a string w, and incrementally computes a set $F(w) \subseteq Q$ of states from which Juliet can win the game as defined in Lemma 4.9. For $a \in \Sigma$, $w' \in \Sigma^*$, the set $F(aw')$ can be computed from the set $F(w')$ by checking, for each $q \in Q$, whether $A_{\mathrm{J}}^{q,F(w')}$ accepts a. As there are only exponentially many subsets of Q it is not hard to prove by a standard pumping argument that w can be chosen to be of exponential size and that its computation can be actually carried out in polynomial space. With $F(\epsilon) = F$ the correctness of this incremental computation follows by a simple induction argument.

Algorithm 2 Test for $G \in$ L2RAll

```
1:  Guess f ∈ Γ and a set of states X̂_uf ⊆ Q
2:  while Guessing a string u in a streaming fashion do
3:    Use A_J to compute the set U = E[G, u](q_0)
4:    Use A_R to verify X̂_uf ∈ Ê[G, uf](q_0)
5:  Guess a string w and compute F(w) by simulating A_J backwards
6:  if X̂_uf ∩ F(w) = ∅ then
7:    // ufw ∉ JWin(G)
8:    for all v ∈ R_f with |v| ⩽ |Q_f| · 2^|Q| do
9:      Guess a set U_i ∈ U
10:     for all q ∈ U_i do
11:       Simulate A_J^{q,F(w)} on input v
12:       if A_J^{q,F(w)} accepts v then
13:         // uvw ∈ JWin(G)
14:       else
15:         Reject
      Accept
16: Reject
```

The algorithm $\mathcal{A}$ then checks whether $\hat{X}_{uf}$ contains a state from $F(w)$. If it does not, we know that $ufw \notin \mathrm{JWin}(G)$. If it does, $\mathcal{A}$ immediately rejects.

Finally, $\mathcal{A}$ checks for all strings $v \in R_f$ of length at most $|Q_f| \cdot 2^{|Q|}$, whether $uvw \in \mathrm{JWin}(G)$. This can be done by (1) cycling through all strings v of this length, (2) checking if $v \in R_f$ by simulating $A(R_f)$ on v and (3) in case $A(R_f)$ accepts v, guessing a set $U_i \in \mathcal{U}$ and testing whether for every $q \in U_i$ there is a set $X \in \mathcal{E}[G, v](q)$ such that $X \subseteq F(w)$.

To perform test (3), $\mathcal{A}$ simulates, for each $q \in U_i$, a run of $A_{\mathrm{J}}^{q,F(w)}$ on v. This can be done in polynomial space. If the run for each q succeeds, $\mathcal{A}$ concludes that $uvw \in \mathrm{JWin}(G)$, otherwise it rejects.

Altogether, $\mathcal{A}$ only requires exponential space; it remains to show that $\mathcal{A}$ accepts iff $\mathrm{JWin}^{(\leftarrow)}(G) \backslash \mathrm{JWin}(G) \neq \varnothing$.

If $\mathcal{A}$ accepts, then there exists a string ufw such that (a) $ufw \notin \mathrm{JWin}(G)$ (this follows directly from Lemma 4.9) and (b) for all $v \in R_f$ of length at most $|Q_f| \cdot 2^{|Q|}$ there exists a set $U_i \in \mathcal{E}[G, u](q_0)$ such that v is accepted by $A_{\mathrm{J}}^{q,F(w)}$ for all $q \in U_i$.

With Lemmas 4.9 and 4.10, it follows from (b) that for every $v \in R_f$ there is a strategy σ_v of Juliet on u such that for all states $q \in \mathrm{states}_G(q_0, u, \sigma_v)$, Juliet has a winning strategy on vw starting at q.

This yields a winning L2R$^+$strategy for Juliet on ufw: In the first pass, Juliet calls f. On the second pass, depending on Romeo's choice of v, Juliet plays according to σ_v on u and is guaranteed to reach a state starting from which she has a winning strategy on vw.

For the "only if" part, assume $\mathrm{JWin}^{(\leftarrow)}(G) \backslash \mathrm{JWin}(G) \neq \varnothing$ holds. Then there exists a word on which Juliet has a winning L2R$^+$strategy, but no winning L2R strategy. This

word must be of the form ufw with f being the symbol Juliet calls on her first pass for some winning L2R$^+$strategy σ. In lines 1 through 4, $\mathcal{A}$ guesses this word.

Since Juliet has no winning L2R strategy on ufw, Romeo must have a strategy τ on uf enforcing a set of states $\hat{X}_{uf} = \{\delta^*(q_0, \mathrm{word}_G(uf, \sigma, \tau)) \mid \sigma \in \mathrm{STRAT}_\mathrm{J}(G)\}$ such that $\hat{X}_{uf} \cap F(w) = \varnothing$. Since this set of states can be guessed by $\mathcal{A}$, the test on line 6 can be passed.

Let σ_v be Juliet's strategy on u in case Romeo replaces f by $v \in R_f$ and $U_i = \mathrm{states}_G(q_0, u, \sigma) \in \mathcal{E}[G, u](q_0)$. Since σ is winning on uvw, Juliet has a winning strategy on vw starting at q for any $q \in U_i$. Using Lemma 4.9, this means that for any $v \in R_f$, $\mathcal{A}$ can guess a set $U_i \in \mathcal{E}[G, u](q_0) = \mathcal{U}$ on line 9 such that all $A_\mathrm{J}^{q,F(w)}$ accepts v for $q \in U_i$. This condition is checked in lines 10 through 15, and since it is fulfilled for all $v \in R_f$, $\mathcal{A}$ accepts. □

For games G with explicitly enumerated replacement languages, this algorithm can be modified to run in exponential time in $|G|$.

Proposition 4.12. *If all replacement languages are finite and explicitly given in the input,* L2RAll *is in* EXPTIME.

Proof. We modify Algorithm 2 such that it runs in exponential time. This works because the only NFAs of doubly exponential size that Algorithm 2 uses can be replaced by NFAs of exponential size, if the replacement sets R_f are finite and explicitly given in the input.

Algorithm 2 uses nondeterminism for two kinds of purposes: for guessing sets of states contained in a given effect, and for guessing strings. The latter can be delegated to standard polynomial space non-emptiness tests for exponential size automata, while the former can be done by cycling through all possible candidates (as there are always only exponentially many).

To this end, the algorithm $\mathcal{A}'$ contains an outer loop over all $f \in \Gamma$, sets $W \subseteq Q$ and vectors of sets $U_1, \ldots, U_{|R_f|} \in \mathscr{P}(Q)$. Inside this loop, similar to algorithm 2, $\mathcal{A}'$ checks if there are strings u and w such that $U_1, \ldots, U_{|R_f|} \in \mathcal{E}[G, u](q_0)$ and $W = F(w)$; then, all $\mathcal{A}'$ needs to do is check for all $i = 1, \ldots, |R_f|$ whether $\delta_\mathrm{J}^*(U_i, v_i) \cap \mathscr{P}(F(w)) \neq \varnothing$ (where δ_J is the transition relation of A_J, and $R_f = \{v_1, \ldots, v_{|R_f|}\}$) and A_R accepts ufw.

To verify the existence of a string u with $U_1, \ldots, U_{|R_f|} \in \mathcal{E}[G, u](q_0)$, $\mathcal{A}'$ computes the product automaton of $|R_f|$ copies of A_J and checks whether the product state $(U_1, \ldots, U_{|R_f|})$ is reachable in polynomial space (and thus exponential time).

To find a string w with $W = F(w)$, $\mathcal{A}'$ computes the product automaton with one copy of ${A_\mathrm{J}}^{q,F}$, for each $q \in W$; again, the verification of the existence of w is by a non-emptiness test.

Finally, $\mathcal{A}$ runs one copy of A_J with starting state U_i and final state set $\mathscr{P}(W)$ on v_i for each $i = 1, \ldots, |R_f|$ and runs A_R on ufw; if all copies of A_J and A_R accept, $\mathcal{A}$ accepts, since a separating string ufw has been found. The correctness of this algorithm follows similar to the proof of Proposition 4.11, and since it loops an exponential number of times and takes no more than exponential time in each iteration, $\mathcal{A}'$ is an EXPTIME algorithm deciding L2RAll for games with finite replacement languages that are given explicitly as part of the input. □

4.2. Lower bounds

In this subsection, we prove the lower bounds in Theorem 4.1, showing that L2RAll is EXPSPACE-hard in general and EXPTIME-hard with finite replacement languages that are explicitly enumerated as part of the input.

4.2.1. Regular replacement languages

Proposition 4.13. L2RAll *is hard for* EXPSPACE.

The proof of this lower bound does follow some of the standard approaches for lower bounds explained in Section 3.3: reducing from a tiling problem and using the protest technique. It is, however, somewhat unusual in that it casts Juliet as the player who is supposed to show that a valid tiling exists, while Romeo is supposed to find flaws in a given tiling. As mentioned in Section 3.3, the distribution of roles in hardness proofs for JWin is generally the other way around, with Romeo trying to verify and Juliet trying to falsify the existence of a valid tiling.

This reversal of roles comes from the fact that, unlike JWin, L2RAll is itself an existence problem. While in hardness proofs for JWin, Romeo usually has to *construct* a valid tiling from a given input string, here, existence of a valid tiling should be equivalent to the existence of a string on which Juliet has a winning strategy with, but not without, left steps. This makes it very natural to put Juliet into the role of verifier, basically tasking her to use a strategy without left steps to prove the validity of a given tiling (against Romeo's objections) and extending the input string in such a way that a single additional call and left step are required.

Proof. We prove the EXPSPACE hardness of L2RAll by reduction from the Exponential Corridor Tiling problem as defined in Section 2.5. This problem is well known to be EXPSPACE-complete; see, e.g., [Chl86; EB97].

Given an input instance $\mathcal{I} = (U, V, H, u_i, u_f, n)$ for Exponential Corridor Tiling, we construct from $\mathcal{I}$ a context-free game $G = (\Sigma, \Gamma, R, T)$ such that there exists a valid corridor tiling for $\mathcal{I}$ if and only if there is a string for which Juliet has a winning L2R$^+$strategy but no L2R strategy in G. The claim then follows from this reduction by Lemma 4.3.

The rough idea of the construction of G is as follows. Let 2^n be the target width for a tiling. Tilings are encoded by strings of the form $v = ((uc)^*)\#)^*$, where u is a tile and c a 0-1-string of length n that should encode the column number of the position of u. A sequence $(uc)^*$ encodes a row of a tiling and rows are separated by $\#$. For each column number $i \in \{0, 1, \ldots, 2^n - 1\}$, we denote by $c(i)$ the encoding of i as a binary string of length n over $\{0, 1\}$.

We will construct G in such a way that the strings in $\mathrm{JWin}^{(\leftarrow)}(G) \backslash \mathrm{JWin}(G)$ are of the form gvf, where v is the encoding of a correct tiling.

The main task of Juliet in the game is to show that the middle part v of the input string indeed represents a correct tiling, while Romeo tries to disprove her. For this, we

utilise a protest technique, in which we force Juliet to call potentially inconsistent symbols in the input, allowing Romeo to flag constraint violations. The additional symbols f and g are primarily meant to ensure that Juliet needs a L2R$^+$strategy to win; f will also be needed to identify violations of vertical constraints, as we shall describe later.

Before giving G in formal detail, we shall first describe the ways in which a string v of the form $(U0^n(U\{0,1\}^n)^*U1^n\#)^*$ may fail to encode a valid tiling. After that, we examine how to deal with these types of violations.

- *Horizontal error:* v violates the horizontal constraints, i.e. v contains a substring of the form $u\{0,1\}^n u'$ with $(u,u') \notin H$;
- *Constant error:* The first (last) symbol from U in v is not u_i (u_f);
- *Increment error:* Two subsequent column number encodings are inconsistent, i.e. v contains a substring of the form $c(i)Uc(j)$ with $j \neq i+1$;
- *Vertical error:* v violates the vertical constraints, i.e. v contains a substring of the form $uc(i)(U \cup \{0,1\})^*\#(U \cup \{0,1\})^*u'c(i)$ with $(u,u') \notin V$ for some $i \leqslant 2^n - 1$.

We will construct G such that Romeo can win without any effort on inputs with horizontal or constant errors and by pinpointing positions with increment or vertical errors otherwise. Horizontal and constant errors can be basically tested by the target DFA, so we merely need to make certain that strings with these kinds of errors can never be rewritten into the target language.

In the main part of the game, during the second pass, Juliet calls all positions of tiling symbols and gives Romeo the possibility to mark a violating position. If v contains an increment error at some position, Romeo can prove this with a simple subgame. Verifying vertical errors is slightly more complicated. To this end, Juliet has to allow Romeo to add an n-digit number c_f to the end of v in the single move of the first pass. Romeo should pick c_f as the encoding of the number of a column in which a vertical error occurs. In the main part, Romeo can then indicate the positions of the two tiles of that error and in a subgame it is verified that they are actually in the same column (with number c_f) on consecutive rows.

We force Juliet to call all positions of tiling symbols by introducing into the alphabet a disjoint copy $\hat{U}$ of U, the set of *marked tiles*, as well as a *protest symbol* @. The idea is that for as long as Juliet keeps calling tile positions in order, Romeo replaces those tiles with their corresponding marked tiles, but as soon as Juliet skips a tile, Romeo protests by returning @ the next time Juliet plays a call move. By including only appropriate strings in the target language, we make sure that Romeo wins on strings on which Juliet has tried to "cheat" by skipping a tile and Romeo has rightfully protested, and that Romeo loses on strings on which he protests without just cause.

Increment errors are dealt with in a similar manner by use of a *number protest symbol* $@_N$. As soon as Juliet calls the tile position immediately before the violating substring $c(i)Uc(j)$, Romeo returns $@_N$, signifying that Juliet now has to call each of the n bits to the right of $@_N$ in turn until Romeo returns a *flag bit* 0_N or 1_N to pinpoint a position

in $c(i)$ that witnesses $j \neq i+1$. (The correctness of this flagging procedure will follow from Lemma 4.15 below.) Similarly as for tiles, we use additional *marked bits* $\hat{0}, \hat{1}$ and the protest symbol @ to force JULIET to call each position of $c(i)$.

Finally, to handle vertical errors, we add another disjoint copy U^V of U, called *flagged tiles* to the alphabet. As described above, after giving the encoding c_f of a column where vertical constraints are violated, ROMEO replaces two tiles involved in this violation by their corresponding flagged tiles. Again, we need to make sure via the target language that ROMEO always wins on rewritten strings with correctly flagged vertical errors and loses on strings with incorrect claims of vertical errors.

Now, we can begin constructing the game $G = (\Sigma, \Gamma R, F)$ from the tiling input $\mathcal{I} = (U, u_i, u_f, V, H, n)$ according to the ideas laid out above

The alphabet Σ consists of the union of $U = \{u_1, \ldots, u_k\}$ with two disjoint copies of U, called $\hat{U}$, and U^V with symbols $\hat{u}_1, \ldots, \hat{u}_k$ and $u_1^V, \ldots, u_k^V$, respectively and the additional symbols $0, 1, \hat{0}, \hat{1}, 0_N, 1_N, \#, @, @_N, f, g, g', h, h'$. To make the definition of T somewhat more concise, we also give names to several subsets of Σ:

- the set of *base symbols* $\Sigma_B = U \cup \{0, 1, \#\}$;
- the set of *processed symbols* $\Sigma_P = \hat{U} \cup \{0, 1, \#\}$;
- the set of *extended bits* $\hat{B}_N = \{0, 1, \hat{0}, \hat{1}, 0_N, 1_N\}$.

The set R consists of the following replacement rules:[3]

$$\begin{array}{rcl} g & \to & g' \\ u_1 & \to & \hat{u}_1 \mid u_1^V \mid @ \mid @_N \\ & \vdots & \\ u_k & \to & \hat{u}_k \mid u_k^V \mid @ \mid @_N \\ 0 & \to & \hat{0} \mid 0_N \mid @ \\ 1 & \to & \hat{1} \mid 1_N \mid @ \\ f & \to & \{0,1\}^n\{h, h'\} \mid @ \end{array}$$

The target language T is the union of several languages described below. To improve readability, we give these languages as regular expressions, but it is easy to verify that a DFA of polynomial size in $|\mathcal{I}|$ accepts each of them. In these regular expressions, we use the abbreviations $(\alpha)_{=n}$ and $(\alpha)_{<n}$ to denote strings of length exactly n (respectively less than n) described by the regular expression α. Again, it is easy to verify that the size of a DFA for $(\alpha)_{=n}$ and $(\alpha)_{<n}$ is at most n times the size of a DFA for α.

As further shorthand notation,

[3]Note that, even though all replacement languages are finite, Proposition 4.12 does not apply here, since the 2^{n+1} replacement strings of the replacement rule $f \to \{0,1\}^n\{h, h'\}$ are not given explicitly but by a DFA of size $O(n)$.

- let W denote the set of all *well-formed tiling encodings*, which are strings of the form $(U0^n(U\{0,1\}^n)^*U1^n\#)^*$;
- let E denote the set of *horizontally correct encodings*, which is obtained by taking the complement of the language described by $\bigcup_{(u,u')\notin H} \Sigma_B^* u\{0,1\}^* u'\Sigma_B^*$; and
- let C denote the set of all *tiling candidates* which are strings that belong to W (i.e. have column numbers of length n that start at 0 and end at 2^n-1 in each line), are also in E (i.e. satisfy the horizontal constraints) and have u_i as their first and u_f as their last tile.

Let $\hat{W}, \hat{E}, \hat{C}$ be defined as W, E and C, but with $\hat{U}$ in place of U.

It is straightforward to construct a polynomial-size DFA for the set C.

With these notations, we define the target language T as the union of the following languages (a short intuitive description follows below):

(1) $gC@$

(2) $g\hat{C}\{0,1\}^n h + g'\hat{C}\{0,1\}^n h'$

(3) $(g+g')(\Sigma_P + U^V)^*@\Sigma_B^*(h+h')$

(4) $(g+g')(\Sigma_P + U^V)^*@_N F_I \Sigma_B^*(h+h')$, where F_I is the language described by

$$(\hat{0}+\hat{1})^*0_N(1^*U(0+1)^*)_{=n}10^*(U+\#)+ \quad \text{(i)}$$
$$(\hat{0}+\hat{1})^*1_N(1^*U(0+1)^*)_{=n}00^*(U+\#)+ \quad \text{(ii)}$$
$$(\hat{0}+\hat{1})^*0_N(0(0+1)^*U(0+1)^*)_{=n}0(0+1)^*(U+\#)+ \quad \text{(iii)}$$
$$(\hat{0}+\hat{1})^*0_N((0+1)^*U(0+1)^*)_{=n}01(0+1)^*(U+\#)+ \quad \text{(iv)}$$
$$(\hat{0}+\hat{1})^*1_N(0(0+1)^*U(0+1)^*)_{=n}1(0+1)^*(U+\#)+ \quad \text{(v)}$$
$$(\hat{0}+\hat{1})^*1_N((0+1)^*U(0+1)^*)_{=n}11(0+1)^*(U+\#)+ \quad \text{(vi)}$$
$$(\hat{0}+\hat{1})^*(0_N+1_N)(0+1)^*\#+ \quad \text{(vii)}$$
$$(\hat{0}+\hat{1})^*\hat{0}(\hat{0}+\hat{1})^*U+\hat{1}^*\# \quad \text{(viii)}$$

(5) $(g+g')\Sigma_P^*(U^V\Sigma_P^*)^?(U^V(0+1)^* + @_N(\hat{0}+\hat{1})^*)@\Sigma_B^*(h+h')$

(6) $(g+g')(\Sigma_P^* U^V \hat{B}_N^*)^3\Sigma_B^*(h+h')$

(7) $(g+g')\Sigma_P^* U^V \hat{B}_N^*(\hat{U}+\#)\Sigma_P^*(h+h')$

(8) $\bigcup_{(u,v)\in V}(g+g')\Sigma_P^* u^V(\Sigma_P+\hat{B}_N)^* v^V\Sigma_B^*(h+h')$

(9) $(g+g')\Sigma_P^* U^V(\Sigma_P\backslash\{\#\})^*((\#(\Sigma_P\backslash\{\#\})^*)^{2+})^? U^V\Sigma_B^*(h+h')$

(10)

$$\begin{aligned}&\big\{(g+g')\Sigma_P^* U^V (0+1)^*(\hat{0}+0_n)\,[((\Sigma_P\backslash\{\#\})^n+\\&(\Sigma_P^*\#\Sigma_P^*)_{=n+1})(0+1)]^*\,1(0+1)_{<n-1}(h+h')\big\}+\\&\big\{(g+g')\Sigma_P^* U^V (0+1)^*(\hat{1}+1_n)\,[((\Sigma_P\backslash\{\#\})^n+\\&(\Sigma_P^*\#\Sigma_P^*)_{=n+1})(0+1)]^*\,0(0+1)_{<n-1}(h+h')\big\}\end{aligned}$$

Recall that the purpose of this construction is to allow Juliet to win in G with a L2R$^+$but not a L2R strategy on exactly the strings gvf where v encodes a valid tiling for $\mathcal{I}$. To this end, we force her to perform her initial first-pass call on the final f, allowing Romeo to fix a column number $c_f \in \{0,1\}^n$ for finding vertical errors. To ensure that Juliet needs a L2R$^+$strategy to win on gvf, Romeo may append either h or h' and we expect the first symbol of words in T to match their last symbol; therefore, part (2) of the above definition corresponds to the case where Juliet and Romeo play according to the intuitive rules described above, v encodes a valid tiling and Romeo never tries to protest. All the other parts of the target language serve only to prevent unjustified protest by Romeo and to make sure that he loses immediately if he tries to cheat by claiming an inconsistency where there is none.

Parts (1), (3) and (5) of the target language address unjustified protests against the sequence in which Juliet calls input symbols, forcing Romeo to reserve the protest symbol @ for cases when Juliet skips an input symbol she is supposed to call. Part (4) prevents unjustified claims of an incremental error, with sub-expressions roughly corresponding to the different cases of Lemma 4.15 (given below). Parts (6) to (10) deal with attempts to wrongly claim a vertical error: Flagging too many tiles (6) or not enough tiles (7), flagging two tiles not violating the vertical constraints (8), flagging tiles in non-subsequent rows (9) and flagging tiles not in the column determined by c_f(10).

We now state the main ingredient of the proof:

Lemma 4.14. *Let $c_f \in \{0,1\}^n$ be the binary encoding of $n_f \in \{0,1,\ldots,2^n-1\}$ and let v be a string of the form $(U0^n(U\{0,1\}^n)^*U1^n\#)^*$. Then the following statements are equivalent:*

(a) v is a tiling candidate for $\mathcal{I}$ that has no vertical errors in column n_f and no incremental errors

(b) $gvc_fh \in JWin(G)$

(c) $g'vc_fh' \in JWin(G)$

Proof of Lemma 4.14. We only prove $(a) \Leftrightarrow (b)$, the proof of $(a) \Leftrightarrow (c)$ is analogous.

"$(a) \Rightarrow (b)$":

We describe a winning L2R strategy on gvc_fh for Juliet.

During her left-to-right pass, Juliet proceeds to call every symbol from U for as long as Romeo returns only symbols from $\hat{U}$. If she reaches the end of the string this way, she wins due to part (2) of T. If, at some point, Romeo decides to raise a false protest

that JULIET is not calling all symbols from U in sequence (i.e. returns @), JULIET wins by (3).

If ROMEO raises a false protest about an incorrectly encoded index (i.e. returns $@_N$ to one of JULIET's calls), JULIET proceeds to call all the bits to the right of $@_N$ in a left-to-right manner. This can result in the following:

- ROMEO returns @ at some point: In this case, JULIET wins by (5);
- ROMEO returns $\hat{0}$ or $\hat{1}$ for every bit of the index string to the right of $@_N$: In this case, JULIET wins by line (viii) of (4).
- ROMEO returns $0_N or 1_N$ at some point: In this case, JULIET wins by one of the other parts of (4) and Lemma 4.15 as will be explained below.

Let $c(i)$ be the column index to the right of $@_N$ and k the index of the bit in $c(i)$ for which ROMEO returned a flagged bit.

If $c(i)$ is followed by #, JULIET wins by line (vii) of (4). Therefore assume that $@_N$ is followed by a string of the form $c(i)Uc(j)$. Since v contains no increment errors, the negation of parts (a), (b) and (c) of Lemma 4.15 holds.

If $c(i)_k \neq c(j)_k$, then JULIET wins by (i) or (ii) due to part (a) of Lemma 4.15; if $c(i)_k = c(j)_k$, JULIET wins by one of (iii)-(vi) due to part (b) of Lemma 4.15. Since there are no other cases, JULIET can always play to win if ROMEO tries to protest an incremental error.

It remains to be explained how JULIET plays if ROMEO returns a symbol from U^V to one of JULIET's calls on the symbols from U (i.e. protests falsely about a vertical error). If this happens and the n-bit string $c(i)$ to the right of ROMEO's protest does not equal c_f, JULIET calls a bit in $c(i)$ on which $c(i)$ and c_f differ. If ROMEO answers this call with @, he loses by (5), if he answers with $\hat{0}$ or 0_N ($\hat{1}$ or 1_N), he loses by the first (second) term of (10).

If c_i matches c_f, JULIET continues calling all occurrences of symbols from U. If ROMEO raises no further protest (or protests with @ or $@_N$, as above), he loses by (7) (or (3), (5) or (4) as above). If he flags another tile by returning a symbol from U^V with column index string $c(j)$, he loses as described above if $c(j) \neq c_f$. If $c(j) = c_f$, again, JULIET continues calling all symbols from U. Should ROMEO then return anything but a symbol from $\hat{U}$ to any of JULIET's calls, she wins as described above by reaching a word in (3), (4), (5) or (6).

Finally, if the word reached after JULIET has called all the occurrences of symbols from U contains exactly two tiles $u_1^V, u_2^V \in U^V$, by the above strategy, their column index strings $c(i)$ and $c(j)$ both have to match c_f. Therefore $i = j$, which means that u_1 and u_2 have to be in the same column of the tiling candidate encoded in v. Thus, u_1 and u_2 are either in non-subsequent lines (in which case JULIET wins by (9)) or conform to the vertical constraints (in which case JULIET wins by (8)).

As the above cases cover all possible counter-strategies of ROMEO, the above is a winning L2R^+strategy for JULIET, and it follows that $gvc_fh \in \text{JWin}(G)$

"$(b) \Rightarrow (a)$":

Assume that Juliet has a winning L2R strategy σ on gvc_fh.

On her pass through v according to σ, Juliet has to call every occurrence of a symbol from U for as long as Romeo keeps returning symbols from $\hat{U}$, because if she were to skip a symbol from U, Romeo could respond to her next call with @ and she would lose. Leaving any symbols from U uncalled to deny Romeo the option of protesting with @ is not an option, either, because there are no words in T containing symbols from U without also containing symbols from $U^V \cup \{@, @_N\}$. Also, for as long as Romeo keeps returning symbols from $\hat{U}$, Juliet may not call any symbol not in U (i.e. 0 or 1), since all words containing $\hat{0}, \hat{1}, 0_N$ or 1_N also have to contain a symbol from $U^V \cup \{@_N\}$ further to the left.

As Juliet has to win the game if Romeo only returns symbols from $\hat{U}$, it follows that $v \in C$, since the only winning condition not involving any protest symbols is (2). Therefore v encodes a tiling candidate, i.e. contains no horizontal or constant errors.

Now assume for the sake of contradiction that v contains an increment error, i.e. a substring of the form $uc(i)u'c(j)$ with $u, u' \in U, c(i), c(j) \in \{0,1\}^n$ and $i + 1 \neq j$.

In this case, Romeo has a winning strategy in which he responds with $@_N$ as soon as Juliet calls u. After this move, Juliet is forced to call all bits from $\{0,1\}$ to the right of u until Romeo responds with 0_N, 1_N or @ or the next symbol not in $\{0,1\}$ is reached. This is because every string in T that contains $@_N$ requires one of these characters to its right, separated from $@_N$ by only characters from $\{\hat{0}, \hat{1}\}$.

Since $uc(i)u'c(j)$ is an increment error, one of the three conditions of the conclusion of Lemma 4.15 holds. If (c) holds, then $c(i) = 1^n$ and Juliet may win by replacing each bit of $c(i)$ by $\hat{1}$, since line (viii) of part (4) of the target language only allows $\hat{1}^n$ to be followed by #, not u'. If (a) or (b) hold, then there exists a position k in $c(i)$ such that Romeo may return a flagged bit 0_N or 1_N on Juliet's call on $c(i)_k$, (a) prevents her from winning according to lines (i) or (ii) of (4) and (b) prevents her from winning by lines (iii)-(vi) of (4).

Therefore, by contradiction to the assumption that Juliet has a winning strategy, v may contain no increment errors.

It remains to be shown that the tiling encoded by v contains no vertical errors in column n_f.

Again, suppose for contradiction's sake that there is a vertical error in column $i = n_f$, i.e. that v contains a substring of the form $uc(i)(U\{0,1\}^n)^*\#(U\{0,1\}^n) * u'c(i)$ with $(u, u') \notin V$. In this case, Romeo has a winning strategy. As argued above, Juliet has to call every symbol from U in v, to which Romeo keeps returning symbols from $\hat{U}$, except for u and u', where Romeo returns u^V (respectively u'^V).

Since according to this strategy, Romeo will never return @ or $@_N$ to a call as long as Juliet keeps calling in sequence (and Juliet loses as described above if she doesn't), Juliet cannot win by parts (1) or (3)-(5) of T. Since Romeo only returns exactly two symbols from U^V in subsequent lines (and therefore the rewritten string can never be in $\hat{C}$), parts (2), (6), (7) or (9) can not be reached, either. The fact that the two marked tiles $\hat{u}, \hat{u'}$ indeed violate a vertical constraint prevents Juliet from winning by (8). All that is left to show is that Juliet cannot win by part (10) of T.

Winning by (10) requires Juliet to call exactly one of the bits of an occurrence of

$c(i)$ right after either u^V or u'^V. Let us assume without loss of generality that this is the k-th bit of $c(i)$ and that its value is 0. Now, ROMEO wins by replacing it with either $\hat{0}$ or 0_N. This is because the $[((\Sigma_P \backslash \{\#\})^n + (\Sigma_P^* \# \Sigma_P^*)_{=n+1})(0+1)]^* 1$ part in (10) requires that the k-th bit of some n-bit substring in the input string does not match the k-th bit of $c(i)$, and since there may be at most $n-1$ bits between this bit and the final h or h', the string thus compared to $c(i)$ has to be the final $c_f = c(i)$. As no differing bit can be found, JULIET has no way of winning by (10), either, and therefore, ROMEO has a winning strategy. This yields the desired contradiction, so v encodes a tiling without any vertical constraint violation in column c_f. □

We now go on to show that (a) there is a valid tiling for instance $\mathcal{I}$ if and only if (b) $\mathrm{JWin}^{(\leftarrow)}(G) \backslash \mathrm{JWin}(G) \neq \varnothing$.

"$(a) \Rightarrow (b)$":

Assume there is a valid tiling for $\mathcal{I}$ and let v be the string encoding one such tiling. We argue that JULIET can win with a L2R$^+$ but not with a L2R strategy on gvf.

For a winning L2R$^+$ strategy, JULIET first calls the final f. If ROMEO responds with @, JULIET wins by part (1) of T since v encodes a valid tiling and therefore $v \in C$. Otherwise, f is replaced by an n-bit binary number c_f followed by either h or h'. If ROMEO chose to end with h', JULIET next calls the initial g, otherwise she plays *Read* on it. In the former case, JULIET plays a L2R pass on gvc_fh, in the latter case on $g'vc_fh'$, and by Lemma 4.14, JULIET has a winning L2R strategy in both of these cases because v contains no vertical errors.

To show that JULIET does not have a winning L2R strategy on gvf, observe first that no word in T ends with f, and therefore JULIET has to call the final f at some point. If doing so is her first (and therefore only) move, she loses if ROMEO returns a string ending with h or h', since T contains no string gvh or gvh' where v contains exclusively symbols from Σ_B.

For similar reasons, so long as ROMEO never protests using @ or $@_N$, JULIET is forced to call all symbols from U in v. ROMEO's winning strategy, here, is to simply return symbols from $\hat{U}$ on every call on a symbol from U and answer calls on bits by returning a corresponding bit from $\hat{B}_N$. This eventually transforms v into a string $v' \in \hat{C}$ (or causes JULIET to lose the game, if she calls anything other than symbols from U).

Finally, JULIET has to call f. To this, ROMEO replies with an arbitrary n-bit string c_f and h' if JULIET hasn't called the initial g, or h if she has replaced it by g'. Since the only strings whose middle part v' contains only symbols from Σ_P are those from part (2) of T, JULIET then loses the game on $gv'c_fh'$ or $g'v'c_fh$. As we have shown that ROMEO can always win against a L2R strategy on gvf, it holds that $gvf \notin \mathrm{JWin}(G)$.

"$(b) \Rightarrow (a)$":

Let $w \in \mathrm{JWin}^{(\leftarrow)}(G) \backslash \mathrm{JWin}(G)$. We begin with some observations on the structure of w.

From the construction of T and the rules in R, it follows that w must begin with g or g' and end with f, h, h' or @. If w ends with @, then $w = gv@$ with $v \in C$, because no string v containing symbols not in Σ_B can be rewritten into C; however, in this case JULIET already has a trivial winning L2R strategy on w. Therefore, w can not end with @ and is of the form $w = xvy$ with $x \in \{g, g'\}, y \in \{f, h, h'\}$. Also, v may not contain any

of the symbols $\{f, g, g', h, h'\}$, or w cannot be rewritten into T at all.

If w ends with h or h', if JULIET has a winning L2R$^+$strategy on w, then she also has a winning L2R strategy. To prove this, assume that JULIET has a winning L2R$^+$strategy σ on w, and let s be the symbol on which JULIET plays her initial *Call*. By the replacement rules of G, $s \in U \cup \{g, 0, 1\}$.

- $s = g$: In this case, JULIET's first *Call* takes place on the first letter of w, since (as stated above) g may appear nowhere else in the input string. Therefore, σ is already a L2R strategy, since no left step is necessary to start the L2R pass after a single *Call* on the initial symbol of w.

- $s \in U \cup \{0, 1\}$: In this case, JULIET may not call any symbol in v to the left of s after calling s, because ROMEO can always return @ on this second call and T contains no strings with an @ to the left of a symbol in $\hat{u} \cup U^V \cup \{\hat{0}, \hat{1}, 0_N, 1_N, @, @_N\}$. Therefore, the only symbol in w left of s which JULIET may call after calling s is an initial g. Doing so can only be part of a winning strategy if w ends with h'; however, in that case, σ can be transformed into an equivalent L2R strategy by calling g before s.

Since $w \in \text{JWin}^{(\leftarrow)} \backslash \text{JWin}$, this implies that w cannot end with h or h'.

If w ends with f, then $w = gvf$ with $v \in C$. This is the case because no word in T ends with f, so JULIET inevitably has to call the final f and ROMEO can always win on words not of the structure gvf by replacing f with @.

By the same argument, JULIET's initial call has to be on the final f; if she doesn't start by calling f, ROMEO can later on reply to the necessary call on f with @. After the initial call on f, play proceeds with a left-to-right pass over a string of the form gvc_fh or gvc_fh'.

On a string of the form gvc_fh', JULIET's first move in her L2R pass has to be a *Call* on g, since (as argued above) ROMEO may always replace tiles from U in v by marked tiles from $\hat{U}$ and the only strings in T ending with h' and containing only symbols from Σ_P in v are of the form $g'vc_fh'$.

Therefore, JULIET has a winning strategy on a string of the form gvc_fh or $g'vc_fh'$, and by Lemma 4.14, this is the case if and only if v encodes a tiling candidate with no vertical errors in the column n_f encoded by c_f and no increment errors. Since ROMEO is free to chose any column index string $c_f \in \{0, 1\}^n$ and by our prerequisite JULIET has a winning strategy for any of these, v may not contain vertical errors in any column and therefore encodes a valid tiling.

This concludes our proof that the existence of a word on which JULIET has a L2R$^+$but no L2R winning strategy in G implies the existence of a valid tiling for $\mathcal{I}$. □

Lemma 4.15. *For a number $i \in \{0, 1, \ldots, 2^n - 1\}$ let $c(i)$ be the n-bit binary encoding of i, and for an n-bit string c let c_k denote the k-th position of c. For any two numbers $i, j \in \{0, 1, \ldots, 2^n - 1\}$, it holds that $j \neq i + 1$ if and only if there exists a number $k \leqslant n$ such that one of the following conditions holds:*

(a) $c(i)_k \neq c(j)_k$, $c(i) \neq 1^n$ and for some $k' > k$, it holds that $c(i)_{k'} = 0$ or $c(j)_{k'} = 1$;

(b) $c(i)_k = c(j)_k$ *and it holds that either* $k = n$ *or* $c(i)_{k+1} = 1$ *and* $c(j)_{k+1} = 0$

(c) $c(i) = 1^n$

Proof. ($\Rightarrow$) Let $c(i), c(j)$ be as described with $j \neq i + 1$. If $i = j$, then the first part of (b) holds with $k = n$.

If $i + 1 < j$, then let k be 1 plus the length of the longest common prefix of $c(i)$ and $c(j)$ (i.e. k is the smallest index such that $c(i)_k \neq c(j)_k$). Since $i + 1 < j \leqslant 2^n - 1$, $c(i) \neq 1^n$, $c(i)_k = 0$ and $c(j)_k = 1$. If $c(i)_{k'} = 1$ and $c(j)_{k'} = 0$ were to hold for all $k' > k$ then it would follow that $i + 1 = j$, so condition (a) must be fulfilled.

If $i > j$ and $c(i) = 1^n$, then (c) holds. Let therefore $c(i) \neq 1^n$. If $c(i)_1 \neq c(j)_1$, then $c(i)_1 = 1$ and $c(j)_1 = 0$; however, since $c(i) \neq 1^n$, there exists a k' with $c(i)_{k'} = 0$, and therefore (a) holds with $k = 1$. Otherwise, let k be the length of the longest common prefix of $c(i)$ and $c(j)$. Then, since $i > j$, it holds that $c(i)_k = c(j)_k$ and $c(i)_{k+1} = 1$,$c(j)_{k+1} = 0$, and thus (b) holds.

($\Leftarrow$) Assume that $i + 1 = j$ for a given substring of v of the form $c(i)Uc(j)$. Then $c(i) \neq 1^n$, since $i < j \leqslant 2^n - 1$, and therefore (c) cannot hold. Let $k \leqslant n$.

If $c(i)_k = 0$ and $c(j)_k = 1$, then $k = n$; if $c(i)_k = 1$ and $c(j)_k = 0$, then $k < n$. In both cases, since $i + 1 = j$, it follows that $c(i)_{k'} = 1$ and $c(j)_{k'} = 0$ for all $k' > k$, and therefore (a) cannot hold.

If $c(i)_k = c(j)_k$, then either $c(i)_{k+1} = c(j)_{k+1}$ or (again because $i + 1 = j$) $c(i)_{k+1} = 0$,$c(j)_{k+1} = 1$. In any of these cases, (b) does not hold. □

4.2.2. Finite replacement languages

As already seen for the upper bounds in Subsection 4.1, L2RAll becomes easier if replacement languages are explicitly enumerated as part of the input.[4] We show here that the EXPTIME upper bound proven in Proposition 4.12 is tight as well.

Proposition 4.16. L2RAll *is hard for* EXPTIME*, even for games with explicitly enumerated finite replacement languages.*

The proof of this proposition is interesting in that it uses a reduction from a variant of JWin with some additional construction to enforce that a single left step is required for Juliet to win.

Proof. The proof is by a polynomial time reduction from JWin on string cfGs with finite replacement languages and unbounded replay, i.e., given a game $G = (\Sigma, \Gamma, R, T)$ and a string u, decide whether $u \in \mathrm{JWin}(G)$. This problem is EXPTIME-complete by Theorem 3.4 (a).

To this end, we show how to construct in polynomial time a game $G' = (\Sigma', \Gamma', R', T')$ from G and u such that the following statements are equivalent.

(a) $u \in \mathrm{JWin}(G)$.

[4] Note that this reduction in complexity does *not* hold if replacement languages are merely finite but represented by DFAs, cf. footnote 3 on page 65.

(b) $\mathrm{JWin}^{\leftarrow}(G') \backslash \mathrm{JWin}(G') \neq \varnothing$.

The construction of G' will ensure that Juliet can deduce a winning strategy on a string g_0uh_0 wrt G' with a single *Call* move in a first phase followed by an L2R phase if and only if she has an L2R winning strategy on u in G. In G' we use additional symbols $g_0, g_1, g_2, h_0, h_1, h_2, \#, @$, where

- $g_0, g_1, g_2, h_0, h_1, h_2$ are used to rule out L2R strategies for many strings,
- @ can be used by Romeo to "protest" if Juliet deviates from the intended flow of the game, and
- $\#$ is used to force Juliet to follow an L2R strategy on u (or otherwise Romeo can "protest").

The alphabet Σ' is $\Sigma \cup \{g_0, g_1, g_2, h_0, h_1, h_2, \#, @\}$ and we assume that the latter eight symbols do not belong to Σ.

For each rule $a \to w_1 \mid \cdots \mid w_\ell$ of R, there is a rule $a \to \#w_1 \mid \cdots \mid \#w_\ell \mid @$ in R'. Furthermore, R' contains the following rules.

- $g_0 \to g_1 \mid @$
- $g_1 \to g_2 \mid @$
- $h_0 \to h_1 \mid h_2 \mid @$

For a string $w \in (\Sigma \cup \{\#\})^*$, we write $\mathrm{cl}(w)$ for the string that results from w by eliminating all occurrences of $\#$.

The target language T' of G' contains

- all strings g_1wh_1 with $\mathrm{cl}(w) \in T$;
- all strings g_2wh_2 with $\mathrm{cl}(w) \in T$;
- the string $g_0u@$;
- all strings of the form gwh where $g \in \{g_0, g_1, g_2\}$, $h \in \{h_0, h_1, h_2\}$, and in w there is at least one occurrence of @ but no occurrence of $\#$ to the right of an occurrence of @;
- all strings $@wh_1$ and $@wh_2$, where w only contains symbols from Σ.

Clearly, G' can be constructed in polynomial time from G and u, in particular a DFA for T' (assuming a DFA for T).

It remains to be shown that (a) and (b) are indeed equivalent.

"(a) $\Rightarrow$ (b)":

We show that if $u \in \mathrm{JWin}(G)$ it follows that $g_0uh_0 \in \mathrm{JWin}^{\leftarrow}(G') \backslash \mathrm{JWin}(G')$.

First $g_0uh_0 \in \mathrm{JWin}^{\leftarrow}(G')$ as Juliet can choose the last position (carrying h_0) first. If Romeo answers with @, Juliet immediately wins as $g_0u@ \in T'$. Otherwise, she enforces

g_1 as first symbol if Romeo chose h_1 and g_2 if Romeo chose h_2. If Romeo chooses @ for the first symbol, Juliet wins directly as strings of the forms $@wh_1$ and $@wh_2$ with $w \in \Sigma^*$ are in T'. Then Juliet can basically follow her L2R winning strategy on u. It is easy to see that she wins the game in this fashion.

We show next that $g_0uh_0 \notin \mathrm{JWin}(G')$. Clearly, Juliet needs to play a *Call* on the last position as she cannot enforce a win otherwise (Romeo simply never protests). However, Romeo can reply by h_1 just if the first position of the string is not g_1, enforcing a win for Romeo.

Thus, (a) $\Rightarrow$ (b).

"(b) $\Rightarrow$ (a)":

Assume that there is a word $v \in \mathrm{JWin}^{\leftarrow}(G')\backslash\mathrm{JWin}(G')$. We start with some observations on what v can look like.

1. The word v must begin with some $g_i \in \{g_0, g_1, g_2\}$. Indeed, no letter not in $\{g_0, g_1, g_2\}$ can ever be rewritten into a letter in $\{g_0, g_1, g_2\}$ and the only strings that are accepted that do not begin with such a letter are strings on the form $@wh_1$ or $@wh_2$. If Juliet wins on a string that begins with @, it must be by never playing *Call*, since otherwise Romeo could protest with a second @ and win. Thus, if Juliet wins, she wins with an L2R-strategy.

2. The word v must end with some $h_j \in \{h_0, h_1, h_2\}$. There is only a single accepted string that is accepted that does not end with such a letter and no other letter can be rewritten into them. If the string v ends with @, it is either $g_0u@$, in which case Juliet wins with an L2R-strategy by just reading it, or it is another string, in which case she cannot win, since any *Call* move can be answered by Romeo with a second @ symbol.

3. If Juliet has a winning strategy on g_iwh_j, then the strategy must play left-to-right on g_iw. If Juliet plays *Call* on a symbol a in w, Romeo can answer with a string $\#w_j \in R_a$, introducing a # symbol into the word. If Juliet ever plays a *Call* on a position to the left of this symbol, Romeo can protest with an @ symbol, creating a word with an @ to the left of a #. After this, Juliet cannot win.

4. That Juliet can win on $v = g_iwh_j$, but not with an L2R-strategy means that she needs to call on the last position before completing play on g_iw. This implies $h_j = h_0$.

5. On strings of the form g_1wh_0 or g_2wh_0, Juliet has no winning strategy. Indeed, since no accepted string ends with h_0, Juliet will sooner or later have to play *Call* on the last position. When she does this, Romeo can answer with @. The only string ending with @ that is accepted is $g_0u@$, but g_1 or g_2 can never be rewritten into g_0, so Juliet loses.

From (1)–(5) above, we can conclude that if $v \in \mathrm{JWin}^{\leftarrow}(G')\backslash\mathrm{JWin}(G')$, then v has the form g_0wh_0. When Juliet starts play on a word g_0wh_0 by calling on the last position,

Romeo can answer with @. The only accepted string that ends with @ is $g_0u@$. This means that the string v must be g_0uh_0. Romeo can, however, also answer the call on h_0 with h_1 or h_2. In this case, Juliet must play an L2R-strategy that transforms g_0u into some g_iw' with $\mathrm{cl}(w') \in T$. This same strategy, restricted to u and ignoring the #-symbols, is a winning L2R-strategy on u in G. □

4.3. Outlook and bibliographical remarks

This chapter introduced and examined the L2RAll problem for context-free games on strings: Does Juliet have a winning L2R strategy on every string where she has an arbitrary winning strategy? As it turns out, the complexity of this problem is somewhat high, requiring at least exponential time, but the algorithm presented here may still be of practical interest, as the problem is basically one of static analysis, required mostly in designing schema languages for web services, where sufficient time for pre-computations may be available.

As a tool for solving this problem, this chapter also introduced the concept of dual effects, which allows for analysing context-free games "from Romeo's perspective". While this technique is not used in any of the following chapters, it seems interesting enough of its own right to merit inclusion in this dissertation.

So far, no further variants of the L2RAll problem have been researched for more advanced classes of context-free games; nonetheless, it should not be too difficult to generalise the results given here to any sort of context-free games that allow for the formulation of effects, such as those on nested words examined in Chapter 6.

Bibliographic remarks. The L2RAll problem was first introduced in [AMB05], where it was falsely claimed to be undecidable. Decidability in doubly exponential time was proven by Joscha Kulbatzki as part of his Diploma thesis [Kul10]. The proof idea was then refined by the author in collaboration with Henrik Björklund and Thomas Schwentick using the effect technique and published in [BSSK13]. For more bibliographical remarks on effects, see also Section 3.5.

Dual effects also originate in [BSSK13]; however, the concept of duality is a common one in all kinds of symmetric games. For instance, the relationship between primal and dual effects given in Lemma 4.6 is based on a corresponding result for effectivity functions for logic games [PP03].

Part II.

Context-free games on nested words

5. Nested words and automata

Context-free games on strings, as examined in the previous chapter, already shed some light on the complexity of the Active XML schema rewriting problem, and they yield techniques and methods of dealing with context-free games in general. However, they still fail to capture the full extent of the schema rewriting problem, as (Active) XML documents are generally abstracted as *trees*. While Milo et al. [MAABN05] presented a technique of solving the schema rewriting problem by iteratively solving context-free games on strings, their technique is only applicable if relevant schemas are given as DTDs and does not carry over to more expressive schema languages like XML Schema or Relax NG.

To this end, we now examine *nested words* as a sequential representation of trees that corresponds very naturally to XML documents. In Section 5.2, we introduce *nested word automata* as a schema formalism that is equivalent to *regular tree languages* (a natural superclass of the languages described by XML Schema). A restriction of these automata that captures XML Schema and has somewhat nicer complexity properties is then presented in Section 5.3. Finally, Section 5.4 gives definitions and complexity results for corresponding models of alternating automata (in accordance with our basic idea of using alternating automata for solving games, cf. Section 1.3).

5.1. Nested words

Essentially, nested words over a label alphabet Σ are a natural representation of node-labelled trees with label alphabet Σ. As such, their structure is very similar to that of XML documents, and our notation (using opening and closing tags) is chosen to reflect this.

Definition 5.1. For a finite alphabet Σ, $\langle\Sigma\rangle \stackrel{\text{def}}{=} \{\langle a\rangle \mid a \in \Sigma\}$ denotes the set of all *opening* Σ*-tags* and $\langle/\Sigma\rangle \stackrel{\text{def}}{=} \{\langle/a\rangle \mid a \in \Sigma\}$ the set of all *closing* Σ*-tags*. The set $\mathrm{WF}(\Sigma) \subseteq (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$ of *(well-)nested words over* Σ is the smallest set such that

- $\epsilon \in \mathrm{WF}(\Sigma)$, and
- if $u, v \in \mathrm{WF}(\Sigma)$ and $a \in \Sigma$, then also $u\langle a\rangle v\langle/a\rangle \in \mathrm{WF}(\Sigma)$.

We associate with every nested word w its *canonical forest representation*, such that words $\langle a\rangle\langle/a\rangle$, $\langle a\rangle v\langle/a\rangle$ and uv correspond to an a-labelled leaf, a tree with root a (and subforest corresponding to v), and the forest of u followed by the forest of v, respectively. Similarly, we associate with each unranked forest t its *linearisation*, which is the nested word obtained from t by reversing this transformation. We slightly abuse notation by usually not distinguishing between forests and nested words, as justified by this

correspondence; particularly, if L is a language of trees (such as one defined by a tree grammar, cf. Section 2.2), we also use the symbol L to refer to the nested word language of linearisations of trees in L.

A nested word w is *rooted*, if its corresponding forest is a tree. In a nested word $w = w_1 \dots w_n \in \mathrm{WF}(\Sigma)$, two tags $w_i \in \langle\Sigma\rangle$ and $w_j \in \langle/\Sigma\rangle$ with $i < j$ are *associated* if the substring $w_i \dots w_j$ of w is rooted.

The *width* of a nested word is the maximum number of children of any node in its corresponding forest. Its *root width* is just the number of trees in its forest. The *depth* of a nested word is the depth of its canonical forest representation.

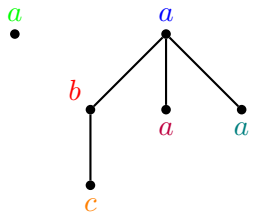


Figure 5.1.: Canonical forest representation of the nested word from Example 5.2; colours correspond to the ones used in Example 5.2.

Example 5.2. *The nested word* $w = \langle a\rangle\langle/a\rangle\langle a\rangle\langle b\rangle\langle c\rangle\langle/c\rangle\langle/b\rangle\langle a\rangle\langle/a\rangle\langle a\rangle\langle/a\rangle\langle/a\rangle$ *over the label alphabet* $\Sigma = \{a, b, c\}$ *corresponds to the canonical forest shown in Figure 5.1; pairs of associated opening and closing tags are marked in the same colour, as are node labels of the canonical forest representation corresponding to those pairs. The word* w *has a root width of* 2 *(as its forest representation consists of two trees), a width of* 3 *(as the blue* a*-labelled node has three children, and no other node has more than that), and a depth of* 2 *(as the longest root-to-leaf path in its forest representation, from the blue* a*-labelled node to the orange* c*-labelled node, uses two edges).*

To stress the distinction from nested words in $\mathrm{WF}(\Sigma)$, we refer to strings in Σ^* as *flat strings* (over Σ). What we describe as opening and closing tags is often referred to as *call symbols* and *return symbols* in the literature on nested words; we do not use these terms to avoid confusion with *Read* and *Call* moves used in context-free games.

For the purpose of reductions, we will sometimes need to encode flat strings as nested words; for a string $w = w_1 \cdots w_n \in \Sigma^*$ with $w_1, \dots, w_n \in \Sigma$, the *standard nested word encoding* $\widehat{w}$ *of* w is $\widehat{w} = \langle w_1\rangle\langle/w_1\rangle \dots \langle w_n\rangle\langle/w_n\rangle \in \mathrm{WF}(\Sigma)$.

Trees vs nested words As already mentioned, the standard way of looking at (Active) XML models documents as trees. There exists already a plethora of research dedicated to trees and automata operating on trees in the context of XML (for some surveys see e.g. [Sch07; Nev02; Sch12] and references given there), so our focus on nested words over trees merits some discussion.

The standard formalism of schemas for trees in the context of XML is that of (subclasses of) *regular tree languages*, recognised by various sorts of *tree automata*. Probably the most important of these variants are *unranked tree automata* [BKMW01], which go

through a tree top-down or bottom-up, assigning states to nodes depending on their labels and parent (child) states and accept depending on the states assigned to leaf (root) nodes. While these automata are well-researched, their operation is inherently *parallel* and generally requires constructing from an XML document its corresponding tree in-memory. As such, the important concept of *left-to-right* traversal (cf. the discussion at the start of Section 3.2) can only be defined on the level of child strings of single nodes, not globally on the entire tree.

Nested words, on the other hand, reflect much more closely the structure of XML documents themselves, rather than the trees they represent. As such, nested words offer themselves up much more naturally for a definition of left-to-right traversal, which corresponds to *document order* traversal of the corresponding XML document. Additionally, even though this matter is outside of the scope of this dissertation, nested words presumably lend themselves much better as far as streaming data processing (and the corresponding class of *one-pass* context-free games with imperfect information, cf. [AMB05]) is concerned.

Additionally, using nested words instead of trees does not incur any penalty as far as complexity or expressiveness are concerned. In fact, regular unranked tree languages and regular languages of nested words (introduced in the next section) are equivalent up to transformation from trees to nested words and back and have basically the same closure and complexity properties for appropriate automata models (cf. [AM09]).

5.2. Nested word automata

Adapting the definition of context-free games to nested words requires a concept of *regular nested word languages* to define (possibly infinite) target and replacement languages by some finite representation with an efficiently decidable word problem. To this end, we use a slightly simplified adaptation of *nested word automata* [AM09]. Nested word automata can be viewed as a generalisation of finite-state automata that adds a component of *hierarchical states* generated when an automaton reads an opening tag and consumed when it reads the corresponding closing tag. Alternatively, one may see nested word automata as a restricted variant of pushdown automata that push a single symbol on opening and pop a single symbol on closing tags, where pushdown symbols are states from a hierarchical state set.

Note that we define the semantics of nested word automata in such a way that runs and acceptance of a nested word automaton are only defined if the input is a well-nested word. While technically, these definitions could easily be extended to arbitrary strings from $(\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, we are only interested in languages of well-nested words (as linearisations of XML trees) in this dissertation.

Definition 5.3. A *Nested Word Automaton (NWA)* $A = (Q, P, \Sigma, \delta, q_0, F)$ consists of

- a set Q of *linear states*,
- a set P of *hierarchical states*,

- an alphabet Σ,
- a *transition relation* $\delta \subseteq (Q \times \langle\Sigma\rangle \times Q \times P) \cup (Q \times P \times \langle/\Sigma\rangle \times Q)$,
- an *initial state* $q_0 \in Q$, and
- a set of *accepting states* $F \subseteq Q$.

We also write $(q', p) \in \delta(q, \langle a\rangle)$ (resp. $q' \in \delta(q, p, \langle/a\rangle)$) instead of $(q, \langle a\rangle, q', p) \in \delta$ (resp. $(q, p, \langle/a\rangle, q') \in \delta$).

A *configuration* κ of A is a tuple $(q, \alpha) \in Q \times P^*$, with a *linear state* q and a sequence α of *hierarchical states*, reflecting the pushdown store. A *run of* A *on* $w = w_1 \cdots w_n \in WF(\Sigma)$ is a sequence $\rho = \kappa_0, \ldots, \kappa_n$ of configurations $\kappa_i = (q_i, \alpha_i)$ of A such that for each $i \in [n]$ and $a \in \Sigma$ it holds that

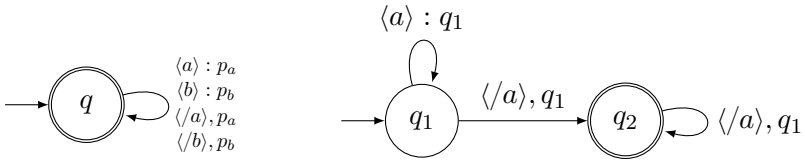
- if $w_i = \langle a\rangle$, then $(q_i, p) \in \delta(q_{i-1}, \langle a\rangle)$ (for some $p \in P$), and $\alpha_i = p\alpha_{i-1}$, or
- if $w_i = \langle/a\rangle$, then $q_i \in \delta(q_{i-1}, p, \langle/a\rangle)$ (for some $p \in P$), and $p\alpha_i = \alpha_{i-1}$.

In this case, we also write $\kappa_0 \overset{w}{\rightsquigarrow}_A \kappa_n$. We say that A *accepts* w if $(q_0, \epsilon) \overset{w}{\rightsquigarrow}_A (q', \epsilon)$ for some $q' \in F$. The language $L(A) \subseteq \mathrm{WF}(\Sigma)$ is defined as the set of all strings accepted by A and is called a *regular language* (of nested words).

An NWA is *deterministic* (or a DNWA) if $|\delta(q, \langle a\rangle)| = 1 = |\delta(q, p, \langle/a\rangle)|$ for all $p, q \in Q$ and $a \in \Sigma$. In this case, we simply write $\delta(q, \langle a\rangle) = (q', p')$ instead of $\delta(q, \langle a\rangle) = \{(q', p')\}$ (and accordingly for $\delta(q, p, \langle/a\rangle)$), and $\delta^*(q, w) = q'$ if q' is the unique state, for which $q \overset{w}{\rightsquigarrow}_A q'$.

An NWA is in *normal form* if $P = Q$ and every transition function $\delta(q, \langle a\rangle)$ only uses pairs of the form (q', q).

Informally, when an NWA in normal form reads an opening tag it always pushes its current linear state (before the opening tag) and therefore can see this state when it reads the corresponding closing tag. Since in this case the hierarchical state is just the origin state q of the transition, we write $\delta(q, \langle a\rangle) = q'$ as an abbreviation of $\delta(q, \langle a\rangle) = (q', q)$, for DNWAs in normal form.



(a) NWA A_1 (b) NWA A_2

Figure 5.2.: NWAs A_1 and A_2 from Example 5.4

Example 5.4. *Figure 5.2 shows two examples of NWA. The NWA* A_1 *(Fig. 5.2a) checks that its input string is well-nested by pushing hierarchical state* p_a *(resp.* p_b*) to the stack on each opening* $\langle a\rangle$ *(resp.* $\langle b\rangle$*) tag and popping an according hierarchical state with each*

matching closing tag. In this manner, A_1 decides the set WF($\{a, b\}$) of all well-nested words over $\{a, b\}$. The NWA A_2 (Fig. 5.2b) initially pushes a hierarchical state q_1 each time it reads $\langle a\rangle$ in linear state q_1, then changes linear state to q_2 on reading the first $\langle/a\rangle$ and accepts iff each initial $\langle a\rangle$ is matched by a $\langle/a\rangle$. In this manner, it decides the language $\{\langle a\rangle^n\langle/a\rangle^n \mid n \geqslant 1\}$.

Both of these NWAs are deterministic, and only A_2 is in normal form.

Note that any NWA $A = (Q, P, \Sigma, \delta, q_0, F)$ can always be transformed into an equivalent NWA whose linear state set Q' and hierarchical state set P' coincide by just setting $Q' = P' = Q \cup P$. For NWA of this type (specifically for NWA in normal form), we simply write $A = (Q', \Sigma, \delta, q_0, F)$ instead of $A = (Q', Q', \Sigma, \delta, q_0, F)$.

For the following result on DNWA normal forms, this transformation is not required, however.

Lemma 5.5. *There is a polynomial-time algorithm that computes for every deterministic NWA an equivalent deterministic NWA in normal form.*

Proof. Let $A = (Q, P, \Sigma, \delta, q_0, F)$ and let δ_1 and δ_2 the projections of δ to its first and second component (for opening tags only), respectively, i.e., $\delta(q, \langle a\rangle) = (\delta_1(q, \langle a\rangle), \delta_2(q, \langle a\rangle))$. An equivalent DNWA $A' = (Q, Q, \Sigma, \delta', q_0, F)$ in normal form can be constructed by letting $\delta'(q, \langle a\rangle) \stackrel{\text{def}}{=} (\delta_1(q, \langle a\rangle), q)$ and $\delta'(q, p, \langle/a\rangle) \stackrel{\text{def}}{=} \delta(q, \delta_2(p, \langle a\rangle), \langle/a\rangle)$. □

It should be stressed that the transformation described in the proof of Lemma 5.5 relies on the fact that the semantics of NWA is only defined on well-nested words, by assuming that the label of every closing tag matches that of its associated opening tag.

The following properties were proven in [AM09] for a slightly different definition of NWA; their proofs are easily adapted to carry over to NWA as defined here. They basically show that NWA behave quite similar to finite-state automata with slightly increased complexity (or, alternatively, similar to finite-state tree automata), which further justifies using the term "regular" for the languages accepted by NWA.

Lemma 5.6 ([AM09], Theorem 3.3)**.** *For each NWA A, there exists a DNWA A' of size at most exponential in $|A|$ such that $L(A) = L(A')$.*

Lemma 5.7 ([AM09], Theorem 3.5)**.** *For all NWA A and B, it is possible to construct NWA deciding WF(Σ)$\backslash L(A)$, $L(A) \cup L(B)$ and $L(A) \cap L(B)$. The constructions for $L(A) \cup L(B)$ and $L(A) \cap L(B)$ are possible in polynomial time; if A is deterministic, the construction for WF(Σ)$\backslash L(A)$ is also possible in polynomial time.*

Theorem 5.8. *(a) The NWA membership and emptiness problem are in* PTIME.

(b) The DNWA emptiness problem is PTIME*-complete with respect to logspace reductions.*

(c) The NWA universality, equivalence and inclusion problem are EXPTIME*-complete.*

(d) Deciding, given an NWA A and a DNWA B, whether $L(A) \subseteq L(B)$ is PTIME*-complete with respect to logspace reductions.*

Proof. Statements (a) and (c) were proven in [AM09] as Theorems 6.1 and 6.2.

The upper bound in (b) follows directly from (a), and the lower bound in (b) can be proven by a straightforward (if technical) reduction from the emptiness problem for deterministic top-down tree automata (cf. Section 7.2 in [AM09] and Theorem 1 in [Vea97]).

The upper bound in (d) follows from the fact that DNWA can be efficiently complemented and the fact that $L(A) \subseteq L(B)$ holds if and only if $L(A) \cap \overline{L(B)} = \varnothing$. By Lemma 5.7 and part (a) of this theorem, this can be checked in polynomial time.

The lower bound in (d) follows from the lower bound in (b) by reduction. Let B be a DNWA to be checked for emptiness. We can construct in logarithmic space an NWA A deciding $\mathrm{WF}(\Sigma)$, and (since B is deterministic) a DNWA B' deciding the complement of $L(B)$. It then holds that $L(A) \subseteq L(B')$ if and only if $L(B') = \mathrm{WF}(\Sigma)$, which is the case if and only if $L(B)$ is empty. □

5.3. Simple nested word automata and XML

While regular nested word languages are a suitable way to define schemas for nested words, we will see in Chapter 6 that the complexity of solving context-free games on nested words is rather high compared to the flat string setting. In addition, while regular languages of nested words can be interpreted to subsume regular languages of unranked trees [AM09] and can therefore represent any practical schema for XML (such as DTDs or XML Schema), they are something of an over-approximation of those practical schema formalisms (cf. [MNSB06]).

In order to remedy both of these problems at once, we define *simple* DNWAs, a restriction of DNWAs in normal form that captures all languages specified by single-type tree grammars. In simple DNWAs, states are typed, i.e. each state has a component in some type alphabet Δ.

Informally, when a simple DNWA A reads a subword $w = \langle a \rangle v \langle / a \rangle$ in (linear) state q, it determines already on reading $\langle a \rangle$ which state q' it will take after processing w, and this state will be of the same type as q. After reading $\langle a \rangle$, the linear state of A only depends on the *type* of q, not the exact state; this models the single-type restriction. After reading $\langle a \rangle$, A goes on to validate v, and if this validation fails, A enters a failure state $\perp$ instead of q'. Thus, the state of A at a position basically only depends on its ancestor positions (in the tree view of the document) and their left siblings. The only way in which other nodes in subtrees of these nodes can influence the state is by assuming the sink state $\perp$. Thus, in the spirit of [MNSB06], we could call such DNWAs *ancestor-sibling-based* but we prefer the term *simple* for succinctness.

For the following definition, we recall that, by convention, deterministic NWA in normal form do not include a hierarchical state set, because it is always equal to the set of linear states.

Definition 5.9. A deterministic NWA $A(T) = (Q, \Sigma, \delta, q_0, F)$ in normal form is *simple* (a *SNWA*) if there exist a *type alphabet* Δ and *state set* S with $Q \subseteq S \times \Delta$, a *local*

acceptance function $F_{\text{loc}} : \Sigma \to \mathscr{P}(Q)$, a *target state function* $t : Q \times \Sigma \to Q$ and a *failure state* $\bot \in Q \backslash F$, such that the following conditions are satisfied for every $a \in \Sigma$:

- for every $p, p' \in S, X \in \Delta$: $\delta((p, X), \langle a \rangle) = \delta((p', X), \langle a \rangle)$;
- for every $q \in F_{\text{loc}}(a)$: $\delta(q, p, \langle / a \rangle) = t(p, a)$;
- for every $q \in Q \backslash F_{\text{loc}}(a)$: $\delta(q, p, \langle / a \rangle) = \bot$;
- for every $q \in Q$: $\delta(\bot, \langle a \rangle) = \delta(\bot, q, \langle / a \rangle) = \bot$, and
- for every $(p, X) \in Q$: $t((p, X), a) = (p', X)$ for some $p' \in S$.

We first show that, as already stated above, simple DNWAs are at least as expressive as single-type tree grammars. The idea behind this is rather straightforward, as we only need to combine DFAs for each type's content model and, on reading some opening tag $\langle a \rangle$, start some DFA in a sub-computation to check the nested word between $\langle a \rangle$ and the associated $\langle / a \rangle$ for compliance with the content model of some type X. Thanks to the single-type property, the type X is uniquely determined by a and the context from which $\langle a \rangle$ was read, so we obtain a deterministic automaton as desired.

Proposition 5.10. *From every single-type tree grammar T, a simple DNWA A can be computed in polynomial time, such that $L(A) = L(T)$.*

Proof. Let $T = (\Sigma, \Delta, X_0, P, \lambda)$ be a single-type tree grammar. We will construct a SNWA A such that $L(T) = L(A)$.

Due to the single-type property, for each type $X \in \Delta$ and each $a \in \Sigma$, there is at most one type X' in the content model of X with $\lambda(X') = a$. Without loss of generality, assume that there is exactly one such type for each X and a (which can be done by adding a "dummy type" $X_\bot$ with $r_{X_\bot} = \varnothing$ to T), and denote this type by $\nu(X, a)$.

For each $X \in \Delta$, let $A_X = (S_X, \Delta, \delta_X, p_{0,X}, F_X)$ be a DFA deciding $L(r_X)$ (which can be computed from the *deterministic* regular expression r_X in polynomial time). Assume w.l.o.g. that all S_X, S_Y are disjoint for $X \neq Y$. Then, the SNWA $A = (Q, \Sigma, \delta, (p_0, 0), \{(p_f, 0)\})$ is defined as follows:

- $Q = \{\bot\} \cup S \times \Delta'$, with
  - $S = \{p_0, p_f\} \cup \bigcup_{X \in \Delta} S_X$ and
  - $\Delta' = \Delta \cup \{0\}$, with $0 \notin \Delta$
- δ is defined by
  - $\delta((p_0, 0), \langle \lambda(X_0) \rangle) = (q_{0,X_0}, X_0)$,
  - $\delta((p, X), \langle a \rangle) = (p_{0,\nu(X,a)}, \nu(X, a))$ for each $a \in \Sigma$, $p \in P$, $X \in \Delta$,
  - $\delta(q, q', \langle / a \rangle)$ is defined by t below as per the definition of SNWA,
- $F_{\text{loc}}(a) = \bigcup_{X \in \Delta : \lambda(X) = a} (F_X \times \{X\})$, and
- t is defined by

	Membership	Non-emptiness
ANWAs	PSPACE	2-EXPTIME
simple ANWAs	PTIME	PSPACE

Table 5.1.: Summary of complexity results for ANWAs. All results are completeness results.

- $t((p_0, 0), \lambda(X_0)) = (p_f, 0)$ and
- $t((p, X), a) = (\delta_X(p, a), X)$ for each $a \in \Sigma$, $p \in S$, $X \in \Delta$.

To show that $L(T) = L(A)$, we can prove by a simple induction that for every $w \in \mathrm{WF}(\Sigma)$ and $X \in \Delta$, it holds that $\langle\lambda(X)\rangle w \langle/\lambda(X)\rangle \in L(X)$ if and only if $\delta^*((q_{0,X}, X), w) \in F_{\mathrm{loc}}(\lambda(X))$, where $L(X)$ is defined like $L(T)$ with root type X. The claimed equality then follows with $L(T) = L(S)$. □

A detailed investigation of complexities of algorithmic problems and closure properties for SNWA is beyond the scope of this dissertation; it should be noted, however, that since SNWA are deterministic NWA, they inherit polynomial-time-complexity on the membership and emptiness problem by Theorem 5.8.

5.4. Alternating nested word automata

As mentioned in Section 1.3 of the introduction, one of the basic ideas behind our algorithms for solving context-free games is pushing the alternation inherent in the game into an appropriate type of alternating automata. As seen for context-free games on flat strings in the previous chapter, these automata are constructed by adding alternation to the target language automata for cfGs. In this section, we therefore define and examine alternating variants of the nested word automata defined in Section 5.2 and simple nested word automata defined in Section 5.3.

The complexity results for decision problems of alternating nested word automata and simple alternating nested word automata are summarised in Table 5.1. Note that, similar to their non-alternating counterparts, the complexities for simple alternating NWA match those for alternating finite-state automata, while the complexities for general alternating ANWA rise beyond that.

To simplify presentation, the variants of alternating nested word automata considered here do not use a separate set of hierarchical states, i.e. we generally assume that the sets of linear and hierarchical states coincide. This is justified by the observation preceding Lemma 5.5, as well as the fact that all alternating automata in Chapter 6 will be constructed from deterministic automata in normal form.

5.4.1. Alternating NWA

An *alternating nested word automaton (ANWA)* $A = (Q, \Sigma, \delta, q_0, F)$ is defined like an NWA with $P = Q$, except that the transition relation δ is the union of one function each

from $Q \times \langle\Sigma\rangle$ to $\mathcal{B}^+(Q \times Q)$ and from $Q \times Q \times \langle/\Sigma\rangle$ to $\mathcal{B}^+(Q)$), where $\mathcal{B}^+(Q)$ denotes the set of all positive boolean combinations over elements of Q using the binary operators $\wedge$ and $\vee$ (and likewise for $\mathcal{B}^+(Q \times Q)$).

Intuitively, on a transition from a linear state q with a closing tag $\langle/a\rangle$ and hierarchical state p, an ANWA A proceeds as follows. First, it guesses nondeterministically a set $X \subseteq Q$ such that setting all occurrences of exactly the elements from X to "true" in $\delta(q, p, \langle/a\rangle)$ satisfies that formula (which we write as $X \models \delta(q, p, \langle/a\rangle)$). Afterwards, A branches universally into all states from X to continue its computation. Transitions on opening tags are handled analogously, with the difference being that A guesses and branches into *pairs* of states, with the first component denoting the (linear) follow-up state and the second component denoting the (hierarchical) state to be pushed to the stack.

The semantics of ANWA is defined via *runs*, which are labelled unranked trees (D, λ) over some tree domain D with a labelling function $\lambda : D \to (Q \times Q^*)$ mapping node addresses to configurations (as defined in Def. 5.3). For any function $\lambda : D \to (Q \times Q^*)$ and node address $x \in D$, we denote by $\lambda(x)_{\text{lin}}$ the linear state component of $\lambda(x)$, i.e. if $\lambda(x) = (q, \alpha)$, then $\lambda(x)_{\text{lin}} = q$.

Definition 5.11. A *run* $\rho = (D, \lambda)$ of an ANWA A over a nested word $w = w_1 \dots w_n$ is a finite tree of depth at most n, represented by a tree domain D and a labelling function $\lambda : D \to (Q \times Q^*)$ such that $\lambda(\epsilon) = (q_0, \epsilon)$ and, for every $x \in D$ of length i with ℓ children, it holds that

- if $w_{i+1} \in \langle\Sigma\rangle$ and $\lambda(x) = (q, \alpha)$, then $\lambda(x \cdot i) = (q_i, p_i\alpha)$ for each $i \in [\ell]$, where $\{(q_1, p_1), \dots, (q_\ell, p_\ell)\} \models \delta(q, w_{i+1})$, and
- if $w_{i+1} \in \langle/\Sigma\rangle$ and $\lambda(x) = (q, p\alpha)$, then $\lambda(x \cdot i) = (q_i, \alpha)$ for each $i \in [\ell]$, where $\{q_1, \dots, q_\ell\} \models \delta(q, p, w_{i+1})$.

The run ρ is called *accepting*, if the linear state component of $\lambda(x)$ is in F, i.e. $\lambda(x)_{\text{lin}} \in F$, for every $x \in D$ of length $|w|$. The *language* $L(A)$ *of* A is defined in the natural way as the set of all nested words on which there is an accepting run of A. The run ρ is called *minimal* if no proper subtree of ρ is a run.

We note that a run over a nested word w may contain leaves of depth $i < |w|$. This happens if and only if such a leaf is labelled with a linear state q (and, possibly, a top hierarchical state p) for which $\delta(q, w_{i+1}) = \texttt{true}$ (resp. $\delta(q, p, w_{i+1}) = \texttt{true}$), because $\varnothing \models \texttt{true}$ and $\varnothing \not\models \varphi$ for any other positive boolean formula φ. Conversely, no run may ever contain any node at any depth i labelled with linear state q (and possibly top hierarchical state p) such that $\delta(q, w_{i+1}) = \texttt{false}$ (resp. $\delta(q, p, w_{i+1}) = \texttt{false}$).

Example 5.12. *Any NWA* $A = (Q, Q, \Sigma, \delta, q_0, F)$ *may be interpreted as an ANWA* $A' = (Q, \Sigma, \delta', q_0, F)$ *by setting* $\delta'(q, \langle a\rangle) = \bigvee_{(q', p) \in \delta(q, \langle a\rangle)} (q', p)$ *and* $\delta'(q, p, \langle/a\rangle) = \bigvee_{q' \in \delta(q, p, \langle/a\rangle)} q'$ *for each* $a \in \Sigma$ *and* $q, p \in Q$. *It is easy to see that each (accepting) run of* A *on some nested word* w*, when interpreted as a tree consisting of only a single root-to-leaf path, directly induces an (accepting) run of* A' *on* w*, and vice versa for minimal runs of* A'.

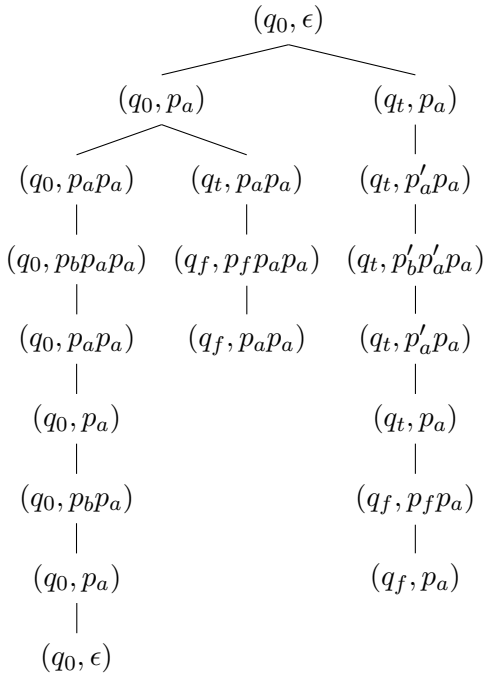


Figure 5.3.: Example run of the ANWA A_{ex} from Example 5.13 on the string $w = \langle a\rangle\langle a\rangle\langle b\rangle\langle/b\rangle\langle/a\rangle\langle b\rangle\langle/b\rangle\langle/a\rangle$.

Example 5.13. *Consider the ANWA $A_{ex} = (Q, \Sigma, \delta, q_0, F)$ over the alphabet $\Sigma = \{a, b\}$, with state set $Q = \{q_0, q_e, q_f, p_a, p_b, p'_a, p'_b, p_f\}$, accepting state $F = \{q_0\}$, and transition function δ defined as follows:*

$q \in Q$	$x \in \Sigma$	$\delta(q, \langle x\rangle)$	$(q,p) \in Q \times Q$	$x \in \Sigma$	$\delta(q, p, \langle/x\rangle)$
q_0	a	$(q_0, p_a) \wedge (q_t, p_a)$	(q_0, p_a)	a	q_0
q_0	b	(q_0, p_b)	(q_0, p_b)	b	q_0
q_t	a	(q_t, p'_a)	(q_t, p'_a)	a	q_t
q_t	b	$(q_t, p'_b) \vee (q_f, p_f)$	(q_t, p'_b)	b	q_t
			(q_f, p_a)	a	`true`
			(q_f, p_f)	b	q_f

An accepting run of A_{ex} on the nested word $w = \langle a\rangle\langle a\rangle\langle b\rangle\langle/b\rangle\langle/a\rangle\langle b\rangle\langle/b\rangle\langle/a\rangle$ is displayed in Figure 5.3, so $w \in L(A_{ex})$ holds.

The language $L(A_{ex})$ consists of all nested words in whose canonical forest representation each a-labelled node has a rightmost child labelled b. The basic idea behind A_{ex} is that it traverses its input string in state q_0, branching universally into the test state q_t each time it reads an $\langle a\rangle$. In this state q_t, it tests whether the nested subword up to the associated $\langle/a\rangle$ ends with $\langle b\rangle\langle/b\rangle$, ending up in state q_f if this is indeed the case and thus successfully terminating this sub-run when the associated $\langle/a\rangle$ is read. The test in

state q_t whether the current subword ends with $\langle b\rangle\langle /b\rangle$ is done by simply traversing the subword (pushing and popping hierarchical states p'_a, p'_b in order to not interfere with the prior traversal in state q_0), guessing existentially on each $\langle b\rangle$ tag whether it is the final one (entering state q_f if this is the case and staying in q_t otherwise), and accepting if it is immediately followed by a single $\langle /b\rangle$ tag and the $\langle /a\rangle$ tag ending the subword to be tested.

As the complexity results for context-free games on nested words in Chapter 6 will be proven by a procedure that relies strongly on ANWA (much like the one in Section 3.4 relies on AFA), the complexity for both the membership and non-emptiness problem for ANWA are of relevance for the algorithms in that chapter.

Theorem 5.14.

(a) Non-emptiness for ANWAs is 2-EXPTIME*-complete.*

(b) The membership problem for ANWAs is PSPACE*-complete.*

Proof. Statement (a) follows easily from [Boz07] where 2-EXPTIME-completeness of Emptiness for alternating visibly pushdown automata was shown. The lower bound in that paper only requires finite well-nested words.

Towards the upper bound in (b), it is easy to see that an ANWA $A = (Q, \Sigma, \delta, q_0, F)$ on a nested word w can be simulated by an alternating Turing machine with polynomial time bound, hence the classical results from [CKS81] yield a polynomial space upper bound.

For future reference, we note that this computation can be done in polynomial space in $|w|$ and the size $|Q|$ of A's set of states, whenever the following two conditions can be tested in polynomial space.

- For a given set $X \subseteq Q \times Q$ of pairs of states, a symbol $a \in \Sigma$ and a state $q \in Q$, does $X \models \delta(q, a)$ hold?
- For a given set $X \subseteq Q$ of states, a symbol $a \in \Sigma$ and states $p, q \in Q$, does $X \models \delta(q, p, a)$ hold?

Note that these conditions are satisfied if A has a polynomial-sized state set and an exponential-sized transition function, which is the case for all ANWAs required in our constructions in Chapter 6. The proof of this statement is along the same lines as the proof that alternating polynomial time is contained in polynomial space: The tree of all possible computations has polynomial depth and can be analysed with polynomial space.

The lower bound in (b) is shown by a reduction from QBF, that is, the problem to decide whether a quantified Boolean formula evaluates to true. We assume that the given formula is of the form $\Phi = Q_1 x_1 \dots Q_n x_n \varphi(x_1, \dots x_n)$ with $Q_i \in \{\exists, \forall\}$ and a boolean formula φ with m clauses in conjunctive normal form.

The idea behind this reduction is to transform Φ into an ANWA A and a nested word w such that Φ is true if and only A accepts w. Actually, w is of a very simple form: $\langle v_1\rangle \cdots \langle v_n\rangle\langle X\rangle\langle /X\rangle\langle /v_n\rangle \cdots \langle /v_1\rangle$.

If the automaton A reads an opening tag $\langle v_i \rangle$, it guesses existentially an assignment for the variable x_i, if x_i is existentially quantified, and it branches universally, if x_i is universally quantified, thus choosing a truth assignment α for the variables. It saves this assignment in its hierarchical state stack by pushing the assignment to each x_i when $\langle v_i \rangle$ is read. Finally, when it reads $\langle X \rangle$, A branches universally, picking one of the m clauses of φ in every branch and saving that clause to its linear state. When it reads the suffix $\langle /v_n \rangle \cdots \langle /v_1 \rangle$ of w, A tests in a straightforward way that α satisfies the chosen clause by checking whether some hierarchical state chosen on the prefix $\langle v_1 \rangle \cdots \langle v_n \rangle$ corresponds to a literal appearing in that clause. □

5.4.2. Simple ANWA

We now define the alternating variant of SNWA by defining a notion of simplicity for ANWA. The details of the definition are somewhat technical but guarantee a comparatively low complexity for algorithmic problems and make sure that (like for all alternating automata models examined so far), SNWA themselves may also be interpreted as alternating SNWA.

Definition 5.15. An ANWA $A = (Q, \Sigma, \delta, q_0, F)$ with $Q \subseteq S \times \Delta$ (for some state set S and type alphabet Δ) is *simple* (SANWA), if it has the following two properties.

- *(Horizontal simplicity)* There are a *local acceptance function* $F_{\text{loc}} : \Sigma \to \mathscr{P}(Q)$, a *test state* $q_? \in Q$, and a *target state function* $t : Q \times \Sigma \to Q$, such that the transition function δ of A satisfies the following conditions:
  - $\delta(q, q', \langle /a \rangle) = t(q', a)$ for all $q \in Q$ and $q' \neq q_?$;
  - $\delta(q, q_?, \langle /a \rangle) = \begin{cases} \text{true}, & \text{if } q \in F_{\text{loc}}(a) \\ \text{false}, & \text{if } q \notin F_{\text{loc}}(a) \end{cases}$

  Furthermore, for each $(p, X) \in Q$ and $a \in \Sigma$, it holds that $t((p, X), a) = (p', X)$ for some $p' \in S$.

- *(Vertical Simplicity)* For each $X \in \Delta$ and $a \in \Sigma$, there is a $q \in Q$ such that for all $p \in S$ it holds that $\delta((p, X), \langle a \rangle) \in \mathcal{B}^+(\{q\} \times ((S \times \{X\}) \cup \{q_?\}))$.

Essentially, horizontal simplicity states that A has two kinds of computations on a well-nested subword: (1) computations starting from a pair $(q, q_?)$ test a property of the subword and can either succeed or fail at the end of the subword (and thus influence the overall computation); (2) computations starting from a pair (q, q') for $q' \neq q_?$ basically ignore the subword. Even though they may branch in an alternating fashion, the state after the closing tag $\langle /a \rangle$ is the same in all subruns, is determined by $t(q', a)$ and has the same type as q'.

Vertical simplicity, on the other hand, states that all alternation in A happens in the choice of hierarchical states – while, on an opening tag, A may branch into sub-runs pushing different hierarchical states onto the stack, the choice of linear follow-up state is "locally deterministic", depending only the type of the previous state of A and the label

of the tag being read, and the current type is preserved in all hierarchical states except for $q_?$. Together, these two conditions guarantee that SNWAs may also be interpreted as SANWAs.

Proposition 5.16.

(a) Non-emptiness for SANWA is PSPACE*-complete.*

(b) The membership problem for SANWA is decidable in polynomial time.

To prove that non-emptiness for SANWAs is in PSPACE, we start off by proving a somewhat stronger result: that the problem to determine, given a NWA A and a SANWA B, whether there is a nested word accepted by both A and B, is in PSPACE. The standard approach for proving a result of this sort (a product construction between two NWA or two SANWA) is generally not feasible here, as SANWA are less expressive than NWA (so A cannot in general be transformed into a SANWA) and transforming B into a NWA might incur a doubly exponential blow-up in size. Therefore, a PSPACE algorithm has to be constructed especially for this problem and uses the following pumping property for strings in $L(A) \cap L(B)$.

Lemma 5.17. *Let $A = (Q_A, P_A, \Sigma, \delta_A, q_{0,A}, F_A)$ be a NWA, and let furthermore $B = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$ be a SANWA with type alphabet Δ, final state function F_{loc}, target state function t and test state $q_? \in Q_B$. Then $L(A) \cap L(B) \neq \varnothing$ if and only if there exists a string in $L(A) \cap L(B)$ of width at most $2^{|Q_B|} \cdot |\Sigma| \cdot |Q_A|$ and depth at most $3(|\Sigma| + 1)|Q_A|^2|\Delta|$.*

The proof of Lemma 5.17, as well as several others in this section, use the notion of *successful* (sub-)runs. So, in the remainder of this section, if ρ is a run of a SNWA B on some string w and ρ' is a sub-run of ρ on a nested substring w' of w, we call ρ' *successful* if all leaves of ρ' are accepting with respect to the context of w', i.e. if the linear state component of all leaves of ρ' at depth $|w'|$ is in F in case $w' = w$, or in $F_{\mathrm{loc}}(a)$ in case $\langle a \rangle w' \langle / a \rangle$ is a substring of w. Note that due to the definition of runs, all test subruns of a successful subrun ρ' on w' (i.e. all sub-subruns starting with horizontal state $q_?$) have to accept by terminating in a leaf of depth less than $|w'|$.

Proof. The "if" direction is trivial. For "only if", assume for the sake of contradiction that $L(A) \cap L(B) \neq \varnothing$, but there is no string in $L(A) \cap L(B)$ fulfilling the claimed upper bounds on both width and depth.

First, we observe that for all words $w \in L(B)$, all nodes of any depth i in an arbitrary accepting run of B on w contain only linear states of the same type, i.e. if $\rho = (D, \lambda)$ is an accepting run of B on w, and $x, y \in D$ with $|x| = |y| = i$ for any $i \in \mathbb{N}$, and if $\lambda(x)_{\mathrm{lin}} = (p, X)$ and $\lambda(y)_{\mathrm{lin}} = (p', Y)$, then $X = Y$. This can be proven by a simple induction on i.

By this observation, all subtrees of a successful subrun ρ' of ρ immediately below its root start from the same state, as that state is uniquely given by the tags enclosing w' and the root type of ρ'.

First off, let $w \in L(A) \cap L(B)$ be a string of width greater than $2^{|Q_B|} \cdot |\Sigma| \cdot |Q_A|$ and minimal length. We now prove that $L(A) \cap L(B)$ contains a string shorter than w, in contradiction to the assumed minimality.

Let w' be a maximum-length nested substring of w with root width greater than $2^{|Q_B|} \cdot |\Sigma| \cdot |Q_A|$. Let ρ be an accepting run of B on w and ρ' its sub-run on w'. Similarly, since A may also be viewed as an ANWA, there is an accepting run π of A on w in which each non-leaf node has only a single child. Let π' be the sub-run of π on w'. For $k = 1, \ldots, |w'|$, let k-layer$(w') \in \Sigma \times ((\mathscr{P}(Q_B) \times P_A) \cup (\mathscr{P}(Q_B^2) \times (Q_A \times P_A)))$ such that

- if $w'_k = \langle a \rangle$, $(q_1, p_1), \ldots, (q_\ell, p_\ell)$ are all pairs of linear state and top-most hierarchical state at depth k in ρ' (i.e. after reading $\langle a \rangle$) and (q, p) is the pair of linear and top-most hierarchical state of depth k in π', then k-layer$(w') = (a, \{(q_1, p_1), \ldots, (q_\ell, p_\ell)\}, (q, p))$, and
- if $w'_k = \langle / a \rangle$, $(q_1), \ldots, (q_\ell)$ are all linear states at depth k in ρ' and q is the linear state at depth k in π', then k-layer$(w') = (a, \{q_1, \ldots, q_\ell\}, q)$.

As the root width of w' is greater than $|\Sigma \times \mathscr{P}(Q_B) \times Q_A|$, there are numbers $i < j < |w'|$ such that i-layer$(w') = j$-layer(w') and the substrings $w'_1 \cdots w'_i$ and $w'_1 \cdots w'_j$ (and therefore also the substring $w'_{i+1} \cdots w'_j$) are well-nested. It is easy (if tedious) to prove that there are accepting runs of A and B on the string $\tilde{w}$ derived from w by deleting $w'_{i+1} \cdots w'_j$ from w', yielding the desired contradiction.

Assume now, again for the sake of contradiction, that there is no string in $L(A) \cap L(B)$ of width at most $2^{|Q_B|} \cdot |\Sigma| \cdot |Q_A|$ and depth at most $3(|\Sigma| + 1)|Q_A|^2|\Delta|$. By the above part of the proof, this means that all strings fulfilling the requirement on width must be of a depth exceeding $3(|\Sigma| + 1)|Q_A|^2|\Delta|$. Let w be such a string of minimal length, and let ρ be an accepting run of B and π an accepting run of A on w.

As the nesting depth of w is greater than $3(|\Sigma| + 1)|Q_A|^2|\Delta|$, there exist well-nested words w' and w'' such that for some $a \in \Sigma$,

(1) $\langle a \rangle w' \langle / a \rangle$ is a substring of w,

(2) $\langle a \rangle w'' \langle / a \rangle$ is a substring of w',

(3) either all sub-runs of ρ on w' and w'' are unsuccessful or there exist successful runs in ρ on both w' and w'',

(4) all sub-runs of ρ on w' and w'' start from the same state $q_a \in Q_B$, and

(5) the linear states of A according to π before and after reading $\langle a \rangle w' \langle / a \rangle$ are the same as those before and after reading $\langle a \rangle w'' \langle / a \rangle$.

Conditions (1), (2) and (3) are due to the fact that the given size bound ensures that, for appropriately chosen w', w'' and a, the states of B before reading each $\langle a \rangle$ have the same type (as per the definition of SANWA); condition (4) then follows by the fact that the state of a SANWA after reading some opening tag is uniquely defined by the type of its previous state and the opening tag's label.

These conditions imply that both A and B have accepting runs on the string $\tilde{w}$ derived from w by replacing w' with w''. As w'' is a proper substring of w', this yields the desired contradiction to the minimal length of w and thus the statement of the lemma. □

Proposition 5.18. *There is an alternating algorithm that tests in polynomial time whether, for an NWA A and a SANWA B it holds $L(A) \cap L(B) \neq \varnothing$.*

Proof. We formulate the claimed algorithm as a game for two players, whom we will call Adam and Eve to avoid confusion with the players for context-free games. This game will always terminate after at most polynomially many rounds, so an alternating polynomial-time algorithm can easily be constructed from it by branching nondeterministically (resp. universally) for the moves for Eve (resp. Adam) and accepting the input if and only if Eve wins.

We will construct the game such that that Eve has a winning strategy on input NWA $A = (Q_A, P_A, \Sigma, \delta_A, q_{0,A}, F_A)$ and SANWA $B = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$ with final state function F_{loc}, test state $q_?$ and target state function t if and only if $L(A) \cap L(B) \neq \varnothing$. Eve's goal in this game is to prove that there is a string that is accepted by both A and B without writing down that string explicitly; by Lemma 5.17, it does suffice to examine strings of at most exponential width and polynomial depth, but such a string can still not be explicitly spelled out using only polynomial space. We therefore represent a string implicitly by the behaviour it induces in A and B.

Game positions for Eve consist of two states $p_1, p_2 \in Q_A$, a function $S : Q_B \to \mathscr{P}(Q_B)$ and two numbers $c, n \geqslant 0$, and the game is constructed in such a way that Eve has a winning strategy from position (q_1, q_2, S, c, n) if and only if there is a string $w \in \mathrm{WF}(\Sigma)$ of root width at most 2^c and nesting depth at most n such that $q_1 \overset{w}{\rightsquigarrow}_A q_2$, and for every $q \in Q_B$ there is a run of B on w beginning in q whose branches all end inside $S(q)$. We write c_0 for $|Q_B| \log(|\Sigma| \cdot |Q_A|)$ and n_0 for $3(|\Sigma| + 1)|Q_A|^2|\Delta|$. Lemma 5.17 then guarantees that $L(A) \cap L(B)$ is nonempty if and only if there is a state $q_f \in F_A$ and a function S with $S(q_{0,B}) \subseteq F_B$ such that Eve has a winning strategy from position $(q_{0,A}, q_f, S, c_0, n_0)$.

In any position (q_1, q_2, S, c, n), Eve has the following options:

- If $c > 0$, she may choose to play a *concatenation round*, asserting that $w = v_1 v_2$ for strings $v_1, v_2 \in \mathrm{WF}(\Sigma)$ whose root width is at most half that of w. In this case, she chooses two functions $S_1, S_2 : Q_B \to \mathscr{P}(Q_B)$, corresponding to strings v_1, v_2 as above and an "in-between" state $q' \in Q_A$. The functions S_1 and S_2 have to fulfil the condition that for each $q \in Q_B$ it holds that $S(q) = \bigcup_{p \in S_1(q)} S_2(p)$; if S_1 and S_2 do not fulfil this condition, Adam wins. Otherwise, Adam has a choice of which part of Eve's assertion he wants to contest, so he may choose as a follow-up position either $(q_1, q', S_1, c-1, n)$ or $(q', q_2, S_2, c-1, n)$.

- If $n > 0$, Eve may choose to play a *nesting round* (with $a \in \Sigma$), asserting that $w = \langle a \rangle v \langle / a \rangle$ for some $v \in \mathrm{WF}(\Sigma)$. To this end, she first chooses an alphabet symbol a and a function S' corresponding to v as above, as well as states $q_1', q_2' \in Q_A$, $p \in P_A$ such that $\delta_A(q_1, \langle a \rangle) = (q_1', p)$ and $\delta_A(q_2', p, \langle / a \rangle) = q_2$ (if no such states exist, Adam

wins immediately). Next, Adam chooses some $q \in Q_B$ on which to contest Eve's claim. In response, Eve picks a state $p' \in Q_B$ and a set of states $X = \{p_1, .., p_k\} \subseteq Q_B$ such that $(\{p'\} \times X) \models \delta_B(q, \langle a \rangle)$ and $S(q) = \bigcup_{p \in X \setminus \{q_?\}} \{t(p, a)\}$. If she cannot choose such a set, Adam wins.

If Adam has not won by this point, he has to contest Eve's claim that there is a string v such that B has a successful run on v. If $q_? \in X$ and $S'(p') \not\subseteq F_{\mathrm{loc}}(a)$, the string v claimed by Eve fails the test subrun mandated by B branching with $q_?$, so in this case, Adam wins. Otherwise, the game continues from position $(q'_1, q'_2, S', c_0, n-1)$, as the root width of the substring v is bounded by 2^{c_0}.

- Eve may choose to *solve (with $a \in \Sigma$)*, asserting that $w = \langle a \rangle \langle / a \rangle$. In this case, she chooses a symbol $a \in \Sigma$. Similar to a nesting round, Adam then picks a state $q \in Q_B$ on which to contest Eve's claim, to which Eve responds by choosing a state $p \in Q_B$ and a set $X \subseteq Q_B$. The game then ends and a winner is determined. Eve wins if and only if all of the following conditions are fulfilled:
  (a) There are states $q' \in Q_A$, $p' \in P_A$ such that both $\delta_A(q_1, \langle a \rangle) = (q', p')$ and $\delta_A(q', p', \langle / a \rangle) = q_2$;
  (b) $(\{p\} \times X) \models \delta_B(q, \langle a \rangle)$;
  (c) $S(q) = \bigcup_{p \in X \setminus \{q_?\}} \{t(p, a)\}$;
  (d) If $q_? \in X$, then $p \in F_{\mathrm{loc}}(a)$.
- Eve may choose to *solve with ϵ*, asserting that $w = \epsilon$. In this case, the game ends and Eve wins if and only if $q_1 = q_2$ and for each $q \in Q_B$ it holds that $S(q) = \{q\}$.

Each round that does not end the game decreases the number of remaining nesting or concatenation rounds; furthermore, the number of remaining concatenation rounds only increases at the end of a nesting round. Therefore, the total number of rounds starting from $(q_{0,A}, q_f, S, c_0, n_0)$ is bounded by $c_0 n_0$, which is polynomial in the size of A and B. It is easy to see that each choice by Eve or Adam requires only a polynomial-size certificate, and that each check for winning conditions is computable in polynomial time. Therefore, an alternating algorithm checking whether Eve has a winning strategy on this game (as described above) has a polynomial upper bound on its running time. It remains to be shown that this algorithm indeed tests A and B for intersection emptiness, i.e. that Eve has a winning strategy from $(q_{0,A}, q_f, S, c_0, n_0)$ for some $q_f \in F_A$ if and only if $L(A) \cap L(B) \neq \varnothing$.

To prove this claim, we show that the following statements are equivalent:

(1) Eve has a winning strategy from position (q_1, q_2, S, c, n);

(2) There is a string $w \in \mathrm{WF}(\Sigma)$ of width at most $2^{|Q_B|}|\Sigma||Q_A|$, root width at most 2^c and depth at most n such that there is a run of A on w from q_1 to q_2, and for each $q \in Q$, there is a successful run of B on w from q ending with linear state components inside $S(q)$.

(1) ⇒ (2): Assume that Eve has a winning strategy σ from position (q_1, q_2, S, c, n). We prove (2) by induction on the structure of σ.

If Eve solves with ϵ as her first move according to σ, the string $w = \epsilon$ obviously fulfils the claim of (2).

If Eve's first move according to σ is to solve with some $a \in \Sigma$, then $w = \langle a \rangle \langle / a \rangle$ fulfils the claim of (2). Since $c, n \geqslant 0$, w fulfils the desired upper bounds on nesting depth and width; winning condition (a) ensures the existence of a run of A; and as for each $q \in Q$ that Adam chooses, Eve can respond with a set of horizontal states compliant with the transition formulae of B according to winning conditions (b) to (d), the desired runs of B on w exist as well.

If Eve begins with a concatenation round according to σ, it follows that there exist a state $q' \in Q_A$ and functions $S_1, S_2 : Q_B \to \mathscr{P}(Q_B)$ such that for each $q \in Q_B$ it holds that $S(q) = \bigcup_{p \in S_1(q)} S_2(p)$ and Eve has a winning strategy on both $(q_1, q', S_1, c-1, n)$ and $(q', q_2, S_2, c-1, n)$. By induction, this implies that there are strings $v_1, v_2 \in \mathrm{WF}(\Sigma)$ of width at most $2^{|Q_B|}|\Sigma||Q_A|$, root width at most 2^{c-1} and depth at most n for which there exist appropriate runs of A and B; it is easy to see that $w = v_1 v_2$ fulfils the width and depth requirements of the claim, and that the claimed runs of A and B on w can be constructed by combining those on v_1 and v_2.

If Eve starts by playing a nesting round with some $a \in \Sigma$, there exists a function S' as well as states $q_1', q_2' \in Q_A$, $p' \in P_A$ such that $\delta_A(q_1, \langle a \rangle) = (q_1', p')$ and $\delta_A(q_2', p', \langle / a \rangle) = q_2$. Furthermore, for each $q \in Q_B$, there is a state $p \in Q_B$ and a set $X \subseteq Q_B$ such that $(\{p\} \times X) \models \delta_B(q, \langle a \rangle)$ and $S(q) = \bigcup_{p \in X \setminus \{q_?\}} \{t(p, a)\}$, and if $q_? \in X$ then $S'(p) \subseteq F_{\mathrm{loc}}(a)$. Finally, Eve has a winning strategy starting from position $(q_1', q_2', S', c_0, n-1)$.

By induction, there exists a string v of width at most $2^{|Q_B|}|\Sigma||Q_A|$ and depth at most $n-1$ for which there exist appropriate runs of A and B; the string $w = \langle a \rangle v \langle / a \rangle$ therefore fulfils the claimed restrictions on depth and width. Again, it is easy to see that a run of A on w can be constructed from the one on v. To construct the desired runs of B on w, denote the run on v starting at p and ending with leaf labels in $S'(p) \times Q^*$ by ρ and let $q \in Q_B$. A run on w starting at q is then constructed as follows: The root node, labelled q, has $\{p\} \times P$ as the set of labels of its children. Each of these nodes (p, p') is the root of a copy of ρ, modified by replacing all node labels of the form (r, α) with $(r, \alpha p')$, whose leaves all have labels inside $S'(p) \times Q^*$; if $p' = q_?$, the leaves of the corresponding copy of ρ have no further children; otherwise, their only child is labelled with $(t(p', a), \epsilon)$. Using the above properties and the definition for SANWA semantics, it is easy to verify that the tree thus constructed is indeed a run of B on w starting at q and ending with linear states inside $S(q)$.

(2) ⇒ (1): This part of the proof is by an induction on the structure of w analogous to the above proof of (1) ⇒ (2). □

The polynomial-time upper bound for the membership problem uses an algorithm similar to the one for the membership problem for AFA (cf. Proposition 2.7). Similar to AFA, we state here a somewhat more precise result, which also implies that the membership problem for SANWA with a polynomial-sized state set and an exponential-sized

transition function can still be solved in polynomial space (as opposed to exponential time).

Proposition 5.19. *There is an algorithm deciding, given a SANWA $A = (Q, \Sigma, \delta, q_0, F)$ and nested word $w \in WF(\Sigma)$, whether $w \in L(A)$ holds. This algorithm has a runtime in $\mathcal{O}(|w| \cdot |Q| \cdot poly(|\delta|))$ and uses space in $\mathcal{O}(|w| \cdot |Q| + \log(|\delta|))$.*

Proof. The basic idea behind the algorithm is the same as the one for the algorithm from Proposition 2.7 deciding the membership problem for AFAs – it reads the input string w backwards and stores in a set S all the states from which there is an accepting run on the suffix of the input string it has read so far. The only complication compared with the algorithm from Proposition 2.7 is the fact that the algorithm for SANWAs has to take nesting of strings into account. This complication, however, can be dealt with quite easily due to the fact that, in substrings of the form $\langle a \rangle v \langle / a \rangle$ (with $v \in \mathrm{WF}(\Sigma)$), the only influence of the substring v on the behaviour of the SANWA A is the possibility of forcing A to reject the input string if its sub-run on v is unsuccessful. Apart from this, the sub-run of A on v has no influence on the state of A after reading $\langle a \rangle v \langle / a \rangle$, so our algorithm can basically just use the idea of the algorithm for AFAs on each "level" of the input string (i.e. on the string of root labels and the child-string of each node, when interpreting w as a forest).

Before describing the algorithm in detail, however, we first have to introduce some notation. By the vertical simplicity of A, for each type $X \in \Delta$ and alphabet symbol $a \in \Sigma$, there is some unique state $q_{X,a} \in Q$ such that A always enters linear state $q_{X,a}$ when reading $\langle a \rangle$ from a state of type X. To simplify presentation of the algorithm, we imagine w to be rooted inside of "pseudo-tags" with label $0 \notin \Sigma$ and type $X_0 \notin \Delta$ such that $q_{X_0,0} = q_0$ is the starting state of A and $F_{\mathrm{loc}}(0) = F$ are the accepting states of A. If A enters a substring $v \in \mathrm{WF}(\Sigma)$ of w with an $\langle a \rangle$-tag (for $a \in \Sigma \cup \{0\}$) from a state with type $X \in \Delta \cup \{X_0\}$, we call (X, a) the *context* of v, and we note that the context of each substring of w has to be the same in each run of A on w due to the simplicity conditions, and is therefore well-defined.

We can now define the algorithm more formerly. Algorithm 3 summarises the algorithm $\mathrm{ExistsRun}(A, w, X, a)$ which, on input of a SANWA A, string $w \in \mathrm{WF}(\Sigma)$, type $X \in \Delta \uplus \{X_0\}$ and alphabet symbol $a \in \Sigma \uplus \{0\}$, checks whether there is a successful sub-run of A on w with respect to the context (X, a). Clearly, when called on the context $(X_0, 0)$, this algorithm then tests whether there is an accepting run of A on w, i.e. if $w \in L(A)$.

Intuitively, each iteration of the while-loop in Algorithm 3 consumes a rooted suffix of w', determining and updating the set S of states from which there is a successful sub-run on the complete suffix that has been deleted from w so far; once all of w has been consumed in this matter, the algorithm accepts if the initial state in the given context is one such state. To that end, for a suffix $\langle b \rangle v \langle / b \rangle$, Line 6 determines the set S' of states from which a valid $\langle / b \rangle$ transition is possible, with Lines 7 and 8 adding $q_?$ to that set if and only if there is a successful sub-run of A on v. Line 9 then updates S to the set of states allowing a valid $\langle b \rangle$ transition before that.

Towards the correctness of Algorithm 3, we prove that $\mathrm{ExistsRun}(A, w, X, a)$ accepts if and only if there is a successful sub-run of A on w in the context (X, a), i.e. a run of

Algorithm 3 ExistsRun(A, w, X, a)

```
1:  S ← F_loc(a)
2:  w' ← w
3:  X' ← (type of q_{X,a})
4:  while w' ≠ ε do
5:     Let w' = u⟨b⟩v⟨/b⟩ with u, v ∈ WF(Σ), b ∈ Σ
6:     S' ← {q ∈ Q | t(q, b) ∈ S}
7:     if ExistsRun(A, v, X', b) accepts then
8:        S' ← S' ∪ {q_?}
9:     S ← {q ∈ Q | ({q_{X',b}} × S') ⊨ δ(q, ⟨a⟩)}
10:    w' ← u
11: Accept iff q_{X,a} ∈ S
```

a on w starting at $q_{X,a}$ and ending with linear states inside $F_{\text{loc}}(a)$ whose test subruns are all successful. The main proof is by induction on the length of w.

For $|w| = 0$, i.e. $w = \epsilon$, Algorithm 3 doesn't enter its while-loop and simply tests whether $q_{X,a} \in F_{\text{loc}}(a)$. This is obviously the case if and only if the desired sub-run on ϵ indeed exists.

For $|w| > 0$, we prove that the while-loop in Algorithm 3 has the following invariant: If, at the beginning of the loop, $w = w'w''$ holds, i.e. if w'' is a (possibly empty) suffix of w that has already been deleted by previous iterations of Line 10, then for each $q \in S$, there is a successful sub-run of A on w'' in context (X, a) starting at q. Before the first execution of the loop (i.e. for $w'' = \epsilon$), this is obviously true.

During a single iteration of the while-loop, the invariant is conserved, as the following argument shows. Let S_0 be the value of S before and S_1 its value after an execution of the loop, and let $q \in S_1$. We show that there is a successful sub-run of A on $\langle b\rangle v\langle/b\rangle w''$ from q, assuming successful sub-runs on w'' from each $q' \in S_0$. To that end, we show that there is a sub-run ρ of A on $\langle b\rangle v\langle/b\rangle$ from q whose leaves are labelled with linear states in S_0; this sub-run can then easily be extended to a sub-run on $\langle b\rangle v\langle/b\rangle w''$. The root of ρ is labelled q, and its children with all the states in $\{q_{X',b}\} \times S'$. Any such child labelled $(q_{X',b}, q'')$ for some $q'' \neq q_?$ is the root of an arbitrary (not necessarily successful) sub-run of A on v, all leaves of which are extended by an additional child labelled $(t(q'', b), \epsilon)$, with $t(q'', b) \in S_0$ by Lines 6 and 9 of the algorithm. If there is a child of the root labelled $(q_{X',b}, q_?)$, then $q_? \in S'$; by induction and Line 7, this means that there is a successful sub-run of A on v which can be used instead of the arbitrary sub-run on v used before. Lines 6-9 then guarantee that the run ρ thus constructed is indeed a sub-run of A on $\langle b\rangle v\langle/b\rangle$ starting at q and ending with linear states inside S_0.

To prove the upper bound on the runtime of Algorithm 3, we observe that, including recursive calls, Lines 6 and 9 are executed exactly once for each closing and opening tag in w, respectively. Line 6 can be executed in time $\mathcal{O}(|Q|)$, and line 9 in time $\mathcal{O}(|Q| \cdot |\delta|)$, using the fact that evaluating boolean formulas is in NC^1[BCGR92]. This yields the desired $\mathcal{O}(|w| \cdot |Q| \cdot |\delta|)$ upper bound on running time. For the upper bound on space,

Lines 6 and 9 use space at most $\mathcal{O}(|Q|+\log(|\delta|))$ (again using boolean formula evaluation on NC^1), and each recursive call of Algorithm 3 uses at most $\mathcal{O}(|Q|)$ additional storage space. Finally, the recursion depth of Algorithm 3 is bounded by the depth of w, leading to the $\mathcal{O}(|w| \cdot |Q| + \log(|\delta|))$ bound on space. □

Now we are prepared to prove the complexity results for SANWAs.

Proof of Proposition 5.16. The PTIME upper bound for membership in (b) was proven as Proposition 5.19.

That non-emptiness for SANWAs is in PSPACE follows directly from Proposition 5.18, as alternating polynomial time equals polynomial space.

PSPACE-hardness can be proven by a reduction from the nonemptiness problem for alternating finite automata, interpreting flat strings $w_1 \dots w_n \in \Sigma^*$ as nested words $\langle w_1\rangle\langle/w_1\rangle \dots \langle w_n\rangle\langle/w_n\rangle \in \mathrm{WF}(\Sigma)$ of nesting depth 0 (and vice versa). It is then quite easy to construct from an AFA B' a SANWA B such that $L(B') \neq \varnothing$ if and only if B accepts some nested word of depth 0. The SANWA B uses test subruns to verify that its input string has depth 0, stays in its current state with each opening transition and simulates a transition of B' with input a on each closing transition on some $\langle/a\rangle$. To do so, we simply assign the same type to all states of B' for constructing B and set the target state function of B to be equal to the transition function of B'.

Together, PSPACE-completeness of non-emptiness follows.

□

5.5. Outlook and bibliographical remarks

This chapter summarised the basic prior results on nested words and nested word automata. It also introduced the concept of simple nested word automata, which can be generated by a polynomial-time transformation from single-type tree grammars, and thus from practical schema languages for XML. If we relax the definition of single-type tree grammars to allow for arbitrary regular content models given by DFAs (instead of deterministic regular content models given by DREs), the transformation from the proof of Proposition 5.10 can easily be reversed, yielding an equivalence result between this kind of tree grammar and SNWAs. Since the rest of this dissertation only requires the result of Proposition 5.10, however, details on the inverse transformation are omitted here.

In addition, this chapter presented alternating versions of both types of nested word automata, as well as complexity results for the basic decision problems of membership and non-emptiness of alternating automata. It turns out that, while even the membership problem for alternating NWA is intractable (assuming PTIME ≠ PSPACE), simple alternating NWA have complexity properties very similar to alternating *finite-state* automata. Intuitively, this may be chalked up to the fact that a SANWA can be seen as nothing more than a collection of (polynomially many) AFAs.

While not a focus of this dissertation, simple nested word automata may indeed be of independent scientific interest. While there is lot of automata-theoretic study of XML schema languages, most of it focusses on types of automata that operate on trees instead

of nested words (cf. e.g. [Sch07]). Notable exceptions include [SV02; SS07], which examine validation of (nested) string representations of XML documents by DFAs.

Bibliographical remarks. Nested word automata were introduced in [AM09] as a refinement of *visibly pushdown automata* [AM04] working on nested words. As a notable difference between this work and [AM09], their formalisation allowed for internal symbols and incomplete nested words (with unmatched opening/closing tags), which are not considered here.

The "ancestor-sibling-based" behaviour of simple NWA is basically that of *deterministic top-down NWA* from [AM09]; the typing of states introduced here, which yields equivalence to tree grammars, however, is (to the best of the author's knowledge) new.

All original work in this chapter was first published as joint work with Thomas Schwentick in [SS15] (and that paper's journal version [SS16]), although the proof given there for the complexity of SANWA was severely revised to fit the more precise statement of Proposition 5.19. The idea behind simple NWA is based on a characterisation of XML Schema by *extended DTDs with ancestor-sibling based types* [MNSB06].

6. Winning problems for games on nested words

Having defined nested words, as well as automata models operating on nested words, in the previous chapter, we now turn towards context-free games on nested words, defined in Section 6.1. As in Chapter 3, the focus of this chapter is on characterising the complexity of the winning problem for Juliet in various classes of context-free games on nested words. The main classes of interest here are those where target languages are given by (deterministic) nested word automata or by simple NWA; complexity results for these classes are summarised in Table 6.1. The upper bound proofs for these results, given in Section 6.2, allow us to reap the rewards of having set up a generic framework for dealing with context-free games in Chapter 3; in fact, apart from some modifications to deal with the more complex structure of nested words, the upper bound proofs follow the exact same pattern as in Section 3.4.

The corresponding lower bounds, proven in Section 6.3, present an interesting addition, as they effectively show a very close connection between the winning problem for Juliet with or without replay and the nonemptiness or membership problem of alternating automata.[1]

In previous literature (most notably [MSS06]), slight variations to the basic workings of context-free games on flat strings were already explored. One example of this are games with *symmetric rule choice*, where we add function symbols for which Juliet instead of Romeo may choose a replacement string after a *Call* move; a different sort of variation, which was first examined in [SS15] in the context of cfGs on nested words, are games where function call results do not replace the original context but are inserted after it instead. Sections 6.4 and 6.5 deal with these two exemplary variations and explain how the generic framework from Chapter 3 can be modified to accommodate them.

6.1. Definitions

This section defines context-free games (cfGs) on nested words with regular replacement. For the most part, the definitions given here follow those for cfGs on flat strings with regular replacement in Section 3.1; however, due to the more intricate structure of nested words, the semantics of *Call* moves for Juliet gets slightly more involved.

Intuitively, just like for cfGs on flat strings, Juliet moves her focus through the input word from left to right, processing each opening or closing tag in turn, deciding on each

[1] In fact, a similar connection could have been used to prove lower bounds in Chapter 3, but the actual lower bounds given there are somewhat stronger for using DREs instead of DFAs.

	No replay	Bounded	Unbounded
Regular target language			
Regular replacement	PSPACE	2-EXPTIME	2-EXPTIME
Finite replacement	PSPACE	PSPACE	EXPTIME
DTD or XML Schema target language			
Regular replacement	PTIME	PSPACE	EXPTIME
Finite replacement	PTIME	PTIME	EXPTIME

Table 6.1.: Summary of complexity results for cfGs on nested words. All results are completeness results.

closing tag labelled with a function symbol whether to play *Read* or *Call*. On opening tags, tags labelled by a non-function symbol as well as closing tags on which JULIET plays *Read*, the focus simply shifts one symbol to the right; however, if JULIET plays *Call* on a closing tag $\langle/a\rangle$ with a function symbol a, the entire rooted subword between that $\langle/a\rangle$ tag and its associated $\langle a\rangle$ tag gets replaced with a nested word chosen by ROMEO from the regular replacement language R_a for a, and the focus is re-set immediately to the left of the newly inserted subword.

It should be noted that the winning chances of a game do not change if JULIET is allowed to play *Call* moves at opening tags: if JULIET wants to play *Call* at an opening tag she can simply play *Read* until the focus reaches the corresponding closing tag and play *Call* then. On the other hand, if she can win a game by calling a closing tag, she can also win it by calling the corresponding opening tag, thanks to the fact that context-free games are reachability games with determinacy and complete information (cf. [GTW02]) and that ROMEO'S possible responses to a *Call* on some subword depend only on the root label of that subword and nothing else.

Definition 6.1. A *context-free game (cfG) on nested words* $G = (\Sigma, \Gamma, R, T)$ consists of a finite alphabet Σ, a set $\Gamma \subseteq \Sigma$ of *function symbols*, a *(replacement) rule set* $R \subseteq \Gamma \times \mathrm{WF}(\Sigma)$ and a *target language* $T \subseteq \mathrm{WF}(\Sigma)$. Again, we only consider the case where T and, for each symbol $a \in \Gamma$, the set $R_a \stackrel{\text{def}}{=} \{u \mid (a, u) \in R\}$ are non-empty regular nested word languages.

A *configuration* is a tuple $\kappa = (p, u, v) \in \{\mathrm{J}, \mathrm{R}\} \times (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^* \times (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$ where p is the player to move, $uv \in \mathrm{WF}(\Sigma)$ is the *current word*, and, if $v \neq \epsilon$, the first symbol of v is *the current symbol*. We call $|u| + 1$ the *current position* of κ. A configuration (p, u, v) with $v \neq \epsilon$ is called *playable*, otherwise it is *final*.

The configuration $\kappa' = (p', u', v')$ is a *successor configuration* of the playable configuration $\kappa = (p, u, v)$ (Notation: $\kappa \to \kappa'$) if one of the following holds:

(1) $p' = p = \mathrm{J}$, $u' = us$, and $sv' = v$ for some $s \in \langle\Sigma\rangle \cup \langle/\Sigma\rangle$ (JULIET plays *Read*);

(2) $p = \mathrm{J}$, $p' = \mathrm{R}$, $u = u'$, $v = v' = \langle/a\rangle z$ for $z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $a \in \Gamma$, (JULIET plays *Call*);

(3) $p = \mathrm{R}$, $p' = \mathrm{J}$, $u = x\langle a\rangle y$, $v = \langle/a\rangle z$ for $x, z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $y \in \mathrm{WF}(\Sigma)$, $u' = x$ and $v' = y'z$ for some $y' \in R_a$ (ROMEO plays y').

Using these definitions for (successor) configurations, each context-free game G again induces a reachability game as described in Definition 3.1, with an arena and winning positions as defined there. The definitions of plays and (winning) strategies again carry over from Section 2.3, with strategies σ for Juliet mapping configurations κ to moves $\sigma(\kappa) \in \{\mathit{Read}, \mathit{Call}\}$ and strategies τ for Romeo mapping configurations κ to nested words $\tau(\kappa) \in \mathrm{WF}(\Sigma)$. Strategies for Juliet will usually be denoted by $\sigma, \sigma', \sigma_1, \ldots$ and strategies for Romeo by $\tau, \tau', \tau_1, \ldots$.

The play $\Pi(w, \sigma, \tau)$ induced by strategies σ for Juliet and τ for Romeo on a nested word $w \in \mathrm{WF}(\Sigma)$ and the *Call* depth $\mathrm{Depth}^G(\sigma, w)$ of strategy σ on a nested word w, as well as finite strategies and strategies of bounded *Call* depth are defined as for cfGs on flat strings in Definition 3.1.

We denote the set of all finite strategies for Juliet in the game G by $\mathrm{STRAT}_\mathrm{J}(G)$, the set of all finite strategies of k-bounded *Call* depth in G by $\mathrm{STRAT}^k_\mathrm{J}(G)$, and the set of all strategies for Romeo by $\mathrm{STRAT}_\mathrm{R}(G)$. By $\mathrm{JWin}(G)$ we denote the set of all words for which Juliet has a winning strategy in $\mathrm{STRAT}_\mathrm{J}(G)$, and likewise for $\mathrm{JWin}^k(G)$ and $\mathrm{STRAT}^k_\mathrm{J}(G)$.

Example 6.2 (Play of a cfG on nested words)**.** *Consider the context-free game $G = (\Sigma, \Gamma, R, T)$ with alphabet $\Sigma = \{r, a, b, f, g\}$, function symbols $\Gamma = \{f, g\}$, replacement languages $R_f = \{\langle b\rangle(\langle a\rangle\langle /a\rangle)^n\langle /b\rangle \mid n \geqslant 1\}$, $R_g = \{\langle f\rangle^n\langle /f\rangle^n \mid n \geqslant 1\}$ and target language $T = \{\langle r\rangle\langle b\rangle(\langle a\rangle\langle /a\rangle)^n\langle /b\rangle\langle /r\rangle \mid n \geqslant 1\}$. A play on the input word $w = \langle r\rangle\langle g\rangle\langle b\rangle\langle /b\rangle\langle /g\rangle\langle /r\rangle$ might proceed as follows:*

- Juliet *moves the focus to $\langle /g\rangle$ and plays Call.*
- Romeo *plays $\langle f\rangle\langle f\rangle\langle /f\rangle\langle /f\rangle \in R_g$, replacing $\langle g\rangle\langle b\rangle\langle /b\rangle\langle /g\rangle$ and leading to the nested word $w' = \langle r\rangle\langle f\rangle\langle f\rangle\langle /f\rangle\langle /f\rangle\langle /r\rangle$.*
- Juliet *moves the focus to the first $\langle /f\rangle$ and plays Read, leaving w' unchanged.*
- Juliet *moves the focus to the second $\langle /f\rangle$ and plays Call.*
- Romeo *plays $\langle b\rangle\langle a\rangle\langle /a\rangle\langle a\rangle\langle /a\rangle\langle a\rangle\langle /a\rangle\langle /b\rangle \in R_f$, thus rewriting w' into the nested word $w'' = \langle r\rangle\langle b\rangle\langle a\rangle\langle /a\rangle\langle a\rangle\langle /a\rangle\langle a\rangle\langle /a\rangle\langle /b\rangle\langle /r\rangle$.*
- Juliet *wins the game, as $w'' \in T$.*

The canonical tree representations of nested words in this play are depicted in Figure 6.1.

Example 6.3 (Strategies for cfGs on nested words)**.** *In the game $G = (\Sigma, \Gamma, R, T)$ with alphabet $\Sigma = \{a, b, f, g\}$, function symbols $\Gamma = \{f, g\}$, replacement languages $R_f = \{\langle a\rangle\langle /a\rangle, \langle b\rangle\langle /b\rangle\}$, $R_g = \{\langle f\rangle^n\langle /f\rangle^n \mid n \geqslant 1\}$ and target language*

$$T = \{\langle a\rangle\langle /a\rangle\langle g\rangle\langle /g\rangle, \langle b\rangle\langle /b\rangle\langle a\rangle\langle /a\rangle, \langle b\rangle\langle /b\rangle\langle b\rangle\langle /b\rangle\},$$

Juliet *has a winning strategy σ on the input nested word $w = \langle f\rangle\langle /f\rangle\langle g\rangle\langle /g\rangle$, which proceeds as follows:*

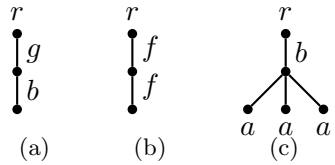


Figure 6.1.: Example play for a cfG as described in Example 6.2.
6.1a: Initial nested word;
6.1b: After *Call* to g and replacement by $\langle f\rangle\langle f\rangle\langle/f\rangle\langle/f\rangle$;
6.1c: After *Call* to f and replacement by $\langle b\rangle\langle a\rangle\langle/a\rangle\langle a\rangle\langle/a\rangle\langle a\rangle\langle/a\rangle\langle/b\rangle$.

- Juliet *plays Call on the initial* $\langle/f\rangle$*, leading (after a replacement move by* Romeo*) either to* $w_1 = \langle a\rangle\langle/a\rangle\langle g\rangle\langle/g\rangle$ *or to* $w_2 = \langle b\rangle\langle/b\rangle\langle g\rangle\langle/g\rangle$*. As can easily be seen,* Juliet *wins on* w_1 *by playing Read on* $\langle/g\rangle$*.*
- *On* w_2*,* Juliet *plays Call on the final* $\langle/g\rangle$*, leading (after replacement by* Romeo*) to a nested word of the form* $w_n = \langle b\rangle\langle/b\rangle\langle f\rangle^n\langle/f\rangle^n$ *for some* $n \geqslant 1$ *chosen by* Romeo*, with the focus being on the first* $\langle f\rangle$*.*
- *On* w_n*,* Juliet *plays Read on each* $\langle/f\rangle$ *but the last, and Call on the final* $\langle/f\rangle$*. This causes* Romeo *to replace the subword* $\langle f\rangle^n\langle/f\rangle^n$ *either by* $\langle a\rangle\langle/a\rangle$ *or by* $\langle b\rangle\langle/b\rangle$*. This yields one of the nested words* $\langle b\rangle\langle/b\rangle\langle a\rangle\langle/a\rangle$ *or* $\langle b\rangle\langle/b\rangle\langle b\rangle\langle/b\rangle$*, both of which are in* T*, so* Juliet *wins in either case.*

Since, according to this strategy, Juliet *may have to place a Call on a* $\langle/f\rangle$ *tag returned by* Romeo *in response to a Call on* $\langle/g\rangle$*, but never plays Call inside the replacement subword returned by this secondary Call move, the winning strategy* σ *has Call depth 2.*

The canonical forest representations of nested words in a play according to σ *are depicted in Figure 6.2.*

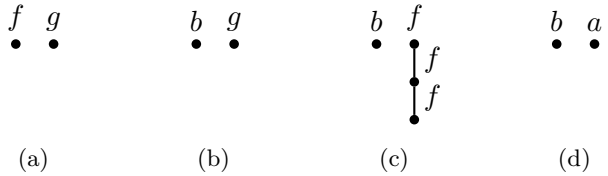


Figure 6.2.: Example play for a cfG as described in Example 6.3.
6.2a: Initial nested word; 6.2b: After *Call* to f and replacement by $\langle b\rangle\langle/b\rangle$; 6.2c: After *Call* to g and replacement by $\langle f\rangle\langle f\rangle\langle f\rangle\langle/f\rangle\langle/f\rangle\langle/f\rangle\langle/f\rangle$; 6.2d: After *Call* to f and replacement by $\langle a\rangle\langle/a\rangle$.

In this chapter, again, our focus is the winning problem for Juliet for various classes of cfGs, this time on nested words.

JWin($\mathcal{G}$)	
Given:	A context-free game $G \in \mathcal{G}$ and a nested word w.
Question:	Is $w \in \mathrm{JWin}(G)$?

The relevant parameters of a class $\mathcal{G}$ of cfGs in this chapter, as in Chapter 3, are mostly the representation of replacement and target languages as well as the amount of replay permitted; different from Chapter 3, however, this chapter puts a somewhat greater focus on the representation of the target language, as we will see in Section 6.3 that the winning problem for JULIET is intractable even in replay-free games with finite replacement languages and a target language represented by a deterministic NWA. For this reason, we investigate in Section 6.2.3 cfGs whose target language is given as a SNWA (as defined in Section 5.3); we call such games *simple* cfGs. Somewhat extended classes of cfGs on nested words that do not easily fit into the categorisation by parameters given above are investigated in Sections 6.4 and 6.5.

As in Chapter 3, we denote the automata representing the target and replacement languages by $A(T)$ and $A(R_a)$ (for $a \in \Gamma$), respectively, by $|R|$ the combined size of all $A(R_a)$ (for $a \in \Gamma$), and by $|G|$ the size of (a sensible representation of) G, i.e. $|G| = |\Sigma| + |R| + |A(T)|$. Throughout this chapter, we generally assume that $A(T)$ is a deterministic automaton in normal form, and therefore omit the set of hierarchical states from the definition of $A(T)$.

6.2. Upper bounds

The basic outline of the algorithms for the upper bounds summarised in Table 6.1 is the same as the generic Algorithm 1 introduced in the upper bound proofs for cfGs on flat strings (Section 3.4): First, the algorithm computes from a cfG G its call effect $\mathcal{C}[G]$, then it transforms that effect into an alternating automaton $A_{\mathcal{C}[G]}$ deciding $\mathrm{JWin}(G)$, and finally it evaluates the automaton $A_{\mathcal{C}[G]}$ on the input nested word w to determine whether $w \in \mathrm{JWin}(G)$.

Some parts of the groundwork laid in Section 3.4, such as definitions and some basic properties for effects, as well as parts of the correctness proofs for the constructions given there, directly carry over to the setting of cfGs on nested words. Adapting the proofs and methods from that section to nested words, however, requires a bit more effort than just switching out (alternating) nested word automata for (alternating) finite automata – specifically, the nesting structure of nested words necessitates defining the additional operation of *hierarchical composition* on word effects, and several correctness proofs need to be extended to account for nesting and deletion of subwords.

This section covers upper bound proofs for context-free games on nested words, focussing mostly on how to adapt the results and techniques already developed in Section 3.4. To that end, we first recall generic results on *Call* effects and word effects in Section 6.2.1, and then examine in detail how the generic Algorithm 1 can be adapted to cfGs on nested words – first for the case of general games (i.e. games with a target language represented by a DNWA) in Section 6.2.2, then for simple games (i.e. games with a target language represented by a SNWA) in Section 6.2.3. It should be noted that, unless stated otherwise, the precise representation of regular replacement languages is irrelevant to the complexity of the winning problem, i.e. the complexity of JWIN is the same regardless of whether replacement languages are given as NWA, DNWA or SNWA.

6.2.1. Call effects and word effects

We recall and adapt to cfGs on nested words the following definitions from Section 3.4.

Definition 6.4. Let $G = (\Sigma, \Gamma, R, T)$ be a nested cfG with target DNWA $A(T) = (Q, \Sigma, \delta, q_0, F)$ in normal form. For any $w \in \mathrm{WF}(\Sigma)$, we define the following notation.

(a) For any fixed $\sigma \in \mathrm{STRAT}_{\mathrm{J}}(G)$, $\mathrm{words}_G(w, \sigma) \stackrel{\mathrm{def}}{=} \{\mathrm{word}_G(w, \sigma, \tau) \mid \tau \in \mathrm{STRAT}_{\mathrm{R}}(G)\}$ denotes the set of final words that can be reached through strategies of Romeo against σ.

(b) For any fixed $\sigma \in \mathrm{STRAT}_{\mathrm{J}}(G)$ and $q \in Q$, $\mathrm{states}_G(q, w, \sigma) \stackrel{\mathrm{def}}{=} \{\delta^*(q, v) \mid v \in \mathrm{words}_G(w, \sigma)\}$ denotes the set of final states that can be reached from q through strategies of Romeo against σ.

(c) The *call effect* $\mathcal{C}[G] : \Gamma \times Q \to \mathscr{P}(\mathscr{P}(Q))$ is defined, for every $a \in \Gamma$, $q \in Q$, by $\mathcal{C}[G](a, q) \stackrel{\mathrm{def}}{=} [\{\mathrm{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma) \mid \sigma \in \mathrm{STRAT}_{\mathrm{J},\mathit{Call}}(G)\}]_{\min}$, where the set $\mathrm{STRAT}_{\mathrm{J},\mathit{Call}}(G)$ consists of all strategies of Juliet that start by playing *Read* on $\langle a\rangle$ and *Call* on $\langle/a\rangle$.

(d) The *word effect, $\mathcal{E}[G, w] : Q \to \mathscr{P}(\mathscr{P}(Q))$, of G on w* is defined, for every $q \in Q$, by $\mathcal{E}[G, w](q) \stackrel{\mathrm{def}}{=} [\{\mathrm{states}_G(q, w, \sigma) \mid \sigma \in \mathrm{STRAT}_{\mathrm{J}}(G)\}]_{\min}$.

Recall that the operator $[\cdot]_{\min}$ removes from a set of sets all non-minimal sets.

The idea behind effects, as in Section 3.4, is the abstraction of a (sub-)game into a two-round game in which first Juliet fixes a strategy σ (choosing a set $X \subseteq Q$ of states from the corresponding effect) and then Romeo decides on a counter-strategy (choosing a final state $q \in X$).

The only difference between the definitions given in Definition 6.4 and the ones from Section 3.4 (except for switching out flat strings and automata for nested ones) is the fact that, instead of a single function symbol a, the basis for $\mathcal{C}[G](a, \cdot)$ is a nested word of the form $\langle a\rangle\langle/a\rangle$. Despite the definition being based on this single nested word $\langle a\rangle\langle/a\rangle$, it still allows $\mathcal{C}[G](a, \cdot)$ to summarise the subgame on *any* nested word of the form $\langle a\rangle v\langle/a\rangle$ starting with a *Call* by Juliet to the final $\langle/a\rangle$, since the (possibly rewritten) subword v gets deleted as soon as that *Call* is played and has no further impact on the *Call* results available to Romeo to choose from.

Just as for flat cfGs, the basic idea behind the ANWA $A_{\mathcal{C}[G]}$ is that, on input $w \in \mathrm{WF}(\Sigma)$, it computes some $X \in \mathcal{E}[G, w](q_0)$ and checks whether $X \subseteq F$ to test whether Juliet has a winning strategy on w in G. To this end, we now examine how word effects can be computed from call effects in cfGs on nested words.

Sequential composition of word effects on nested words, in fact, directly carries over from word effects on flat strings. Since composition is defined on the abstract level of arbitrary functions from Q to $\mathscr{P}(\mathscr{P}(Q))$ in Definition 3.17, we do not even need to adapt it in any way for the setting of nested words. It is easy to see that the proof of Lemma 3.19, with nested words substituted for flat strings, directly yields the following analogous result.

Lemma 6.5. *For every nested cfG $G = (\Sigma, \Gamma, R, T)$ and $u, v \in WF(\Sigma)$ it holds that*

$$\mathcal{E}[G, uv] = \mathcal{E}[G, u] \circ \mathcal{E}[G, v].$$

Some more care has to be taken, though, to handle nesting of words and compute word effects from call effects. An analogue of Lemma 3.16 indeed yields that, for any $q \in Q$ and $a \in \Gamma$, it holds that $\mathcal{E}[G, \langle a\rangle\langle/a\rangle](q) = \mathcal{C}[G](a, q) \cup \{\{\delta^*(q, \langle a\rangle\langle/a\rangle)\}\}$. However, this equality is of no direct use for computing effects of the form $\mathcal{E}[G, \langle a\rangle v\langle/a\rangle]$ with $v \neq \epsilon$, as in case of a *Read* by Juliet on $\langle/a\rangle$, the subgame on v (represented by the effect $\mathcal{E}[G, v]$) has to be taken into account.

To describe hierarchical composition of word effects we define, for every $a \in \Sigma$ the following operator $H_a : Q \to \mathscr{P}(\mathscr{P}(Q))$. For every two functions $E : Q \to \mathscr{P}(\mathscr{P}(Q))$ and $C : \Sigma \times Q \to \mathscr{P}(\mathscr{P}(Q))$ and $q, q' \in Q$ such that $\delta(q, \langle a\rangle) = q'$, let

$$H_a[E, C](q) \stackrel{\text{def}}{=} \left[\bigcup_{X \in E(q')} \textsc{Mix}\left(\left\{ C(a, q) \cup \left\{ \delta(r, q, \langle/a\rangle) \right\} \mid r \in X \right\} \right) \right]_{\min}.$$

Informally, interpreting E as a word effect and C as a call effect, the first set inside the Mix operator accounts for *Call* moves and the second for *Read* moves of Juliet, with the intuition behind this definition being as follows. On a nested word $\langle a\rangle v\langle/a\rangle$, with E being the word effect of v, for each set X of states that Juliet can enforce by choosing some strategy in the subgame on v and for each state $r \in X$, Juliet has the choice of either playing *Read* on $\langle/a\rangle$ or playing *Call* on $\langle/a\rangle$ and choosing a substrategy to follow on Romeo's chosen replacement word. The former corresponds to Juliet choosing $\{\delta(r, q, \langle/a\rangle)\}$ as a follow-up set of states, the latter corresponds to choosing some set in $C(a, q)$. Since Juliet has to make such a choice for *each* state $r \in X$, and since Romeo can choose both the state $r \in X$ and the resulting state in Juliet's chosen follow-up set for r, we have to take the union of follow-up sets for all $r \in X$. Note that $C(a, q)$ has to be included *inside* the Mix operator since Juliet needs to have the choice between *Read* and *Call* for each single state $r \in X$ that may result from the sub-game on v.

Using this intuition, we can now formulate how effects behave hierarchically.

Lemma 6.6. *For every cfG $G = (\Sigma, \Gamma, R, T)$, $v \in WF(\Sigma)$, and $a \in \Sigma$, it holds*

$$\mathcal{E}[G, \langle a\rangle v\langle/a\rangle] = H_a[\mathcal{E}[G, v], \mathcal{C}[G]].$$

Proof. We once again use Lemma 2.1 to show that for every $q \in Q$, it holds that

$$\mathcal{E}[G, \langle a\rangle v\langle/a\rangle](q) = H_a[\mathcal{E}[G, v], \mathcal{C}[G]](q).$$

The proof itself is rather technical but straightforward, requiring only the definitions of the sets in question and simple logical steps but no major ideas.

($\supseteq$): Let $X \in \mathcal{E}[G, \langle a\rangle v\langle/a\rangle](q)$. Then there is some strategy $\sigma \in \mathrm{STRAT}_J(G)$ such that $X = \mathrm{states}_G(q, \langle a\rangle v\langle/a\rangle, \sigma)$. Let σ_v be the sub-strategy of σ on v, let $\delta(q, \langle a\rangle) = (q_a, p)$

and let $X_v = \{q_1, \ldots, q_k\} = \text{states}_G(q_a, v, \sigma_v)$. For each $i \in [k]$, let $v_i \in \text{words}_G(v, \sigma_v)$ such that $\delta^*(q_a, v_i) = q_i$ and let σ_i be the sub-strategy of σ starting at $(\langle a\rangle v_i, \langle/a\rangle)$. If JULIET'S move on $\langle/a\rangle$ according to σ_i is *Read*, let $X_i = \{\delta(q_i, p, \langle/a\rangle)\}$, otherwise let $X_i = \text{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma_i)$.[2] Clearly, $X' = X_1 \cup \ldots \cup X_k$ has a subset in $\text{MIX}(\{\mathcal{C}[G](a,q) \cup \{\delta(r, p, \langle/a\rangle)\} \mid r \in X\})$ and therefore in $H_a[\mathcal{E}[v], \mathcal{C}[G]](q)$. It remains to be proven that $X' \subseteq X$.

Let $q' \in X'$. Then $q' \in X_i$ for some $i \in [k]$. If $X_i = \{\delta(q_i, p, \langle/a\rangle\}$, then clearly $q' = \delta^*(q, \langle a\rangle v_i\langle/a\rangle) \in \text{states}_G(q, \langle a\rangle v\langle/a\rangle, \sigma) = X$. Otherwise, $q' \in \text{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma_i)$ and σ_i coincides on $\langle a\rangle\langle/a\rangle$ with σ on $\langle a\rangle v_i\langle/a\rangle$, it follows again that $q' \in X$.

($\sqsubseteq$): Let $X \in H_a[\mathcal{E}[G, v], \mathcal{C}[G]](q)$ and let $\delta(q, \langle a\rangle) = (q', p)$. Then, there exists some $X_v = \{q_1, \ldots q_k\} \in \mathcal{E}[G, v](q')$ such that $X = X_1 \cup \ldots \cup X_k$, where each X_i is either in $\mathcal{C}[G](a, q)$ or of the form $\{\delta(q_i, p, \langle/a\rangle)\}$. By the definition of $\mathcal{E}[G, v]$, there exists some strategy $\sigma_v \in \text{STRAT}_\text{J}$ on v such that $\text{states}_G(q', v, \sigma_v) = X_v$, and by the definition of $\mathcal{C}[G](a, q)$, for each i with $X_i \in \mathcal{C}[G](a, q)$ there exists a strategy $\sigma_i \in \text{STRAT}_{\text{J},\mathit{Call}}$ such that $X_i = \text{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma_i)$. We extend σ_v to a strategy σ on $\langle a\rangle v\langle/a\rangle$ as follows: JULIET reads the initial $\langle a\rangle$, then plays on v according to σ_v. The nested word v' resulting from this play on v has to fulfil $\delta^*(q', v') = q_i$ for some $i \in [k]$; if, for this i it holds that $X_i = \{\delta(q_i, p, \langle/a\rangle)\}$, then JULIET plays *Read* on $\langle/a\rangle$, otherwise she plays *Call* on $\langle/a\rangle$ and plays according to σ_i in the resulting sub-game. Let $X' = \text{states}_G(q, \langle a\rangle v\langle/a\rangle, \sigma)$; it is easy to see that $X' \subseteq X$, and since X' has a subset in $\mathcal{E}[G, \langle a\rangle v\langle/a\rangle](q)$ by definition, this proves the claim. □

The auxiliary result of Lemma 3.24 also carries over to nested cfGs with an identical proof:

Lemma 6.7. *For a cfG $G = (\Sigma, \Gamma, R, T)$ with a target DNWA in normal form $A(T) = (Q, \Sigma, \delta, q_0, F)$, it holds, for every $a \in \Gamma$ and $q \in Q$:*

$$\mathcal{C}[G](a, q) = \text{MIX}(\{\mathcal{E}[G, w](q) \mid w \in R_a\}).$$

With these results on word effects, we are now ready to describe the algorithms for computing $\mathcal{C}[G]$ and $A_{\mathcal{C}[G]}$ (Steps 2 and 3 of the generic Algorithm 1 on page 39) on a cfG G on nested words.

6.2.2. General games

As in Section 3.4, we begin by describing how to compute $A_{\mathcal{C}[G]}$ from $\mathcal{C}[G]$ (Step 3 of the generic algorithm), as this construction is the same both for arbitrary regular replacement languages and for finite replacement languages. In later subsections, we will then turn to the algorithms computing $\mathcal{C}[G]$ (Step 2 of the generic algorithm) for each of those two cases separately.

Like for cfGs on flat strings, $A_{\mathcal{C}[G]}$ uses alternation to guess (abstracted) strategy choices for JULIET and ROMEO and tracks a state in $A(T)$. The intuition is basically the same as explained after the proof of Lemma 3.19; the only difference for nested cfGs

[2] As, in this case, $\sigma_i \in \text{STRAT}_{\text{J},\mathit{Call}}$ is a strategy playing *Call* on $\langle/a\rangle$, we can omit v here.

is that opening and closing tags are handled differently, with $A_{\mathcal{C}[G]}$ always simulating $A(T)$ on opening tags and making strategy choices on closing tags.

Formally, $A_{\mathcal{C}[G]} = (Q, \Sigma, \delta_C, q_0, F)$ is an ANWA where δ_C is defined as follows. (Recall that the target language DNWA $A(T) = (Q, \Sigma, \delta, q_0, F)$ is in normal form.)

- For $a \in \Sigma, q \in Q$:
$$\delta_C(q, \langle a \rangle) \stackrel{\text{def}}{=} (\delta(q, \langle a \rangle), q).$$
- For $a \in \Sigma, q, p \in Q$:
$$\delta_C(q, p, \langle / a \rangle) \stackrel{\text{def}}{=} \delta(q, p, \langle / a \rangle) \vee \bigvee_{X \in \mathcal{C}[G](a,p)} \left(\bigwedge_{r \in X} r \right).$$

Again, $A_{\mathcal{C}[G]}$ has the same state space as $A(T)$, but possibly an exponential-sized transition relation.

The proof of correctness for $A_{\mathcal{C}[G]}$ is only slightly more involved than that of Lemma 3.20. Recall that a run ρ of an ANWA A on a nested word w is called *minimal* if no proper subtree of ρ is a run of A on w.

Lemma 6.8. *Let $q \in Q$, $w \in WF(\Sigma)$ and $X \subseteq Q$. Then, $X \in \mathcal{E}[G, w](q)$ if and only if there is a minimal run of $A_{\mathcal{C}[G]}$ on w starting at q and ending in linear states from X.*

Proof. Let $q \in Q$, $X \subseteq Q$ and $w \in \mathrm{WF}(\Sigma)$. The proof is by induction on the structure of w and mostly analogous to that of Lemma 3.20. Specifically, the cases $w = \epsilon$ and $w = uv$ (with $u, v \in \mathrm{WF}(\Sigma)$) are easily adapted from that proof.

Let therefore $w = \langle a \rangle v \langle / a \rangle$ for $a \in \Sigma$, $w \in \mathrm{WF}(\Sigma)$. Let further $\delta(q, \langle a \rangle) = q'$. For "only if", Lemma 6.6 implies that $X \in H_a[\mathcal{E}[G, v], \mathcal{C}[G]](q)$. This means that there is a set $X_v = \{q_1, \ldots, q_k\} \in \mathcal{E}[G, v](q')$ and sets $X_w^1, \ldots, X_w^k$ such that $X = X_w^1 \cup \ldots \cup X_w^k$ and for each $i \in [k]$ either $X_w^i \in \mathcal{C}[G](a, p)$ or $X_w^i = \{\delta(q_i, q, \langle / a \rangle)\}$. By induction, there exists a minimal run ρ_v of $A_{\mathcal{C}[G]}$ on v starting at q' and ending inside X_v. We extend ρ_v to a run ρ on w as follows: The root of ρ is labelled (q, ϵ) and has as its only child the root of a modified copy of ρ_v, where each node label of the form $(r, \alpha) \in (Q \times Q^*)$ is replaced by $(r, \alpha q)$; each leaf of this copy labelled q_i has as its children exactly the configurations in $X_w^i \times \{\epsilon\}$. Using the definition of $A_{\mathcal{C}[G]}$, it is easy to verify that ρ is indeed a run of $A_{\mathcal{C}[G]}$ on w, and it is also clear that ρ starts at q and ends with linear states inside X. Finally, ρ is minimal because its subrun on v is minimal, and for each q_i, the set X_w^i is an inclusion-minimal set fulfilling $\delta_C(q_i, q, \langle / a \rangle)$ (for $X_w^i \in \mathcal{C}[G](a, q)$, this follows from $\mathcal{C}[G](a, q)$ being normalised). Again, the "if" part is proven analogously. □

It is obvious that $A_{\mathcal{C}[G]}$ can be computed from G and $\mathcal{C}[G]$ in polynomial time, so with Lemma 6.8 we directly get our main result for Step 3 of the generic algorithm.

Proposition 6.9. *There is an algorithm that computes from the call effect $\mathcal{C}[G]$ of a game G in polynomial time in $|\mathcal{C}[G]|$ and $|G|$ an ANWA $A_{\mathcal{C}[G]}$ such that $L(A_{\mathcal{C}[G]}) = JWin(G)$.*

Next, we examine how the construction of call effects (Step 2 of the generic algorithm) can be performed for regular and for finite replacement languages.

Regular replacement languages

The main result for general cfGs on nested words is as follows.

Theorem 6.10. *For the class of cfGs on nested words with regular replacement languages, the following complexity results hold for* JWIN*:*

(a) with unbounded or bounded replay, it is 2-EXPTIME*-complete, and*

(b) without replay, it is PSPACE*-complete.*

The lower bounds of Theorem 6.10 are proven as Proposition 6.18 in Section 6.3.

As for flat cfGs, computing $\mathcal{C}[G]$ from a given nested cfG G follows a fixpoint-based approach, computing inductively, for $k = 1, 2, \ldots$ the call effect of the restricted game of maximum *Call* depth k. We once again define for every cfG G, $a \in \Sigma$, $q \in Q$, and $k \geqslant 1$,

$$\mathcal{C}^k[G](a,q) \stackrel{\text{def}}{=} \Big[\{\text{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma) \mid \sigma \in \text{STRAT}^k_{\text{J},\mathit{Call}}(G)\}\Big]_{\min}.$$

The main interesting difference to flat cfGs, here, is that a single iteration of computing $\mathcal{C}^{k+1}[G]$ from $\mathcal{C}^k[G]$ actually takes doubly exponential time, which means that the exponential number of iterations for the fixpoint process is dominated by the effort of a single iteration. In particular, this means that doubly exponential time is already required for games with *Call* depth 2, and by the corresponding hardness results, it is unlikely that this requirement can be lowered.

For *Call* depth 1, however, a simple characterisation similar to Lemma 3.21 (with an analogous proof) holds.

Lemma 6.11. *For every $q \in Q$ and $a \in \Sigma$, it holds that*

$$\mathcal{C}^1[G](a,q) = \{\{\delta^*(q,v) \mid v \in R_a\}\}.$$

In particular, $\mathcal{C}^1[G]$ can be computed from G in polynomial time.

The computation of $\mathcal{C}^{k+1}[G]$ from $\mathcal{C}^k[G]$ is done in an analogous manner to the proof of Lemma 3.22, using (A)NWA instead of (alternating) finite-state automata.

Lemma 6.12. *Given a state $q \in Q$, an alphabet symbol $a \in \Gamma$, and $\mathcal{C}^k[G]$, for some $k \geqslant 1$, the call effect $\mathcal{C}^{k+1}[G](a,q)$ can be computed in doubly exponential time in $|G|$.*

Proof. As in the proof of Lemma 3.22, we show that, given $\mathcal{C}^k[G]$ and a set $X \subseteq Q$, it can be decided in doubly exponential time in $|Q|$ and polynomial time in $|R|$ whether a subset of X is in $\in \mathcal{C}^{k+1}[G](a,q)$. Let $A_{\mathcal{C}^k[G]}$ be as defined for the proof of Lemma 3.20 with $\mathcal{C}^k[G]$ as its basic *Call* effect, and let A be its modification with initial state q and set X of accepting states. By using a standard product construction with an NWA A_a for replacement language R_a and a complementation of an ANWA, we can test whether $R_a \subseteq L(A)$ holds, i.e. whether for every word in R_a, JULIET has a strategy of *Call* depth k rewriting it into a nested word that takes $A(T)$ from state q into some state guaranteed to be in X. This test is possible in doubly exponential time due to Theorem 5.14. □

Again, since for every q and a, $\mathcal{C}^k[G](a,q)$ is contained in the closure of $\mathcal{C}^{k+1}[G](q,a)$ under supersets and contains at most $2^{|Q|}$ sets, the process computing $\mathcal{C}^k[G]$ for increasing values of k reaches a fixpoint after at most exponentially many iterations. For $a \in \Gamma$ and $q \in Q$, we denote this fixpoint by

$$\mathcal{C}^*[G](a,q) \stackrel{\text{def}}{=} \left[\bigcup_{k=1}^{\infty} \mathcal{C}^k[G](a,q)\right]_{\min}.$$

This fixpoint indeed captures game effects for arbitrary *Call* depth, as the following proposition shows.

Proposition 6.13. *For every cfG G it holds: $\mathcal{C}^*[G] = \mathcal{C}[G]$.*

Proof. The proof follows along the same lines as that of Proposition 3.23. We construct from an initial cfG $G = (\Sigma, \Gamma, R, T)$ a game $G' = (\Sigma, \Gamma, R', T)$, where R' consists of finite sublanguages $R'_a \subseteq R_a$, for every $a \in \Gamma$. Then we show

(a) $\mathcal{C}^*[G] = \mathcal{C}^*[G']$,

(b) $\mathcal{C}^*[G'] = \mathcal{C}[G']$, and finally

(c) $\mathcal{C}[G'] = \mathcal{C}[G]$.

As in the proof of Proposition 3.23, for each $a \in \Gamma$, $k \geqslant 1$ and $w \in \Sigma^*$, let $v(a,w,k)$ be a nested word of minimum length such that $\mathcal{E}^k[G,w] = \mathcal{E}^k[G, v(a,w,k)]$ and $v(a,w,k) \in R_a$, and let $v(a,w)$ be a nested word of minimum length such that $\mathcal{E}[G,w] = \mathcal{E}[G, v(a,w)]$ and $v(a,w) \in R_a$. Let $V_a = \{v(a,w) \mid w \in \Sigma^*\}$ and let $W_a = \{v(a,w,k) \mid w \in \Sigma^*, k \geqslant 1\}$. We then define, for each $a \in \Gamma$, the replacement language $R'_a \stackrel{\text{def}}{=} W_a \cup V_a$ in G'. Claims (a) and (b) then follow as in the proof of Proposition 3.23.

The proof of claim (c) requires a little more work. We again prove the stronger claim that for all $q \in Q$ and $w \in \Sigma^*$, it holds $\mathcal{E}[G,w](q) = \mathcal{E}[G',w](q)$ by proving that each set in $\mathcal{E}[G',w](q)$ has a subset in $\mathcal{E}[G,w](q)$ and vice versa, which proves the desired equality by Lemmas 2.1 and 6.7.

The direction $\text{states}_{G'}(q,w,\sigma') \subseteq \text{states}_G(q,w,\sigma)$ is again quite obvious. For the other direction, let $q \in Q$, $w \in \Sigma^*$ and let $\sigma' \in \text{STRAT}_{\text{J},Call}(G')$ with $X = \text{states}_{G'}(q,w,\sigma') \in \mathcal{E}[G',w](q)$, and let $d \stackrel{\text{def}}{=} \text{Depth}^{G'}(\sigma', w)$. We prove by nested induction over d and the structure of w that there exists a strategy σ in G with $\text{states}_G(q,w,\sigma) \subseteq X$, which implies that X has a subset in $\mathcal{E}[G,w](q)$.

If $d > 0$, Juliet must play *Call* on w at some point, and therefore it holds that $w \neq \epsilon$.

The case $w = uv$ with $u, v \in \text{WF}(\Sigma)$ again carries over from the proof of Proposition 3.23. Therefore let $w = \langle a \rangle v \langle / a \rangle$.

If $a \notin \Gamma$ with $\delta(q, \langle a \rangle) = q'$, then there is a strategy σ'_v on v such that $X = \{\delta(q'', q, \langle / a \rangle) \mid q'' \in \text{states}_{G'}(q', v, \sigma'_v)\}$. By induction, there exists a strategy σ_v on v in G such that $\text{states}_G(q', v, \sigma_v) \subseteq \text{states}_{G'}(q', v, \sigma'_v)$, i.e. the strategy σ in G that plays on v as σ_v would fulfils the claim.

If $w = \langle a \rangle v \langle / a \rangle$ for some $a \in \Gamma$, $v \in \mathrm{WF}(\Sigma)$, let $\delta(q, \langle a \rangle) = q'$, let σ'_v be the sub-strategy of σ' on v, and let $\{q_1, \ldots, q_k\} = \mathrm{states}_{G'}(q', v, \sigma'_v)$. By induction (as the depth of v is smaller than the depth of w), there exists a strategy σ_v on v in G with $\mathrm{states}_G(q', v, \sigma_v) \subseteq \mathrm{states}_{G'}(q', v, \sigma'_v)$. In the strategy σ on w, JULIET plays according to σ_v on v. The play on v from q' according to σ is bound to reach some state q_i for $i \in [k]$. If there is some nested word $v_i \in \mathrm{words}_{G'}(v, \sigma'_v)$ with $\delta^*(q', v_i) = q_i$ such that JULIET would play *Read* on $\langle / a \rangle$ according to σ' in G' if the play on v yields v_i, then JULIET also plays *Read* on $\langle / a \rangle$ according to σ; obviously, in this case, the resulting state from the play according to σ is in X. Otherwise, JULIET plays *Call* on $\langle / a \rangle$ in σ. Let $z \in R_a$ be some arbitrary response for ROMEO to this *Call* move in G; we now explain how JULIET plays on z according to σ.

By construction, the replacement language R'_a in G' contains the nested word $v(a, z)$, so this word is a valid response for ROMEO to the *Call* by JULIET on $\langle / a \rangle$ in G'. Let $\sigma'_{v(a,z)}$ be the sub-strategy of σ' if ROMEO chooses this response. As $\sigma'_{v(a,z)}$ has a *Call* depth of at most $d - 1$, by induction there exists a strategy $\sigma_{v(a,z)}$ for JULIET on $v(a, z)$ in G with $\mathrm{states}_G(q, v(a, z), \sigma_{v(a,z)}) \subseteq \mathrm{states}_{G'}(q, v(a, z), \sigma'_{v(a,z)})$. By the definition of $v(a, z)$, it holds that $\mathcal{E}[G, z] = \mathcal{E}[G, v(a, z)]$, which implies that there is a strategy σ_z for JULIET on z in G such that $\mathrm{states}_G(q, z, \sigma_z) \subseteq \mathrm{states}_G(q, v(a, z), \sigma_{v(a,z)})$. In σ, JULIET then plays on z according to σ_z, and the above set inclusions show that all states resulting from this play are in X as well, which completes the case $w = \langle a \rangle v \langle / a \rangle$ for $a \in \Gamma$ and concludes the proof. □

Finally, we adapt the proof of Theorem 3.9 to yield the upper bounds for Theorem 6.10.

Proof of Theorem 6.10. We justify here only the upper bounds, as lower bounds are stated and proven as Proposition 6.18 in Section 6.3. Let G be a cfG and w a word. By Lemma 6.11, $\mathcal{C}^1[G]$ can be computed in polynomial time from G. For the replay-free case, we can immediately construct an ANWA for $\mathrm{JWin}(G)$ and evaluate it on w, yielding the claimed PSPACE upper bound by Theorem 5.14.

For (a), $\mathcal{C}[G]$ and $\mathcal{C}^k[G]$, respectively, can be computed in doubly exponential time, $A_{\mathcal{C}[G]}$ can be computed in exponential time (in the size of G), and whether $w \in L(A_{\mathcal{C}[G]})$ can then be tested in polynomial space in $|A_{\mathcal{C}[G]}|$ and $|w|$, that is, in at most exponential space in $|G|$ and $|w|$.

Corresponding hardness results are stated in Proposition 6.18, thereby completing the proof of Theorem 6.10. □

Finite replacement languages

We now explain how to adapt the upper bound proofs for general nested cfGs with arbitrary regular replacement languages to yield lower complexities in the case of finite replacement languages. The complexity results for this setting are as follows.

Proposition 6.14. *For the class of cfGs on nested words with finite replacement languages (explicitly enumerated or represented by NWAs), the following complexity results hold for* JWIN*:*

(a) with unbounded replay, it is EXPTIME*-complete, and*

(b) with bounded or without replay, it is PSPACE*-complete.*

The lower complexity in the cases with replay is owed to the fact that, with finite replacement languages, it is possible to replace the non-emptiness test of an ANWA in the computation of $\mathcal{C}^{k+1}[G]$ from $\mathcal{C}^k[G]$ by a finite number of membership tests. The main determining factor for the complexity is therefore no longer the complexity of the non-emptiness test, but rather the depth to which call effects have to be computed – exponential with unbounded replay, and polynomial with at most bounded replay.

Proof. The lower bounds follow directly from the proofs of Theorem 3.4(a) and Proposition 6.18(b), since these proofs already use only finite replacement languages. We therefore only prove the upper bounds here.

We first show that the computation of $\mathcal{C}^{k+1}[G]$ from $\mathcal{C}^k[G]$ (Lemma 6.12) is feasible in polynomial space. To this end, we replace the non-emptiness test for $R_a \cap \overline{L(A)}$ in the proof of Lemma 6.12 with membership tests checking whether $v \notin L(A)$ for all $v \in R_a$. If R_a is represented by a NWA A_a, then it follows from standard pumping arguments that words in R_a may have at most linear width and quadratic depth in $|A_a|$, as any word in R_a violating these upper bounds would induce a run of A_a in which the same configuration occurs both before and after reading some subword, so that subword could be repeated arbitrarily often, contradicting finiteness of R_a. Therefore, all nested words in R_a can be iterated over using at most polynomial space, and for each of these nested words, membership in $L(A)$ can be tested in polynomial space by Theorem 5.8(b).

The exponential time upper bound in (a) immediately follows because the number of iterations of the fixpoint process for computing $\mathcal{C}[G]$ is at most exponential, and the final test whether w is accepted by $A_{\mathcal{C}[G]}$ needs only polynomial space. This again uses the fact that $A_{\mathcal{C}[G]}$ has a state set of polynomial size, even if its transition relation may be exponential.

For (b), the initial call effect $\mathcal{C}^1[G]$ can be computed in polynomial time. For each i, $\mathcal{C}^{i+1}[G]$ can be computed from $\mathcal{C}^i[G]$ in polynomial space and finally, whether $A_{\mathcal{C}^k[G]}$ accepts w can be tested in polynomial space in $|w|$ and the number of states of $A_{\mathcal{C}^k[G]}$, that is, in the number of states of the target automaton of G. Note that, as in the proof of Theorem 3.9, $A_{\mathcal{C}^k[G]}$ and all intermediate automata cannot be stored explicitly but have to be recomputed on-the-fly in polynomial space whenever needed; again, it is crucial here, that (as observed in the proof of Theorem 5.14) the evaluation of $A_{\mathcal{C}^k[G]}$ is possible in polynomial space in $|w|$ and the number of states of $A_{\mathcal{C}^k[G]}$. □

6.2.3. Simple games

As seen in Section 6.2.2, the complexity of solving context-free games on nested words is generally rather high. The algorithms provided in that section (as well as the lower bound proofs in Section 6.3) show that this problem mainly stems from the complexity of decision problems for alternating NWA – intuitively, regular nested word languages

as a formalism for target languages are just somewhat too expressive compared with regular languages of flat strings.

Therefore, we now examine context-free games with target languages given by *simple* NWAs. By Proposition 5.10, these are expressive enough to encompass XML Schema and, by extension, also DTDs (cf. [MNSB06]). We will see that the complexity results for such *simple* games actually match those for cfGs on flat strings, implying that it makes no difference (from a complexity standpoint) whether target languages are given as DTDs, XML Schemas, or SNWAs.

The main work in this section consists of adapting the methods and results from Section 6.2.2 to simple games. As in that section, we first describe how the alternating automaton $A_{\mathcal{C}[G]}$ (here: a SANWA) is computed from G and $\mathcal{C}[G]$, i.e. how to instantiate Step 3 of Algorithm 1 on simple cfGs, before going into the construction of $\mathcal{C}[G]$ from G (Step 2 of the generic algorithm) for regular and finite replacement languages.[3]

Proposition 6.15. *Given a nested cfG G with target SNWA, there is an algorithm that computes from the call effect $\mathcal{C}[G]$ in polynomial time in $|\mathcal{C}[G]|$ and $|G|$ a SANWA $A_{\mathcal{C}[G]}$ such that $L(A_{\mathcal{C}[G]}) = JWin(G)$.*

Proof. We construct $A_{\mathcal{C}[G]}$ almost as in the proof of Proposition 6.9. However, as they are mimicking games, the alternating transitions in the automaton constructed there occur at closing tags, whereas the definition of simple ANWAs requires that alternating transitions occur only at opening tags. Thus we slightly adapt the construction as follows.

Let $A(T) = (Q, \Sigma, \delta, q_0, F_0)$ be a SNWA in normal form and let S, Δ, F_{loc}, t, $\bot$ witness the simplicity of $A(T)$. With $q_? \notin Q$, we define the SANWA $A_{\mathcal{C}[G]} \stackrel{\text{def}}{=} ((Q \cup \{q_?\}), \Sigma, \delta_C, q_0, F_0)$, with δ_C defined as follows

- For every $q \in Q$ and $a \in \Sigma$, where $q' = \delta(q, \langle a \rangle)$,

$$\delta_C(q, \langle a \rangle) \stackrel{\text{def}}{=} \big((q', t(q,a)) \wedge (q', q_?)\big) \vee \bigvee_{X \in \mathcal{C}[G](a,q)} \bigwedge_{r \in X} (q', r).$$

- For every $q, q' \in Q$ and $a \in \Sigma$, $\delta_C(q, q', \langle / a \rangle)$ is defined via a target state function t_C as per the definition of SANWA.

The target state function t_C and final state function $F_{\mathrm{loc,C}}$ witnessing the simplicity of $A_{\mathcal{C}[G]}$ are defined by $t_C(q,a) \stackrel{\text{def}}{=} q$ and $F_{\mathrm{loc,C}}(a) \stackrel{\text{def}}{=} F_{\mathrm{loc}}(a)$, respectively. This automaton obviously fulfils both simplicity conditions, by construction and the simplicity of $A(T)$. The correctness of the automaton is proven analogously to the proof of Lemma 6.8. □

We now turn towards adapting the construction of call effects from simple games.

[3] Note that, while we restrict the *target* language automaton to be a SNWA, the *replacement* languages may still be represented by arbitrary NWA.

Regular replacement languages

The complexity results for simple cfGs on nested words are as follows.

Theorem 6.16. *For classes of games on nested words with target languages given by SNWAs, XML Schemas or DTDs, respectively, the following complexity results hold for* JWIN*:*

(a) with unbounded replay, it is EXPTIME*-complete,*

(b) with bounded replay, it is PSPACE*-complete, and*

(c) without replay, it is PTIME*-complete (under logspace-reductions).*

To prove this theorem, it suffices to prove the lower bounds for DTDs and the upper bounds for SNWAs. In fact, the lower bounds follow almost directly from the lower bounds on flat cfGs, and the upper bounds only require careful analysis of the generic algorithm instantiated for target SNWAs.

Proof. All lower bounds follow by the same reduction from corresponding lower bounds for flat cfGs, which were proven as Propositions 3.7, 3.8 and 3.6.

The idea for the reduction from flat cfGs to simple (nested) cfGs is as follows: All input and replacement strings $w = w_1 \dots w_n \in \Sigma^*$ are replaced by their standard nested word encoding $\widehat{w} = \langle w_1 \rangle \langle / w_1 \rangle \dots \langle w_n \rangle \langle / w_n \rangle \in \mathrm{WF}(\Sigma)$; to this end the equivalent DFA $A(T)$ computed from a target DRE T is simulated by a SNWA in normal form with an extra state q_n such that $\delta(q, \langle a \rangle) = q_n$ for each q and a, $F_{\mathrm{loc}}(a) = q_n$ for each a, and $t(q, a)$ is the transition function of $A(T)$. Replacement DREs are similarly transformed into (S)NWAs.

For the upper bounds, it suffices to prove suitable upper bounds for the ingredients needed by Algorithm 1.

More precisely, Proposition 5.16 (b) and Lemma 6.11 yield a polynomial time bound for replay-free games. Proposition 5.16 (a) guarantees that the inductive step in the computation of $\mathcal{C}[G]$ can be carried out in polynomial space (as opposed to doubly exponential time). The upper bounds for games with unrestricted replay follows immediately and the upper bound for bounded replay can be shown similarly as in Proposition 6.14 (b). □

Finite replacement languages

Seeing as the complexities for solving simple cfGs on nested words with regular replacement languages (Theorem 6.16) mirror those on flat cfGs with regular replacement (Theorem 3.5), it should not be too surprising that a similar correspondence holds for finite replacement languages as well. In this subsection, we prove that this is indeed the case.

Proposition 6.17. *For the class of games with target languages specified by SNWAs, XML Schemas or DTDs, and explicitly enumerated finite replacement languages, the following complexity results hold for* JWIN*:*

(a) with unbounded replay, it is EXPTIME*-complete, and*

(b) with bounded replay or without replay, it is PTIME*-complete (under logspace-reductions).*

The interesting thing about the proof for this theorem is that it actually deviates from the general idea of upper bounds in this chapter so far (using restrictions on the class of games to reduce the complexity of key components in the generic algorithm) and instead follows the approach of directly simulating the game by using alternation.

Proof. The upper bound in (a) follows from Theorem 6.16. So do the lower bounds in (a) and (b), as the corresponding reductions are from flat cfGs with finite replacement languages, and the reductions do not increase the size of replacement languages. It thus only remains to show the upper bound in (b).

We prove this upper bound by giving an alternating logarithmic-space algorithm for JWin. This algorithm uses alternation to simulate plays of the game on the input word and additionally uses universal branching to divide, at each opening tag $\langle a \rangle$, the processing of the remaining word into the processing of the subword until the corresponding closing tag $\langle /a \rangle$ and the processing of the remaining word after that $\langle /a \rangle$. It is crucial that the computation only requires logarithmic space.

We first describe how the game on an input word w can be simulated by an alternating logspace-computation. This part of the proof is very similar to the proof of the upper bound of Theorem 5.8 in [MSS06]. Let k be the bound on the *Call* depth. We consider the equivalent version of cfGs in which Juliet decides already when she reads an opening tag $\langle a \rangle$, whether she wants Romeo to rewrite a subword $u = \langle a \rangle \cdots \langle /a \rangle$.

The idea is that the choices of Juliet and Romeo are simulated by existential and universal branching of the algorithm in the obvious fashion. However, if Juliet calls an opening tag $\langle a \rangle$ at some position i and Romeo replaces the corresponding subword $u = \langle a \rangle \cdots \langle /a \rangle$ of w by a word v from R_a then the algorithm does not actually replace u but rather stores the information that u has been replaced by a pointer to position i and another pointer to v (which is stored in the representation of G). As the *Call* depth is bounded by k, at any given time no more than k such pairs of pointers are active, consuming at most $\mathcal{O}(\log(|G|))$ bits of space. The test whether the resulting word (of each branch) is accepted by the target automaton $A(T)$ is integrated into this branching process as follows. Each process maintains a current linear state p reflecting the state of $A(T)$ in the unique computation on the prefix of the current word, that is, if the current game configuration is (J, w_1, w_2), the current state is the one obtained in $A(T)$ after reading w_1.

Whenever Juliet reads an opening tag $\langle a \rangle$, the computation universally branches into two subcomputations. The first subcomputation checks whether Juliet has a winning strategy in the game on the subword between $\langle a \rangle$ and its corresponding closing tag. The other subcomputation continues after that closing tag in the state determined by the target state function. Juliet only wins if both subcomputations accept. Each subcomputation may recursively branch in the same way. When a subcomputation reaches a closing tag $\langle /a \rangle$, it accepts if the current linear state is in $F_{\mathrm{loc}}(a)$ and rejects otherwise.

It is not hard to see that this algorithm has an accepting run on a word w if and only if JULIET has a winning strategy on w. As the algorithm only uses logarithmic space it witnesses the desired PTIME upper bound. □

6.3. Lower bounds

In this section, we prove lower bounds on the complexity of solving cfGs on nested words with arbitrary regular replacement languages, thereby completing the proof of Theorem 6.10. In proving the corresponding upper bounds, the non-emptiness and membership problem for ANWA played an integral part; we will see in the lower bound proofs that algorithmic problems for ANWA and the problem of determining a winner in nested cfGs are actually quite closely connected, as we prove the lower bounds by reduction from non-emptiness and membership for ANWA, respectively. That is, much like in Lemma 6.8, where we constructed ANWA from given cfGs to obtain upper complexity bounds, we prove matching lower bounds by transforming ANWA into cfGs.

Proposition 6.18. *For the class of context-free games on nested words with regular replacement languages, the following complexity results hold for* JWIN*:*

(a) with bounded replay, it is 2-EXPTIME*-hard, and*

(b) without replay, it is PSPACE*-hard.*

We will see that both claims of Proposition 6.18 follow from the corresponding parts of Theorem 5.14. To this end, the following lemma is an essential component.

Lemma 6.19. *There is a polynomial time algorithm that computes, given an ANWA A and a nested word w, a cfG $G = (\Sigma, \Gamma, R, T)$ and a nested word w' such that $w \in L(A)$ if and only if* ROMEO *has a winning strategy on w' in G, against all replay-free strategies of* JULIET*. Furthermore, G only depends on A (not on w) and can be computed in polynomial time in the size of A.*

The reader might feel that it would be more natural to associate existential moves to JULIET, and therefore expect a winning strategy for JULIET iff $w \in L(A)$ holds. However, Lemma 6.19 follows the idea of reductions from existence problems (already sketched in the discussion of lower bounds in Section 1.3 and used in several previous lower bounds), where it is ROMEO who chooses a potential witness for existence and JULIET who aims to show that this witness is incorrect. With this intuition, the association of ROMEO with existential choices actually makes more sense, as will become clear in the proof of Proposition 6.18 given below.

Proof. Let $A = (Q, \Sigma, q_0, \delta, \{q_F\})$ be an ANWA and $w \in \mathrm{WF}(\Sigma)$ a nested word. The idea is to simulate the alternation of A in the game G on w'. We design G to only admit replay-free strategies for JULIET. To make this possible, we construct w' from w by adding subwords that offer enough "space" for this simulation.

We assume without loss of generality that Σ does not contain any symbols from $(Q \times Q) \cup Q \cup \{b, 1, 2, \vee, \wedge, \texttt{false}, \texttt{true}\}$. For any formula $\varphi \in \mathcal{B}^+(Q \times Q) \cup \mathcal{B}^+(Q)$, the *encoding* $\mathrm{Enc}(\varphi)$ is the well-nested word over the alphabet $(Q \times Q) \cup Q \cup \{\vee, \wedge\}$ derived from φ as follows:

- If $\varphi \in \{\texttt{false}, \texttt{true}\}$, then $\mathrm{Enc}(\varphi) = \langle\varphi\rangle\langle/\varphi\rangle$;
- If $\varphi = (q, p) \in Q \times Q$, then $\mathrm{Enc}(\varphi) = \langle(q, p)\rangle\langle/(q, p)\rangle$;
- If $\varphi = q \in Q$, then $\mathrm{Enc}(\varphi) = \langle q\rangle\langle/q\rangle$;
- If $\varphi = \varphi_1 \vee \varphi_2$, then $\mathrm{Enc}(\varphi) = \langle\vee\rangle\langle/\vee\rangle\langle b\rangle\mathrm{Enc}(\varphi_1)\mathrm{Enc}(\varphi_2)\langle/b\rangle$;
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $\mathrm{Enc}(\varphi) = \langle\wedge\rangle\langle/\wedge\rangle\langle b\rangle\mathrm{Enc}(\varphi_1)\mathrm{Enc}(\varphi_2)\langle/b\rangle$.

Let $q_1, \ldots, q_m$ be some canonical ordering of the states in Q. Let Σ' and Σ'' be two distinct copies of Σ with symbols of the form a' and a'', respectively, for every $a \in \Sigma$.

For each $a \in \Sigma$, we define

- $v(\langle a\rangle) \stackrel{\text{def}}{=} \langle a'\rangle\langle/a'\rangle\mathrm{Enc}(\delta(q_1, \langle a\rangle)) \cdots \mathrm{Enc}(\delta(q_m, \langle a\rangle))\langle a\rangle$, and
- $v(\langle/a\rangle) = \langle a''\rangle\langle/a''\rangle\mathrm{Enc}(\delta(q_1, q_1, \langle/a\rangle)) \cdots \mathrm{Enc}(\delta(q_m, q_m, \langle/a\rangle))\langle/a\rangle$.

We note that in $v(\langle a\rangle)$, for each $i \leqslant m$, there is a subword $\mathrm{Enc}(\delta(q_i, \langle a\rangle))$, whereas in $v(\langle/a\rangle)$ there is a subword $\mathrm{Enc}(\delta(q_i, q_j, \langle/a\rangle))$, for every $i, j \leqslant m$. The word w' is defined as the nested word $v(w)$, that results from w by replacing every tag $x \in \langle\Sigma\rangle \cup \langle/\Sigma\rangle$ with $v(x)$.

As explained above, the purpose of the game G is to simulate the alternation of A. We associate existential branching with Romeo and universal branching with Juliet. To this end, the replacement languages for $\langle\vee\rangle$ and $\langle\wedge\rangle$ are as follows.

- $R_\vee = \{\langle 1\rangle\langle/1\rangle, \langle 2\rangle\langle/2\rangle\}$;
- $R_\wedge = \{\langle 2\rangle\langle/2\rangle\}$;

All other symbols should not be replaced in the game, so we set $\Gamma = \{\vee, \wedge\}$.

The intention of the construction is that the behaviour of A on w is simulated as follows in the game on $v(w)$ in G. Choices corresponding to $\vee$-gates in transitions are taken by Romeo (and we force Juliet to call every symbol $\vee$ as words containing $\vee$-tags will not be accepted by the target NWA). The choice of $\langle 1\rangle\langle/1\rangle$ by Romeo is interpreted by the choice of the first branch of the formula by A and likewise for $\langle 2\rangle\langle/2\rangle$ and the second branch. Choices corresponding to $\wedge$-gates in transitions are taken by Juliet: we interpret $\langle\wedge\rangle\langle/\wedge\rangle$ just as $\langle 1\rangle\langle/1\rangle$ in the $\vee$-case. Therefore, if Juliet reads $\langle\wedge\rangle\langle/\wedge\rangle$ this corresponds to choosing the first branch of the formula, if she calls it, she chooses the second branch.

The target automaton follows the choices taken by the two players. At opening tags of the form $\langle(q, p)\rangle$ it interprets q and p as the next horizontal and hierarchical state,

respectively. It accepts if it ends in an accepting state or reaches $\langle\texttt{true}\rangle\langle/\texttt{true}\rangle$ at some point. If it reaches $\langle\texttt{false}\rangle\langle/\texttt{false}\rangle$ at some point, it rejects.

It remains to show that indeed w is accepted by A if and only if Romeo has a winning strategy on $v(w)$ in G.

We call a strategy for Juliet on $v(w)$ *valid* if Juliet plays *Call* on every $\vee$ symbol. Since Juliet can never win with a strategy that is not valid, we restrict our attention to valid strategies for Juliet on $v(w)$.

We will now show that each run of A on w corresponds to some strategy τ of Romeo on $v(w)$ in G, and that an accepting run on w induces a winning strategy on $v(w)$ and vice versa.

Let τ be a strategy for Romeo on $v(w)$, and let θ be some tag in w. We say that a subformula φ' encoded in $v(\theta)$ is *enabled* according to τ and some counterstrategy for Juliet if the resulting sub-play on $v(\theta)$ yields a subword of the form $\langle 1\rangle\langle/1\rangle\langle b\rangle\mathrm{Enc}(\varphi')\mathrm{Enc}(\psi)\langle/b\rangle$ or $\langle 2\rangle\langle/2\rangle\langle b\rangle\mathrm{Enc}(\psi)\mathrm{Enc}(\varphi')\langle/b\rangle$ (for some formula ψ). By the construction of $v(\theta)$, for each $q \in Q$ (and $p \in Q$, if $\theta \in \langle/\Sigma\rangle$) the set of all states $q' \in Q$ such that q' might be enabled in the sub-play on $\mathrm{Enc}(\delta(q,\theta))$ (resp. $\mathrm{Enc}(\delta(q,p,\theta))$) according to τ and some valid counter-strategy for Juliet satisfies the formula $\delta(q,\theta)$ (resp. $\delta(q,p,\theta)$). In this way, the strategy τ induces a run ρ of A on w such that for each valid counter-strategy of Juliet, the resulting rewriting of $v(w)$ corresponds to one path in ρ.

Similarly, a run ρ of A on w induces a strategy τ for Romeo on $v(w)$; if, for some tag θ in w and state $q \in Q$, $X \subseteq Q \times Q$ (resp. $X \subseteq Q$) is the follow-up state set satisfying $\delta(q,\theta)$ (resp. $\delta(q,p,\theta)$ for some appropriate $p \in Q$) in ρ, τ can be constructed to enable exactly the states from X for all counter-strategies of Juliet.

As the target automaton in G accepts a rewriting of $v(w)$ if and only if it encodes a path in a run of A on w ending in an accepting state, the correspondence between runs of A on w and strategies of Romeo on $v(w)$ in G implies that there exists a winning strategy for Romeo on $v(w)$ in G if and only if there is an accepting run of A on w. □

From Lemma 6.19, Proposition 6.18 easily follows, thus completing the proof of Theorem 6.10.

Proof of Proposition 6.18. The proof that JWin is 2-EXPTIME-hard for general games with *Call* depth $k \geqslant 2$ is by a reduction from the emptiness problem for ANWA, which is 2-EXPTIME-hard according to Theorem 5.14(a).

Given an ANWA A, let G' be the cfG constructed by the algorithm of Lemma 6.19. Let G be the game with an additional new function symbol s which Romeo is allowed to rewrite by any word of the form $v(w)$ as defined in the proof of Lemma 6.19. Then $L(A)$ is non-empty if and only if Romeo has a winning strategy on $\langle s\rangle\langle/s\rangle$ in G. This yields the desired reduction from emptiness for ANWA to JWin.

PSPACE-hardness of JWin for replay-free general games follows directly from the corresponding hardness result for the ANWA membership problem (Theorem 5.14(b)) along with the reduction of Lemma 6.19. □

6.4. Games with symmetric rule choice

This section considers an extension of context-free games on nested words that was first established for flat cfGs with finite replacement languages in [MSS06]: *Symmetric rule choice*. Different from standard cfGs, where it is always ROMEO who chooses a replacement after a *Call* by JULIET, in games with symmetric rule choice the set Γ of function symbols is partitioned into two sets Γ_{J} and Γ_{R}. Just as for standard cfGs, ROMEO chooses replacement words after *Call* moves on function symbols from Γ_{R}, but a *Call* move on a symbol from Γ_{J} actually allows JULIET to choose a replacement word instead.

Example 6.20. *Consider again the game $G = (\Sigma, \Gamma, R, T)$ from Example 6.3 with alphabet $\Sigma = \{a, b, f, g\}$, function symbols $\Gamma = \{f, g\}$, replacement languages $R_f = \{\langle a\rangle\langle/a\rangle, \langle b\rangle\langle/b\rangle\}$, $R_g = \{\langle f\rangle^n\langle/f\rangle^n \mid n \geqslant 1\}$ and target language*

$$T = \{\langle a\rangle\langle/a\rangle\langle g\rangle\langle/g\rangle, \langle b\rangle\langle/b\rangle\langle a\rangle\langle/a\rangle, \langle b\rangle\langle/b\rangle\langle b\rangle\langle/b\rangle\}.$$

Obviously, JULIET *doesn't have a winning strategy in G on the input word $w = \langle g\rangle\langle/g\rangle\langle f\rangle\langle/f\rangle$, since* JULIET *will have to Call the leading $\langle/g\rangle$, and afterwards the right-most $\langle/f\rangle$ in the replacement word chosen by* ROMEO*, and* ROMEO *can answer this Call with $\langle a\rangle\langle/a\rangle$. This causes* JULIET *to lose the game, as the subword $\langle f\rangle\langle/f\rangle$ cannot be rewritten into $\langle g\rangle\langle/g\rangle$.*

In the game G' with symmetric rule choice resulting from G by setting $\Gamma_{\mathrm{J}} = \{f\}$ and $\Gamma_{\mathrm{R}} = \{g\}$ (i.e. leaving JULIET *instead of* ROMEO *the choice what replacement word a Call to a $\langle/f\rangle$ tag yields), however,* JULIET *has a winning strategy, since she can Call the initial $\langle/g\rangle$, again Call the right-most $\langle/f\rangle$ returned by* ROMEO *and then choose to replace the called subword by $\langle b\rangle\langle/b\rangle$. After a final Call to the input word's right-most $\langle/f\rangle$ tag, she then wins the game, e.g. by replacing the final $\langle f\rangle\langle/f\rangle$ with $\langle a\rangle\langle/a\rangle$.*

Originally, games with symmetric rule choice were introduced in [MSS06] (and proven to be equivalent to standard games) as a tool to simplify the presentation of some algorithms and lower bound proofs. However, symmetric rule choice may also interpreted as a way to more accurately model the Active XML rewriting scenario where some of the services responsible for function calls are actually also under the control of the server responsible for rewriting the source document into the target schema.

The methodology in this section is somewhat different from the previous ones. This is because the results from previous sections mostly fit within the same basic framework, with only target and replacement language formalisms changed between settings. The results for this section, however, do not fit neatly within that framework, since they change more fundamental concepts about the semantics of context-free games regarding function calls.

We begin by formally defining cfGs with symmetric rule choice.

Definition 6.21. A context-free game on nested words with *symmetric rule choice* $G = (\Sigma, \Gamma_{\mathrm{J}}, \Gamma_{\mathrm{R}}, R, T)$ consists of a finite alphabet Σ, sets $\Gamma_{\mathrm{J}}, \Gamma_{\mathrm{R}} \subseteq \Sigma$ of *function symbols*

for JULIET *and* ROMEO with $\Gamma_J \cap \Gamma_R = \varnothing$, a *rule set* $R \subseteq \Gamma \times \mathrm{WF}(\Sigma)$, where $\Gamma = \Gamma_J \uplus \Gamma_R$, and a *target language* $T \subseteq \mathrm{WF}(\Sigma)$.

Configurations of G are defined as for standard cfGs on nested words (Definition 6.1). Different from that definition, the configuration $\kappa' = (p', u', v')$ is a *successor configuration* of the playable configuration $\kappa = (p, u, v)$ (Notation: $\kappa \to \kappa'$) if one of the following holds:

(1) $p' = p = \mathrm{J}$, $u' = us$, and $sv' = v$ for some $s \in \langle\Sigma\rangle \cup \langle/\Sigma\rangle$ (JULIET plays *Read*);

(2) $p = \mathrm{J}$, $p' = \mathrm{R}$, $u = u'$, $v = v' = \langle/a\rangle z$ for $z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $a \in \Gamma_R$, (JULIET plays *Call*);

(3) $p = \mathrm{R}$, $p' = \mathrm{J}$, $u = x\langle a\rangle y$, $v = \langle/a\rangle z$ for $x, z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $y \in \mathrm{WF}(\Sigma)$, $a \in \Gamma_R$, $u' = x$ and $v' = y'z$ for some $y' \in R_a$ (ROMEO plays y').

(4) $p' = p = \mathrm{J}$, $u = x\langle a\rangle y$, $v = \langle/a\rangle z$ for $x, z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $y \in \mathrm{WF}(\Sigma)$, $a \in \Gamma_J$, $u' = x$ and $v' = y'z$ for some $y' \in R_a$ (JULIET plays *Call* and replaces with y').

Based upon this definition for successor configurations, the induced reachability game, plays, (winning/finite) strategies for JULIET and ROMEO, *Call* depth and the winning problem are defined as in Definition 6.1.

For context-free games on flat strings with finite replacement languages, it can be proven (cf. [MSS06]) that cfGs with symmetric rule choice can be simulated by standard cfGs. In fact, this simulation uses a technique frequently reproduced in lower bound proofs in this thesis – on a *Call* to a function symbol from Γ_J in the simulating standard cfG, ROMEO returns a "choice string" containing one function symbol for each possible replacement word, of which JULIET may then *Call* exactly one to simulate her choice of replacement word.

This technique suffers from two disadvantages, however. Firstly, it necessitates increasing the allowed *Call* depth by 1, meaning that it could potentially increase complexities, particularly in the replay-free case. Secondly, and much more importantly, it does not directly transfer to the case of arbitrary (i.e. not necessarily finite) regular replacement languages, as it is impossible for ROMEO to directly present JULIET the choice among an infinite number of replacement words.

However, the techniques presented in Section 6.2 (and, in particular, the generic algorithm introduced in Chapter 3) can easily be adapted to cfGs with symmetric rule choice without increasing the complexity of the winning problem for JULIET. In the rest of this section, we sketch how this can be done for cfGs with DNWA target languages, that is, we prove the following analogue of Theorem 6.10. This is mostly intended as an explanation how proof techniques can be adapted to games with symmetric rule choice; similar adaptations can be made for all other results based on effects and the generic Algorithm 1.

Theorem 6.22. *For the class of context-free games on nested words with symmetric rule choice and regular replacement languages, the following complexity results hold for* JWIN*:*

(a) with bounded or unbounded replay, it is 2-EXPTIME*-complete, and*

(b) without replay, it is PSPACE*-complete.*

The lower bounds of this theorem follow directly from Theorem 6.10, as standard cfGs can be interpreted as games with symmetric rule choice where $\Gamma = \Gamma_{\mathrm{R}}$ and $\Gamma_{\mathrm{J}} = \varnothing$.

The upper bound proof again follows the outline of that in Section 6.2.2. Just as in Definition 6.4, call effects are defined for each $q \in Q$ and $a \in \Gamma$ by

$$\mathcal{C}[G](a,q) \stackrel{\text{def}}{=} \left[\{\mathrm{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma) \mid \sigma \in \mathrm{STRAT}_{\mathrm{J},\mathit{Call}}(G)\}\right]_{\min},$$

and word effects for $q \in Q$ and $w \in \mathrm{WF}(\Sigma)$ by

$$\mathcal{E}[G,w](q) \stackrel{\text{def}}{=} \left[\{\mathrm{states}_G(q, w, \sigma) \mid \sigma \in \mathrm{STRAT}_{\mathrm{J}}(G)\}\right]_{\min}.$$

Results on sequential as well as hierarchical composition of word effects (Lemmas 6.5 and 6.6) carry over directly to cfGs with symmetric rule choice, as the proofs of those lemmas are based only on the definition of effects via strategies. This implies that the ANWA $A_{\mathcal{C}[G]}$ from Section 6.2.2 can be constructed exactly as described there, and its correctness proof (Lemma 6.8) also carries over. Altogether, this yields the following result for Step 3 of the generic algorithm:

Proposition 6.23. *There is an algorithm that computes from the call effect $\mathcal{C}[G]$ of a game G with symmetric rule choice in polynomial time in $|\mathcal{C}[G]|$ and $|G|$ an ANWA $A_{\mathcal{C}[G]}$ such that $L(A_{\mathcal{C}[G]}) = \mathit{JWin}(G)$.*

Thus, the main difference for games with symmetric rule choice lies in the way that the call effect $\mathcal{C}[G]$ is computed from a game G (Step 2 of the generic algorithm).

The connection between call effects and word effects of replacement words is somewhat more involved for games with symmetric rule choice. For function symbols $a \in \Gamma_{\mathrm{R}}$ (just as in general cfGs), the idea is that a strategy of Juliet on $\langle a\rangle\langle/a\rangle$ starting with a *Call* on $\langle/a\rangle$ basically corresponds to choosing one set in $\mathcal{E}[G,w]$ for each replacement word $w \in R_a$ (corresponding to a sub-strategy on w) and then taking the union of these sets (corresponding to Romeo's choice of $w \in R_a$). For function symbols $a \in \Gamma_{\mathrm{J}}$, on the other hand, it is Juliet who may choose both the word $w \in R_a$ and her sub-strategy on it, so any set of states that Juliet can enforce on *any* $w \in R_a$, she can also enforce on $\langle a\rangle\langle/a\rangle$.

Lemma 6.24. *For any cfG $G = (\Sigma, \Gamma_{\mathrm{J}}, \Gamma_{\mathrm{R}}, R, T)$ with symmetric rule choice and a target DNWA in normal form $A(T) = (Q, \Sigma, \delta, q_0, F)$, it holds, for every $a \in \Gamma$ and $q \in Q$:*

$$\mathcal{C}[G](a,q) = \begin{cases} \left[\bigcup_{w \in R_a} \mathcal{E}[G,w](q)\right]_{\mathit{min}}, & \text{if } a \in \Gamma_{\mathrm{J}}, \\ \textsc{Mix}(\{\mathcal{E}[G,w](q) \mid w \in R_a\}), & \text{if } a \in \Gamma_{\mathrm{R}}. \end{cases}$$

Proof. The case $a \in \Gamma_{\mathrm{R}}$ was already proven as Lemma 6.7. It only remains to prove the case $a \in \Gamma_{\mathrm{J}}$, which again uses Lemma 2.1 to show that each element of a set on one side of the equation has a subset on the other side.

($\sqsupseteq$): Let $a \in \Gamma_J$, $q \in Q$ and $X \in \mathcal{C}[G](a,q)$. By definition of $\mathcal{C}[G]$, there exists a strategy $\sigma \in \mathrm{STRAT}_{J,Call}$ such that $X = \mathrm{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma)$. Again by definition, Juliet plays *Call* on $\langle/a\rangle$ according to σ. Juliet then picks some word $w \in R_a$ and keeps playing on w according to some sub-strategy σ_w of σ. It follows that $\mathrm{words}_G(\langle a\rangle\langle/a\rangle, \sigma) = \mathrm{words}_G(w, \sigma_w)$ and therefore $X = \mathrm{states}_G(q, w, \sigma_w)$ has a subset in $\mathcal{E}[G,w](q)$.

($\sqsubseteq$): Let $a \in \Gamma_J$, $q \in Q$ and $X \in \left[\bigcup_{w\in R_a} \mathcal{E}[G,w](q)\right]_{\min}$. Then $X = \mathrm{states}_G(q,w,\sigma_w)$ for some $w \in R_a$ and $\sigma_w \in \mathrm{STRAT}_J$. Obviously, also $X = \mathrm{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma)$ for the strategy $\sigma \in \mathrm{STRAT}_{J,Call}$ that plays *Call* on $\langle/a\rangle$, selects w as a replacement and then keeps playing according to σ_w on w, so X has a subset in $\mathcal{C}[G](a,q)$. □

As in previous sections, the construction of $\mathcal{C}[G]$ involves a fixpoint process iteratively computing $\mathcal{C}^{k+1}[G]$ from $\mathcal{C}^k[G]$. For the initial call effect of replay-free games, a similar characterisation to Lemma 6.11 holds.

Lemma 6.25. *For every $q \in Q$ and $a \in \Sigma$, it holds that*

$$\mathcal{C}^1[G](a,q) = \begin{cases} \{\{\delta^*(q,v)\} \mid v \in R_a\}, \text{ if } a \in \Gamma_J, \\ \{\{\delta^*(q,v) \mid v \in R_a\}\}, \text{ if } a \in \Gamma_R. \end{cases}$$

In particular, $\mathcal{C}^1[G]$ can be computed from G in polynomial time.

Proof. For the case $a \in \Gamma_R$, a proof sketch was given with Lemma 3.21. For $a \in \Gamma_J$, the claim also follows directly from the definitions, as any strategy for Juliet on $\langle a\rangle\langle/a\rangle$ that plays *Call* on $\langle/a\rangle$ simply consists of choosing a replacement word from R_a and playing *Read* on that replacement word. The polynomial-time computation is a simple adaptation of the one given for Lemma 3.21. □

The construction of $\mathcal{C}^{k+1}[G]$ from $\mathcal{C}^k[G]$ also has to be slightly modified, but the complexity result remains the same as in Lemma 6.12.

Lemma 6.26. *Given a state $q \in Q$, an alphabet symbol $a \in \Gamma$, and $\mathcal{C}^k[G]$, for some $k \geqslant 1$, the call effect $\mathcal{C}^{k+1}[G](a,q)$ can be computed in doubly exponential time in $|G|$.*

Proof. For $a \in \Gamma_R$, the proof is the same as for Lemmas 3.22 and 6.12. Let therefore $a \in \Gamma_J$. We again show that, given $\mathcal{C}^k[G]$ and a set $X \subseteq Q$, it can be decided in doubly exponential time in $|Q|$ and polynomial time in $|R|$ whether a subset of X is in $\in \mathcal{C}^{k+1}[G](a,q)$. As before, let A be defined as $A_{\mathcal{C}^k[G]}$ constructed from $\mathcal{C}^k[G]$ with starting state q and accepting state set X.

By definition, X has a subset in $\mathcal{C}^{k+1}[G](a,q)$, if Juliet has a strategy σ of call depth $k+1$ on $\langle a\rangle\langle/a\rangle$ that plays *Call* on $\langle/a\rangle$ and fulfils $\mathrm{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma) \subseteq X$. Since $a \in \Gamma_J$, such a strategy exists if and only if there exists a word $w \in R_a$ and a strategy σ_w of Juliet on w with $\mathrm{states}_G(q,w,\sigma_w) \subseteq X$, thus if and only if $R_a \cap L(A) \neq \varnothing$. As in the proof of Lemma 6.12, this can be checked by a non-emptiness test for ANWA, and therefore in doubly exponential time by Theorem 5.14. □

Finally, small adjustments are necessary to prove the adequacy of the fixpoint process. To that end, let again $\mathcal{C}^*[G]$ be defined as the result of the fixpoint computation.

Proposition 6.27. *For every cfG G with symmetric rule choice it holds: $\mathcal{C}^*[G] = \mathcal{C}[G]$.*

Proof. The proof mostly proceeds like the proof of Proposition 6.13. Let the game G' with finite replacement languages be defined as in that proof. Parts (a) and (b) of the claim in the proof of Proposition 6.13 directly carry over, as they only use effect and game definitions.

For part (c), we again prove that $\mathcal{E}[G, w](q) = \mathcal{E}[G', w](q)$ by proving that each set contained in one side of the equation has a subset contained in the other side.

Unlike in the proof of Proposition 6.13, the direction $\mathcal{E}[G, w](q) \supseteq \mathcal{E}[G', w](q)$ is not entirely trivial for games with symmetric rule choice, as JULIET actually has a bigger selection of words to choose from in G than in G', so a strategy for JULIET in G that chooses some word $z \in R_a$ on some *Call* move might not be directly transferable to G' if $z \notin R'_a$. However, as R'_a contains the word $v(a, z)$, by a similar argument as the one in the proof of Proposition 6.13, JULIET can select $v(a, z)$ instead in G' and follow a strategy on $v(a, z)$ yielding the desired set of states.

For the direction $\mathcal{E}[G, w](q) \sqsubseteq \mathcal{E}[G', w](q)$ of part (c), one can easily adapt the nested induction proof for Proposition 6.13.

The cases for $d = 0$ as well as $d > 0$ with $w = uv$ or $w = \langle a\rangle v\langle/a\rangle$ for $a \in \Gamma_{\mathrm{R}}$ or $a \notin \Gamma$ in that induction carry over from the proof of Proposition 6.13. The case where $d > 0$ and $w = \langle a\rangle v\langle/a\rangle$ with $a \in \Gamma_{\mathrm{J}}$ and $v \in \mathrm{WF}(\Sigma)$ is similar to the case with $a \in \Gamma_{\mathrm{R}}$, but much simpler, since each choice $z \in R'_a$ for JULIET in G' after a *Call* to $\langle/a\rangle$ is also a valid choice in G because $R'_a \subseteq R_a$. □

As it could be shown that all parts of the generic algorithm for cfGs on nested words can be adapted to games with symmetric rule choice without any increase in complexity, Theorem 6.22 follows.

6.5. Games with insertion semantics

This section focusses on an alternative semantics for context-free games, which allows that returned trees do not *replace* their call nodes but are inserted next to them.

Similar to the previous section, the results and methods presented in this section do not exactly match the framework presented in Sections 6.2 and 6.3. Different from the previous section, however, the results presented here do not follow directly from that framework with a few minor modifications; instead, the central technique used in this section are reductions between cfGs with replacement semantics (as examined in previous sections) and cfGs with insertion semantics, detailed below.

In the original definition of *Call* moves, we define the successor configurations of a configuration $(\mathrm{R}, u\langle a\rangle v, \langle/a\rangle w)$ to be of the form $(\mathrm{J}, u, v'w)$, that is, $\langle a\rangle v\langle/a\rangle$ is *replaced* by a word $v' \in R_a$. However, Active XML also offers an "append" option, where results of function calls are inserted as siblings after the calling function node (cf. [ABM08]). There are (at least) three possible semantics of a *Call* move for insertion (as opposed to replacement) based games: the next configuration could be (1) $(\mathrm{J}, u, \langle a\rangle v\langle/a\rangle v'w)$, (2) $(\mathrm{J}, u\langle a\rangle v\langle/a\rangle, v'w)$, or (3) $(\mathrm{J}, u\langle a\rangle v\langle/a\rangle v', w)$, depending on "how much replay" we allow

for JULIET. We refer to (1) as the general setting, (2) as the setting with *weak replay* and (3) as the setting *without replay*. It turns out that the weak replay setting basically corresponds to the (unrestricted) setting with replacement rules and that setting (3) corresponds to the replay-free setting with replacement rules. Setting (1), however, gives JULIET a lot of power and makes JWIN undecidable.

Formally, we consider cfGs of the form $G = (\Sigma, \Gamma, I, T)$ where the insertion relation $I \subseteq \Gamma \times \mathrm{WF}(\Sigma)$ takes the place of the replacement relation R from cfGs as defined in Section 6.1. The semantics is similar to standard cfGs, except for the definition of follow-up configurations after a *Call* move by JULIET, as explained above.

The restriction to games having *only* insertion rules is primarily to simplify the presentation of proofs. It is relatively easy (if tedious) to prove that games can be extended to contain both replacement and insertion rules without changing the complexity of JWIN, as long as appropriate semantics for insertion and replacement rules are chosen.

We generally assume $G = (\Sigma, \Gamma, I, T)$ to be an insertion game with target language T represented by a DNWA $A(T)$ and insertion languages I_a represented by an arbitrary NWA for each $a \in \Gamma$.

Theorem 6.28. *For the class of games on nested words with insertion semantics, target languages given as DNWAs and replacement languages given as NWAs, the following decidability and complexity results hold for* JWIN*:*

(a) in general, it is undecidable;

(b) for games with weak replay, it is 2-EXPTIME*-complete, and*

(c) for games without replay, it is PSPACE*-complete.*

The proof idea for Theorem 6.28 is to simulate insertion-based games by replacement-based games and vice versa; part (a) additionally uses the undecidability of JWIN for unconstrained cfGs on flat strings (Theorem 3.3).

Before proving Theorem 6.28, we prove two auxiliary results showing a strong correspondence between replacement games and insertion games (with appropriate semantics). We recall that for a replacement game G, $\mathrm{JWin}(G)$ denotes the set of all winning words for JULIET in G with unbounded replay and $\mathrm{JWin}^1(G)$ denotes the set of all winning words for JULIET in G for strategies without replay; similarly, we denote the winning set for JULIET in an insertion game G' by $\mathrm{JWin}(G')$ in the general setting, by $\mathrm{JWin}^{1+}(G')$ with weak replay, and by $\mathrm{JWin}^1(G')$ without replay.

Lemma 6.29. *There exists a polynomial-time algorithm that, given a replacement cfG $G = (\Sigma, \Gamma, R, T)$ and nested word $w \in WF(\Sigma)$, outputs an insertion cfG $G' = (\Sigma', \Gamma, I, T')$ and word $w' \in WF(\Sigma')$ such that*

- *$w \in JWin(G) \Leftrightarrow w' \in JWin^{1+}(G')$, and*
- *$w \in JWin^1(G) \Leftrightarrow w' \in JWin^1(G')$.*

Proof. The main observation we need is that replacement in cfGs is generally very localised, i.e. a *Call* on $\langle/a\rangle$ in a word of the form $w\langle a\rangle v\langle/a\rangle$ only affects $\langle a\rangle v\langle/a\rangle$, the shortest well-nested suffix of the current word up to $\langle/a\rangle$.

The obvious idea behind the proof is to simulate replacement rules with insertion rules. The crucial insight for this simulation is that, while we cannot delete the rooted suffix $u = \langle a\rangle v\langle/a\rangle$ from a current word, the new target automaton $A(T')$ can "undo" the effect of u on $A(T)$ by reverting it to the state it had before reading u. To this end, $A(T')$ simulates $A(T)$, all the while memorising (in its state) a "fallback state" that $A(T)$ was in before beginning to read u. That way, $A(T')$ can always revert its simulation of $A(T)$ to the point before u was read, effectively making $A(T)$ "forget" u and thus simulating a replacement of u.

In this way, it is easy to simulate deletion of suffixes that would be replaced in G, so we only need some way of knowing *when* such a deletion should take place. To this end, we encapsulate replacement words u' for G within *backspace* tags as $\langle b\rangle u'\langle/b\rangle$ to obtain insertion words for G' (with $b \notin \Sigma$). Now, when the automaton A' reads $\langle b\rangle$, it knows that what follows after is supposed to be a replacement word, so it "forgets" the last rooted suffix of the current word, jumps back to the last fallback state and continues simulating $A(T)$ on u', re-setting its fallback state along the way as necessary.

Formally, let $A(T) = (Q, \Sigma, \delta, q_0, F)$ be a DNWA in normal form for T and let $b \notin \Sigma$. We define G' as follows:

- $\Sigma' = \Sigma \cup \{b\}$
- $I_a = \{\langle b\rangle u'\langle/b\rangle \mid u' \in R_a\}$ for all $a \in \Gamma$ and
- $T' = L(A')$ for the DNWA A' defined below.

The automaton $A' = (Q', \Sigma \cup \{b\}, \delta', q_0', F')$ is defined by

- $Q' = Q \times Q$;
- $q_0' = (q_0, q_0)$;
- $F' = F \times Q$ and
- $\delta'((p,q), \langle a\rangle) = (\delta(p, \langle a\rangle), p)$ for all $a \in \Sigma$,
- $\delta'((p,q), \langle b\rangle) = (q, q)$,
- $\delta'((p,q), (p',q'), \langle/a\rangle) = (\delta(p, p', \langle/a\rangle), q')$ for all $a \in \Sigma$ and
- $\delta'((p,q), (p',q'), \langle/b\rangle) = (p, q)$.

In keeping with the above intuition, A' tracks in its state (p,q) a *current* state p and a *fallback* state q of $A(T)$. When A' reads an $\langle a\rangle$ (resp. $\langle/a\rangle$), it knows that the rooted word immediately to the left of $\langle a\rangle$ has *not* been replaced, so it simulates a step of $A(T)$ to obtain a new current state and sets the new fallback state to be the state A had immediately before reading $\langle a\rangle$ (respectively the $\langle a\rangle$ associated with the current $\langle/a\rangle$).

On reading $\langle b\rangle$, A' knows that the last rooted nested subword has been replaced in G, so it returns its simulation of $A(T)$ to the fall-back state and simulates $A(T)$ on the replacement word following after $\langle b\rangle$ from there. On $\langle/b\rangle$, neither the current nor fallback state changes, as the last rooted word to the right of $\langle/b\rangle$ may be considered the last minimal suffix of the current word in the replacement game.

If, after reading a word and simulating $A(T)$ on it as described above, $A(T)$ accepts (i.e. the current state of A' is in F), A' accepts as $A(T)$ would.

We can obviously construct G' from G in polynomial time; setting $w' = w$, the claim of Lemma 6.29 follows by the above considerations. □

Lemma 6.30. *There exists a polynomial-time algorithm that, given an insertion cfG $G = (\Sigma, \Gamma, I, T)$ and nested word $w \in WF(\Sigma)$, outputs a replacement game $G' = (\Sigma', \Gamma', R, T')$ and word $w' \in WF(\Sigma')$ such that*

- $w \in JWin^{1+}(G) \Leftrightarrow w' \in JWin(G')$, *and*
- $w \in JWin^{1}(G) \Leftrightarrow w' \in JWin^{1}(G')$.

Proof. The basic idea behind simulating insertion games using replacement games is to replace every subword $\langle a\rangle v\langle/a\rangle$ of w by $\langle a\rangle v'\langle/a\rangle\langle a'\rangle\langle/a'\rangle$ in w' (where a' is a new "copy" of a) and to simulate the insertion of a new subword to the right of $\langle a\rangle v\langle/a\rangle$ by the replacement of $\langle a'\rangle\langle/a'\rangle$. We refer to the additional subwords of the form $\langle a'\rangle\langle/a'\rangle$ as "anchors".

To this end, we need to ensure that (a) no non-anchor subword ever gets replaced, and (b) each replacement word contains new anchors for further insertions. For part (a), we add extra symbols to the input alphabet, while part (b) is done through the transformation from $w \in \mathrm{WF}(\Sigma)$ to $w' = \mu(w) \in \mathrm{WF}(\Sigma')$ hinted at in the claim's statement.

More formally, we set $\Sigma' = \Sigma \cup \{a' \mid a \in \Sigma\}$, i.e. we add a second disjoint copy of Σ to itself. Words will generally be transformed using a function $\mu : \mathrm{WF}(\Sigma) \to \mathrm{WF}(\Sigma')$ defined inductively by

- $\mu(\epsilon) = \epsilon$
- $\mu(uv) = \mu(u)\mu(v)$ for all $u, v \in \mathrm{WF}(\Sigma)$ and
- $\mu(\langle a\rangle v\langle/a\rangle) = \langle a\rangle\mu(v)\langle/a\rangle\langle a'\rangle\langle/a'\rangle$ for all $a \in \Sigma$, $v \in \mathrm{WF}(\Sigma)$.

The target language automaton $A(T')$ for G' simply simulates the target language DNWA $A(T)$ for G and ignores any uncalled anchors, i.e. when reading any symbol from $\Sigma'\backslash\Sigma$, it simply stays in its current state, and otherwise it behaves as $A(T)$ does. Clearly, $A(T')$ can be constructed from $A(T)$ in polynomial time.

The set of function symbols in G' is just $\Gamma' = \{a' \mid a \in \Gamma\}$, and the replacement languages are defined by

$$R_{a'} = \{\mu(w) \mid w \in R_a\}.$$

Again, it is easy to see that automata for each $R_{a'}$ can be computed from those for R_a in polynomial time.

Finally, the input string gets transformed (in polynomial time) via μ as well: $w' = \mu(w)$. □

Proof of Theorem 6.28. Parts (b) and (c) follow directly from Lemmas 6.30, 6.29 as well as Theorem 6.10. All that remains to be proven is therefore the undecidability of JWin in the general setting.

Intuitively this holds because, on a string of the form $\langle a\rangle v\langle/a\rangle$, jumping back to the start after calling $\langle/a\rangle$ effectively allows Juliet to play arbitrarily many left-to-right passes on v, thereby enabling her to simulate a strategy with left steps on v. We utilise this fact to give a reduction from the winning problem on unconstrained flat cfGs, which is undecidable by Theorem 3.3.

For the reduction, we construct a nested insertion cfG $G' = (\Sigma \uplus \{r, b, t\}, \Gamma, I, T')$ from a given input flat cfG $G = (\Sigma, \Gamma, R, T)$ with the DFA $A(T) = (Q, \Sigma, \delta, q_0, F)$ for the target language T. The idea is to simulate a strategy with left steps for Juliet on some flat string $w \in \Sigma^*$ in G by means of a standard (left-to-right) strategy on the standard nested word encoding $\widehat{w} \in \mathrm{WF}(\Sigma)$ derived from $w \in \Sigma^*$ by replacing each symbol a in w with $\langle a\rangle\langle/a\rangle$.

The idea for the reduction, now, is to transform the input word w into a string of the form $\langle r\rangle\widehat{w}\langle/r\rangle$ (for some $r \notin \Sigma$), simulate each left-to-right pass for Juliet on w appropriately on $\widehat{w}$ and then use a *Call* on $\langle/r\rangle$ to simulate a LS move, appending some irrelevant "tail" $\langle t\rangle\langle/t\rangle$ (for $t \notin \Sigma$) to the current nested word in the process.

The only minor conceptual difficulty is how to simulate a left-to-right pass of Juliet on $\widehat{w}$ using insertion rules, as flat context-free games are defined using only replacement. This can be done with a similar technique as described in the proof of Lemma 6.29 – replacement words v from some replacement language $R_a \subseteq \Sigma^*$ are transformed into nested words $\widehat{v}$ as above and encapsulated in "backspace" tags as $\langle b\rangle\widehat{v}\langle/b\rangle$ (for $b \notin \Sigma$); on reading $\langle b\rangle$, the target DNWA for G' "forgets" the last nested word before the $\langle b\rangle$ by restoring a fallback state of $A(T)$. The only difference to the proof of Lemma 6.29 is that here, the target DNWA for G' merely has to simulate a DFA, not a DNWA. This can be done in the same way as in the reduction for the lower bound proofs in Theorem 6.16. □

6.6. Outlook and bibliographical remarks

This chapter introduced context-free games on nested words and gave a complete classification of the complexity of solving these games for the cases of no replay, bounded replay and unbounded replay, both for games with target NWAs and for games where target languages are given as SNWAs. While in the former case, complexities rise above those for context-free games on strings, complexities in the latter case (which, notably, includes cases where relevant languages are given by practical specifications for XML, such as DTDs or XML Schema) match those for games on strings.

In addition to this characterisation of complexities, this chapter introduced two extensions to the basic setting of context-free games: Games with symmetric replacement rules, which address some of the basic asymmetry inherent in context-free games by

adding function symbols which allow JULIET (instead of ROMEO) the choice of replacement word, and games with insertion semantics, which insert function call results into the document in addition to the originally called subword instead of replacing that subword. As it turns out, neither of these extensions change the complexity of the winning problem for JULIET– in fact, insertions may be simulated by replacement and vice versa, and the effect technique used to solve context-free games in the standard setting can easily be extended to accommodate symmetric replacement as well.

This goes to show that the basic model of context-free games is somewhat robust with regards to extensions, as was already proven for symmetric rules choice in context-free games on strings (as well as some other extensions like *navigation constraints*) in [MSS06]. In this vein, it might be interesting to find some characterisation for extensions that conserve complexity properties in this manner, since more expansive extensions like input dependencies (examined in the following chapters) radically increase complexity.

Bibliographical remarks All results presented in this chapter, except for those in Section 6.4 were originally published as joint work with Thomas Schwentick in [SS15] (and that paper's journal version [SS16]), based on an extension of the effect technique used in [BSSK13]. For more bibliographical remarks on effects, see also Section 3.5.

The results given in Section 6.4 are original work and have not been published before; [MSS06] give comparable results for context-free games on flat strings, but their proof techniques are quite different from the ones used here (as discussed at the start of Section 6.4).

Part III.

Parameter-dependent context-free games

7. Context-free games with parameter validation

As mentioned in Section 1.2, in the practical application of Active XML rewriting, subtrees that get overwritten by a call to a function node are supposed to serve as input parameters to the external service that gets called. The semantics of flat or nested context-free games as discussed in the previous chapters so far does not take these parameters into account.

In general, one would expect the output of function calls to depend upon the input parameters in some sensible (i.e. efficiently representable) way; in this chapter, we examine a somewhat limited dependency of outputs on inputs, namely that outputs for a function call can be arbitrarily chosen in accordance with some fixed output schema, as long as the input is correct with respect to some input schema. This is also the setting considered in the seminal theoretical work on Active XML schema rewriting [MAABN05].

As an example, consider the online book store aggregator from Section 1.2 (p. 7). In the practical setting underlying this example, the search engine represented by the @search-engine node might expect its parameters to contain a node labelled "max-price", and, optionally, a node labelled "type", which has to occur to the left of the "max-price" node. If, for instance, the maximum price parameter is not present in a function call, or if it occurs to the left of the type parameter, the search engine might return an error message instead of a search result, or simply return nothing at all.

Transferred to the setting of cfGs this means that JULIET should only receive a response to a *Call* move in a configuration $(\mathrm{J}, u\langle a\rangle v, \langle/a\rangle w)$ if $\langle a\rangle v\langle/a\rangle$ is in V_a for some set V_a of words that are valid for calls of $\langle/a\rangle$.[1] *Call* moves with invalid parameters, on the other hand, should yield some sort of validation error, which we model as an *error string* $\langle\mathrm{err}\rangle\langle/\mathrm{err}\rangle$ being returned.[2]

We do not investigate all possible game types in combination with parameter validation but rather concentrate on the most promising setting with respect to tractable algorithms. Specifically, we restrict our attention to games without replay, as we are seeking to identify tractable cases, and JWIN is already PSPACE-hard for bounded-replay games on nested words with target languages specified by DTDs *without* parameter validation (cf. Theorem 6.16).

It turns out that games without replay and with DTDs to specify target, replacement and validation languages have a tractable winning problem as long as the number of

[1] Note that we focus solely on nested word cfGs here, as it is not clear how to define parameters corresponding to a subtree in the flat cfG setting.

[2] Alternative semantics where, for instance, JULIET is unable to perform a *Call* move when a parameter subword is invalid, are also imaginable; this is discussed to some extent in Section 7.3 and [SS16].

	DNWAs	XML Schema	General DTDs	Bounded DTDs
Without validation	PSPACE	PTIME	PTIME	PTIME
With validation	EXPTIME	EXPTIME	PSPACE	PTIME

Table 7.1.: Comparison of complexity results for nested standard cfGs and validation cfGs. All results are completeness results for games without replay.

different validation DTDs is bounded by some constant. The winning problem becomes intractable if the number of validation DTDs can be unbounded; with target and validation languages specified by XML Schemas, on the other hand, it is already intractable with only one validation schema. A comparison of complexity results between replay-free standard games without validation and replay-free games with validation can be found in Table 7.1. Note that the result for a bounded number of validation DTDs implies polynomial-time *data complexity* for arbitrary replay-free games on nested words with DTD target, replacement and validation languages.

As far as methodology is concerned, we do not use the basic concept of effects from Chapters 3 to 6 here (although similar ideas of representing subgames by succinct representations of their outcomes are used in some upper bounds). This is because the use of validation languages would greatly complicate both the definition and the computation of call effects; also, due to the absence of replay, effects would only be of limited utility.

7.1. Definitions

Following the intuition given above, we define context-free games with validation (or *validation cfGs*) in a manner similar to the standard cfGs on nested words of Chapter 6, with the main difference being that validation cfGs are additionally equipped with a *validity relation* V and an *error symbol* err. The idea behind the semantics is that, if Juliet plays *Call* on some rooted subword $\langle a\rangle v\langle/a\rangle$, then depending on whether the subword v is *valid* for a (i.e. whether $(a, \langle a\rangle v\langle/a\rangle) \in V$ holds), Romeo either replaces $\langle a\rangle v\langle/a\rangle$ with some string from the replacement language for a (as in standard cfGs) if v is valid for a, or returns an *error string* $\langle\text{err}\rangle\langle/\text{err}\rangle$ instead, otherwise.

Definition 7.1. A *context-free game on nested words with validation* (or *validation cfG* for short) $G = (\Sigma, \Gamma, R, V, T, \text{err})$ consists of a finite alphabet Σ, a set $\Gamma \subseteq \Sigma$ of *function symbols*, a *(replacement) rule set* $R \subseteq \Gamma \times \text{WF}(\Sigma)$, a *validity relation* $V \subseteq \Gamma \times \text{WF}(\Sigma)$, a *target language* $T \subseteq \text{WF}(\Sigma)$ and an *error symbol* $\text{err} \in (\Sigma \backslash \Gamma)$. As for standard cfGs, we generally assume that the target language T, each *replacement language* $R_a \stackrel{\text{def}}{=} \{w \in \text{WF}(\Sigma) \mid (a, w) \in R\}$ and, additionally, each *validation language* $V_a \stackrel{\text{def}}{=} \{w \in \text{WF}(\Sigma) \mid (a, w) \in V\}$ are nonempty regular languages of nested words.

The semantics for validation cfGs is defined based on successor configurations, where a configuration is again a tuple $\kappa = (p, u, v) \in \{\text{J}, \text{R}\} \times (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^* \times (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$ as in Definition 6.1.

The configuration $\kappa' = (p', u', v')$ is a *successor configuration* of the playable configuration $\kappa = (p, u, v)$ (Notation: $\kappa \to \kappa'$) if one of the following holds:

(1) $p' = p = \mathrm{J}$, $u' = us$, and $sv' = v$ for some $s \in \langle\Sigma\rangle \cup \langle/\Sigma\rangle$ (Juliet plays *Read*);

(2) $p = \mathrm{J}$, $p' = \mathrm{R}$, $u = u'$, $v = v' = \langle/a\rangle z$ for $z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $a \in \Gamma$, (Juliet plays *Call*);

(3) $p = \mathrm{R}$, $p' = \mathrm{J}$, $u = x\langle a\rangle y$, $v = \langle/a\rangle z$ for $x, z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $\langle a\rangle y\langle/a\rangle \in V_a$, $u' = x$ and $v' = y'z$ for some $y' \in R_a$ (y is valid for a and Romeo plays y'), or

(4) $p = \mathrm{R}$, $p' = \mathrm{J}$, $u = x\langle a\rangle y$, $v = \langle/a\rangle z$ for $x, z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $\langle a\rangle y\langle/a\rangle \notin V_a$, $u' = x$ and $v' = \langle\mathrm{err}\rangle\langle/\mathrm{err}\rangle z$ (y is invalid for a and Romeo returns an error).

Again, all other semantic definitions like the induced reachability game, plays, strategies and the winning problem are defined as in Definition 6.1.

Note that the symbol err may be used freely (like any other symbol) in the target language and additional validation languages. Furthermore, the semantic definition calls for validity of strings of the form $\langle a\rangle v\langle/a\rangle$ for a; while it would be sufficient to only test v in place of $\langle a\rangle v\langle/a\rangle$, the latter has the advantage of making V_a a language of trees with root node labelled a.

7.2. Complexity results

In this section, we examine the complexity of the winning problem for Juliet in replay-free validation games, with different settings characterised by the representation of target, replacement and validation languages.

Upper bound proofs again follow the basic idea of succinct representation of replacement strings as sketched in Section 1.3; since (as already stated) the effect technique cannot be easily transferred to the setting of validation games, the details of this succinct representation get a little bit more involved here. Lower bound proofs, on the other hand, are by relatively straightforward reductions from the intersection emptiness problem for XML Schema (Prop. 7.3) and DTDs (Prop. 7.6), and the emptiness problem for DTDs (Prop. 7.9), which all follow a basic idea from Section 3.3: tasking Romeo to provide a witness for non-emptiness and Juliet with finding an error in the witness provided by Romeo.

7.2.1. Regular nested word languages and XML Schema

In this subsection, we prove an exponential-time upper bound for games with target, validation and replacement languages given by DNWA. Furthermore, we prove that a corresponding lower bound already holds for games with a single function symbol, DTD replacement and target languages, and a single XML Schema validation language, so these bounds are tight for arbitrary combinations of XML Schema and DNWA target, validation and replacement languages.

Proposition 7.2. *For the class of validation games with target, validation and replacement languages specified by DNWA, and strategies without replay,* JWin *is in* EXPTIME.

Proof. We describe an alternating polynomial-space algorithm $\mathcal{A}$ which, given a validation game $G = (\Sigma, \Gamma, R, V, T, \mathrm{err})$ and an input string $w \in \mathrm{WF}(\Sigma)$, decides whether Juliet has a winning strategy on w in G. To simplify the algorithm's presentation, we assume $\Gamma = \{1, \ldots, k\}$.

The basic idea behind the algorithm $\mathcal{A}$ is to use alternation to simulate a play of Juliet and Romeo on w while aggregating over sub-plays on replacement strings in a manner similar to the effect technique used in Chapters 3 to 6. In other words, while the algorithm simulates a play on a given input string, it does not track the current string itself through the play, but only stores enough relevant information about sub-plays regarding the target and validation automata to decide validity of substrings and containment of the final string in the target language.

Thanks to the lack of replay, succinct representations do not need to take *Call* moves inside of replacement strings into account. On the other hand, unlike in Chapters 3 to 6, it does not suffice to aggregate sub-plays into states of the target language automaton, due to the need to take care of validation. As an additional complication, just tracking states of the validation automata is not enough, either, since we need to deal with nested function symbols and thus potentially with the impact of strings on several copies of the same validation automaton.

To solve this problem, $\mathcal{A}$ tracks *aggregated transition functions* for each relevant DNWA, i.e. instead of keeping some string v obtained by an alternating rewriting in memory, it instead tracks the transitions induced by v starting from each state of each DNWA. It does so in a relatively straightforward manner by going through the input string in a left-to-right fashion; any time it encounters an opening tag, it initialises a new aggregated transition function for each automaton (storing the old ones on a stack), and every time it encounters a closing tag, it guesses existentially a move by Juliet and universally a response by Romeo (if applicable) and aggregates those onto the transition functions it has stored so far. After the entire input string has been processed in this manner, all that remains is to check whether the aggregated transition function for the rewriting that $\mathcal{A}$ has guessed corresponds to a string in T.

More concretely, $\mathcal{A}$ tracks a $(k+1)$-tuple t of aggregated transition functions, one for each validation DNWA as well as the target DNWA. That is, if $\mathcal{A}$ has already processed a prefix $u\langle a\rangle v$ of w, with $v \in \mathrm{WF}(\Sigma)$ being the longest well-nested suffix, the tuple t contains, for each DNWA with transition function δ, the function $\delta^*(\cdot, v')$, where v' is the rewriting of v according to a play which $\mathcal{A}$ has guessed using alternation. If $\mathcal{A}$ next reads some opening $\langle b\rangle$ tag, it pushes t to a stack and proceeds on the string nested inside the b tags, starting with a new tuple of identity functions; if it reads a closing $\langle/a\rangle$ tag instead, it guesses Juliet's strategy choice on $\langle/a\rangle$ and (if applicable) a reply by Romeo, pops the function tuple t' it has pushed with the corresponding $\langle a\rangle$ tag, and computes from t, t' and the strategy choices of Juliet and Romeo a new function tuple

corresponding to the rewriting of $\langle a\rangle v\langle/a\rangle$.[3]

To describe $\mathcal{A}$ in formal detail, let $A_i = (Q_i, \Sigma, \delta_i, s_i, F_i)$ be a DNWA in normal form for V_i (for each $i \in [k]$), and let $A_T = (Q_T, \Sigma, \delta_T, s_T, F_T)$ be a DNWA in normal form for T. A *function tuple* is a tuple $(f_1, \ldots, f_k, f_T)$, where $f_T : Q_T \to Q_T$, and (for each i) $f_i : Q_i \to Q_i$. By id we denote the function tuple that has an identity function id_i in each entry. For two function tuples $t = (f_1, \ldots, f_k, f_T)$, $t' = (f'_1, \ldots, f'_k, f'_T)$, let $t \circ t' = (f_1 \circ f'_1, \ldots, f_k \circ f'_k, f_T \circ f'_T)$ denote their component-wise composition. Furthermore, for $a \in \Sigma$ and a function tuple $t = (f_1, \ldots, f_k, f_T)$, let $a(t) = (a(f_1), \ldots, a(f_k), a(f_T))$, where, for each $i \in \{1, \ldots, k\} \cup \{T\}$, $a(f_i) : Q_i \to Q_i$ is the function defined for each $q \in Q_i$ by $a(f_i)(q) = \delta(f_i(\delta(q, \langle a\rangle)), q, \langle/a\rangle)$, i.e. the aggregated transition function simulating a transition with $\langle a\rangle$ before, and with $\langle/a\rangle$ after applying f_i.

The algorithm $\mathcal{A}$ tracks a current function tuple t, initialised to id, and a stack of function tuples, initialised to ϵ. It reads the input string w from left to right. With the current tuple t being $(f_1, \ldots, f_k, f_T)$, $\mathcal{A}$ proceeds as follows:

- On an opening tag $\langle a\rangle$ with $a \in \Sigma$, $\mathcal{A}$ pushes $(f_1, \ldots, f_k, f_T)$ to the stack and then resets t to id.
- On a closing tag $\langle/a\rangle$, $\mathcal{A}$ guesses existentially a strategy choice for Juliet on $\langle/a\rangle$, if $a \in \Gamma$.
  - (i) Depending on the precise case, $\mathcal{A}$ then summarises the information about the subgame on the current substring into a function tuple t'' as follows.
    - (a) If $a \in \Sigma\backslash\Gamma$ (and Juliet must therefore play *Read* on $\langle/a\rangle$), or if $a \in \Gamma$ and Juliet plays *Read* on $\langle/a\rangle$, then $\mathcal{A}$ sets $t'' := a(t)$;
    - (b) If $a = i$ for some $i \in \Gamma$, Juliet plays *Call* on $\langle/i\rangle$, and $f_i(s_i) \notin F_i$ (i.e. Juliet's *Call* on $\langle/a\rangle$ is invalid), $t'' := \mathrm{err}(t)$;
    - (c) If $a = i$ for some $i \in \Gamma$, Juliet plays *Call* on $\langle/i\rangle$, and $f_i(s_i) \in F_i$ (i.e. Juliet plays a valid *Call* on $\langle/a\rangle$), $\mathcal{A}$ guesses universally a function tuple t'' corresponding to Romeo's choice of replacement string, and verifies that there is indeed a replacement string in R_i inducing t''.
  - (ii) Finally $\mathcal{A}$ pops $t' = (f'_1, \ldots, f'_k, f'_T)$ from the stack and replaces t by $t' \circ t''$.

At the end of w, $\mathcal{A}$ accepts if and only if $f_T(s_T) \in F_T$ holds for the final function tuple $t = (f_1, \ldots, f_k, f_T)$.

Assuming that the verification in case (iii) is possible in alternating polynomial space, it is clear that $\mathcal{A}$ indeed only requires alternating polynomial space; furthermore, under the same assumption, it is easy to see (if somewhat tedious to prove) that $\mathcal{A}$ is correct, as it directly simulates a play of G on w, aggregating well-nested rewritten strings into their induced transition functions.

It remains to be explained how we verify, for a function tuple $t'' = (f''_1, \ldots, f''_k, f''_T)$ and $i \in \Gamma$, whether there exists a string $v \in R_i$ inducing the transition functions in t'', i.e.

[3] This technique is somewhat similar to (and inspired by) the algorithm for determinising nondeterministic NWA [AM09].

with $f''_j(q_j) = \delta^*_j(q_j, v)$ for each $j \in \{1, \ldots, k, T\}$ and $q_j \in Q_j$. For a single tuple, this is possible in deterministic exponential time by computing a product automaton consisting, for each $j \in \{1, \ldots, k, T\}$, of $|Q_j|$ many copies of A_j, each starting in a different state $q \in Q_j$ and having $f_j(q)$ as its only accepting state, and a single copy of A_i without any modification. Obviously, this product automaton is of exponential size and is nonempty if and only if there is a string $v \in R_i$ such that for each $j \in \{1, \ldots, k, T\}$ and each $q_j \in Q_j$ it holds that $\delta^*_j(q_j, v) = f(q_j)$, and emptiness for this product automaton can be checked in deterministic exponential time. Since EXPTIME = APSPACE, this yields the desired complexity for verification in case (iii) above and completes the proof. □

The corresponding lower bound proof once again follows one of the standard techniques for lower bounds: Reducing from the complement of an existence problem (in this case the intersection emptiness problem for XML Schema), tasking Romeo with providing a witness for existence (here: a nested word encoding a tree in the intersection), and having Juliet verify the correctness of this witness.

Proposition 7.3. *For the class of validation games with target, validation and replacement languages specified by XML Schemas, and strategies without replay,* JWin *is* EXPTIME*-hard. This lower bound already holds for games with a DTD target language and one single function symbol, whose replacement language is given by a DTD.*

Proof. The proof is by reduction from the intersection emptiness problem for XML Schema: Given XML Schemas $S_1, \ldots, S_n$ over a common label alphabet Σ, is $L(S_1) \cap \ldots \cap L(S_n) = \varnothing$? This problem is known to be EXPTIME-complete [MNS09].

The basic idea behind the reduction is to construct from $S_1, \ldots, S_n$ a game in which Romeo chooses some string $v \in \mathrm{WF}(\Sigma)$, and Juliet picks an index $i \in [n]$ such that $v \notin L(S_i)$; in this way, Romeo has a winning strategy (which basically consists of choosing a string from $L(S_1) \cap \ldots \cap L(S_n)$) if and only if $L(S_1) \cap \ldots \cap L(S_n)$ is nonempty. The rest of the proof consists of setting up this game G and an input string w such that the above idea can be carried out with a replay-free game with only a single function symbol, and making sure that none of the players is able to win the game by cheating (like for instance Juliet not playing any *Call* moves and denying Romeo the option of choosing a string v).

The input string is $w = \langle f \rangle^{n+1} \langle / f \rangle^{n+1}$, with $f \notin \Sigma$ being the only function symbol in G. The idea is that Juliet is supposed to *Call* the innermost $\langle / f \rangle$, to which Romeo replies with a string $v \in \mathrm{WF}(\Sigma)$, leading to a string of the form $\langle f \rangle^n v \langle / f \rangle^n$. The validation language for f should be set up such that it contains, for all $i \in [n]$, all strings from $L(S_i)$ enclosed in i opening and closing f-tags. In this way, Juliet can provoke a return of $\langle \mathrm{err} \rangle \langle / \mathrm{err} \rangle$ to a *Call* on some $\langle / f \rangle$ if and only if the string v given by Romeo is not in $L(S_i)$ for some i.

In order for Romeo to be able to return the string v, Juliet's first *Call* to the innermost $\langle / f \rangle$ must be valid; therefore, the validation language V_f must contain the string $\langle f \rangle \langle / f \rangle$. At first glance, it might seem that having the validation language V_f consist of $\langle f \rangle \langle / f \rangle$ as well as all $\langle f \rangle^i v \langle / f \rangle^i$ for all $i \in [n]$ and $v \in L(S_i)$ would suffice; however, with a validation language of this shape, Juliet would always be able to win

the game by playing *Call* on some $\langle/f\rangle$ that is *not* the innermost, since it is JULIET'S goal to provoke a return of $\langle\text{err}\rangle\langle/\text{err}\rangle$ and this naïve validation language would yield such an error on an input of multiple nested f tags. Therefore V_f must also contain all strings of the form $\langle f\rangle^i\langle/f\rangle^i$ for $i \in [n]$. In the above sketch, however, this would allow ROMEO the option to cheat by always returning the empty string to *any Call* by JULIET, never allowing her to make an invalid *Call*. Therefore, we preface all replacement strings yielded by ROMEO with a special symbol $\# \notin \Sigma$, i.e. the replacement language for f is

$$R_f = \{\langle\#\rangle\langle/\#\rangle v \mid v \in \mathrm{WF}(\Sigma)\}.$$

Combining this with our previous considerations, we obtain the following validation language for f:

$$V_f = \{\langle f\rangle^i\langle/f\rangle^i \mid i \in [n+1]\} \cup \{\langle f\rangle^i\langle\#\rangle\langle/\#\rangle v\langle/f\rangle^i \mid i \in [n], v \in L(S_i)\}$$

Finally, the target language consists of all strings in which JULIET let ROMEO choose a candidate string v and successfully showed that $v \notin L(S_i)$ for some i:

$$T = \{\langle f\rangle^i\langle\text{err}\rangle\langle/\text{err}\rangle\langle/f\rangle^i \mid i \geqslant 0\}$$

From the above considerations, it should be clear that JULIET has a winning strategy on w in G if and only if $L(S_1) \cap \ldots \cap L(S_n) = \varnothing$. Furthermore, it is easy to see that DTDs for R_f and T can be efficiently constructed.

An XML Schema S_f for V_f is also quite easy to construct: Assume without loss of generality that the type alphabets for $S_1, \ldots, S_n$ (as defined in Section 2.2) are pairwise disjoint. We take additional types f_i for each $i \in \{0, \ldots, n\}$ which are all labelled with f, with f_0 being the root type for S_f, and add for each $i \in [n]$ rules $f_{i-1} \to f_i \mid \#r_i$, where r_i is the root type of S_i. Finally, we add rules $f_n \to \epsilon$ and $\# \to \epsilon$. The content models of these additional rules are clearly deterministic, and since no f_i or $\#$ appears in any S_i, the schema thus constructed is single-type. This concludes the proof. □

7.2.2. DTDs with unbounded number of function symbols

Restricting ourselves to DTDs for target, replacement and validation languages slightly improves the complexity of the winning problem for replay-free validation games; however, the problem remains intractable in general.

Theorem 7.4. *For the class of validation games with target, validation and replacement languages specified by DTDs, and strategies without replay,* JWIN *is* PSPACE*-complete.*

We prove the upper and lower bound in two separate propositions.

Proposition 7.5. *For the class of validation games with target, validation and replacement languages specified by DTDs, and strategies without replay,* JWIN *is in* PSPACE.

Proof. We give an alternating polynomial-time algorithm $\mathcal{A}$ that, given a validation game $G = (\Sigma, \Gamma, R, V, T, \text{err})$ and a string w, decides whether JULIET has a winning strategy in the replay-free game according to G on w.

At its very core, $\mathcal{A}$ simply simulates play on w, going through the input string from left to right, guessing existentially JULIET'S choices of *Read* or *Call*, and guessing universally (where applicable) ROMEO'S choices of nested replacement strings. The main obstacle to a "naïve" simulation is once again the fact that ROMEO may in general have infinitely many possible replacement strings, which may accordingly be of an arbitrary length. To address this problem, similar to the effect technique of Chapters 3 to 6 and the proof of Proposition 7.2, $\mathcal{A}$ does not store the concrete replacement strings chosen by ROMEO, but instead aggregates the essential information needed for the rest of the play that they (and the substring that has been read so far) contain.

Fortunately, the information needed for these considerations is rather simple, as target and validation languages are given by DTDs. In this scenario, the only impact of a substring v in $\langle a\rangle v\langle/a\rangle$ is in the schemas violated by v, and if v violates some schema, this violation carries over to all superstrings of $\langle a\rangle v\langle/a\rangle$. Therefore, if in some string of the form $\langle a\rangle u_1\langle f\rangle u_2\langle/f\rangle u_3\langle/a\rangle$ (with $u_1, u_3 \in \mathrm{WF}(\Sigma)$ and $u_2 \in V_f$), ROMEO replaces $\langle f\rangle u_2\langle/f\rangle$ by some string of the form $\langle a_1\rangle v_1\langle/a_1\rangle \cdots \langle a_n\rangle v_n\langle/a_n\rangle$, all the information needed to determine the outcome of the play is

- the set P of (validation or target) schemas violated by (substrings of) $v_1, \ldots, v_n$, and
- the state transitions induced by the flat string $a_1 \cdots a_n$ in all content model DFAs with root a for schemas not in P.

The algorithm $\mathcal{A}$ basically computes recursively in left-to-right order, for each substring $\langle a\rangle v\langle/a\rangle$ of the input string, the set P_v of schemas violated by a rewriting of v according to a play that $\mathcal{A}$ guesses; it then uses this information as well as the label a (or corresponding information for replacement strings if JULIET plays *Call* on $\langle/a\rangle$) to compute the set of schemas violated by rooted superstrings of $\langle a\rangle v\langle/a\rangle$.

To formulate $\mathcal{A}$ more concretely, we first introduce a bit of notation. Assume for the sake of simplicity that $\Gamma = \{1, \ldots, k\}$ for some $k \in \mathbb{N}$, and that for each alphabet symbol $a \in \Sigma$ and each schema index $i \in \Gamma \cup \{T\}$, where T denotes the target schema, the content model for a in schema i is given by a DFA $A_{i,a} = (Q_{i,a}, \Sigma, \delta_{i,a}, F_{i,a}, s_{i,a})$. Without loss of generality, assume further that each of these automata contains a non-accepting sink state $\bot_{i,a} \in Q_{i,a}$.

The recursive alternating algorithm $\mathcal{A}$ takes as an input a rooted nested string $\langle a\rangle v\langle/a\rangle$, guesses in an alternating fashion a rewriting v' of v according to some play on v according to G and outputs a set $P_v \subseteq \{1, \ldots, k\} \cup \{T\}$ such that exactly the schemas with indices in P_v are violated on v' with an a-labelled root. To that end, it simulates the automata $A_{1,a}, \ldots, A_{k,a}$ and $A_{T,a}$ on the root strings of the rewriting it guesses for v, tracking a tuple $(q_1, \ldots, q_k, q_T) \in Q_{1,a} \times \ldots \times Q_{k,a} \times Q_{T,a}$, which is initialised as $(s_{1,a}, \ldots, s_{k,a}, s_{T,a})$. For as long as v has not been completely processed, $\mathcal{A}$ takes the next rooted prefix $\langle b\rangle u\langle/b\rangle$ of the unprocessed suffix of v and determines, by a recursive call of $\mathcal{A}$ on $\langle b\rangle u\langle/b\rangle$, the set $P_u \subseteq \{1, \ldots, k\} \cup \{T\}$ of schemas violated by rewriting of u. It then updates the simulation of content model DFAs for a as follows:

1. If $b \in \Sigma \backslash \Gamma$, then all content model DFAs with indices not in P_u are simply updated according to b, while all DFAs with indices in P_u are forced into their non-accepting sink state, carrying over the schema violation from the rewriting of u to its proper superstring. That is, $\mathcal{A}$ sets q_i to $\bot_{i,a}$ if $i \in P_u$, and to $\delta_{i,a}(q_i, b)$ otherwise.

2. If $b = j$ for some $j \in \Gamma$, then $\mathcal{A}$ guesses existentially whether Juliet plays *Read* or *Call* on $\langle /j \rangle$;
   (a) if Juliet plays *Read*, then $\mathcal{A}$ updates the state tuple as stated in 1. above;
   (b) if Juliet plays *Call* and $j \in P_u$ (i.e. Juliet's *Call* move is invalid and returns $\langle \text{err} \rangle \langle /\text{err} \rangle$), then each state q_i is set to $\delta(q_i, \text{err})$, i.e. $\mathcal{A}$ simulates a transition with err for all DFAs;
   (c) if Juliet plays *Call* and $j \notin P_u$ (i.e. Juliet's *Call* is valid and Romeo may return some replacement string $u' \in R_j$), then $\mathcal{A}$ guesses universally a set $P'_u \subseteq \{1, \ldots, k\} \cup \{T\}$ of schemas violated by u' and follow-up states q'_i for each $i \notin P'_u$, verifies (using an alternating polynomial-time subalgorithm $\mathcal{B}$ detailed below) whether there exists a string $u' \in R_j$ that violates exactly the schemas from P'_u and whose root label string induces the transitions from q_i to q'_i for each $i \notin P'_u$ and, if this verification turns out positive, updates the current state tuple accordingly (i.e. q_i is set to $\bot_{i,a}$ if $i \in P'_u$ and to q'_i otherwise).

Finally, once all of v has been processed in this way, $\mathcal{A}$ returns the set P_v based upon the final state tuple $(q_1, \ldots, q_k, q_T)$ as $P_v = \{i \in \{1, \ldots, k\} \cup \{T\} \mid q_i \notin F_{i,a}\}$.

Assuming the correctness and alternating polynomial-time complexity of $\mathcal{B}$, it is easy to see that $\mathcal{A}$ is indeed correct (as it simply simulates gameplay on its input string by means of alternation) and runs in alternating polynomial time (as $\mathcal{A}$ is invoked exactly once for each closing tag of the input string and each such invocation adds at most an alternating polynomial time computation). Applying $\mathcal{A}$ to a rooted input string $\langle r \rangle w \langle /r \rangle$ and accepting in an alternating fashion if and only if $T \notin P_w$ also directly yields an algorithm deciding JWin. Therefore, the only remaining part of this proof is to describe the sub-algorithm $\mathcal{B}$ and to show its correctness and alternating polynomial-time complexity.

As input, $\mathcal{B}$ receives a replacement language schema R, validation schemas $V_1, \ldots, V_k$, a target schema T, an index set $P_u \subseteq \{1, \ldots, k\} \cup \{T\}$, a root label a and pairs of states $(q_{i,a}, q'_{i,a}) \in Q_{i,a} \times Q_{i,a}$ for every $i \notin P_u$. Its task is to determine whether there is a nested string $u \in R$ which violates exactly the schemas with indices from P_u rooted at a and whose root label string induces a transition from $q_{i,a}$ to $q'_{i,a}$ in $A_{i,a}$ for each $i \notin P_u$.

First, we argue that such a string, if it exists, can be chosen to have a width of at most $(k+1) \cdot |\Sigma| \cdot q_{\max}^{k+2}$ (where $q_{\max}$ is the maximum size of any content model DFA for any of the schemas $R, T, V_1, \ldots, V_k$) and a depth of at most $(k+2)|\Sigma|$. This is due to standard pumping arguments similar to (but somewhat easier than) those in Lemma 5.17:

- Assume that there is a string u fulfilling the above requirements with a width of more than $(k+1) \cdot |\Sigma| \cdot q_{\max}^{k+2}$, i.e. some node in the tree representation of u has a child string $v \in \Sigma^*$ of length more than $(k+1) \cdot |\Sigma| \cdot q_{\max}^{k+2}$. Then there are more

than $(k+1)$ (i.e. at least $(k+2)$) positions in v labelled with the same alphabet symbol at which all $(k+2)$ relevant content model DFAs are in the same states. If we label each of the $(k+1)$ substrings starting with one such position and ending before the next with the indices of schemas from P_u that are violated somewhere inside that substring in u, we can still cover all $|P_u| \leqslant k+1$ indices in P_u with a string shorter than v by cutting out at least one of the $(k+1)$ substrings.

- Assume now that there is a string u fulfilling the above requirements with a depth greater than $(k+2)|\Sigma|$. Then there is a path from the root to a leaf of (the tree representation of) u with at least $(k+2)|\Sigma|$ nodes. If we annotate each node v in this path with the set of schemas from $\{1, \ldots, k, T\}$ violated in the subtree rooted at v, then there are at least two nodes v_1, v_2 (with v_2 properly contained in the subtree below v_1) in this path that are labelled with the same alphabet symbol from Σ and have the same annotation. This is because the sequence of annotations is monotone non-increasing with regard to the subset relation from root to leaf (if a tree violates some schema, then so do all trees containing it as a subtree). Therefore, we can just replace the subtree rooted at v_1 by the one rooted at v_2 and obtain a string of lower depth.

The algorithm $\mathcal{B}$ uses nondeterministic polynomial space to verify the existence of a string u with the above properties. Intuitively, $\mathcal{B}$ guesses in a streaming fashion the root string of u and simulates all appropriate content model DFAs on the string it guesses. For each alphabet symbol c that $\mathcal{B}$ guesses, it also guesses a subset of the schemas in P_u to be violated in the subtree nested below that c and then recursively verifies that there is an appropriate substring nested below c of appropriate length and smaller depth. If, at the end of this procedure (i.e. after at most $(k+1) \cdot |\Sigma| \cdot q_{\max}^{k+2}$ alphabet symbols have been guessed), all content model DFAs end up in the desired states and all schemas from P_u have been violated (either in substrings or in the root string of u), $\mathcal{B}$ accepts.

The correctness of $\mathcal{B}$ can be proven by a relatively simple (if tedious) nested induction over the depth and width of suitable strings. Since the recursion depth of $\mathcal{B}$ has a polynomial bound, and since each instance of $\mathcal{B}$ only tracks a $(k+2)$-tuple of states as well as a length counter of polynomial length and guesses only witnesses of polynomial length, $\mathcal{B}$ is a nondeterministic polynomial-space algorithm. As NPSPACE = PSPACE = APTIME, this means that $\mathcal{B}$ can be simulated in alternating polynomial time, as was to be proven. □

Proposition 7.6. *For the class of validation games with target, validation and replacement languages specified by DTDs, and strategies without replay,* JWin *is* PSPACE*-hard.*

Proof. The proof is by reduction from the intersection emptiness problem for DTDs: Given DTDs $S_1, \ldots, S_n$ over an alphabet Σ, is $L(S_1) \cap \ldots L(S_n) = \varnothing$? This problem is known to be complete for PSPACE [MNS09].

At its core, the proof is just a slight reworking of the proof of Proposition 7.3. In that proof, we already constructed a game whose target and replacement languages

were DTDs, which however needed the added flexibility of XML Schema to make up for the fact that we had only one function symbol available (and account for the fact that we reduced from the intersection emptiness problem for XML Schema). The basic construction of this proof will therefore be the same as the one for Proposition 7.3, with separate function symbols f_i and corresponding validation languages for each S_i ($i \in [n]$) instead of types for a common XML Schema.

The input nested word for the game G constructed by the reduction is the word $w = \langle f_1\rangle \cdots \langle f_n\rangle\langle f_{n+1}\rangle\langle /f_{n+1}\rangle \cdots \langle /f_1\rangle$, with $f_1, \ldots, f_{n+1} \notin \Sigma$ being the function symbols of G. Again, in a winning strategy, JULIET should play *Call* on $\langle /f_{n+1}\rangle$ to have ROMEO replace it by some string $\langle\#\rangle\langle/\#\rangle v$, and then play *Call* on some f_i with $v \notin L(S_i)$ (if such an i exists). The validation languages for each f_i should basically test the string returned by ROMEO for inclusion in $L(S_i)$, so JULIET should win if that *Call* to f_i returns an error. As in the proof of Proposition 7.3, we have to construct G in a way that punishes the following ways in which JULIET might try to get an an $\langle$err$\rangle\langle/$err$\rangle$ return by cheating:

- JULIET might initially *Call* some f_i with $i \neq n+1$. To that end, each validation language for f_i has to contain the string $\langle f_i\rangle \cdots \langle f_{n+1}\rangle\langle /f_{n+1}\rangle \cdots \langle /f_i\rangle$, and ROMEO has to return some string as a reply to such a *Call* move. To indicate the fact that JULIET has tried to cheat in this fashion, the replacement languages for each f_i with $i \neq n+1$ contain only the string $\langle\top\rangle\langle/\top\rangle$ for a symbol $\top \notin \Sigma$.
- After playing a valid *Call* on f_{n+1} and receiving some string as a reply from ROMEO, JULIET might try to play *Call* on some f_j with $j < n+1$, even though the string chosen by ROMEO is valid for S_j. In that case, ROMEO should also return $\langle\top\rangle\langle/\top\rangle$.
- Once one of her *Call* moves to some f_j has returned $\langle\top\rangle\langle/\top\rangle$, JULIET might attempt another *Call* on some f_i with $i < j$. This should not allow JULIET to provoke an error, either, so each validation language for f_i should contain all strings of the shape $\langle f_i\rangle \cdots \langle f_j\rangle\langle\top\rangle\langle/\top\rangle\langle/f_j\rangle \cdots \langle/f_i\rangle$.

This yields the replacement languages

- $R_{f_{n+1}} = \{\langle\#\rangle\langle/\#\rangle v \mid v \in \mathrm{WF}(\Sigma)\}$, and
- $R_{f_i} = \{\langle\top\rangle\langle/\top\rangle\}$, for each $i \in [n]$.

Similar to the proof of Proposition 7.3, the target language contains all strings of the form $\langle f_1\rangle \cdots \langle f_i\rangle\langle$err$\rangle\langle/err\rangle\langle/f_i\rangle \cdots \langle/f_1\rangle$ for all $i \in \{0, \ldots, n-1\}$.

The main difference to the proof of Proposition 7.3 is the fact that there is a validation language V_{f_i} for *each* $i \in [n+1]$ and that these validation languages are all different. Their construction follows from the above considerations regarding attempts by JULIET to cheat. The validation languages are defined as follows:

$$\begin{aligned} V_{f_i} \;=\; & \{\langle f_i\rangle \cdots \langle f_n\rangle\langle\#\rangle\langle/\#\rangle v\langle/f_n\rangle \cdots \langle/f_i\rangle \mid v \in L(S_i))\} \\ & \cup\{\langle f_i\rangle \cdots \langle f_{n+1}\rangle\langle/f_{n+1}\rangle \cdots \langle/f_i\rangle\} \\ & \cup\{\langle f_i\rangle \cdots \langle f_j\rangle\langle\top\rangle\langle/\top\rangle\langle/f_j\rangle \cdots \langle/f_i\rangle \mid i < j < n+1\} \end{aligned}$$

for each $i \in [n]$, and

$$V_{f_{n+1}} \quad = \quad \{\langle f_{n+1}\rangle\langle/f_{n+1}\rangle\}.$$

A somewhat more subtle difference is that we now actually need a separate DTD for each validation language V_{f_i}. These DTDs, constructed below, share several rules, but have to be kept separate, due to the fact that their rules for f_n depend on the root label f_i. For an arbitrary $i \in [n]$, the rules of a DTD S_{f_i} for V_{f_i} consist of all the rules P_i of S_i as well as the following additional productions:

$$\begin{aligned} f_j &\rightarrow f_{j+1} \mid \top \text{ for each } i \leqslant j \leqslant n-1 \\ f_n &\rightarrow f_{n+1} \mid \#r_i, \\ f_{n+1} &\rightarrow \epsilon, \\ \# &\rightarrow \epsilon \text{ and} \\ \top &\rightarrow \epsilon, \end{aligned}$$

where r_i denotes the root symbol of S_i.

It is easy to verify that the DTDs thus given indeed describe the verification languages defined above and that Juliet has a winning strategy in G on w if and only if $L(S_1) \cap \ldots \cap L(S_n)$ is empty. □

7.2.3. DTDs with bounded number of function symbols

While the winning problem for Juliet in validation games without replay is generally intractable even under the restriction that target, validation and replacement languages are given as DTDs, it is still possible to prove tractability if games are further restricted to only contain a fixed number of function symbols.

Theorem 7.7. *For every fixed $d \geqslant 1$,* JWin *is* PTIME*-complete for the class of validation games with at most d validation DTDs, target languages specified by DTDs, and strategies without replay.*

We prove the upper and lower bound separately.

Proposition 7.8. *For every $d \geqslant 1$,* JWin *is in* PTIME *for the class of validation games with at most d validation DTDs, target languages specified by DTDs, and strategies without replay.*

Proof. Without loss of generality, assume that all content models of DTDs are given by DFAs. As content models are normally given by deterministic regular expressions, these DFAs can be computed efficiently.

The basic proof idea for this result follows a similar approach to that used in Section 4 of [MAABN05]: Traversing the input string (interpreted as a tree) in a bottom-up fashion, the algorithm checks, for each node's child string, whether it (and the subtrees below

it) can be rewritten in a replay-free manner to fit the target and validation languages. This allows us to tell whether Juliet is able to safely play *Read* or *Call* on the node whose child string was just examined, and possibly on ancestor nodes as well. In this manner, deciding $\mathrm{JWin}(G)$ basically boils down to performing a polynomial number of safe rewritability tests for replay-free games on flat strings, which are each feasible in polynomial time by Theorem 3.5.

For the sake of simple presentation, we identify trees and their nested word linearisations throughout this proof. As a slight abuse of notation, we consider a tree t *valid* for a DTD D if the child string of each node v in t respects the content model of the label of v in D. Differing from the standard notion of validity, it is *not* required that the root of t is labelled by the unique root symbol of D.

As described above, our goal is to replace subtrees by leaves bit by bit in a bottom-up manner, and to consider only flat strings of leaf node labels in each step. More precisely, each removal step replaces a subtree of depth one, that is, a node v whose children are all leaves, by a single node with a label that contains all relevant information about its (former) subtree with respect to the game.

In particular, if the subtree below a node v with function symbol f cannot be rewritten to conform to the corresponding part of some DTD V_f, this information will be encoded into the label of v and Juliet will always receive $\langle\mathrm{err}\rangle\langle/\mathrm{err}\rangle$ as a reply to a *Call* on v or any of its ancestors labelled with function symbol f, no matter her rewriting capabilities on other parts of the input tree.

Let t be the tree representing some well-nested rooted[4] word w. By $\mathrm{label}(v)$ we denote the label of a node v. By S we denote the set $\{T, V_1, \ldots, V_d\}$ of schemas of the game. The *profile* $P(t') \subseteq S$ of a tree t' is the set of schemas for which t' is valid.

We first consider subgames on subtrees t_v rooted at some node v with some label a. With each replay-free strategy σ on t_v that does *not* play *Call* on v itself, we associate the *profile set* $\mathfrak{P}_\sigma(t_v)$ of profiles P, for which Romeo has a counterstrategy yielding a tree t' with $P = P(t')$. In the rest of this proof, we denote the set of such strategies for Juliet as $\mathrm{STRAT}_{\mathrm{J},\mathit{Read}}$. The *dossier* $\mathfrak{D}(v)$ of v is the minimised set of all profile sets $\mathfrak{P}_\sigma(t_v)$ for strategies $\sigma \in \mathrm{STRAT}_{\mathrm{J},\mathit{Read}}$ of this kind, i.e.

$$\mathfrak{D}(v) = \left[\{\mathfrak{P}_\sigma(t_v) \mid \sigma \in \mathrm{STRAT}_{\mathrm{J},\mathit{Read}}\}\right]_{\min}.$$

Recall that the $[\cdot]_{\min}$ operator removes from a set of sets all sets that are not inclusion-minimal.

It is crucial for this proof that, since the number $|S|$ of relevant schemas is constant, the number of all profiles, profile sets and dossiers are bounded by constants, as well. Therefore, an iteration over all profile sets is possible in polynomial time.

The bottom-up computation mentioned above successively replaces the subtree below a node v with label a by a leaf with label $(a, \mathfrak{D}(v))$. Once this process reaches the root $\mathrm{root}(t)$ of the tree, it can be instantly decided whether Juliet has a wining strategy on w. Indeed, this is the case if and only if $\mathfrak{D}(\mathrm{root}(t))$ contains some profile set $\mathfrak{P}$, each profile of which contains the target schema T. This approach is somewhat related to the

[4]For simplicity, we do not consider non-rooted words in this proof. They can be handled similarly.

idea behind effects (used in previous chapters) – summarising the subgame in a subtree t_v (excluding its root) by a single-round game in which Juliet first selects some profile set $\mathfrak{P}_\sigma(t_v) \in \mathfrak{D}(v)$, then Romeo selects a profile $P \in \mathfrak{P}_\sigma(t_v)$.

Note that, even though this bottom-up rewriting process does not strictly proceed in left-to-right order, it still captures only strategies for Juliet respecting that order. While dossiers of sibling nodes are computed independently of each other, the computation considers (and stores information) for *all* possible strategies of Juliet on the corresponding subtrees, and that information is then aggregated and propagated further up the tree. In effect, the choice of strategy for Juliet only happens once a dossier for the root node has been computed (in deciding whether Juliet has a winning strategy as described above).

We now start with the detailed description of the algorithm.

First, for all leaf nodes, their dossier is computed. As there is no actual subgame on a leaf node v (that does not play *Call* on that node), each such dossier is just $\{P(t_v)\}$. In this case, $P(t_v)$ is just the set of schemas in which the (original) label of v has the empty string in its content model and is therefore allowed at leaf nodes.

The key step that the algorithm performs is to compute the dossier of a node v with children $u_1, \ldots, u_m$ (where $m \geqslant 1$), all of whose dossiers are already given. From these, it computes $\mathfrak{D}(v)$ iteratively by checking, for each potential profile set $\mathfrak{Q}$ (out of $2^{2^{|S|}}$ possible sets of profiles), whether $\mathfrak{Q}$ has a subset in $\mathfrak{D}(v)$, i.e. whether there is a strategy $\sigma \in \mathrm{STRAT}_{\mathrm{J},\mathit{Read}}$ with $\mathfrak{P}_\sigma(t_v) \subseteq \mathfrak{Q}$. In order to obtain a minimised set, the algorithm first checks all profile sets containing only a single profile, then proceeds in order of ascending cardinality. If some subset of the profile set $\mathfrak{Q}$ currently examined is already in $\mathfrak{D}(v)$, $\mathfrak{Q}$ is not considered for inclusion. Otherwise the algorithm checks whether $\mathfrak{Q}$ should be added to $\mathfrak{D}(v)$ by way of a replay-free game $G_\mathfrak{Q}$ on a flat string.

The basic idea behind this game is that its initial string is constructed by transforming each u_i into a string representing the dossier of u_i. Juliet traverses this string from left to right and chooses, for each such dossier $\mathfrak{D}(u_i)$, one profile set $\mathfrak{P}_i \in \mathfrak{D}(u_i)$ corresponding to her sub-strategy in the tree rooted at u_i, to which Romeo replies by picking a profile $P \in \mathfrak{P}_i$. Afterwards, Juliet decides whether to play *Read* or *Call* on u_i. In the latter case, Romeo replies with a flat string derived from the corresponding replacement language and annotated with profiles in a similar manner to the bottom-up subtree-cutting method intuitively described above. At the end of this game, the target automaton then aggregates all the profiles and string labels chosen by Juliet and Romeo to check whether $u_1, \ldots, u_m$ would be rewritten (by the chosen strategies of Juliet and Romeo) into a forest conforming to exactly the schemas of some profile in $\mathfrak{Q}$ when rooted at v. If Juliet has a winning strategy in this game, $\mathfrak{Q}$ is added to $\mathfrak{D}(v)$, otherwise, it is not. As the game $G_\mathfrak{Q}$ will be of polynomial size, this can be checked in polynomial time thanks to Theorem 3.5(c).

Towards the construction of $G_\mathfrak{Q}$, we first examine how this game's replacement languages are constructed. For every function symbol f, the flat replacement language R'_f consists of strings s_F over the alphabet $\Sigma \times \mathscr{P}(S)$ that are obtained from replacement forest F in R_f as follows. The string s_F has one position per tree t of F, which is labelled

by (a, P), where a is the label of the root node of t and P is a the profile of all schemas for which t is valid. A similar replacement language R'_{err} is constructed for the return $\langle\text{err}\rangle\langle/\text{err}\rangle$ of invalid calls. As replacement strings represent strings in which no further *Call* moves are possible, the labels of their positions do not include dossiers but rather the profile of the actual tree that they represent.

Each set R'_f for $f \in \Gamma$ can be computed as follows. Let L_f denote the content model of f in V_f (represented by some DFA A_f). For each symbol a occurring in L_f, let $\Sigma_{f,a}$ be the set of all pairs (a, P), such that there is a tree t' with profile P and root label a that is valid with respect to R_f. For each f, a and P, it can be decided in polynomial time whether $(a, P) \in \Sigma_{f,a}$ by constructing a deterministic tree automaton (cf e.g. [GS84; Com+]) that accepts all trees that are valid with respect to R_f and the schemas in P, and invalid with respect to the schemas in $S\backslash P$. As d is fixed, this amounts to an emptiness test for the polynomial-size product of $d + 1$ deterministic tree automata. It follows that all sets $\Sigma_{f,a}$ can be computed in time polynomial in the size of G.

The set R'_f consists of all strings over $\bigcup_{a\in\Sigma} \Sigma_{f,a}$ whose Σ-projection is in L_f. Given the sets $\Sigma_{f,a}$, a DFA for R'_f can be easily (and efficiently) computed. The set R'_{err} simply consists of the single one-character string $(\text{err}, P_{\text{err}})$, where P_{err} is the set of all schemas in which err occurs as a valid symbol for a leaf node. This can easily be computed in polynomial time as well.

Now, with the schemas R'_f at hand, we describe the computation of $\mathfrak{D}(v)$ from $u_1, \ldots, u_m$ and their dossiers in more detail.

For a dossier $\mathfrak{D} = \{\mathfrak{P}_1, \ldots, \mathfrak{P}_\ell\}$ (assuming some arbitrary but canonical ordering on profile sets) and a symbol a, let $s(a, \mathfrak{D})$ denote the string $(a, \mathfrak{D})(a, \mathfrak{P}_1) \cdots (a, \mathfrak{P}_\ell) \cdot \$_a \cdot \#_a$.

The idea behind the construction of the flat game is as follows.

The original game on a tree t_z with root label g (where z is a child of the current root node v) can be viewed as follows: Juliet chooses a strategy for the first phase of the game before the closing tag $\langle/g\rangle$ of z is reached. This strategy corresponds to some profile set $\mathfrak{P}_i \in \mathfrak{D}(z)$. By choosing a counterstrategy for this subgame, Romeo basically picks a profile $P \in \mathfrak{P}_i$. Then Juliet decides whether she plays *Call* at $\langle/g\rangle$ (subject to validity with respect to V_g) and Romeo replaces z, in case she plays *Call*.

In the flat game on $(g, \mathfrak{D})(g, \mathfrak{P}_1) \cdots (g, \mathfrak{P}_\ell)\$_a\#_g$ this is mimicked as follows: Juliet chooses her strategy by playing *Call* at $(g, \mathfrak{P}_i)$. Romeo replaces $(g, \mathfrak{P}_i)$ by some pair (g, P) with $P \in \mathfrak{P}_i$. So far the game exactly mimics the original game *before* reaching $\langle/g\rangle$. If P allows Juliet to play a valid *Call* at $\langle/g\rangle$ (that is, if $V_g \in P$), she can call the follow-up symbol $\#_g$ which is then replaced by Romeo with a string from R'_g; if $V_g \notin P$, Romeo replaces $\#_g$ by the only string $(\text{err}, P_{\text{err}})$ in V_{err}. If both players play like this on the string $s(\text{label}(u_1), \mathfrak{D}(u_1)), \ldots, s(\text{label}(u_m), \mathfrak{D}(u_m)))$ (i.e. if no one tries to cheat), the target language DFA for $G_{\mathfrak{Q}}$ can then aggregate all string labels and profiles chosen by Juliet and Romeo to check whether the resulting forest in the original game G would be valid with respect to exactly the schemas in some profile $P_f \in \mathfrak{Q}$.

To this end, $G_{\mathfrak{Q}}$ has replacement languages

- $R''_{\#_a} = R'_a \cup R'_{\text{err}}$, for every symbol $\#_a$, and
- $R''_{(a,\mathfrak{P})} = \{(a, P_1), \ldots, (a, P_j)\}$, for each set $\mathfrak{P} = \{(a, P_1), \ldots, (a, P_j)\}$.

We will now first examine how the target language DFA $A_{\mathfrak{Q}}$ for $G_{\mathfrak{Q}}$ aggregates alphabet symbols and profiles. We then look at how to prevent either player from cheating and how $A_{\mathfrak{Q}}$ has to be adapted to reflect this.

The DFA $A_{\mathfrak{Q}}$ primarily tracks in its state the following two informations (along with some auxiliary information to determine which input symbols are relevant and for cheating prevention, as described later on):

- A profile P_A of schemas that are so far fulfilled by "hidden" subtrees (initialised to $P_A = S$), and
- one state of the DFA for the content model of label(v), for each of the schemas from S.

As sketched above, $A_{\mathfrak{Q}}$ aggregates the information from a play on the forest $u_1, \ldots, u_m$ simulated by Juliet and Romeo on $s_1, \ldots, s_m$, where $s_i = s(\text{label}(u_i), \mathfrak{D}(u_i))$, for each i. To that end, we call a pair (a, P) consisting of an alphabet symbol $a \in \Sigma \cup \{\text{err}\}$ and a profile $P \subseteq S$ *relevant* if one of the following conditions holds:

- (a, P) occurs between a pair of the form $(a, \mathfrak{D})$ (with a dossier $\mathfrak{D}$) and a substring $\$_a \#_a$, and all other pairs in this interval are of the form $(a, \mathfrak{P})$ for profile sets $\mathfrak{P}$ (i.e. (a, P) represents the root label and subgame information for a subtree that Juliet played *Read* on); or
- (a, P) occurs between $\$_a$ and a pair of the form $(b, \mathfrak{D})$ (with alphabet symbol b and dossier $\mathfrak{D}$), and all other pairs in this interval are also of the form (a', P') for alphabet symbols a' and profiles P' (i.e. (a, P) occurs in some response by Romeo to one of Juliet's calls on some $\#_g$).

Note that in the latter case, all pairs between $\$_a$ and $(b, \mathfrak{D})$ are relevant.

It is easy (if technical) to construct $A_{\mathfrak{Q}}$ in such a way that it only aggregates relevant pairs and ignores all non-relevant symbols (except for the purpose of determining relevancy and for cheating prevention as described below).

The aggregation itself proceeds in a straightforward way: On reading a relevant pair (a, P), $A_{\mathfrak{Q}}$ simulates a step of all content model DFAs with input symbol a and updates their states accordingly; furthermore, $A_{\mathfrak{Q}}$ sets the new value of P_A to be $P_A \cap P$. The DFA $A_{\mathfrak{Q}}$ accepts an input string if there is some profile $P_f \in \mathfrak{Q}$ such that its final value of P_A equals P_f and exactly the content model DFAs for schemas in P_f are in an accepting state. Recall that $|S|$ is constant, and therefore $A_{\mathfrak{Q}}$ can be constructed to be of polynomial size in $|G|$.

Assuming that no player cheats (i.e. play proceeds in the manner sketched above), it is again relatively easy to see but tedious to prove that Juliet has a winning strategy in the game on flat strings with target automaton $A_{\mathfrak{Q}}$ and replacement languages and input string described above if and only if there exists a strategy σ for Juliet in the original validation game such that $\mathfrak{P}_\sigma(t_v) \subseteq \mathfrak{Q}$.

We finally describe how to prevent either player from cheating. The following possibilities of deviating from the play sequence sketched above exist:

- In some substring $s(\text{label}(u_i), \mathfrak{D}(u_i))$, JULIET might play *Call* on either none or on more than one pair of the form $(\text{label}(u_i), \mathfrak{P})$. In this case, there is not exactly one pair of the form $(\text{label}(u_i), P)$ occurring between $(\text{label}(u_i), \mathfrak{D}(u_i))$ and $\$_{\text{label}(u_i)}$.

- ROMEO might answer some *Call* by JULIET on some $\#_g$ with a string from R'_g although $V_g \notin P$ for the corresponding relevant P (or with $(\text{err}, P_{\text{err}})$ although $V_g \in P$).

Both of these cases can easily be recognised by regular conditions, which can be added into $A_{\mathfrak{Q}}$ with at most polynomial blow-up to ensure that any player who cheats loses the game immediately.

This completes the construction of the flat game used to check, for some profile set $\mathfrak{Q}$, whether $\mathfrak{Q} \in \mathfrak{D}(v)$. With this, each of the bottom-up reduction steps amounts to a (large but) constant number of tests whether JULIET has a winning strategy in a flat game without replay and therefore can be done in overall polynomial time.

It is not too difficult but tedious to verify that the algorithm as a whole is also correct. □

Proposition 7.9. *For the class of validation games with one validation DTD, target languages specified by DTDs, and strategies without replay,* JWIN *is* PTIME*-hard (under logspace reductions).*

Proof. The proof is by reduction from the emptiness problem for DTDs: Given a DTD D, is $L(D) = \varnothing$? This problem is complete for PTIME, as can be shown by a trivial reduction from the emptiness problem for context-free grammars [JL76].

Let D be a DTD over alphabet Σ with root symbol S. We construct from D a game G with a single function symbol $f \notin \Sigma$ such that JULIET has a winning strategy on $\langle f\rangle\langle/f\rangle$ in G if and only if $L(D) = \varnothing$ holds. Basically, the game G will have JULIET play *Call* on f, to have ROMEO replace $\langle f\rangle\langle/f\rangle$ by some nested word from $L(D)$, with JULIET winning if and only if ROMEO is unable to produce such a string. The only minor technical difficulty stems from the fact that replacement languages have to be nonempty.

To that end, we extend D to a DTD D' with root symbol S' such that $L(D') = \{\langle S'\rangle v\langle/S'\rangle \mid v \in L(D)\} \cup \{\langle S'\rangle\langle\#\rangle\langle/\#\rangle\langle/S'\rangle\}$ for some $\# \notin \Sigma$, i.e. $L(D)$ is empty if and only if $L(D')$ contains only the single word $\langle S'\rangle\langle\#\rangle\langle/\#\rangle\langle/S'\rangle$. This simply amounts to adding the rules $S' \to S \mid \#$ and $\# \to \epsilon$ to D.

We let $G = (\Sigma \cup \{f, \#\}, \{f\}, R, V, T)$ with the replacement language for f given by the DTD D' and the validation language for f consisting only of $\langle f\rangle\langle/f\rangle$, i.e. $R_f = L(D')$ and $V_f = \{\langle f\rangle\langle/f\rangle\}$. The target language is $T = \{\langle S'\rangle\langle\#\rangle\langle/\#\rangle\langle/S'\rangle\}$.

Clearly, G can be constructed from D in logarithmic space. It is also quite easy to see that JULIET has a winning strategy (consisting of playing *Call* on f) on $\langle f\rangle\langle/f\rangle$ in G if and only if ROMEO has no choice but to respond to that initial *Call* with $\langle S'\rangle\langle\#\rangle\langle/\#\rangle\langle/S'\rangle$. By construction of R_f, this is the case if and only if $L(D) = \varnothing$. □

7.3. Outlook and bibliographical remarks

This chapter presented a complete characterisation of the complexity of solving context-free validation games without replay whose target languages are given by deterministic nested word automata or by practical schema languages for XML. Further investigation of the impact that adding replay has on the complexity of the winning problem might be of interest, but considering the results of this chapter, complexities are likely to get quite high.

Another open question concerns the gap between the complexities of the algorithms witnessing the upper bounds in Propositions 7.5 and 7.8; the former gives a PSPACE upper bound for validation games with an unbounded number of function symbols, while the latter has a time bound that is polynomial for a constant number of function symbols but rises in a doubly exponential fashion in the number of function symbols, and therefore requires strictly more than polynomial space. The interesting open question, then, is whether some algorithm can be found that combines both upper bounds, requiring only polynomial space in general and additionally running in polynomial time for a fixed number of function symbols.

It should be noted that the semantics given here for validation games is only one of several possible options. A different version, called *error-rejecting semantics* (as opposed to the *error-marking* semantics detailed in this chapter), where Juliet is not allowed to play *Call* on any function node with an invalid parameter subtree (as opposed to receiving an error as a call result) is presented in [SS15] and yields the same upper bounds as the ones given here. The lower bound proofs given here, however, do not carry over to error-rejecting semantics, and no corresponding lower bounds have so far been proven using error-rejecting semantics. The main difficulty in proving lower bounds in error-rejecting semantics, intuitively, stems from the fact that, unlike in error-marking semantics, it is impossible to tell whether, on some invalid subtree, Juliet *decided* to play *Read* or *wanted* to play *Call* but couldn't. The author conjectures, however, that upper and lower bounds actually coincide in both semantics.

Bibliographical remarks. All results presented in this chapter were originally published in [SS15] (and that paper's journal version [SS16]) as joint work with Thomas Schwentick. The polynomial-time algorithm of Proposition 7.8 follows the basic idea presented for Active XML schema rewriting in [MAABN05].

8. Nested word transducers

In previous chapters, we explored context-free games where the input of function calls (i.e. the substructures replaced by a *Call* move) had no, or only very limited impact on the choice of *Call* results available to Romeo. In fact, so far, only the validation games of Chapter 7 took parameter substrings into account, and only in a limited fashion that is still very close in nature to context-free games with regular replacement as examined in Chapter 6.

Taking a look back at the practical motivation of Active XML schema rewriting from Section 1.2, one can safely assume that the dependency of the output of function calls on their input goes beyond the purely syntactical validation modelled by validation games. For a first attempt at modelling more complex semantic dependencies between the input and output of function calls, the next chapter will examine context-free games where the replacement relation is given by a *transducer*; in preparation for examining these games, this chapter introduces the transducer models used there.

Much like automata usually offer an efficient finite representation of (potentially infinite) languages, transducers are used to finitely represent transformations on input structures (in our case, nested words). To that end, the most intuitive way to define transducers is to take an automaton model (here, nested word automata) and add an output string to each transition, with the semantics being that, once a transition is taken, its output component is appended to the output string; once the transducer has completely read its input, it then produces its output string. To allow for limiting a transducer's domain, outputs are only produced on runs that reach an accepting state. Since we are interested in transducers that describe a relation on nested words (instead of a function), transducers will generally be nondeterministic.[1]

In using transducers as a replacement mechanism for context-free games, there are several desirable properties for the transducer model to fulfil:

(1) Since our model of context-free games only considers well-nested words, outputs of transducers on well-nested input words should generally be well-nested; additionally, as an extension of context-free games with regular replacement, the image of each input word (i.e. the set of possible outputs on any given input) should be a regular language of nested words.

(2) Since, in the practical application, input and output schemas of function calls are given by (subclasses of) regular languages, regular languages should be closed under transduction, i.e. the image of a regular language after transduction should again be regular.

[1] The special case of *functional* transducers, whose output relation is a function while still potentially requiring nondeterminism, will be discussed later in this chapter (cf. p. 158).

(3) To allow for iterated function calls, it seems sensible to desire that the transducer model should be closed under composition, i.e. several iterated transductions should be expressible as a single transduction without requiring a more powerful model.

(4) In order to simulate context-free games with regular replacement, it should be possible to obtain arbitrary regular languages as images of transduction; and since input strings are generally finite, this requires the transducer model to be able to insert strings of unbounded length.

As we will see, the *non-deleting nested word transducers* introduced in this chapter have all four of these properties. Our initial definition of nested word transducers will be somewhat more general, allowing for deletion of input symbols, at the cost of losing property (2).

In order to obtain property (4), nested word transducers allow for *ϵ-transitions*, which produce an output without reading an input. Transitions of this kind were already researched in the context of finite state automata and finite state transducers (see e.g. [HMU01; Yu97]), where they do not change the expressive power of the underlying model. As a technical tool for analysis, which may be of independent interest, we first examine *nested word automata with ϵ-transitions* in Section 8.1, showing that these automata have the same expressiveness as well as closure and complexity properties as nested word automata. In Section 8.2, we then define nested word transducers (and various restrictions thereof) and examine their closure and complexity properties as well.

8.1. Nested word automata with ϵ-transitions

It is well known (see, for instance, [HMU01]) that extending finite-state automata with ϵ-transitions does not change their expressive power; (nondeterministic) finite-state automata with ϵ-transitions still decide exactly the class of regular languages of flat strings.

Even though nested word automata and the class of regular nested word languages strongly parallel finite-state automata and flat regular languages, a similar investigation has so far not been performed for nested word automata. We now define and examine nested word automata with ϵ-transitions, mainly as a tool for the analysis of nested word transducers with ϵ-transitions in Section 8.2.

Definition 8.1. A *Nested Word Automaton with ϵ-transitions (ϵ-NWA)* is a tuple $A = (Q, P, \Sigma, \delta, q_0, F)$ consisting of

- a set Q of *linear states*,
- a set P of *hierarchical states*,
- an alphabet Σ,
- a *transition relation* $\delta \subseteq (Q \times \langle\Sigma\rangle \times Q \times P) \cup (Q \times P \times \langle/\Sigma\rangle \times Q) \cup (Q \times \{\epsilon\} \times Q)$,
- an *initial state* $q_0 \in Q$, and

- a set of *accepting states* $F \subseteq Q$.

As for standard NWA, we also write $(q', p) \in \delta(q, \langle a \rangle)$ (resp. $q' \in \delta(q, p, \langle /a \rangle)$, $q' \in \delta(q, \epsilon)$) instead of $(q, \langle a \rangle, q', p) \in \delta$ (resp. $(q, p, \langle /a \rangle, q'), (q, \epsilon, q') \in \delta$).

Note that ϵ-transitions are always *internal* transitions that merely change the current linear state, not the hierarchical stack; allowing ϵ-transitions to manipulate the stack as well would yield a strictly more expressive automaton model. For instance, with stack-manipulating ϵ-transitions pushing some extra state on each read $\langle /a \rangle$ and later popping such states before being allowed to read $\langle b \rangle$, it would be easy to construct an automaton deciding the non-regular language $\{(\langle a \rangle \langle /a \rangle)^n (\langle b \rangle \langle /b \rangle)^n \mid n \geqslant 0\}$.

The semantics of ϵ-NWA is defined almost exactly like that of NWA, by way of configurations and (accepting) runs. To account for ϵ-transitions, however, the definition has to be somewhat adapted. The idea behind this adaptation is to virtually decorate the input string into an *internal ϵ-extension* with special symbols $\dot{\epsilon}$.[2] Runs of an ϵ-NWA are then defined on internal ϵ-extensions, with $\dot{\epsilon}$ symbols being read when an ϵ-transition is taken. This semantics definition via ϵ-extensions may seem somewhat unusual when compared to standard semantics definitions of (variants of) pushdown automata (cf. e.g. [HMU01]), but generalises nicely into the semantics definition for nested word *transducers* with ϵ-transitions (Section 8.2).

Definition 8.2. Let $w \in \mathrm{WF}(\Sigma)$, with $\dot{\epsilon} \notin \Sigma$. An *internal ϵ-extension of w* is a string $\hat{w} \in (\langle \Sigma \rangle \cup \langle /\Sigma \rangle \cup \{\dot{\epsilon}\})^*$ obtained by inserting an arbitrary amount of $\dot{\epsilon}$ symbols at arbitrary positions in w.

Like for NWA, a *configuration* κ of an ϵ-NWA A is a tuple $(q, \alpha) \in Q \times P^*$, with a *linear state* q and a sequence α of *hierarchical states*. A *run of A on an internal ϵ-extension* $\hat{w} = \hat{w}_1 \cdots \hat{w}_n$ *of* $w \in WF(\Sigma)$ is a sequence $\rho = \kappa_0, \ldots, \kappa_n$ of configurations $\kappa_i = (q_i, \alpha_i)$ of A such that for each $i \in [n]$ and $a \in \Sigma$ it holds that

- if $\hat{w}_i = \langle a \rangle$, then $(q_i, p) \in \delta(q_{i-1}, \langle a \rangle)$ (for some $p \in P$), and $\alpha_i = p\alpha_{i-1}$, or
- if $\hat{w}_i = \langle /a \rangle$, then $q_i \in \delta(q_{i-1}, p, \langle /a \rangle)$ (for some $p \in P$), and $p\alpha_i = \alpha_{i-1}$, or
- if $\hat{w}_i = \dot{\epsilon}$, then $q_i \in \delta(q_{i-1}, \epsilon)$, and $\alpha_i = \alpha_{i-1}$.

In this case, we also write $\kappa_0 \overset{w}{\rightsquigarrow}_A \kappa_n$. Acceptance of nested words and the language of an ϵ-NWA are defined as for NWA in Definition 5.3.

The following properties of ϵ-NWA follow easily from the proofs for corresponding properties of NWA in [AM09].

Lemma 8.3. *For each ϵ-NWA A, there exists a DNWA A' of size at most exponential in $|A|$ such that $L(A) = L(A')$.*

[2] The term "internal" here serves to stress the fact that ϵ-transitions only affect the linear state, and to distinguish internal ϵ-extensions from the ϵ-extensions used to define the semantics of nested word transducers in Section 8.2.

Proof. The proof of this statement for NWA without ϵ-transitions in Theorem 3.3 of [AM09] uses a modified subset construction, where states of the exponential-sized DNWA A' correspond to sets of pairs of states of A (so-called *summaries*) such that a (not necessarily well-nested) string w induces a (partial) run from the initial state of A' to some summary state $S \in \mathscr{P}(Q \times Q)$ if and only if there are (partial) runs with w from each $q \in Q$ to all $q' \in \{q' \mid (q, q') \in S\}$ in A'.

To account for ϵ-transitions, we modify these summaries to include *ϵ-closures* of target states, i.e. for each pair (q, q') of states contained in a summary S as constructed in [AM09], we add to S all pairs (q, q''), where q'' is reachable from q by a series of ϵ-transitions in A. Otherwise, the construction (and correctness proof) is the same as in [AM09]. □

Lemma 8.4. *For all ϵ-NWA A_1 and A_2, it is possible to construct in polynomial time ϵ-NWA deciding $L(A_1) \cup L(A_2)$ and $L(A_1) \cap L(A_2)$.*

Proof. Let $A_1 = (Q_1, P_1, \Sigma, \delta_1, q_{0,1}, F_1)$ and $A_1 = (Q_2, P_2, \Sigma, \delta_2, q_{0,2}, F_2)$. This proof, like the one for Theorem 3.5 in [AM09], uses a standard product construction simulating A_1 and A_2 simultaneously on the input. The product automaton $A' = (Q_1 \times Q_2, P_1 \times P_2, \Sigma, \delta', (q_{0,1}, q_{0,2}), F')$ is constructed as in [AM09] and simply extended by ϵ-transitions. Note that, unlike reading transitions, ϵ-transitions do *not* have to be synchronised between the two automata, i.e. an ϵ-transition of A' simulates an ϵ-transition of only *one* of the component automata A_1 or A_2. The transition relation δ' of A' is therefore extended by the sets $\{((q_1, q_2), \epsilon, (q_1', q_2)) \mid (q_1, \epsilon, q_1') \in \delta_1, q_2 \in Q_2\}$ and $\{((q_1, q_2), \epsilon, (q_1, q_2')) \mid (q_2, \epsilon, q_2') \in \delta_2, q_1 \in Q_1\}$. □

Theorem 8.5. *(a) The membership and emptiness problem for ϵ-NWA are in* PTIME.

(b) The universality, equivalence and inclusion problem for ϵ-NWA are EXPTIME-*complete.*

(c) Deciding, given an ϵ-NWA A and a DNWA B, whether $L(A) \subseteq L(B)$ is PTIME-*complete with respect to logspace reductions.*

Proof. These complexity properties mostly follow from the corresponding results for NWA without ϵ-transitions. Specifically, as NWA may also be interpreted as ϵ-NWA, all lower bounds carry over directly from Theorem 5.8.

For (a), ϵ-NWA may also be interpreted as pushdown automata, whose membership and emptiness problem are decidable in polynomial time.

The upper bounds for (b) can be proven using Lemmas 8.3 and 8.4 as well as the complement construction for DNWA (Lemma 5.7) and set-theoretic formulations for universality (is $\overline{L(A)} = \varnothing$?), equivalence (is $(\overline{L(A_1)} \cap L(A_2)) \cup (L(A_1) \cap \overline{L(A_2)}) = \varnothing$?) and inclusion (is $L(A_1) \cap \overline{L(A_2)} = \varnothing$?).

As in the proof of Theorem 5.8, the upper bound in (c) also follows from the fact that DNWA can be efficiently complemented and the fact that $L(A) \subseteq L(B)$ holds if and only if $L(A) \cap \overline{L(B)} = \varnothing$. By Lemma 8.4 and part (a) of this theorem, this can be checked in polynomial time. □

8.2. Nested word transducers

In this section, we define nested word transducers and examine their closure properties and complexities of algorithmic problems. Thanks to the definition of nested word transducers putting some rather severe restrictions on the use of ϵ-transitions and the allowed output of transducers, we obtain advantageous closure properties and comparatively low complexities.

Intuitively, a NWT T works much like a NWA with output and additional ϵ-transitions – T reads its input from left to right and decides nondeterministically which available transition to use; on an opening (resp. closing) transition, it reads an opening (closing) input tag, changes its linear state and pushes (pops) a hierarchical state while producing an output. Opening (closing, internal) ϵ-transitions do not consume input symbols but induce state changes and produce outputs. T only produces an output if it accepts the input.

Definition 8.6. A *nested word transducer* (or *NWT*) $T = (Q, P, P_\epsilon, \Sigma, \delta, q_0, F)$ consists of

- a set Q of *linear states*,
- a set P of *hierarchical states*,
- a set $P_\epsilon \subseteq P$ of *hierarchical ϵ-states*,
- an alphabet Σ,
- a *transition relation* δ, which is the union of three relations:
  - *opening* transitions from $Q \times (\langle\Sigma\rangle \cup \{\langle\epsilon\rangle\}) \times Q \times P \times (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$,
  - *internal* transitions from $Q \times \{\epsilon\} \times Q \times \mathrm{WF}(\Sigma)$, and
  - *closing* transitions from $Q \times P \times (\langle/\Sigma\rangle \cup \{\langle/\epsilon\rangle\}) \times Q \times (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$,
- an *initial state* $q_0 \in Q$, and
- a set of *accepting states* $F \subseteq Q$,

such that for all $q, q', r, r' \in Q$, $p \in P$, $a \in \Sigma \cup \{\epsilon\}$ and $u, v \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$ it holds that[3]

- $(q, \langle\epsilon\rangle, q', p, u) \in \delta$ or $(q, p, \langle/\epsilon\rangle, q', u) \in \delta$ if and only if $p \in P_\epsilon$ (*ϵ-consistency*),
- if $(q, \langle a\rangle, q', p, u) \in \delta$ and $(r, p, \langle/a\rangle, r', v) \in \delta$, then $uv \in \mathrm{WF}(\Sigma)$ (*well-formedness*), and
- for each $(q, \langle a\rangle, q', p, u) \in \delta$ (resp. $(r, p, \langle/a\rangle, r', u) \in \delta$) with $u \neq \epsilon$, u contains at least one unmatched opening (resp. closing) tag (*synchronisation*).

[3] These three conditions make NWT roughly correspond to *synchronised visibly pushdown transducers* [RS08]; we mainly require them to ensure closure of regular nested word languages under NWT transduction.

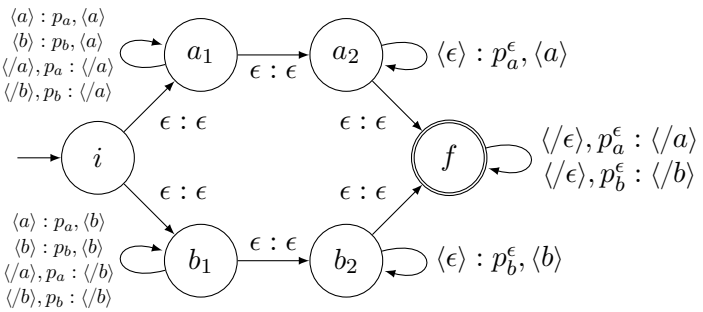


Figure 8.1.: Nested Word Transducer T_{ab} from Example 8.8.

As for standard NWA, we also write $(q', p, u) \in \delta(q, \langle a \rangle)$ (resp. $(q', u) \in \delta(q, p, \langle /a \rangle)$, $(q', u) \in \delta(q, \epsilon)$) instead of $(q, \langle a \rangle, q', p, u) \in \delta$ (resp. $(q, p, \langle /a \rangle, q', u), (q, \epsilon, q', u) \in \delta$).

Like for (ϵ-)NWA, the semantics of NWT is defined by *runs*. Due to the presence of ϵ-transitions and an output in NWT, the precise definition becomes a bit more complicated, using and extending *ϵ-extensions* of nested words, similar to the internal ϵ-extensions introduced for ϵ-NWA in Section 8.1.

Definition 8.7. Let $T = (Q, P, P_\epsilon, \Sigma, \delta, q_0, F)$ be an NWT. A *configuration* $\kappa = (q, \alpha)$ consists of a linear state $q \in Q$ and a stack $\alpha \in P^*$ of hierarchical states.

For any nested word $w \in \mathrm{WF}(\Sigma)$, an *$\epsilon$-extension of w* is a string $\hat{w}$ obtained by inserting symbols $\langle \epsilon \rangle$, $\langle /\epsilon \rangle$ and $\dot{\epsilon}$ into w such that the subsequence of $\hat{w}$ obtained by removing all $\dot{\epsilon}$ from $\hat{w}$ is a well-nested word over $\Sigma \cup \{\epsilon\}$.

A *run* of T on an ϵ-extension $\hat{w} = \hat{w}_1 \dots \hat{w}_n$ of a nested word $w \in \mathrm{WF}(\Sigma)$ starting at configuration $\kappa_0 = (r_0, \alpha_0)$ is a finite sequence $\rho = \kappa_0 \kappa_1 \cdots \kappa_n$ with $\kappa_i = (r_i, \alpha_i)$ for each $i \in [n]$, such that for each $i \in [n]$, one of the following holds:

- $\hat{w}_i = \langle a \rangle$ for some $a \in \Sigma \cup \{\epsilon\}$, $(r_i, p_i, u_i) \in \delta(r_{i-1}, \langle a \rangle)$ and $\alpha_i = p\alpha_{i-1}$,
- $\hat{w}_i = \dot{\epsilon}$, $(r_i, u_i) \in \delta(r_{i-1}, \epsilon)$ and $\alpha_i = \alpha_{i-1}$, or
- $\hat{w}_i = \langle /a \rangle$ for some $a \in \Sigma \cup \{\epsilon\}$, $(r_i, u_i) \in \delta(r_{i-1}, p, \langle /a \rangle)$ and $p\alpha_i = \alpha_{i-1}$.

The run ρ is *accepting* if $r_0 = q_0$, $r_n \in F$, and $\alpha_0 = \alpha_n = \epsilon$; in this case, the string $u_1 u_2 \dots u_n$ is the *output* of T on w according to ρ.[4]

We note once again that the semantics of NWT is defined exclusively on well-nested input words. While NWT could easily be extended to work on partial nested words containing trailing opening and closing tags, this is not required for the purposes of cfGs on nested words and therefore outside the scope of this dissertation.

Example 8.8. *Figure 8.1 shows a NWT T_{ab}, with linear states displayed as circles and transitions as arrows; transition labels show, in that order, the input symbol to be read*

[4]Note that the ϵ-extension $\hat{w}$ on which the output $u_1 u_2 \dots u_n$ is produced is already implicit in the run ρ, so we do not specify it explicitly.

and the hierarchical state to be popped before the colon, and the hierarchical state to be pushed and the output string to be produced after the colon. Obviously, some of these are omitted where they don't make any sense; for instance, opening transitions do not pop any hierarchical states and internal ϵ-transitions do not consume an input or push or pop any hierarchical states.

Consider the input $w = \langle a\rangle\langle b\rangle\langle/b\rangle\langle a\rangle\langle/a\rangle\langle/a\rangle$. One possible ϵ-extension of this nested word is $\hat{w} = \dot{\epsilon}\langle a\rangle\langle b\rangle\langle/b\rangle\langle a\rangle\langle/a\rangle\langle/a\rangle\dot{\epsilon}\langle\epsilon\rangle\dot{\epsilon}\langle/\epsilon\rangle$. There are two accepting runs of T_{ab} on $\hat{w}$; the first starts with an internal ϵ-transition to state a_1 and produces the output $\langle a\rangle\langle a\rangle\langle/a\rangle\langle a\rangle\langle/a\rangle\langle/a\rangle\langle a\rangle\langle/a\rangle$, the second starts with an ϵ-transition to b_1 and produces the output $\langle b\rangle\langle b\rangle\langle/b\rangle\langle b\rangle\langle/b\rangle\langle/b\rangle\langle b\rangle\langle/b\rangle$.

The *image* $T(w)$ of a well-nested word $w \in \mathrm{WF}(\Sigma)$ under T is the set of all outputs of T on w according to some accepting run of T on w. This definition extends to sets of input words in the natural way: For a set $S \subseteq \mathrm{WF}(\Sigma)$, we define $T(S) = \bigcup_{w\in S} T(w)$. The *domain* $\mathcal{D}(T)$ of T is the set of all nested words w such that $T(w) \neq \varnothing$, and the *range* $\mathcal{R}(T)$ of T is the set of all strings u such that there exists a $w \in \mathrm{WF}(\Sigma)$ with $u \in T(w)$, i.e. the set of all possible outputs of T.

Example 8.9. *Consider again the NWT T_{ab} from Figure 8.1. As already seen in Example 8.8, for $w = \langle a\rangle\langle b\rangle\langle/b\rangle\langle a\rangle\langle/a\rangle\langle/a\rangle$, it holds that $\langle a\rangle\langle a\rangle\langle/a\rangle\langle a\rangle\langle/a\rangle\langle/a\rangle\langle a\rangle\langle/a\rangle$ and $\langle b\rangle\langle b\rangle\langle/b\rangle\langle b\rangle\langle/b\rangle\langle/b\rangle\langle b\rangle\langle/b\rangle$ are in $T_{ab}(w)$.*

It is easy to see that the domain of T_{ab} is $\mathcal{D}(T_{ab}) = WF(\{a, b\})$, as any well-nested word over $\{a, b\}$ induces a run from a_1 (or b_1) to itself, which can be extended to a successful run by internal ϵ-transitions. The range of T_{ab} is

$$\mathcal{R}(T_{ab}) = \{w\langle a\rangle^n\langle/a\rangle^n \mid w \in WF(\{a\}), n \in \mathbb{N}\} \cup \{w\langle b\rangle^n\langle/b\rangle^n \mid w \in WF(\{b\}), n \in \mathbb{N}\},$$

which is due to the fact that T_{ab} works as follows:

From the initial state i, T_{ab} branches nondeterministically into either state a_1 or b_1. In state a_1, T_{ab} checks that the input string is well-nested just as the NWA A_1 from Example 5.4. During this check, T_{ab} outputs $\langle a\rangle$ (resp. $\langle/a\rangle$) for each opening (resp. closing) input tag, effectively relabelling the input to consist exclusively of a-labelled tags. In state a_2, T_{ab} inserts into the output string an arbitrary number of opening $\langle a\rangle$ tags, for which a matching number of $\langle/a\rangle$ tags are inserted in state f before T_{ab} accepts. The behaviour of T_{ab} in states b_1 and b_2 is analogous, but outputs consist only of b-labelled tags. Altogether, T_{ab} chooses nondeterministically some $x \in \{a, b\}$, relabels all tags of a well-nested input word into x-labelled tags and then appends a string of the form $\langle x\rangle^n\langle/x\rangle^n$.

We next define several restrictions on the expressiveness of NWT.

Definition 8.10. Let $T = (Q, P, P_\epsilon, \Sigma, \delta, q_0, F)$ be a NWT. We call T

- *ϵ-free* if $P_\epsilon = \varnothing$ and δ contains no ϵ-transitions.
- *non-deleting* if the output component of every non-internal transition in δ is a non-empty string;

- *deterministic* (or a DNWT) if it is ϵ-free and for every $q \in Q, p \in P$ and $a \in \Sigma$, it holds that $|\delta(q, \langle a\rangle)| = |\delta(q, p, \langle/a\rangle)| = 1$;
- a *relabelling* transducer if it is ϵ-free and for every $q, q' \in Q, p \in P$, $a \in \Sigma$ and $u \in \Sigma^*$, if $(q', p, u) \in \delta(q, \langle a\rangle)$, then $u \in \langle\Sigma\rangle$, and if $(q', u) \in \delta(q, p, \langle/a\rangle)$, then $u \in \langle/\Sigma\rangle$;
- *functional*, if for every $w \in \mathrm{WF}(\Sigma)$, it holds that $|T(w)| = 1$.

It is easy to see that the length of any output of an ϵ-free NWT is at most linear in the length of the input string, while outputs of general NWT may grow to an arbitrary length.

We note that functionality, unlike the other restrictions defined here, is a *semantic* condition. We do not investigate here decidability or complexity of determining whether a NWT is functional; likely, techniques for Visibly Pushdown Transducers in [FRRST10] could be adapted for this purpose. Also, while determinism implies functionality, the converse does not hold; for instance, a NWT that rewrites all tag labels in an input nested word w to be equal to the label at position $|w| - 1$ can easily be realised as a functional nondeterministic NWT, but not as a deterministic one.

The following lemma shows that we can assume without loss of generality that each transition of a NWT involves at most one input and at most one output symbol, i.e. each opening (closing) transition outputs at most one opening (closing) tag and each internal ϵ-transition outputs nothing.

Lemma 8.11. *Each NWT $T = (Q, P, P_\epsilon, \Sigma, \delta, q_0, F)$ can be transformed in polynomial time into an NWT $T' = (Q', P', P'_\epsilon, \Sigma, \delta', q_0, F)$ with $T(w) = T'(w)$ for each $w \in WF(\Sigma)$, such that for any transition in δ' with output u, it holds that $|u| \leqslant 1$.*

We say that a NWT of this shape is in normal form.

Proof. An arbitrary NWT T is transformed into an NWT T' in normal form by successively replacing each transition that is not of the required form by a sequence of new states and transitions. We only examine this procedure for an opening transition; closing and internal transitions can be handled in a similar manner.

Assume for some $q \in Q$ and $a \in \Sigma$ that $(q', p', v) \in \delta(q, \langle a\rangle)$ with $v = v_1 \cdots v_n \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$ for some $n > 1$. We add new states $q_1, \ldots, q_{n-1}$ to Q and $p_{p',a}$ to P_ϵ. Let $k \leqslant n$ be the position of the last unmatched opening tag in v, i.e. $v_1 \cdots v_{k-1} \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $v_k \in \langle\Sigma\rangle$ and $v_{k+1} \cdots v_n \in \mathrm{WF}(\Sigma)$. We add a transition $(q_1, p_{p',a}, v_1)$ to $\delta(q, \langle\epsilon\rangle)$[5], a transition (q_{k+1}, p', v_k) to $\delta(q_k, \langle a\rangle)$ and, for each $i \in [n-1]$ with $i \neq k$, a transition $(q_{i+1}, p_{p',a}, v_i)$ to $\delta(q_i, \langle\epsilon\rangle)$ if $v_i \in \langle\Sigma\rangle$, or (q_{i+1}, v_i) to $\delta(q_i, p_{p',a}, \langle/\epsilon\rangle)$ if $v_i \in \langle/\Sigma\rangle$, identifying q_n with q'. Finally, we remove the original transition. This takes care of (reading or ϵ-)transitions producing more than one output symbol.

The resulting NWT T' is obviously in normal form. Its equivalence to T is relatively simple (if tedious) to prove by an induction over the structure of input words, with the main argument using the fact that T fulfils the ϵ-consistency and well-formedness condition. Notably, these conditions also justify the above simplification that newly added

[5] Note that $v_1 \in \langle\Sigma\rangle$ due to well-formedness.

ϵ transitions obtained from a transition reading a and pushing the hierarchical state p' use solely the new hierarchical ϵ-state $p_{p',a}$. □

In most of this chapter, as well as Chapter 9, we restrict our attention to non-deleting transducers. This is because regular nested word languages are closed under transduction by non-deleting NWT, which does not hold in the presence of deletions; as an example showing this, consider a NWT deleting all matched opening and closing c-labelled tags on the regular input language $\{(\langle a\rangle\langle/a\rangle\langle c\rangle)^n(\langle/c\rangle\langle b\rangle\langle/b\rangle)^n \mid n \geqslant 0\}$, yielding the non-regular language $\{(\langle a\rangle\langle/a\rangle)^n(\langle b\rangle\langle/b\rangle)^n \mid n \geqslant 0\}$. The practical motivation for desiring this property is the fact that the AXML setting assumes that function call results can be specified by standard XML schema languages, which are subclasses of regular nested word languages.

Moreover, for most of the transducer models examined here, non-deleting transducers are not a significant restriction when it comes to context-free games, as shown in Chapter 9 by Lemma 9.2.

Using Lemma 8.11, it is comparatively easy (if tedious) to prove that non-deleting NWTs are closed under composition. This proof, like most proofs for properties of NWT in this section, follows proof ideas used in [FRRST10; RS08] adapted to the specifics of NWT.

Proposition 8.12. *Let T_1, T_2 be non-deleting NWT. Then there exists a non-deleting NWT T such that for all $w \in WF(\Sigma)$, it holds that $T(w) = (T_2 \circ T_1)(w) \stackrel{def}{=} T_2(T_1(w))$.*[6] *This NWT T can be computed from T_1 and T_2 in polynomial time and is of size $\mathcal{O}(|T_1| \cdot |T_2|)$.*

Proof. The basic idea behind this construction is simple: The transducer T simulates T_1 on its input and directly feeds the output of T_1 into T_2. This is done by a sort of product construction which is, for the most part, quite straightforward. However, there are some subtleties that need to be addressed, which stem from the fact that both T_1 and T_2 can contain ϵ-transitions, so the various possibilities how one of the two transducers might perform a step while the other is idle have to be dealt with. We note that this proof could be extended to general (not necessarily non-deleting) NWT, but since the proof details are quite technical already, we restrict our attention to non-deleting NWT here.

Let $T_1 = (Q^1, P^1, P^1_\epsilon, \Sigma, \delta^1, q^1_0, F^1)$ and $T_2 = (Q^2, P^2, P^2_\epsilon, \Sigma, \delta^2, q^2_0, F^2)$ be two NWT in normal form. We construct from T_1 and T_2 the NWT $T = (Q, P, P_\epsilon, \Sigma, \delta, q_0, F)$ as follows:

- $Q = Q^1 \times Q^2$,
- $P = (P^1 \cup \{\bot\}) \times P^2$ for a special symbol $\bot$ not used in T_1 or T_2,

[6] Note that the order of operands in composition of NWTs differs from that in composition of effects (Def. 3.17) – for effects, $E_1 \circ E_2$ corresponds to first considering E_1, then E_2 on the result, while for NWTs, $T_1 \circ T_2$ corresponds to first applying T_2, and then T_1 on the result. This difference in ordering is due to the fact that effect composition closely corresponds to string concatenation (Lemma 3.19), while composition of NWTs is much closer to "classical" composition of functions or relations.

- $P_\epsilon = (P_\epsilon^1 \times P^2) \cup (\{\bot\} \times P_\epsilon^2)$
- $q_0 = (q_0^1, q_0^2)$,
- $F = F^1 \times F^2$, and
- δ is constructed as detailed below.

The construction of δ is quite straightforward in the case where T_1 produces an output on which T_2 is simulated: If $(q_1', p_1', \langle b\rangle) \in \delta^1(q_1, \langle a\rangle)$ (or $(q_1', p_1', \langle b\rangle) \in \delta^1(q_1, \langle\epsilon\rangle)$, respectively) and $(q_2', p_2', \langle c\rangle) \in \delta^2(q_2, \langle b\rangle)$, then $((q_1', q_2'), (p_1', p_2'), \langle c\rangle) \in \delta((q_1, q_2), \langle a\rangle)$ (or $((q_1', q_2'), (p_1', p_2'), \langle c\rangle) \in \delta((q_1, q_2), \langle\epsilon\rangle)$, respectively), and analogously for the corresponding closing (ϵ-) transitions.

Internal ϵ-transitions for T_1 and T_2 are also easily handled, as for these transitions, neither transducer consumes an input or produces an output. If $(q_1', \epsilon) \in \delta^1(q_1, \epsilon)$, then $((q_1', q_2), \epsilon) \in \delta((q_1, q_2), \epsilon)$ for each $q_2 \in Q^2$, and if $(q_2', \epsilon) \in \delta^2(q_2, \epsilon)$, then $((q_1, q_2'), \epsilon) \in \delta((q_1, q_2), \epsilon)$ for each $q_1 \in Q^1$.

The only case requiring special attention is the one where only one of the two transducers modifies the output. This happens when T_2 produces an output symbol by an ϵ-transition, without T_1 consuming an input symbol. In this case, T_2 produces a hierarchical state while T_1 doesn't; to this end, some states in the set P_ϵ contain a component $\bot$ to indicate a "null transition" for T_1.

More formally, if for some $q_2, q_2' \in Q^2$, $p_2' \in P^2$ and $a \in \Sigma$, $(q_2', p_2', \langle a\rangle) \in \delta^2(q_2, \langle\epsilon\rangle)$, then $((q_1, q_2'), (\bot, p_2'), \langle a\rangle) \in \delta((q_1, q_2), \langle\epsilon\rangle)$ for all $q_1 \in Q^1$ (and analogously for closing transitions).

It follows directly from the construction that T is ϵ-consistent; the well-formedness condition for T follows from well-formedness of T_1 and T_2 by a simple but lengthy case distinction over all the sorts of transitions introduced here. Furthermore, since T is obviously in normal form, it automatically fulfils synchronisation. It remains to be proven that $T(w) = T_2(T_1(w))$ indeed holds for all $w \in \mathrm{WF}(\Sigma)$.

To this end, let $w_2 \in T_2(T_1(w))$; let further ρ_1 be an accepting run of T_1 with output w_1 on an ϵ-extension $\hat{w}$ of w, and let ρ_2 be an accepting run of T_2 with output w_2 on an ϵ-extension $\hat{w}_1$ of w_1. We construct an ϵ-extension $\hat{w}'$ of w and an accepting run ρ of T on $\hat{w}'$ with output w_2.

We denote all positions of $\hat{w}$ in which T_1 outputs some symbol as *1-producing*. Note that there is a bijective correspondence between positions of w_1 and 1-producing positions of $\hat{w}$ and that all $\langle\epsilon\rangle$- and $\langle/\epsilon\rangle$-positions of $\hat{w}$ are 1-producing. Next, we examine all $\langle\epsilon\rangle$- and $\langle/\epsilon\rangle$-positions of $\hat{w}_1$; these, we call *2-producing*. Due to the well-formedness and ϵ-consistency restrictions on T_1 and T_2, it is possible to insert all $\langle\epsilon\rangle$-, $\langle/\epsilon\rangle$- and $\dot{\epsilon}$-positions of $\hat{w}_1$ into $\hat{w}$ in such a way that we obtain an ϵ-extension $\hat{w}'$ of w that has $\hat{w}$ as a subsequences, and such that there is a subsequence of $\hat{w}'$ with a bijective correspondence to $\hat{w}_1$ mapping opening (resp. closing) to opening (resp. closing) tags and $\dot{\epsilon}$ to $\dot{\epsilon}$. In particular, the latter subsequence can be obtained from $\hat{w}'$ by removing only $\dot{\epsilon}$ symbols.

It is now easy to see that an accepting run ρ of T on $\hat{w}'$ with output w_2 can be obtained by combining the transitions used in ρ_1 and ρ_2 – positions in $\hat{w}'$ that are 1-producing correspond to "standard" transitions of T, $\dot{\epsilon}$-positions correspond to internal ϵ-transitions, and 2-producing positions correspond to ϵ-transitions with a $\perp$ component in their hierarchical state. This shows that $T_2(T_1(w)) \subseteq T(w)$.

For the other direction, let ρ be a run of T on an ϵ-extension $\hat{w}'$ of w with output w_2. We label the positions of $\hat{w}'$ according to the transitions taken by T in ρ – positions where "standard" transitions are used are labelled as 1-producing, $\dot{\epsilon}$-positions are labelled as internal, and positions with transitions whose hierarchical stack contains a $\perp$ component are labelled as 2-producing. Similar to the previous part of the proof, we can then use these labels to separate ρ into an accepting run ρ_1 of T_1 on an ϵ-extension $\hat{w}$ of w with output w_1 and an accepting run ρ_2 of T_2 on an ϵ-extension $\hat{w}_1$ of w_1 with output w_2, thus proving that $w_2 \in T_2(T_1(w))$ holds. □

Throughout the rest of this and the next chapter, it will sometimes be necessary to restrict the domain of some NWT to a given regular nested word language. The following corollary to Proposition 8.12 shows that this is indeed possible.

Corollary 8.13. *Let T be a non-deleting NWT and A an ϵ-NWA over alphabet Σ. Then, there exists a non-deleting NWT T' of size $\mathcal{O}(|T| \cdot |A|)$ such that $\mathcal{D}(T') = \mathcal{D}(T) \cap L(A)$ and $T'(w) = T(w)$ for each $w \in \mathcal{D}(T) \cap L(A)$.*

Proof. Let T_A be a NWT with $\mathcal{D}(T_A) = L(A)$ and $T_A(w) = \{w\}$ for each $w \in \mathrm{WF}(\Sigma)$, i.e. T_A accepts exactly the nested words in $L(A)$ and simply outputs its input word. Such a NWT is easy to construct with a size in $\mathcal{O}(|A|)$.

Then, we set T' to be the NWT for $T \circ T_A$ as constructed in Proposition 8.12. For each $w \in L(A) \cap \mathcal{D}(T)$, it holds that $T'(w) = T(T_A(w)) = T(w)$, for each $w \notin L(A)$, $T'(w) = T(\varnothing) = \varnothing$, and for each $w \in L(A) \backslash \mathcal{D}(T)$, it holds that $T'(w) = T(w) = \varnothing$. This proves that $\mathcal{D}(T') = \mathcal{D}(T) \cap L(A)$ and $T'(w) = T(w)$ for each $w \in \mathcal{D}(T) \cap L(A)$. The desired bound on the size of T' follows directly from Proposition 8.12. □

In order to prove closure of regular nested word languages under transduction by non-deleting NWT, we observe another helpful property of these transducers.

Lemma 8.14. *Let T be a non-deleting NWT. Then $\mathcal{R}(T)$ is a regular language of nested words. Furthermore, an ϵ-NWA for $\mathcal{R}(T)$ can be computed from T in polynomial time.*

Proof. Let $T = (Q, P, P_\epsilon, \Sigma, \delta, q_0, F)$ be a non-deleting NWT in normal form. The basic idea behind constructing the ϵ-NWA A for $\mathcal{R}(T)$ is taking the input string for A and verifying it against the output component of T. This way, it is easy to ensure that every string in $\mathcal{R}(T)$ is accepted by A; some care has to be taken, however, to make certain that for every string w' accepted by A, there is a nested word $w \in \mathrm{WF}(\Sigma)$ such that $w' \in T(w)$.

Since T is in normal form, all opening transitions in δ are of the form $(q, \langle a\rangle, q', p, \langle b\rangle)$ and all closing transitions of the form $(q, p, \langle /a\rangle, q', \langle /b\rangle)$ (for $a \in \Sigma \cup \{\epsilon\}$ and $b \in \Sigma$) while internal transitions are of the form $(q, \epsilon, q', \epsilon)$.

The linear state set of A is Q and its set of hierarchical states is $P \times (\Sigma \cup \{\epsilon\})$; its starting and accepting states are those of T. Reading transitions in A are constructed from those of T by taking, for each opening transition $(q, \langle a\rangle, q', p, \langle b\rangle) \in \delta$, a transition $(q, \langle b\rangle, q', (p, a))$, and for each closing transition $(q, p, \langle/a\rangle, q', \langle/b\rangle) \in \delta$, a transition $(q, (p, a), \langle/b\rangle, q')$. Each internal transition $(q, \epsilon, q', \epsilon) \in \delta$ translates to an ϵ-transition (q, ϵ, q') in A. This construction of A from T requires only a simple iteration over all transitions in δ, so it is obviously feasible in polynomial time.

To prove correctness of this construction, we need to show that for any nested word $w' \in \mathrm{WF}(\Sigma)$ it holds that $w' \in L(A)$ if and only if $w' \in \mathcal{R}(T)$.

For the "if" direction, if $w' \in \mathcal{R}(T)$, then there is some $w \in \mathrm{WF}(\Sigma)$ such that T has an accepting run ρ on some ϵ-extension of w with output w'. Translating the transitions of T taken in ρ into transitions of A as per the above construction naturally yields an accepting run of A on w'.

For the "only if" direction, assume that $w' \in L(A)$ for some $w' \in \mathrm{WF}(\Sigma)$. Then, there is an accepting run ρ of A on w'. We can construct from ρ an ϵ-extension $\hat{w}$ of a nested word $w \in \mathrm{WF}(\Sigma)$ by extracting the sequence of hierarchical components of transitions in ρ – every opening (closing) transition in ρ with hierarchical component (p, a) with some $p \in P$, $a \in \Sigma \cup \{\epsilon\}$ corresponds to a symbol $\langle a\rangle$ ($\langle/a\rangle$) in $\hat{w}$, and every ϵ-transition taken in ρ corresponds to a symbol $\dot{\epsilon}$ in $\hat{w}$. It is clear to see that indeed $\hat{w}$ is an ϵ-extension of some nested word $w \in \mathrm{WF}(\Sigma)$ and that there is an accepting run of T on $\hat{w}$ that outputs w', which yields $w' \in \mathcal{R}(T)$ as was to be proven. □

Corollary 8.15. *Regular nested word languages are closed under transduction by non-deleting NWT, i.e. if $L \subseteq WF(\Sigma)$ is regular and T an NWT, then $T(L)$ is regular.*

Proof. Let A be a NWA for L and T a NWT. From A, we can construct by Corollary 8.13 a NWT T_A with $\mathcal{D}(T_A) = L \cap \mathcal{D}(T)$ and $T_A(w) = T(w)$ for each $w \in \mathcal{D}(T_A)$, which implies that $\mathcal{R}(T_A) = T(L)$. By Lemma 8.14, this implies that $T(L)$ is regular as well. □

We now turn to the complexity of standard decision problems for NWT. The upper bounds use relatively simple constructions based on Proposition 8.12, while lower bounds follow from comparable results for NWA.

Theorem 8.16. *The membership problem for non-deleting NWT (Given a non-deleting NWT T and nested words $w, u \in WF(\Sigma)$, is $u \in T(w)$?) is in* PTIME.

Proof. From w, we can easily compute a NWA A of size $\mathcal{O}(w)$ with $L(A) = \{w\}$. By Corollary 8.13, we can compute from A and T in polynomial time a non-deleting NWT T_w with $\mathcal{D}(T_w) = \mathcal{D}(T) \cap \{w\}$ and $\mathcal{R}(T_w) = T(w)$. By Corollary 8.15, $T(w)$ is regular, and by Lemma 8.14 an ϵ-NWA A' for $T(w)$ can be computed in polynomial time. This ϵ-NWA is of polynomial size, and checking u for membership in $L(A)$ is possible in polynomial time by Theorem 8.5 (a). □

Theorem 8.17. *The nonemptiness problem for non-deleting NWT (Given a non-deleting NWT T, is there a nested word $w \in WF(\Sigma)$ with $T(w) \neq \varnothing$?) is* PTIME*-complete with regard to logspace reductions.*

Proof. By definition, it holds that there is some $w \in \mathrm{WF}(\Sigma)$ with $T(w) \neq \varnothing$ if and only if $\mathcal{D}(T) \cap \mathrm{WF}(\Sigma) \neq \varnothing$. By Corollary 8.13, we can compute from T a NWT T' whose domain is exactly $\mathcal{D}(T) \cap \mathrm{WF}(\Sigma)$. Clearly, it holds that $\mathcal{D}(T') \neq \varnothing$ if and only if $\mathcal{R}(T') \neq \varnothing$, so by Lemma 8.14 we can extract from T' an ϵ-NWA A of polynomial size with $L(A) = \mathcal{R}(T')$. All of these transformations, as well as testing whether $L(A) \neq \varnothing$, are feasible in polynomial time. This proves the upper bound.

The lower bound follows by reduction from the emptiness problem for DNWA (cf. Theorem 5.8). Let $A = (Q, P, \Sigma, \delta, q_0, F)$ be a DNWA to be tested for non-emptiness. We construct from A a NWT T with $T(\epsilon) = L(A)$ and $T(w) = \varnothing$ for all $w \neq \epsilon$. The idea behind the reduction is replacing, for each $a \in \Sigma$, each $\langle a \rangle$ (resp. $\langle / a \rangle$) transition of A by an $\langle \epsilon \rangle$ (resp. $\langle / \epsilon \rangle$) transition in T that writes $\langle a \rangle$ (resp. $\langle / a \rangle$) as an output. Formally, we set $T = (Q, P \times \Sigma, P \times \Sigma, \Sigma, \delta', q_0, F)$, where δ' contains a transition $(q, \langle \epsilon \rangle, q', (p, a), \langle a \rangle)$ (resp. $(q, (p, a), \langle / \epsilon \rangle, q', \langle / a \rangle)$) if and only if δ contains a transition $(q, \langle a \rangle, q', p)$ (resp. $(q, p, \langle / a \rangle, q')$). This construction is feasible using logarithmic space, and it holds that $T(\epsilon) = L(A)$ and $T(w) = \varnothing$ for all $w \neq \epsilon$, so $\mathcal{D}(T) \neq \varnothing$ if and only if $L(A) \neq \varnothing$. □

Theorem 8.18. *The type checking problem for non-deleting NWT (Given a non-deleting NWT T and ϵ-NWA A_1, A_2, is $T(L(A_1)) \subseteq L(A_2)$?) is*

(a) EXPTIME*-complete in general, and*

(b) PTIME*-complete (w.r.t. logspace reductions) if A_2 is a DNWA.*

Proof. The lower bound in (a) follows directly by a reduction from the inclusion problem for NWA, which is known to be EXPTIME-complete [AM09]. Let B_1 and B_2 be NWA to be checked for inclusion of $L(B_1)$ in $L(B_2)$. We can construct in polynomial time a NWT T that simply reproduces each input symbol in the output and accepts any input (i.e. $T(w) = \{w\}$ for any $w \in (\langle \Sigma \rangle \cup \langle / \Sigma \rangle)^*$). Then, clearly, $L(B_1) \subseteq L(B_2)$ if and only if $T(L(B_1)) \subseteq L(B_2)$, as $T(L(B_1)) = L(B_1)$.

The lower bound in (b) follows from PTIME-hardness of the emptiness problem for DNWA by a similar reduction to that used in the proof of Theorem 8.5 (c). Let A be a DNWA to be tested for emptiness. We can construct in logarithmic space an ϵ-NWA A_1 deciding $\mathrm{WF}(\Sigma)$, and (since A is deterministic) a DNWA A_2 deciding the complement of $L(A)$. For T, we construct a NWT reproducing its input, i.e. with $\mathcal{D}(T) = \mathrm{WF}(\Sigma)$ and $T(w) = \{w\}$ for each $w \in \mathrm{WF}(\Sigma)$. It then holds that $T(L(A_1)) \subseteq L(A_2)$ if and only if $L(A_2) = \mathrm{WF}(\Sigma)$, which is the case if and only if $L(A)$ is empty.

The upper bounds in both (a) and (b) can easily be proven using prior results from this section. By Corollary 8.13 and Lemma 8.14, we can construct in polynomial time an ϵ-NWA deciding $T(L(A_1))$; checking this ϵ-NWA for inclusion in $L(A_2)$ is generally possible in exponential time, and in polynomial time if A_2 is a DNWA by Theorem 8.5 (b) and (c). □

8.3. Outlook and bibliographical remarks

This chapter introduced nested word transducers as a formalism for specifying transformations on nested words. In their most general form as defined here, these NWT have

a simple normal form presentation; they can also be proven to be closed under composition and inverse transduction following proof ideas from [FRRST10; RS08], even though these proofs are omitted here. However, they lack the property that regular languages of nested words are closed under transduction; non-deleting transducers were introduced to obtain this closure property, while sacrificing closure under inverse transduction. For non-deleting NWT, membership, non-emptiness and type checking against DNWA are tractable. As a technical tool (that may be of independent interest), nested word automata with ϵ-transitions, which slightly extend nondeterministic NWA without increasing their complexity or expressiveness, were also introduced.

An important question regarding nested word transducers is how their expressiveness relates to other types of transducers. For the recognition of tree languages, (bottom-up) finite-state tree automata are the *de facto* standard model, and nested word automata on linearisations of trees can be proven to be equally as expressive as finite-state tree automata; for tree transformations, however, a wide variety of transducer models exists (for some basics see e.g. [GS84; Com+]), and it is not obvious whether the nested word transducers introduced here directly correspond to any of them. In particular, it would be interesting to see how nested word transducers compare to *monadic second order (MSO) logic* as a formalism for specifying transformations, as equivalence to MSO logic has been established as an important property for "sensible" mechanisms in the context of language recognition (cf. [Tho90; TW68; Don70; AM09]). So far, however, this question remains open.

Bibliographical remarks. All results presented here are solely by the author and were previously published in [Sch16]. The formalisation of nested word transducers is based on *visibly pushdown transducers* [RS08; TVY08], specifically as a cross between synchronised VPT [RS08] and well-formed VPT [FRRST10]. Accordingly, most of the proof ideas for closure and complexity properties of NWT are based on the corresponding proofs for variants of VPT.

9. Transducer-based context-free games

Having introduced nested word transducers in the previous chapter, we now examine context-free games whose replacement relation is given by a NWT, i.e. games in which, once Juliet plays *Call* on a nested subword v, Romeo may choose as a replacement for v any string from the image $R(v)$, where R is a transducer representing the replacement rule set of the corresponding game. In order for Romeo to actually have any choice, the replacement transducer R will in general be nondeterministic and non-functional (as defined in Definition 8.10); using a functional (or deterministic) transducer for replacement instead results in a "solitaire" game for Juliet without any choice for Romeo. A proper generalisation of context-free games with regular replacement from Chapter 6 and validation games from Chapter 7 would call for having a separate replacement transducer R_a for each function symbol a, but the discussion in Section 9.1 will show that we can, without loss of generality, restrict our attention to games with only a single transducer used for calls on *any* function symbol.

As in most previous chapters examining context-free games, our main focus here is once again on classifying the complexity of the winning problem for Juliet. In previous variants of context-free games, the main factors influencing said complexity were the amount of replay permitted as well as the representation of target, replacement, and (for validation games) validation languages. With transducer-based replacement, however, the representation of replacement (and validation) languages gets replaced by the class of replacement transducer. As already seen in the previous chapter, there is a plethora of different transducer classes that may be used (general, ϵ-free, relabelling, functional, etc.), and as this chapter will show, the choice of transducer model has a great impact on the decidability and complexity of the winning problem for Juliet. Since this chapter mainly focusses on the impact of the transducer model, we will once again assume target languages to be represented by deterministic NWA.

The goal of the exploration into transducer-based cfGs in this chapter is twofold: Firstly, as in previous chapters, finding a tractable setting (i.e. one in which the winning problem is decidable in polynomial time) is of interest for the practical application of AXML schema rewriting; secondly, it will become clear early on that cases where the winning problem is undecidable when replay is allowed are quite easy to come by, so the other goal of this chapter lies in charting the decidability frontier for context-free games with transducer-based replacement.

With these goals in mind, Sections 9.2 through 9.4 give a complete classification by replay and transducer model of the decidability and complexity of the winning problem for transducer-based cfGs for three main classes of replacement transducers: General NWT (Section 9.2), ϵ-free NWT (Section 9.3), and relabelling NWT (Section 9.4). The results of these sections are summarised in Table 9.1; as can be seen from that table,

	No replay	Bounded	Unbounded
NWT	2-EXPTIME	undecidable	undecidable
ϵ-free NWT	co-NEXPTIME	non-elementary	undecidable
Relabelling	PSPACE	PSPACE	EXPTIME
Functional relabelling	NP	NP	PSPACE

Table 9.1.: Summary of main complexity results for JWin in transducer-based context-free games. All results are completeness results.

complexity results cover a wide range of complexity classes. However, there is no tractable case among these; even in the very limited setting of replay-free games with functional relabelling transducers, the winning problem remains NP-complete.

In an attempt to find tractable cases, as well as to offer an outlook into other potentially sensible restrictions for context-free games with transducer-based replacement, Section 9.5 presents three additional restrictions on games, derived from lower bound proofs for the results of earlier sections: *depth-bounded* replacement NWTs, strategies for Juliet with bounded *Call width*, and, finally, *write-once* games.

9.1. Definitions

With the intuition given above, the syntax and semantics definitions for context-free games with transducer-based replacement follow the pattern of all definitions for cfGs on nested words so far. The main difference to standard cfGs from Chapter 6 is just that the replacement relation in a transducer-based cfG does not simply give replacement words based on function symbols that are supposed to be the root label of a subword to be replaced, but instead based upon the *entirety* of the subword. Accordingly, instead of being represented by a set of nested word automata (one for each replacement language), the replacement relation here is assumed to be given by (some variant of) a nested word *transducer*. Otherwise, most definitions again carry over from Chapter 6.

Definition 9.1. A *context-free game (cfG) on nested words with transducer-based replacement* $G = (\Sigma, \Gamma, R, T)$ consists of a finite alphabet Σ, a set $\Gamma \subseteq \Sigma$ of *function symbols*, a *(replacement) rule set* $R \subseteq \mathrm{WF}(\Sigma) \times \mathrm{WF}(\Sigma)$ and a *target language* $T \subseteq \mathrm{WF}(\Sigma)$. As usual, we assume that the target language is a regular language of nested words; additionally, the rule set will generally be represented by some variant of NWT throughout this chapter. Therefore, we denote by $R(v) = \{w \in \mathrm{WF}(\Sigma) \mid (v, w) \in R\}$ the image of v under R as defined on page 157.

Configurations are defined as tuples $\kappa = (p, u, v) \in \{\mathrm{J}, \mathrm{R}\} \times (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^* \times (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, and $\kappa' = (p', u', v')$ is a *successor configuration* of $\kappa = (p, u, v)$ (Notation: $\kappa \to \kappa'$) if one of the following holds:

(1) $p' = p = \mathrm{J}$, $u' = us$, and $sv' = v$ for some $s \in \langle\Sigma\rangle \cup \langle/\Sigma\rangle$ (Juliet plays *Read*);

(2) $p = \mathrm{J}$, $p' = \mathrm{R}$, $u = u'$, $v = v' = \langle/a\rangle z$ for $z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $a \in \Gamma$, (Juliet plays *Call*);

(3) $p = \mathrm{R}$, $p' = \mathrm{J}$, $u = x\langle a\rangle y$, $v = \langle/a\rangle z$ for $x, z \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, $y \in \mathrm{WF}(\Sigma)$, $u' = x$ and $v' = y'z$ for some $y' \in R(\langle a\rangle y\langle/a\rangle)$ (ROMEO plays y').

All other semantic definitions like the induced reachability game, plays, strategies and the winning problem are defined as in Definition 6.1. Note, however, that configurations of the form $(\mathrm{R}, x\langle a\rangle y, \langle/a\rangle z)$ with $\langle a\rangle y\langle/a\rangle \notin \mathcal{D}(R)$ have no successor configurations by this definition, so it follows from the standard semantics definition that if JULIET plays *Call* on some subword that is not accepted by the replacement transducer, ROMEO automatically wins the play.

Note that the definition of transducer-based cfGs calls only for a *single* replacement transducer, which is unlike the situation for standard cfGs, where a replacement language automaton was given for each function symbol. The definition of transducer-based cfGs could be adapted to call for a different replacement transducer for each function symbol as well, but these two variants are actually equivalent. This is due to the fact that the replacement transducer is called on the string to be replaced, *including its root label*; therefore, a game with a separate transducer for each function symbol can easily be transformed into a game using only a single "global" transducer, which simply branches and simulates the appropriate transducer based on the root label it reads.

In the rest of this chapter, we will once again examine the complexity of the winning problem for JULIET, depending on the class of games allowed.

JWIN($\mathcal{G}$)	
Given:	A context-free game $G \in \mathcal{G}$ and a nested word w.
Question:	Is $w \in \mathrm{JWin}(G)$?

In this chapter, parameters for the classification for games are the type of transducers used (NWT, ϵ-free NWT, relabelling transducers, or other variants) and the types of strategies allowed. For strategies, we again examine primarily strategies with unbounded, bounded or no replay, but some lower bound proofs will give rise to additional restrictions, as discussed in Section 9.5. As in the previous chapters, even though one might take the representation of target languages as another parameter of the class of games, we generally assume the target language T of any transducer-based cfG $G = (\Sigma, \Gamma, R, T)$ to be represented by a deterministic NWA $A(T) = (Q, \Sigma, \delta, q_0, F)$ in normal form (cf. Def. 5.3).

As was already mentioned in the previous chapter, we restrict our attention to cfGs whose rule set is given by a non-deleting transducer; this, however, is not a significant restriction, as the following result shows.

Lemma 9.2. *Any context-free game $G = (\Sigma, \Gamma, R, T)$ with NWT R can be transformed in polynomial time into a game $G' = (\Sigma', \Gamma, R', T')$ such that R' is non-deleting and it holds that $JWin(G') \cap WF(\Sigma) = JWin(G)$.*

Proof. The idea behind this proof is modifying R into R' in such a way that, whenever R would delete some tag $\langle a\rangle$ (or $\langle/a\rangle$), R' instead replaces that tag by a special "strike-out" version $\langle\text{~~a~~}\rangle$ (or $\langle/\text{~~a~~}\rangle$) with $\text{~~a~~} \notin \Sigma$, instead; we therefore set $\Sigma' = \Sigma \uplus \{\text{~~a~~} \mid a \in \Sigma\}$. To ensure that iterated transductions respect deleted tags, we add transitions to each

state of R' that only replace "strike-out" tags by themselves without changing the state of R'. Finally, we similarly modify the target DNWA for T in order to ignore all tags with labels not in Σ (i.e. only check them for their nesting structure without changing states). Since JULIET may only play *Call* on symbols from Σ, ROMEO is unable to rewrite "strike-out" symbols, and the DNWA for T' ignores symbols outside of Σ, it is clear that JULIET has a winning strategy on any string $w \in \mathrm{WF}(\Sigma)$ in G' if and only if she has a winning strategy on w in G. □

9.2. Games with general NWT

The main characteristic distinguishing general NWT from ϵ-free NWT is the fact that, for any input string w and NWT T, transducts in $T(w)$ may be arbitrarily large in the size of w. This behaviour is necessary if we want to simulate games with regular replacement languages (in the sense of Chapter 6) by transducer-based games. As it turns out, however, NWT-based replacement is much more complex than that: the winning problem in games with replay becomes undecidable (as opposed to 2-EXPTIME-complete with regular replacement languages).

Theorem 9.3. *For the class of games with NWT and Call depth $k \geqslant 2$,* JWIN *is not recursively enumerable.*

The proof of this result follows the standard idea for lower bound proofs outlined in Section 3.3 – reducing from the complement of an existence problem (in this case, the halting problem for Turing machines), tasking ROMEO with providing a witness (a halting computation), and JULIET with checking that this witness is correct. Different from the standard approach for context-free games without transducers, however, we do not require JULIET to point out an actual flaw in the witness. Instead, we use a technique that will appear several times in lower bound proofs for transducer-based games – we encode some transformation mechanism (in this case, the transition function of a Turing machine) in the replacement transducer, require JULIET to "trigger" this transformation a set number of times (here: the length of an accepting computation provided by ROMEO) and then use the target language to check whether this iterated transformation process yields a "correct" outcome from a given starting point (here: an accepting configuration from the input TM's starting configuration).

Proof. The proof is by reduction from the complement of the halting problem for Turing machines with an empty input. From a given Turing machine M with working alphabet Σ and state set Q, we construct a game G and string w such that JULIET has a winning strategy on w in G if and only if M does not halt on an empty input. The game G uses two function symbols, s and t, which are not in Σ in order to avoid plays according to G interfering with the workings of M.

The idea behind the reduction is rather simple. The game begins on the input string $\langle s\rangle\langle/s\rangle$, where JULIET is supposed to *Call* $\langle/s\rangle$ as her first move. For his reply, ROMEO then picks a number r such that M halts on the empty input after exactly r steps (if such a number exists) and returns $\langle t\rangle^r \widehat{v}_0 \langle/t\rangle^r$, where $\widehat{v}_0$ is a string representing the initial

configuration of M on the empty input. JULIET should then call each $\langle/t\rangle$ in left-to-right order, making ROMEO simulate a step of M on the current position of M encoded in the current string. After JULIET has called all r closing $\langle/t\rangle$ tags and thus rewritten $\widehat{v}_0$ into a string $\widehat{v}_r$ representing the configuration of M after r computation steps, JULIET wins the game if $\widehat{v}_r$ does not represent a halting configuration.

We can represent configurations of M as flat strings over the alphabet $\Sigma \times (Q \cup \{-\})$ in the standard fashion – a string $(x_1, -) \cdots (x_{k-i}, -)(x_k, q)(x_{k+1}, -) \cdots (x_m, -)$ denotes that the content of M's working tape is $x_1 \cdots x_m$, with the head of M being on the tape's k-th cell and M being in state q. Each such flat string v can then be represented as a nested string $\widehat{v}$ using the standard nested word encoding defined in the beginning of Chapter 5. Without loss of generality, we assume that M only has a single halting state h and always moves its head to the left-most used tape cell before halting, i.e. the flat string encoding of a halting configuration is of the form $(\Sigma \times \{h\})(\Sigma \times \{-\})^*$.

It is easy to construct a polynomial-sized NWT rewriting $\langle s\rangle\langle/s\rangle$ into a string of the form $\langle t\rangle^r \widehat{v}_0 \langle/t\rangle^r$ (for arbitrary $r \in \mathbb{N}$ and a nested string encoding $\widehat{v}_0$ of M's initial configuration) and every string $\langle t\rangle\widehat{v}_i\langle/t\rangle$ where $\widehat{v}_i$ is the nested word encoding of some configuration of M into $\widehat{v}_{i+1}$, where $\widehat{v}_{i+1}$ encodes the successor configuration of the one encoded by $\widehat{v}_i$; in fact, the latter rewriting is even functional (if nondeterministic for requiring a look-ahead when M moves its head to the left). We can also easily construct a DNWA accepting all nested-word encodings of non-halting configurations of M.

It only remains to be explained how we ensure that the game is played in the fashion sketched above, i.e. that JULIET plays *Call* first on $\langle/s\rangle$ and then on each $\langle/t\rangle$ in left-to-right order. To keep JULIET from leaving $\langle/s\rangle$ or some $\langle/t\rangle$ uncalled, we simply set up the target DNWA such that it doesn't accept any nested strings containing tags with labels s or t. Finally, to safeguard against JULIET skipping some $\langle/t\rangle$ before calling the next, we modify the replacement transducer R in such a way that it rejects any input string containing two or more nested t tags, i.e. strings of the form $\langle t\rangle\langle t\rangle v\langle/t\rangle\langle/t\rangle$ with $v \in \mathrm{WF}(\Sigma)$.

In the game G thus constructed, JULIET clearly has a winning strategy on $w = \langle s\rangle\langle/s\rangle$ if and only if there is no r such that M reaches a halting configuration within r steps, i.e. if and only if M doesn't halt. □

Even the replay-free winning problem for JULIET is quite hard when using NWT for replacement – we show that this problem is complete for doubly exponential time. The lower bound uses a rather intricate reduction from a two-player tiling problem, while the upper bound is proven by reduction to the purely NWT-based problem of *alternating iterated transduction*, which can be proven to be in 2-EXPTIME.

Theorem 9.4. *For the class of replay-free games with NWT,* JWIN *is* 2-EXPTIME-*complete.*

We prove the upper and lower bounds for Theorem 9.4 separately. Since context-free games with general and with non-deleting NWT are polynomially equivalent by Lemma 9.2, we prove the lower bound for general NWT and the upper bound for non-deleting NWT.

Proposition 9.5. *For the class of replay-free games with NWT,* JWin *is hard for* 2-EXPTIME.

This lower bound again follows the basic principle of reducing from the complement of an existence problem, with Romeo choosing a witness and Juliet attempting to prove that Romeo's witness is incorrect. The proof of Proposition 9.5 adds a new ingredient, the technique of *alternating deletion*, to this basic principle. This technique is used in proving lower bounds for transducer-based games; it allows Romeo to give a witness that is "too large" to be checked by a polynomial-sized target DNWA and tasks Juliet with repeatedly choosing one half of the witness to be deleted, performing a sort of "binary search" until only a sufficiently small witness remains. For tilings in particular, this technique usually calls for Juliet to repeatedly select either all odd-numbered columns or all even-numbered columns of the tiling candidate given by Romeo for deletion until only a single column remains, which the target language automaton then checks for vertical errors or evidence of attempted cheating.

Proof. The proof is by reduction from the complement of the 2-Player Exponential Corridor Tiling problem as defined in Section 2.5. Recall that this problem is complete for 2-EXPTIME by Theorem 2.9.

Given a tiling instance $\mathcal{I} = (U, V, H, u_i, u_f, n)$, we construct a game $G = (\Sigma, \Gamma, R, T)$ and string w such that Juliet has a winning strategy in G on w if and only if Player 2 has a winning strategy in the exponential-width tiling game on instance $\mathcal{I}$.

The input string is $w = (\langle d\rangle\langle e\rangle)^n\langle s\rangle\langle/s\rangle(\langle/e\rangle\langle/d\rangle)^n$. The basic idea behind the game G is that Juliet is first supposed to play *Call* on $\langle/s\rangle$, to which Romeo should give a witness for the existence of a winning strategy for Player 1 in the tiling game. Juliet should then use calls to the rest of the input string to uncover a flaw in Romeo's witness and thus have a winning strategy in G on w if and only if Romeo cannot prove the existence of a winning strategy for Player 1.

More concretely, Romeo should give as his witness (a slightly modified linearisation of) a *strategy tree* for Player 1, in which nodes are labelled with tiles from two disjoint copies U^1, U^2 of U, representing moves by Player 1 and Player 2. Each node labelled with an element of U^2 has a single child (corresponding to a move of Player 1), each node labelled with an element of U^1 has $|U|$ children (corresponding to the possible moves of Player 2) and the sequence of labels on each path from the root to a leaf (called a *tiling candidate*) either makes up a valid tiling or contains an invalid move by Player 2. Replacement strings for Romeo on a *Call* to $\langle/s\rangle$ will be somewhat extended linearisations of strategy trees, with extensions to be discussed later on.

Proof trees given by Romeo should be restricted (by construction of the replacement transducer R) to only represent tiling candidates that start with the initial tile u_i^1 and contain no horizontal errors. Juliet's task in trying to invalidate Romeo's proof tree therefore consists of uncovering vertical errors and incorrect final tiles (as well as encoding errors within the aforementioned extensions, to be discussed later). To this end, Juliet uses *Call* moves on $\langle/e\rangle$ and $\langle/d\rangle$ tags to select parts of the proof tree given by Romeo that correspond to a fixed column in each tiling candidate represented in the

proof tree; if this column contains an error in at least one of these tiling candidates (which should be detected by the target language DNWA), Juliet wins the game.

We now examine how strategy trees should be encoded. Throughout the rest of the proof, we identify nested words and their forest representations to simplify presentation.

The most intuitive approach to generate strategy trees in which all tiling candidates are horizontally correct would be to fix the replacement transducer in such a way that each node labelled with some $u^1 \in U^1$ has only children labelled by some $v^2 \in U^2$ with $(u, v) \in H$; additionally, to make certain that a strategy tree correctly represents all possible counter-strategies for Player 2, we would have to require that each node corresponding to a move by Player 2 has as children nodes with *all* possible labels v^2 such that $(u, v) \in H$. This intuitive approach leads to two minor problems, though:

- For some tiles u, there may be no tiles v with $(u, v) \in H$. If this happens, we fix the replacement transducer such that it may only follow u up with a special "pseudo-tile" $u_{\text{err}} \notin U$.

- If a tile u is placed at the end of a line, the next tile v does *not* have to fulfil $(u, v) \in H$. In order to account for this fact and still have the replacement transducer produce horizontally correct tiling candidates, we introduce a special *line divider* symbol $\# \notin U$ which may be placed by the replacement transducer at any time it could normally place a node for some player, regardless of its parent tile, and which may be followed up by either *any* tile $u^1 \in U^1$ as a move for Player 1, or by *all* tiles $u^2 \in U^2$ as possible moves for Player 2.

In a correct encoding of a strategy tree, the line divider symbol # should occur as a node label if and only if the corresponding node's depth is a multiple of 2^n. However, the above construction of replacement transducer allows Romeo to construct incorrect encodings of strategy trees. Dealing with attempts by Romeo to cheat in this manner, as well as with vertical errors, is what we discuss next.

We interpret a tree given by Romeo as a tree whose paths from root to leaf encode tiling candidates with 2^n columns and an arbitrary number of lines (i.e. root-to-leaf paths should always have as their length a multiple of 2^n), with the last (i.e. 2^n-th) column in a correctly encoded tiling candidate consisting only of line divider symbols, and no line divider symbol occurring in any other column.

As mentioned above, to check for vertical errors and incorrect encodings, Juliet selects some column number $\ell \in [2^n]$ and removes all nodes not corresponding to some tile in that column, i.e. reduces the original tree t to one containing only the nodes at depths $k \cdot (2^n) + \ell$ for all $k \geqslant 0$. How this is done exactly will be explained later on. Due to the alternation between Players 1 and 2, if the column chosen by Juliet is not the last column, each node with a label from U^1 should only have children with labels from U^2 in this reduced tree, or vice versa.

At first glance, it seems clear that, with the considerations on encoding made above, Juliet can already uncover attempts by Romeo to cheat or give a non-winning strategy, namely

- by picking a column where $\ell \neq 2^n$ and in some path of the resulting tree, there are two directly subsequent labels from U^1 or from U^2 (Romeo tried to cheat by disrespecting the correct player sequence after a #), or
- by picking a column $\ell \neq 2^n$ in which some path contains a #, (Romeo tried to cheat by placing an incorrect #), or
- by picking a column where $\ell \neq 2^n$ and in some path of the resulting tree, there is a tile $u^2 \in U^2$ followed either by u_{err} or some $v^1 \in U^1$ with $(u, v) \notin V$ (the strategy given by Romeo is non-winning due to a horizontal or vertical error), or
- by picking $\ell = 2^n - 1$ when for some path in the resulting tree, its leaf is not u_f (the strategy given by Romeo is non-winning due to a wrong final tile), or
- by picking $\ell = 2^n$ if the last column contains some symbol that is not # in some path (Romeo tried to cheat by not placing a correct #).

All of these conditions can easily be checked by a polynomial-size DNWA.

However, while these conditions are indeed sufficient for checking whether Romeo has given an incorrect or non-winning strategy tree, our construction so far is still too restrictive in that it sometimes fails to recognise a correct winning strategy tree. Consider, for instance, the case where all paths of the strategy tree either contain a vertical error for Player 2 or end with a valid tiling. According to our construction so far, Juliet could still win on a strategy tree of this shape, as our construction requires Romeo to continue giving horizontally correct tiles even after a vertical error has occurred, and these "irrelevant" tiles might not always make it possible to end a line with u_f, thus allowing Juliet to find some path in the strategy tree corresponding to column $2^n - 1$ that doesn't end in u_f.

To address this problem, we further modify the replacement transducer R in such a way that, whenever it is supposed to output a node with label $u^2 \in U^2$ corresponding to a move by Player 2, it may instead nondeterministically choose to output a *marked* version $\hat{u}$ of u^2 instead. This is supposed to indicate that choosing u^2 leads to a vertical error for Player 2, so below the node labelled $\hat{u}$, the replacement transducer may produce a sequence of arbitrary tiles from U^1 for Player 1 and from U^2 for Player 2 (as vertically correct "pseudo-tiles"), terminating with u_f at an odd depth followed by #. If a node labelled u_{err} ever occurs instead of a move of Player 2 due to a horizontal error, it is followed up by a similar path.

In this way, we make it Romeo's responsibility to flag vertical errors for Player 2. This would allow Romeo to cheat by claiming a vertical error when, in fact, there is none, but such attempts at cheating can be penalised by once again adapting the target language in such a way that Juliet wins if she selects a column in which Romeo has falsely flagged a vertical error (i.e. the selected column number ℓ is not 2^n and some path in the corresponding tree contains a marked tile $\hat{v}$ that is not preceded by some $u^1 \in U^1$ with $(u, v) \notin V$).

Next, we examine how Juliet selects a column in the strategy tree given by Romeo. Recall that the input string is $w = (\langle d\rangle\langle e\rangle)^n\langle s\rangle\langle/s\rangle(\langle/e\rangle\langle/d\rangle)^n$, and that the first move by

Juliet is supposed to be a *Call* on $\langle/s\rangle$ to replace $\langle s\rangle\langle/s\rangle$ by an encoding of a strategy tree, in which each path is to be interpreted as a tiling candidate with 2^n-1 columns and an additional column made up of separator symbols #. Juliet now selects one of these 2^n columns by incrementally causing Romeo to delete either all odd-numbered columns or all even-numbered columns (i.e. all nodes at odd or even depths in the strategy tree) until only a single column is left. This is the purpose of the nodes labelled d and e.

By a *Call* to some node labelled e, Juliet causes Romeo to change its label to o (without changing the tree nested below it). A *Call* to a d-node with e-labelled child causes Romeo to delete both of these nodes as well as all nodes at even depths in the strategy tree nested below them; similarly, a *Call* to a d-node with o-child deletes these nodes and all odd-depth nodes in the strategy tree.

To keep Juliet from cheating in this selection process, we have to make certain that she calls *all* d-nodes in left-to-right order. We can easily keep Juliet from leaving uncalled d-nodes by fixing the target language to not contain any strings including the label d; to make certain that she does not skip any d-nodes, the replacement transducer rejects any strings containing a d-node as child of an e- or o-node.

To later check for the right kinds of vertical errors or inconsistencies, we have to keep track of the column selected by Juliet, i.e. the sequence of *Call* moves deleting odd or even columns. The following cases have to be distinguished:

- If Juliet deletes odd columns on all n calls, she selects column 2^n;
- if Juliet deletes even columns as her first call and then only odd columns on the following $n-1$ calls, she selects column 2^n-1;
- otherwise, Juliet selects a column with index $\ell \leqslant 2^n-2$.

To store this information, we use an additional node, initially labelled 0, which Romeo returns as the root of his chosen strategy tree after Juliet's *Call* on $\langle/s\rangle$. This node is then updated throughout the column-deletion process as follows:

- A label 0 signifies that no deletions have been made so far; on a *Call* to d which deletes odd columns, 0 is rewritten to l, on a *Call* deleting even columns, it becomes x
- A label l ("last") signifies that the column Juliet selects may possibly be the 2^n-th line divider column. As long as only odd columns are deleted, the label l remains, if even columns are deleted, l is rewritten to r.
- A label x ("neXt-to-last") signifies that the column Juliet selects may possibly be the (final) column with number 2^n-1. As long as only odd columns are deleted, the label x remains, if even columns are deleted, x is rewritten to r.
- A label r ("standaRd") signifies that the column Romeo selects will definitely not be one of the last two columns, i.e. have a number at most 2^n-2. The label r is not rewritten by any deletion.

To summarise, the game $G = (\Sigma, \Gamma, R, T)$ constructed from the input tiling instance $\mathcal{I} = (U, V, H, u_i, u_f, n)$ is as follows.

The alphabet of G is

$$\Sigma = U \cup \hat{U} \cup U^1 \cup U^2 \cup \{u_{\text{err}}, \#, s, d, e, o, 0, l, x, r\},$$

(where $\hat{U} = \{\hat{u} \mid u \in U\}$, $U^1 = \{u^1 \mid u \in U\}$ and $U^2 = \{u^2 \mid u \in U\}$) with function symbols $\Gamma = \{s, d, e\}$.

The replacement transducer R behaves as follows:

- R rewrites $\langle s\rangle\langle/s\rangle$ into a string of the form $\langle 0\rangle w'\langle/0\rangle$, where w' is the linearisation of a strategy tree as described above, i.e. a tree t with the following properties:
  - t has a root node labelled u_i^1;
  - Each node labelled with some $u^1 \in U^1$ that does not have a node with label from $\hat{U} \cup \{u_{\text{err}}\}$ as ancestor has as its children either a single node labelled $\#$, or nodes either labelled v^2 or $\hat{v}$ for each $v \in U$ with $(u, v) \in H$, or a single node labelled u_{err} if no such v exists.
  - Each node labelled with some $u^2 \in U^2$ that does not have a node with label from $\hat{U} \cup \{u_{\text{err}}\}$ as ancestor has a single child labelled either v^1 for some $v \in U$ with $(u, v) \in H$, or u_{err} if no such v exists, or $\#$.
  - Each node that has a node with label from $\hat{U} \cup \{u_{\text{err}}\}$ as its own label or as an ancestor has a single child labelled either by some $u^1 \in U^1$, or by some $u^2 \in U^2$, or by $\#$.
  - Each node labelled with $\#$ has a single child labelled by some $u^1 \in U^1$, or by $u^2 \in U^2$, or no child at all.
- R rewrites strings of the form $\langle e\rangle v\langle/e\rangle$, for arbitrary $v \in \text{WF}(\Sigma)$, into $\langle o\rangle v\langle/o\rangle$.
- R rewrites strings of the form $\langle d\rangle\langle e\rangle v_1\langle/e\rangle\langle/d\rangle$ into strings v_2 as follows:
  - If $v_1 = \langle 0\rangle v_1'\langle/0\rangle$ (with $v_1' \in \text{WF}(\Sigma)$), then $v_2 = \langle x\rangle v_2'\langle/x\rangle$, where v_2' is derived from v_1' by deleting all nodes at even depths;
  - If $v_1 = \langle l\rangle v_1'\langle/l\rangle$ or $v_1 = \langle x\rangle v_1'\langle/x\rangle$ or $v_1 = \langle r\rangle v_1'\langle/r\rangle$ (with $v_1' \in \text{WF}(\Sigma)$), then $v_2 = \langle r\rangle v_2'\langle/r\rangle$, where v_2' is derived from v_1' by deleting all nodes at even depths.
- R rewrites strings of the form $\langle d\rangle\langle o\rangle v_1\langle/o\rangle\langle/d\rangle$ into strings v_2 as follows:
  - If $v_1 = \langle 0\rangle v_1'\langle/0\rangle$ or $v_1 = \langle l\rangle v_1'\langle/l\rangle$ (with $v_1' \in \text{WF}(\Sigma)$), then $v_2 = \langle l\rangle v_2'\langle/l\rangle$, where v_2' is derived from v_1' by deleting all nodes at odd depths;
  - If $v_1 = \langle x\rangle v_1'\langle/x\rangle$ (with $v_1' \in \text{WF}(\Sigma)$), then $v_2 = \langle x\rangle v_2'\langle/x\rangle$, where v_2' is derived from v_1' by deleting all nodes at odd depths;
  - If $v_1 = \langle r\rangle v_1'\langle/r\rangle$ (with $v_1' \in \text{WF}(\Sigma)$), then $v_2 = \langle r\rangle v_2'\langle/r\rangle$, where v_2' is derived from v_1' by deleting all nodes at odd depths.

- All other strings are rejected by R.

The target language T contains all strings v of the following kinds:

- $v = \langle l \rangle v' \langle / l \rangle$, where the tree represented by v' has some path from root to leaf containing a label different from #.
- $v = \langle r \rangle v' \langle / r \rangle$, where the tree represented by v' has some path from root to leaf
  - containing a label #, or
  - containing two subsequent labels from U^1 or two subsequent labels from U^2, or
  - containing some label u^2 from U^2 followed either by u_{err} or by a u'^1 with $(u, u') \notin V$, or
  - containing some label from $\hat{U}$ that is *not* part of a vertical error.
- $v = \langle x \rangle v' \langle / x \rangle$, where the tree represented by v' has some path from root to leaf
  - containing a label #, or
  - containing two subsequent labels from U^1 or two subsequent labels from U^2, or
  - containing some label u^2 from U^2 followed either by u_{err} or by a u'^1 with $(u, u') \notin V$, or
  - containing some label from $\hat{U}$ that is *not* part of a vertical error
  - containing *no* vertical error, *no* label u_{err} and ending with a label different from u_f^1, u_f^2.

It is relatively easy to see (but tedious to prove formally) that this construction is possible in polynomial time and that JULIET indeed has a winning strategy in G on w if and only if Player 1 has no winning strategy on the tiling instance $\mathcal{I}$. □

For the upper bound, we first prove decidability in 2-EXPTIME for a "purely NWT-based" problem, to which we will later reduce JWIN.

Definition 9.6. The *alternating iterated transduction* problem for non-deleting NWT is defined as the following decision problem:

AIT(NWT)	
Given:	A string $w \in \text{WF}(\Sigma)$, number k (given in unary), DNWA A, and pairs of non-deleting NWTs $(T_{1,0}, T_{1,1}), \ldots, (T_{k,0}, T_{k,1})$
Question:	Is there an $i_1 \in \{0,1\}$ such that for every $w_1 \in T_{1,i_1}(w)$ there exists $i_2 \in \{0,1\}$ such that for every $w_2 \in T_{2,i_2}(w_1)$ ... there exists $i_k \in \{0,1\}$ such that for every $w_k \in T_{k,i_k}(w_{k-1})$ it holds that $w_k \in L(A)$?

Proposition 9.7. AIT(NWT) $\in$ 2-EXPTIME

Proof. We assume, without loss of generality, that each transducer $T_{j,i}$ (for $j \in [k], i \in \{0,1\}$) is in normal form, i.e. that each (reading or ϵ-) transition of each transducer produces exactly one output tag (cf Lemma 8.11).

The idea behind this proof is to eliminate the existential quantification in the problem setting by constructing an NWT T that simulates in parallel *both* transducers $T_{j,1}$ and $T_{j,2}$ at each level $j \in [k]$. More concretely, T takes as input a string w and outputs a 2^k-tuple of strings, each component of which corresponds to a sequence of existential choices $i_1 \dots i_k \in \{0,1\}^k$ of transducers, while the non-determinism in T simulates universal choice. It then holds (as we will prove after the construction of T) that the condition of AIT(NWT) is fulfilled if and only if for each possible transduct w' of w by T, at least one of the 2^k component strings is contained in (an appropriate modification of) the target language $L(A)$.

To construct T, we construct from $T_{j,0}$ and $T_{j,1}$ for each level $j \in [k]$ a transducer T_j that takes as input a nested string w_{j-1} over an alphabet of 2^{j-1}-tuples of alphabet symbols and outputs a nested string w_j over 2^j-tuples, with the intuition being that T_j simulates one run of both $T_{j,0}$ and $T_{j,1}$ on each of the 2^{j-1} input strings encoded in w_{j-1} to produce a total of $2 \cdot 2^{j-1} = 2^j$ output strings, which are encoded in w_j. From all of the transducers T_j for all $j \in [k]$, we then use Proposition 8.12 to construct T as the transducer for $T_k \circ \dots \circ T_1$.

The main difficulty in the construction of each T_j is the fact that, while both $T_{j,0}$ and $T_{j,1}$ should be simulated for exactly one run on each of the components on the input string, all of these runs should be independent of each other. For instance, if T_2 wanted to simulate some run ρ of $T_{2,0}$ on the first and a different run ρ' of $T_{2,0}$ on the second input string component, ρ might start by reading the first tag from the input string while ρ' starts with an ϵ-transition and reads the first input tag afterwards. In such a situation, ρ' would produce an output before the output of ρ even starts. We therefore need to construct each T_j in such a manner that runs are *synchronised*, producing an output for *all* components of the output string, even if only *one* run of some transducer T_{j,i_j} calls for producing an output.

To address this problem, we introduce a special *blank* symbol $\sqcup \notin \Sigma$ and construct each T_j in such a way that reading transitions of T_j simulate synchronous reading transitions of $T_{j,0}$ and $T_{j,1}$ on *all* symbols of the next input tuple; ϵ-transitions of T_j, on the other hand, simulate only *one* of $T_{j,0}$ or $T_{j,1}$ on only *one* of the 2^{j-1} input components while outputting tuples consisting of exactly one symbol from Σ in the corresponding component and $\sqcup$ symbols in all the others. To keep this construction consistent and allow for later transductions, this also means that all transducers $T_{j,i}$ ($j \in [k]$, $i \in \{0,1\}$) have to be modified in such a way that, on reading an opening (closing) $\sqcup$ tag, they always output an opening (closing) $\sqcup$ tag and do not change states.

More formally, if (for each $j \in [k]$ and $i \in \{0,1\}$) $T_{j,i} = (Q_{j,i}, P_{j,i}, P^{\epsilon}_{j,i}, \Sigma, \delta_{j,i}, q^0_{j,i}, F_{j,i})$, then let first $T'_{j,i}$ be an extension of $T_{j,i}$ to strings containing $\sqcup$ tags that ignores and reproduces these tags as described above, i.e. we define the non-deleting NWT $T'_{j,i}$ by $T'_{j,i} = (Q_{j,i}, P_{j,i} \cup \{p_\sqcup\}, P^{\epsilon}_{j,i}, \Sigma \cup \{\sqcup\}, \delta'_{j,i}, q^0_{j,i}, F_{j,i})$, where $p_\sqcup \notin P_{j,i}$ and $\delta'_{j,i}$ consists of $\delta_{j,i}$ extended by transitions $(q, \langle\sqcup\rangle, q, p_\sqcup, \langle\sqcup\rangle)$ and $(q, p_\sqcup, \langle/\sqcup\rangle, q, \langle/\sqcup\rangle)$ for each $q \in Q_{j,i}$.

For any $w \in \mathrm{WF}(\Sigma \cup \{\sqcup\})$, let $\mathrm{Strip}(w) \in \mathrm{WF}(\Sigma)$ denote the nested string obtained from w by deleting all $\sqcup$-labelled tags, and for any set $S \subseteq \mathrm{WF}(\Sigma \cup \{\sqcup\})$, let $\mathrm{Strip}(S) \stackrel{\text{def}}{=} \{\mathrm{Strip}(w) \mid w \in S\}$. Then, it is clear that for any $w \in \mathrm{WF}(\Sigma \cup \{\sqcup\})$, it holds that $\mathrm{Strip}(T'_{j,i}(w)) = T_{j,i}(\mathrm{Strip}(w))$.

Now we examine in detail the construction of T_j from $T'_{j,0}$ and $T'_{j,1}$. Let

$$T_j = (Q_j, P_j, P_j^\epsilon, \Sigma_j^{\text{in}} \cup \Sigma_j^{\text{out}}, \delta_j, q_j^0, F_j),$$

where the linear and hierarchical state sets as well as initial and final states simply derive from a 2^{j-1}-fold product construction of $T'_{j,0}$ and $T'_{j,1}$, i.e.

$$Q_j = (Q_{j,0} \times Q_{j,1})^{2^{j-1}},$$

$$P_j = ((P_{j,0} \cup \{p_\sqcup\}) \times (P_{j,1} \cup \{p_\sqcup\}))^{2^{j-1}},$$

$$q_j^0 = (q_{j,0}^0 \times q_{j,1}^0)^{2^{j-1}},$$

and

$$F_j = (F_{j,0} \times F_{j,1})^{2^{j-1}}.$$

The working alphabet of T_j consists of input alphabet $\Sigma_j^{\text{in}} = (\Sigma \cup \{\sqcup\})^{2^{j-1}}$ and output alphabet $\Sigma_j^{\text{out}} = (\Sigma \cup \{\sqcup\})^{2^j}$. The construction of P_j^ϵ is similar, but has to be adjusted slightly; since ϵ-transitions of T_j are supposed to simulate $T_{j,0}$ or $T_{j,1}$ on only *one* component of the output string and the other components have to be filled in with $\sqcup$ symbols, we add a new hierarchical ϵ-state $p_\sqcup^\epsilon \notin P_{j,0} \cup P_{j,1}$ and set $P_j^\epsilon = ((P_{j,0}^\epsilon \cup \{p_\sqcup^\epsilon\}) \times (P_{j,1}\epsilon \cup \{p_\sqcup^\epsilon\}))^{2^{j-1}}$.

In accordance with the above intuition, we construct the transition relation δ_j of T_j. Reading transitions in δ_j (i.e. transitions that read one input symbol and produce one output symbol) are pretty much products of transitions from $\delta'_{j,0}$ and $\delta'_{j,1}$. That is, δ_j contains an opening transition which, for $\ell \in [2^j]$, starts with linear state q, ends in state q' and produces hierarchical state q in the ℓ-th position of the corresponding state tuples while reading $\langle a\rangle$ in the $\lfloor \frac{\ell}{2} \rfloor$-th position of the input tuple and writing $\langle b\rangle$ in the ℓ-th position of the output tuple if the transition $(q, \langle a\rangle, q', p, \langle b\rangle)$ is in $\delta'_{j,0}$ (for odd ℓ) or in $\delta'_{j,1}$ (for even ℓ).[1] Closing reading transitions in δ_j are constructed accordingly.

As for ϵ-transitions, δ_j contains an opening transition $(\overline{q}, \langle\epsilon\rangle, \overline{q}', \overline{p}, \overline{a})$ if there is an $\ell \in [2^j]$ such that (with $\overline{x}_i$ denoting the i-th component of a tuple $\overline{x}$)

- for all $\ell' \neq \ell$, it holds that $\overline{q}_{\ell'} = \overline{q}'_{\ell'}$, $\overline{p}_{\ell'} = p_\sqcup^\epsilon$ and $\overline{a}_{\ell'} = \sqcup$, and
- for $\overline{q}_\ell = q$, $\overline{q}'_\ell = q'$, $\overline{p}_\ell = p$ and $\overline{a}_\ell = a$, there is a transition $(q, \langle\epsilon\rangle, q', p, \langle a\rangle)$ in $\delta'_{j,0}$ (for odd ℓ) or in $\delta'_{j,1}$ (for even ℓ),

[1] At first glance, it may seem more intuitive to associate even positions with $T_{j,0}$ and odd positions with $T_{j,1}$, but seeing as ℓ is a number between 1 and 2^j which will later be encoded as $\ell - 1$ by a binary sequence of length j, the association described here is indeed the more useful one.

and accordingly for closing ϵ-transitions.

With this construction, T_j is indeed an NWT in normal form (i.e. fulfils the ϵ-consistency, well-formedness and synchronisation properties). This directly implies that T_j transduces well-nested strings over Σ_j^{in} into well-nested strings over Σ_j^{out}. We now identify nested strings over tuples of alphabet symbols with tuples of nested strings as follows: For some j, let $\overline{w} = \overline{w}^1 \cdots \overline{w}^n$ be a well-nested string over $(\Sigma \cup \{\sqcup\})^{2^j}$, and for each $\overline{w}^i \in \langle(\Sigma \cup \{\sqcup\})^{2^j}\rangle$, let $\overline{w}^i_\ell$ denote its ℓ-th component interpreted as an opening tag (and analogously for closing tags). Then the nested string $\overline{w}_\ell$ is defined as $\overline{w}_\ell = \overline{w}^1_\ell \cdots \overline{w}^n_\ell$. It is clear that, if $\overline{w} \in \mathrm{WF}((\Sigma \cup \{\sqcup\})^{2^j})$, then $\overline{w}_\ell \in \mathrm{WF}(\Sigma \cup \{\sqcup\})$ for each ℓ.

Identifying strings of tuples with tuples of strings in this way, it is easy (if tedious) to prove that, for each j and each string $\overline{w} \in \mathrm{WF}(\Sigma_j^{\text{in}})$, interpreted as a 2^{j-1}-tuple $(\overline{w}_1, \ldots, \overline{w}_{2^{j-1}})$ of nested strings, it holds that

$$\mathrm{Strip}(T_j(w)) = \prod_{\ell=1}^{2^{j-1}} T_{j,0}(\mathrm{Strip}(\overline{w}_\ell)) \times T_{j,1}(\mathrm{Strip}(\overline{w}_\ell)),$$

where the Strip operator is applied component-wise, i.e.

$$\mathrm{Strip}((\overline{w}_1, \ldots, \overline{w}_n)) \stackrel{\text{def}}{=} (\mathrm{Strip}(\overline{w}_1), \ldots, \mathrm{Strip}(\overline{w}_n)).$$

In other words, disregarding $\sqcup$ tags, every run of T_j on a 2^{j-1}-tuple of nested words simulates one run each of $T_{j,0}$ and $T_{j,1}$ on each of its component strings, and all combinations of such component runs can be simulated by a run of T_j.

By a simple induction argument, it follows that each transducer of the form $T_j \circ \cdots \circ T_1$ completely describes all possible series of existential choices up to the j-th level. More precisely, denoting by $S_\ell = \{\overline{w}_\ell \mid \overline{w} \in S\}$ the set of all ℓ-th component strings in a set S of tuples of nested strings, we get that for every $j \leqslant k$, $w \in \mathrm{WF}(\Sigma)$ and $\ell \in [2^j]$, it holds that $\mathrm{Strip}((T_j \circ \cdots \circ T_1(w))_\ell) = T_{j,i_j} \circ \cdots T_{1,i_1}(w)$, where $i_1 \cdots i_j \in \{0,1\}^j$ is the binary representation of the number $\ell - 1$.

To use the transducer $T = T_k \circ \cdots \circ T_1$ for solving the alternating iterated transduction problem as described initially, we now need to prove that the defining property of AIT(NWT) (i.e. "There is an $i_1 \in \{0,1\}$ such that for every $w_1 \in T_{1,i_1}(w)$...") is equivalent to the following: For every $\overline{w} \in T(w)$, there is an $\ell \in [2^k]$ such that $\mathrm{Strip}(\overline{w}_\ell) \in L(A)$. We denote this property by (*). As we have already seen, the existence of ℓ in property (*) is equivalent to the existence of an index sequence $i_1 \cdots i_k \in \{0,1\}^k$ representing $\ell - 1$ in binary, so we basically need to prove that the alternation between existential and universal choices in the defining property of AIT(NWT) is equivalent to a single universal choice of iterated transducts for each possible index sequence of existential choices, followed by a single existential choice of index sequence.

The proof of this equivalence is by induction; the central step is proving, for any set S of nested strings, the equivalence of

$$\forall \overline{w} \in T_{j-1} \circ \cdots \circ T_1(w) \exists \ell \in [2^{j-1}] \exists i_j \in \{0,1\} \forall w' \in T_{j,i_j}(\mathrm{Strip}(\overline{w}_\ell)) : w' \in S$$

and

$$\forall \overline{w}' \in T_j \circ T_{j-1} \circ \ldots \circ T_1(w) \exists \ell' \in [2^j] : \mathrm{Strip}(\overline{w}'_{\ell'}) \in S,$$

which follows primarily from the fact that T_j simulates both $T_{j,0}$ and $T_{j,1}$ as was shown before.

To sum up the proof thus far, we have constructed from an AIT(NWT) instance a transducer T such that the original instance is a positive one if and only if for the input string w it holds that all transducts in $T(w)$ have a component string that is in $L(A)$ when stripped of all $\sqcup$-labelled tags. It remains to be seen how we can check for this property in doubly exponential time.

To that end, let A' be a modification of A that ignores $\sqcup$-labelled tags, constructed from A using a similar construction to the one for each T_{j,i_j}. From A', we can construct a DNWA B that gets as input a nested string of tags over 2^k-tuples of symbols from $\Sigma \cup \{\sqcup\}$, simulates a copy of A' on each of the 2^k components and accepts if and only if at least one of its components accepts. It then holds that the original AIT(NWT) instance is a positive one if and only if $T(w) \subseteq L(B)$, which yields an instance of a type checking problem. Each level transducer T_j is of size at most $\mathcal{O}((|T_{j,0}| \cdot |T_{j,1}|)^{2^k})$, therefore T is of doubly exponential size, and so is B. Since the type checking problem with target DNWA is decidable in PTIME by Theorem 8.18(b), this yields a 2-EXPTIME algorithm for AIT(NWT), as was to be proven. □

A reduction to AIT(ϵ-NWT), along with Proposition 9.5 now proves Theorem 9.4.

Theorem 9.4 ***(restated).*** *For the class of replay-free games with NWT,* JWIN *is* 2-EXPTIME*-complete.*

Proof. The lower bound was proven as Proposition 9.5. We prove a matching upper bound by reduction to AIT(ϵ-NWT), which is in 2-EXPTIME according to Proposition 9.7.

Let $G = (\Sigma, \Gamma, R, T)$ be a game with NWT replacement, and let w be an input string for G. Let furthermore $k \leqslant \frac{|w|}{2}$ be the number of occurrences of closing tags from $\langle/\Gamma\rangle$ in w.

The idea behind the reduction to AIT(ϵ-NWT) is taking k rounds of alternating transduction, where the j-th round (with transducers $T_{j,0}, T_{j,1}$) corresponds to the replay-free subgame on the j-th function symbol in w (in left-to-right order). The choice between transducers $T_{j,0}$ and $T_{j,1}$ models JULIET'S choice between *Read* and *Call*; to that end, $T_{j,0}$ basically does not change its input string at all, while $T_{j,1}$ simulates the replacement transducer R on the substring that JULIET chose to be replaced. The only minor technical difficulty in this construction is the fact that, in the game G, the transducer R only rewrites the called substring, while each $T_{j,i}$ rewrites the entirety of the current string. This difficulty can be solved by some minor modifications, which we now examine.

The input string w' for AIT(ϵ-NWT) is derived from w by replacing, in left-to-right order, each substring $\langle f\rangle v\langle/f\rangle$ of w by $\langle j\rangle\langle f\rangle v\langle/f\rangle\langle/j\rangle$, where $\langle/f\rangle$ is the j-th closing function tag in w and $j \notin \Sigma$ for each $j \in [k]$. In other words, the substring on which JULIET has to make her j-th strategy decision is encapsulated in j-tags.

For each $j \in [k]$, the transducer $T_{j,0}$ simply deletes the $\langle j\rangle$ and $\langle/j\rangle$ tags from its input, leaving it otherwise unchanged. The transducer $T_{j,1}$, on the other hand, also directly outputs its input until it reaches the $\langle j\rangle$ tag. It deletes this tag and then starts

simulating the replacement transducer R. Once $T_{j,1}$ reaches the $\langle/j\rangle$ tag, it deletes that tag as well and stops its simulation, rejecting its input if R has not reached an accepting state. Afterwards, $T_{j,1}$ simply outputs its input again. Note that the simulation of R in $T_{j,1}$ will never receive as input any tags with labels not in Σ, as all such tags have a label strictly less than j and have therefore already been removed by earlier transductions.

The target DNWA A for AIT(ϵ-NWT) is simply the target DNWA $A(T)$ of G.

It is easy to see that k, w' and each $T_{j,0}$ for $j \in [k]$ can be computed from w in polynomial time, as can each $T_{j,1}$ from R. As the alternating transduction simulates the replay-free game, it is also clear that JULIET has a replay-free winning strategy on w in G if and only if the constructed instance for AIT(ϵ-NWT) is a positive one, which concludes the reduction. □

9.3. Games with ϵ-free NWT

As seen in the previous section, the winning problem for JULIET on games with general NWT-based replacement is undecidable if any replay is allowed, and even the replay-free case is of quite high complexity. The lower bound proofs for these results (Theorem 9.3 and Proposition 9.5) rely strongly on the fact that ROMEO can choose replacement strings of arbitrary length.

In this section, we examine what happens if we eliminate replacement strings of unbounded length as a source of undecidability, i.e. if we restrict games to use *ϵ-free* NWT as a replacement mechanism. As it turns out, this restriction makes the winning problem for bounded replay decidable and lowers the complexity of the replay-free case; however the complexity of these decidable cases remains quite high (even forbiddingly so for the bounded replay case), and the winning problem remains undecidable with unbounded replay.

Theorem 9.8. *For the class of games with ϵ-free NWT and unbounded replay,* JWIN *is undecidable.*

Proof. The proof is by a straightforward reduction from the halting problem for Turing machines with an empty input. From a given Turing machine M with working alphabet Σ and state set Q, we construct a game G and string w such that JULIET has a winning strategy on w in G if and only if M halts on an empty input. Without loss of generality, we assume that M always moves its head to the left-most used tape cell before halting.

The basic idea behind the reduction is encoding configurations of M by strings nested below a root $r \notin \Sigma$ and using *Call* moves by JULIET to $\langle/r\rangle$ to simulate moves of M. The input string is $w = \langle r\rangle\widehat{v}_0\langle/r\rangle$, where $\widehat{v}_0$ represents the initial configuration of M, and any time JULIET plays *Call* on $\langle/r\rangle$ in some string $\langle r\rangle\widehat{v}_i\langle/r\rangle$ (with $\widehat{v}_i$ representing some configuration of M), that string gets replaced by $\langle r\rangle\widehat{v}_{i+1}\langle/r\rangle$, where $\widehat{v}_{i+1}$ represents the successor configuration of the one represented by $\widehat{v}_i$. The target language of G is constructed to contain all strings of the form $\langle r\rangle\widehat{v}_h\langle/r\rangle$, where $\widehat{v}_h$ represents a halting configuration of M. This way, JULIET has a winning strategy of *Call* depth k on the input string if and only if M halts on the empty input within at most k steps.

We represent configurations of M in the same way as in the proof of Theorem 9.3: a flat string $(x_1, -)\cdots(x_{k-1}, -)(x_k, q)(x_{k+1}, -)\cdots(x_m, -)$ over the alphabet $\Sigma \times (Q \cup \{-\})$ denotes that the content of M's working tape is $x_1 \cdots x_m$, with the head of M being on the tape's k-th cell and M being in state q, and these flat strings are represented as nested strings using the standard nested string representation. Again, by our assumption on the shape of M's halting configurations, the flat string encoding of a halting configuration is of the form $(\Sigma \times \{h\})(\Sigma \times \{-\})^*$ for the halting state h of M.

From M, we can easily construct a ϵ-free NWT implementing M's transition function on configurations represented in this way.[2] A DNWA accepting all strings $\langle r\rangle v\langle/r\rangle$ in which v represents a halting configuration for M is similarly easy to construct. Finally, it is clear that JULIET has a winning strategy on the input string w using at most k *Call* moves if and only if M reaches a halting configuration from its initial configuration within at most k steps, which completes the proof. □

Different from games with general NWT, the winning problem for JULIET in games with ϵ-free NWT and fixed *Call* depth is decidable; however, the complexity of deciding JWIN is already non-elementary for *Call* depth 2.

Theorem 9.9. *For the class of games with ϵ-free NWT and Call depth bounded by $d \geqslant 2$,* JWIN *is decidable, but not decidable in elementary time.*

The upper bound proof for Theorem 9.9 is surprisingly simple, basically just using an enumeration of all possible plays in a given game. The lower bound proof, however, is considerably more intricate. Its main part is a proof that, for games with call depth 2, JWIN is CO-k-NEXPTIME-hard for all $k \geqslant 1$. The following result serves as set-up for this proof.

We recall that $\mathrm{Exp}(k, n)$ is the k-fold exponential tower function in n, defined recursively by $\mathrm{Exp}(0, n) = n$ and $\mathrm{Exp}(k, n) = 2^{\mathrm{Exp}(k-1,n)}$ for all integers $k > 0$ and $n \geqslant 0$.

Lemma 9.10. *An input nested word of length $2 \cdot (n + 2k - 1)$ can be transformed into a word of length $2 \cdot Exp(k, n)$ by a game of Call depth 2 with deterministic ϵ-free NWT replacement.*

Proof. Choose as input the nested word

$$w = \langle k-1\rangle\langle k-2\rangle\cdots\langle 1\rangle\langle c_0\rangle^n\langle c_1\rangle\cdots\langle c_k\rangle\langle/c_k\rangle\cdots\langle/k-1\rangle.$$

The tree represented by w is obviously a path of length $n + 2k - 2$. Play proceeds in k *rounds* as follows: In round $i \in [k]$, JULIET plays *Call* on each node labelled c_{i-1} in bottom-up (i.e. left-to-right) order. Each such *Call* move deletes the called c_{i-1} node and doubles the number of c_i nodes below it (i.e. replaces each $\langle c_i\rangle$ by $\langle c_i\rangle\langle c_i\rangle$ and $\langle/c_i\rangle$ by $\langle/c_i\rangle\langle/c_i\rangle$). Afterwards, if $i < k$, JULIET plays *Call* on the node labelled i, which deletes that node, attaches its child path to its $i + 1$-labelled parent and allows JULIET to play

[2] Note that this ϵ-free NWT is functional, so G is basically a "solitaire" game for JULIET, where ROMEO does not get to make any choices. In fact, the replacement relation can even be implemented using a deterministic finite-state transducer that inserts at most two symbols with each transduction.

again on its child path, since this is the first time JULIET has played *Call* on the node labelled i. By an induction argument, it is easy to show that at the conclusion of round i, the current string contains exactly $2 \cdot \mathrm{Exp}(i, n)$ tags labelled c_i (half of them opening, the other half closing tags). □

Proposition 9.11. *For each $k \geqslant 1$, it holds that for the class of games with ϵ-free NWT, and Call depth bounded by 2,* JWIN *is hard for* CO-k-NEXPTIME.

Proof. Let $k \geqslant 1$. We show CO-k-NEXPTIME-hardness by reduction from the complement of the k-NEXPTIME-complete problem k-EXPONENTIAL TILING of, given a tile set U, vertical and horizontal constraints V, H, initial and final tile u_i, u_f and unary number n, determining whether there exists a valid tiling of height $\mathrm{Exp}(k, n)$ and width $\mathrm{Exp}(k, n) - 1$.[3]

We construct from an instance of k-EXPONENTIAL TILING a game G and a string w such that JULIET has a winning strategy in G starting at w if and only if there exists *no* valid tiling of the given size. We first give a rough overview over the game before describing the construction of G and w in detail.

Play proceeds in three phases:

- In the first phase, the construction from Lemma 9.10 is used to transform the polynomial-sized input string into a *seed string* of size $\mathcal{O}(\mathrm{Exp}(k-1, n))$. This phase is deterministic in the sense that we will design the replacement transducer and target language in a way such that neither JULIET nor ROMEO get to make any choices in phase 1.

- In phase 2, ROMEO constructs from this seed string a *tiling candidate*, i.e. a (representation of a) potential tiling of size $\mathrm{Exp}(k, n) \times (\mathrm{Exp}(k, n) - 1)$. In this phase, JULIET still doesn't get to make any choices; all she is supposed to do is to call certain nodes in order to allow ROMEO to construct the tiling candidate. By fixing the replacement transducer in the proper way, we can ensure that this tiling candidate starts with the initial tile.

- In the final phase, JULIET tries to show that the tiling candidate ROMEO constructed in phase 2 does not represent a valid tiling. Since horizontal errors are easily uncovered using the target automaton, JULIET'S main task in this phase is pointing out vertical or encoding errors. To this end, she repeatedly forces ROMEO to delete either all even or all odd columns in his tiling candidate until only a single column is left; this column is then checked for errors by the target language automaton.

Note that, as in most lower bound proofs so far, during all three phases, we require JULIET to stick to a certain "game plan", in which she only calls certain nodes in a predetermined order. This can be enforced by constructing the replacement transducer and

[3] This definition differs slightly from that in Section 2.5 in that the width of desired tilings is smaller by 1; this is to simplify the presentation of the reduction and obviously does not change the problem's computational complexity.

target language automaton in an appropriate way. We will first examine phase-by-phase how the game G is constructed, assuming JULIET'S compliance, and later describe how G has to be modified to prevent deviations from the game plan.

Let $v_{k,n}$ be the sequence of opening tags from the proof of Lemma 9.10, i.e. $v_{k,n} = \langle k-1\rangle\langle k-2\rangle\cdots\langle 1\rangle\langle c_0\rangle^n\langle c_1\rangle\cdots\langle c_k\rangle$, and let $\overline{v_{n,k}}$ be the complementary sequence of closing tags. The input string w, then, is of the form

$$w = \langle\text{start2}\rangle\langle\text{mv}\rangle\langle\text{dbl}\rangle\langle\text{cp1}\rangle v_{k-1,n}\langle r\rangle\langle p\rangle\langle/p\rangle\langle/r\rangle\overline{v_{k-1,n}}\langle/\text{cp1}\rangle\langle/\text{dbl}\rangle\langle/\text{mv}\rangle\langle/\text{start2}\rangle.$$

The purpose of each tag will be explained when it is first used in the game.

At the start of phase 1, JULIET uses the procedure described in the proof of Lemma 9.10 to rewrite w into the following string of $(k-1)$-fold exponential size:

$$\langle\text{start2}\rangle\langle\text{mv}\rangle\langle\text{dbl}\rangle\langle\text{cp1}\rangle\langle c_{k-1}\rangle^\ell\langle r\rangle\langle p\rangle\langle/p\rangle\langle/r\rangle\langle/c_{k-1}\rangle^\ell\langle/\text{cp1}\rangle\langle/\text{dbl}\rangle\langle/\text{mv}\rangle\langle/\text{start2}\rangle,$$

where $\ell = \text{Exp}(k-1, n)$. Afterwards, JULIET plays her first *Call* on the node labelled cp1 (where "cp" stands for "copy"), allowing her a replay on $\langle c_{k-1}\rangle^\ell\langle r\rangle\langle/r\rangle\langle/c_{k-1}\rangle^\ell$. In this replay, she calls every c_{k-1} in left-to-right order; each such *Call* replaces the called c_{k-1} by c'_{k-1}, replaces $\langle r\rangle$ by $\langle r\rangle\langle d\rangle\langle e\rangle$ and replaces $\langle/r\rangle$ by $\langle/e\rangle\langle/d\rangle\langle/r\rangle$. After this rewriting, the r-labelled node has a child path consisting of ℓ alternating d- and e-nodes and terminating in a single p-labelled node.

Next, JULIET plays *Call* on the node labelled dbl (for "double"), which rewrites the c'_{k-1}-labelled path into a c'_{k-1}-labelled path of double length, i.e. the rewriting transducer replaces each $\langle c'_{k-1}\rangle$ by $\langle c'_{k-1}\rangle\langle c'_{k-1}\rangle$ and each $\langle/c'_{k-1}\rangle$ by $\langle/c'_{k-1}\rangle\langle/c'_{k-1}\rangle$. With her next call to mv (for "move"), JULIET gets another replay on this path of length 2ℓ, calling each c'_{k-1} node in left-to-right order, which causes ROMEO to delete that node and insert a single c (for "create") node below the bottom p-labelled node. Finally, JULIET plays *Call* on the root node labelled start2 to start the second phase of the game; as his reply to this *Call*, ROMEO replaces the start2-node by a v-node and inserts a node labelled u_i as a leaf at the very bottom of the path. This ends phase 1 and leaves JULIET to play a replay-free game on the string

$$w_1 = \langle v\rangle\langle r\rangle(\langle d\rangle\langle e\rangle)^\ell\langle p\rangle\langle c\rangle^{2\ell}\langle u_i\rangle\langle/u_i\rangle\langle/c\rangle^{2\ell}\langle/p\rangle(\langle/e\rangle\langle/d\rangle)^\ell\langle/r\rangle\langle/v\rangle.$$

In the second phase, ROMEO is supposed to set up a tiling candidate. This candidate will be encoded within the leaves of the current string, i.e. in siblings of the initial u_i-labelled leaf. To this end, JULIET calls each of the c-nodes in bottom-up order. On each such call, ROMEO deletes the called node and doubles the number of leaves in the current string. More precisely, the replacement transducer may replace each $\langle/u\rangle$ (for some $u \in U$) either by $\langle/u\rangle\langle u'\rangle\langle/u'\rangle$ for some $u' \in U$, or by $\langle/u\rangle\langle\#\rangle\langle/\#\rangle$ for a divider symbol $\# \notin U$. Since each *Call* to a c-node doubles the length of the current tiling candidate, it is clear that when all c-nodes have been called (and phase 2 ends), the current string is of the form

$$w_2 = \langle v\rangle\langle r\rangle(\langle d\rangle\langle e\rangle)^\ell\langle p\rangle v_t\langle/p\rangle(\langle/e\rangle\langle/d\rangle)^\ell\langle/r\rangle\langle/v\rangle,$$

where v_t, the final tiling candidate, is a string of nesting depth zero consisting a total of $2^{2\ell} = (2^\ell)^2 = \mathrm{Exp}(k,n)^2$ pairs of corresponding opening and closing tags with labels from $U \cup \{\#\}$, beginning with $\langle u_i\rangle\langle/u_i\rangle$.

We interpret the tiling candidate v_t as the concatenation of $\mathrm{Exp}(k,n)$ lines of length $\mathrm{Exp}(k,n)$ each. To encode a valid tiling of size $\mathrm{Exp}(k,n) \times (\mathrm{Exp}(k,n) - 1)$, we expect v_t to be of the form $((\langle U\rangle\langle/U\rangle)^{\mathrm{Exp}(k,n)-1}\#)^{\mathrm{Exp}(k,n)}$, i.e. we expect Romeo to use the symbol # as a line separator, and only as such.

It should be clear from the construction that the string v_t represents a concatenation of several (possibly empty) substrings of tiles, separated by #, the first of which starts with u_i. These substrings may potentially contain horizontal errors; if this is the case, Juliet plays *Call* on the final $\langle/v\rangle$ of w_2, which causes Romeo to relabel v into h. This label then tells the target automaton to accept if and only if its input string is of the form of w_2 with some tile substring of v_t containing a horizontal error. This is obviously feasible using a polynomial-sized DNWA.

Therefore, if Romeo has given a tiling candidate with a horizontal error, Juliet wins the game at this point; if play proceeds further (and Juliet leaves the v-tags unchanged), we can assume that there are no horizontal errors in v_t, i.e. that all substrings of tiles represented in v_t adhere to the horizontal constraints.

The tiling candidate encoded by v_t may, however, still contain one or more of the following types of errors:

- *Vertical error*: Two vertically adjacent tiles u, u' with $(u, u') \notin V$;
- *Incorrect line lengths*: Strictly more or less than $\mathrm{Exp}(k,n) - 1$ symbols from U between two subsequent #;
- *Incorrect final tile*: The last symbol from U in v_t is not u_f.

The main observation needed for this part of the reduction is the following: if a tiling candidate does not represent a correct tiling, then at least one of these errors can be found by examining just a single column of the tiling candidate. If a tiling candidate contains a vertical error, then there is a column containing two subsequent tiles u, u' with $(u, u') \notin V$; if some line length is incorrect, then the last column contains some symbol other than #, or there is a # in a column that is *not* the last column; and finally, if the tiling candidate does not end with u_f, then the next-to-last column (i.e. column number $\mathrm{Exp}(k,n) - 1$) does not end with u_f. It is easy to see that, once a single column of v_t has been isolated, all three of these conditions can easily be checked using a polynomial-sized DNWA.

Juliet's task on the string w_2 therefore consists of isolating a single column containing an error. If and only if she manages to do so, she wins the game (i.e. the target DNWA checking for the existence of one of the above kinds of errors accepts).

To isolate a single column, Juliet plays *Call* moves on all nodes labelled d (for "destroy") in bottom-up order. Each such *Call* removes the called d-node and its child, and forces the replacement transducer to either delete all even-numbered columns or all odd-numbered columns by deleting every second node in v_t. Juliet makes the choice

of whether to delete all even-numbered or all odd-numbered columns by playing either *Read* or *Call* on the e-labelled node just below the d-node she is to call next; a *Read* move leaves the label e ("even") intact, causing the replacement transducer to delete all even-numbered columns on JULIET'S *Call* to the d-node above, while a *Call* move relabels e into o ("odd"), causing all odd-numbered columns to be deleted analogously. Each such deletion step halves the number of remaining columns, which means that after ℓ deletion moves, only a single column of length $\mathrm{Exp}(k, n)$ remains of v_t.

From the above considerations concerning error types, it is clear that some additional information needs to be tracked through the deletion phase, to determine whether the column chosen by JULIET is the last, next-to-last, or some other column. This is the purpose of the node labelled p ("position"), which is rewritten depending on the sequence of JULIET'S choices of even and odd columns:

- To reach the last column, JULIET must successively remove only odd-numbered columns. Therefore, on the first *Call* to a d-node with o-child (and p-grandchild), p is rewritten into l ("last"). Any *Call* to a d-node with e-child and l-grandchild rewrites l into s ("standard", i.e. the column to be checked is not a special case), while calling a d-node with o-child leaves the label l as is.

- To reach the next-to-last column, JULIET must remove all even-numbered columns in the first step and successively remove only odd-numbered columns after that. Therefore, on the first *Call* to a d-node with e-child (and p-grandchild), p is rewritten into x ("neXt-to-last"). Any *Call* to a d-node with o-child and x-grandchild leaves the label x intact, while a *Call* to a d-node with e-child and x-grandchild relabels x to s.

Using the label of the rewritten p-node as an indicator, a polynomial-sized DNWA can now easily check whether a column chosen by JULIET contains some error. Assuming that JULIET is restricted to strategies that follow the described order of *Call* moves, it is straightforward (if tedious) to prove that JULIET has a winning strategy if and only if there does not exist a valid tiling of size $\mathrm{Exp}(k, n) \times (\mathrm{Exp}(k, n) - 1)$. If such a tiling exists, ROMEO can give its encoding as v_t and deny JULIET the opportunity to find any errors, no matter which column she isolates, and if no such tiling exists, then any tiling candidate given by ROMEO necessarily contains at least one error, which JULIET can then point out.

Finally, we examine how JULIET may be restricted to play only according to the game plan described above. The basic idea behind this is to construct the replacement transducer and target DNWA in such a way that any deviation from the game plan causes JULIET to immediately and irrevocably lose the game, thus ensuring that any winning strategy, should one exist, sticks to the game plan.

The construction of the target language so far already ensures that JULIET does not leave any undesired uncalled nodes behind, i.e. since target strings enclosed in v-tags may only have tags from $U \cup \{\#, r, l, x, s\}$, JULIET loses automatically if, for instance, there are any uncalled d-nodes left behind in such a final string.

To prevent Juliet from first performing deletions (i.e. calling d-nodes) and then changing the v-labels to h, trying to trick the target automaton with horizontal "errors" created by her deletions, we fix the replacement transducer in such a way that it rejects its input on a call to $\langle/v\rangle$ if any deletions have been performed beforehand. This is easily detectable by the fact that the p-labelled node gets relabelled as soon as the first deletion is performed; therefore, the replacement transducer rejects any input word rooted with v-tags that does not contain any p-tags.

The only thing that still needs to be ensured is that Juliet does not skip any calls. For instance, we could imagine Juliet trying to cheat in phase 1 by playing *Call* on some i-labelled node without having called all nodes labelled c_{i-1} below it, or in phase 3 by calling some d-labelled node while there are still uncalled c-labelled nodes. There are numerous further situations like these, but they are all handled in the same way: on any *Call* to a function symbol, the nested substrings supposed to be rewritten due to that *Call* are required to be of a specific form; for instance, on a *Call* to some i-labelled node in phase 1, the substring nested below it has to be of the form

$$\langle c_i\rangle^*\langle c_{i+1}\rangle\cdots\langle c_k\rangle\langle r\rangle\langle p\rangle\langle/p\rangle\langle/r\rangle\langle/c_k\rangle\cdots\langle/c_{i+1}\rangle\langle/c_i\rangle^*.$$

If the substring below the called function node is not of the desired form (for instance due to remaining c_{i-1}-nodes in the previous example), the replacement transducer constructed according to the ideas laid out above will not have an accepting run on this incorrect substring, therefore reject that substring, and, according to the semantics of NWT games, Romeo will win the game immediately. This shows that we can safely assume Juliet to be restricted to the game plan laid out above.

Finally, it is relatively easy (but again rather tedious) to prove that the replacement transducer described above can be constructed to be of polynomial size in $|U|$, n and k. □

Using Proposition 9.11, the proof of Theorem 9.9 becomes quite straightforward.

Proof of Theorem 9.9. Decidability follows from the fact that, due to the restriction to ϵ-free NWT, there are only finitely many possible replacements for each substring that Juliet plays *Call* on. This, combined with the finite *Call* depth, means that all strategies for Juliet and all possible plays for each strategy can be enumerated in finite time.

The non-elementary lower bound on complexity is implied by Proposition 9.11: Assume that, for some $d \geqslant 2$, there exists a k such that $\textsc{JWin}(\mathcal{G}_d)$ is decidable in k-fold exponential time, where $\mathcal{G}_d$ denotes the class of games with ϵ-free NWT and *Call* depth d. It then follows (by a trivial reduction) that $\textsc{JWin}(\mathcal{G}_2)$ is also in k-EXPTIME, and therefore in co-k-NEXPTIME. However, by Proposition 9.11, $\textsc{JWin}(\mathcal{G}_2)$ is hard for co-$(k+1)$-NEXPTIME, which yields a contradiction to the nondeterministic time hierarchy theorem [Coo73]. □

A close inspection of the proof of Proposition 9.11 also yields co-NEXPTIME-hardness for the replay-free case with ϵ-free replacement NWTs; as the following theorem states, this lower bound is tight.

Theorem 9.12. *For the class of replay-free games with ϵ-free NWT,* JWIN *is complete for* co-NEXPTIME.

Proof. The lower bound follows as in the proof for Proposition 9.11, omitting the first phase of the game constructed there and setting $k = 1$ in the second and third phase.

The co-nondeterministic exponential-time algorithm yielding a matching upper bound is conceptually very straightforward: It moves through the input string from left to right, recursively trying out all possible strategy decisions for JULIET while guessing universally ROMEO'S strategy decisions.

To formalise this algorithm, we use the shorthand notation $[u, v]$ for game positions (J, u, v) of JULIET, where $uv \in \mathrm{WF}(\Sigma)$, with u denoting the substring that has already been processed and v the substring that is yet to be played on (including the closing tag on which JULIET is to move next). For a string $u\langle/a\rangle \in (\langle\Sigma\rangle \cup \langle/\Sigma\rangle)^*$, let $\mathrm{LAST}(u\langle/a\rangle)$ denote the (unique) rooted substring of $u\langle/a\rangle$ ending at $\langle/a\rangle$. The following algorithm CHECKWIN$(G, [u, v])$ then tests whether JULIET has a replay-free winning strategy in game $G = (\Sigma, \Gamma, R, T)$ starting at position $[u, v]$.

Algorithm 4 CHECKWIN$(G, [u, v])$

```
1:  if v = ε then
2:    if u ∈ T then
3:      Accept
4:    else
5:      Reject
6:  if v = tv′ for t ∈ ⟨Σ⟩ ∪ (⟨/Σ⟩\⟨/Γ⟩) then
7:    // Juliet may not make a strategy choice on t; move to the right.
8:    Return CheckWin(G, [ut, v′])
9:  if v = ⟨/f⟩v′ for f ∈ Γ then
10:   // Try out both strategy options for Juliet; Read first.
11:   if CheckWin(G, [u⟨/f⟩, v′]) accepts then
12:     Accept
13:   else
14:     Guess universally a transduct u_t ∈ R(LAST(u⟨/f⟩))
15:     u′ ← (u⟨/f⟩ with LAST(u⟨/f⟩) replaced by u_t)
16:     if CheckWin(G, [u′, v′]) accepts then
17:       Accept
18:     else
19:       Reject
```

As the algorithm CHECKWIN$(G, [u, v])$ directly mimics the gameplay according to G from position $[u, v]$, a simple induction argument suffices to prove that CHECKWIN$(G, [u, v])$ accepts if and only if JULIET has a replay-free winning strategy in G from $[u, v]$; therefore, CHECKWIN$(G, [\epsilon, w])$ accepts if and only if $w \in \mathrm{JWin}^1(G)$. It remains to be shown that CHECKWIN indeed runs in co-nondeterministic exponential time.

To that end, we first examine the maximum size of positions used as inputs for recursive calls of CheckWin (where the size of a position $[u, v]$ is defined as $|uv|$). On some input $[u, v]$, CheckWin can only increase the size of $[u, v]$ through the transduction in lines 14 and 15. Since R is a NWT without ϵ-transitions, any of its outputs on some input x may only have size $c \cdot |x|$ for some constant c depending only on R. This in turn means that the size of positions may only increase by a factor of c for each recursive call to CheckWin.

Obviously, as each recursive call to CheckWin removes one symbol from the right side of a position and insertions only occur on the left side of positions, the recursion depth of CheckWin on an input position $[\epsilon, w]$ is at most $|w|$. This in turn means that input positions for recursive calls to CheckWin may be of size at most $|w| \cdot c^{|w|}$.

Finally, since each recursive call branches into at most two further calls of CheckWin, the algorithm's recursion tree has at most exponentially many nodes in the size of w. This implies that the membership test in line 2 and the co-nondeterministic choice in line 14 are executed at most exponentially many times on at most exponentially long strings. Since membership testing both for NWA and for NWT is in PTIME, this yields an exponential upper bound on the running time of CheckWin. □

9.4. Games with relabelling NWT

The previous section showed that, even with the limited amount of insertion permitted by ϵ-free NWT as a replacement mechanism, the winning problem for Juliet still remains undecidable with unbounded replay, and the decidable cases of bounded or no replay still have a very high complexity compared to games without parameter dependencies. The intuitive reason for this is that, even though ϵ-free NWT offer only limited insertion capabilities and increase the size of input string at most linearly through transduction, even this slight increase in size allows initial strings to become quite large through repeated transduction and re-transduction of call results.

For the unbounded replay case, JWin even stays undecidable as long as *any* sort of insertion is permitted – the ϵ-free NWT used in the proof of Theorem 9.8 can be implemented by a deterministic finite-state transducer that increases the size of its input by no more than 2 on transduction.[4]

In this section, we therefore investigate the case where replacement NWTs are not allowed any insertion at all, i.e. we examine games with *relabelling* NWTs. As it turns out, using relabelling NWT as a replacement mechanism renders the unbounded-replay winning problem decidable but still leads to an intractable winning problem even in the replay-free case. In order to further restrict the model and try to isolate the cause for this intractability, we then examine *functional* relabelling NWT, which, however still do not yield a tractable case (assuming PTIME $\neq$ NP).

[4] Note that, in the nested word setting, inserting an opening (closing) tag always requires also inserting the corresponding closing (opening) tag to obtain a well-nested word, so 2 is the minimum number of symbols that need to be inserted to turn nested words into nested words of strictly greater length.

The upper bounds in this section all use a nigh-trivial (alternating or nondeterministic) algorithm that simply simulates the game. Lower bounds, on the other hand, are proven by reduction from the word problem for linearly bounded (alternating) Turing machines (Theorems 9.13 and 9.15) and from standard logic-based problems (Theorems 9.14 and 9.16). In these lower bound proofs, reductions from TM-based problems follow the standard technique for transducer-based lower bounds introduced above (Juliet repeatedly triggering a transformation mechanism), while reductions from logic-based problems mostly use techniques introduced in Chapters 3 and 6.

Theorem 9.13. *For the class of games with relabelling transducers and unbounded replay,* JWin *is* EXPTIME*-complete.*

Proof. The upper bound uses a trivial alternating polynomial-space algorithm that moves through the input string in a left-to-right order (resetting its focus as necessary after *Call* moves), guesses existentially for each closing tag a move for Juliet and, in case of a *Call*, guesses universally a relabelling chosen by Romeo and applies it to the input string. As relabellings are generally of linear size, they can easily be guessed on polynomial space, and the verification whether a guessed relabelling is indeed consistent with the replacement transducer is feasible in polynomial time by Theorem 8.16.

The lower bound follows by a reduction from the membership problem for linear bounded alternating Turing machines, which is complete for APSPACE = EXPTIME [CKS81].

Let M be a linear bounded ATM with state set Q and working alphabet Σ (i.e. on input w, M uses at most $|w|$ tape cells in any computation). Assume without loss of generality that any non-halting state of M is either existential or universal, that the transition relation for M has exactly two transitions for each state and tape symbol (i.e. each non-halting configuration of M has exactly two successor configurations), that the initial state of M is universal and that M always moves its head to the left-most tape position before halting.

We construct from M and a given input string w a game G and input string w' such that Juliet has a winning strategy on w' in G if and only if M accepts w. To this end, we use a similar technique as for Theorems 9.3 and 9.8: We represent configurations of M by strings rooted at a function symbol $r \notin \Sigma$ and have each *Call* by Juliet to $\langle/r\rangle$ initiate a transition of M, updating M's configuration by the replacement transduction.

The main difference to the proof of Theorems 9.3 and 9.8 is that we have to take alternation into account. The idea here is to simulate the alternation in M by strategy choices of Juliet and Romeo, with Juliet choosing existential and Romeo choosing universal transitions. Universal choice can simply be encoded into the replacement transducer, such that when Juliet initiates a transition of M, Romeo chooses which of the two possible successor configurations to rewrite the current string to. A little more care has to be taken with existential choice, as Juliet may not select any rewritings but can only choose whether or not a substring should be rewritten.

To allow Juliet to choose between transitions, we extend the nested word representation of M's current configuration by a special *flag substring*, which may be either $\langle 0\rangle\langle/0\rangle$

or $\langle 1\rangle\langle/1\rangle$ or $\langle 2\rangle\langle/2\rangle$. This flag should be 0 if the current configuration is universal or halting, and initially set to 1 once an existential configuration is reached. If it is 1, JULIET has the option to have ROMEO rewrite it to 2 by way of a *Call* move, or leave it as is with a *Read* move. Once JULIET then initiates a transition of M, the flag indicates which of the two possible successor configuration is reached by the rewriting, i.e. ROMEO does not get any choice but rewrites the current string based on the flag's value.

Once again, we represent configurations of M as in the proof of Theorem 9.3: a flat string $(x_1, -)\cdots(x_{k-i}, -)(x_k, q)(x_{k+1}, -)\cdots(x_m, -)$ over the alphabet $\Sigma \times (Q \cup \{-\})$ denotes that the content of M's working tape is $x_1 \cdots x_m$, with the head of M being on the tape's k-th cell and M being in state q, and these flat strings are represented as nested strings using the standard nested string encoding. By our assumption on the shape of M's halting configurations, the flat string encoding of an accepting configuration is of the form $(\Sigma \times \{q_+\})(\Sigma \times \{-\})^*$ for the accepting state q_+ of M.

The input string w' for the game G is constructed as $w' = \langle r\rangle\langle 0\rangle\langle/0\rangle\widehat{v_0}\langle/r\rangle$, where $\widehat{v_0}$ is the nested string representation of M's initial configuration with input w (which is universal by our assumption above, hence the 0 flag). The game G is over the alphabet $(\Sigma \times (Q \cup \{-\})) \cup \{r, 0, 1, 2\}$ with function symbols $\Gamma = \{r, 1\}$.

The replacement transducer simulates a transition of M at a *Call* on $\langle/r\rangle$ as described above. Such a transducer can be computed from M's transition relation in polynomial time. It is also easy to construct in polynomial time a target DNWA accepting all strings of the form $\langle r\rangle\langle 0\rangle\langle/0\rangle\widehat{v}\langle/r\rangle$ where $\widehat{v}$ represents an accepting configuration of M. It follows from the above considerations that JULIET has a winning strategy on w' in G if and only if M accepts w. □

Theorem 9.14. *For any $k \geqslant 1$, for the class of games with relabelling transducers and bounded Call depth k,* JWIN *is* PSPACE*-complete.*

Proof. The upper bound again uses the trivial alternating algorithm that moves through the input string in a left-to-right order, guesses existentially for each closing tag a move for JULIET and, in case of a *Call*, guesses universally a relabelling chosen by ROMEO and applies it to the input string. As in the proof of Theorem 9.13, relabellings can be guessed and verified in polynomial time; additionally, since the game has bounded *Call* depth k, strategy decisions and relabellings are guessed at most $k \cdot |w|$ times, which yields a polynomial time bound for the alternating algorithm.

The lower bound for $k = 1$ follows directly from the PSPACE lower bound proof for games with fixed replacement languages without transducers (Theorem 6.10(c)), as that proof only requires relabelling. □

We now turn to the winning problem for games with functional relabelling transducers. The restriction to functional transducers is already a quite severe one, as games with functional replacement are essentially solitaire games for JULIET without any choice for ROMEO. Even so, the complexity of determining whether JULIET has a winning strategy remains at least NP-hard, even if replacement transducers are assumed to be deterministic instead of functional.

Theorem 9.15. *For the class of games with functional or deterministic relabelling transducers and unbounded replay,* JWin *is* PSPACE*-complete.*

Proof. As for Theorems 9.13 and 9.14, the upper bound uses the trivial algorithm simulating the game with functional relabelling transducers by moving through the input string in a left-to-right order (resetting its focus as necessary after *Call* moves), guessing existentially for each closing tag a move for Juliet and, in case of a *Call*, also guessing the relabelling chosen by Romeo and applying it to the input string. Since there is only at most a single possible rewriting for each string input into the relabelling transducer and relabellings are of linear size in the input, this is a nondeterministic polynomial-space algorithm witnessing membership of JWin in NPSPACE = PSPACE.

The lower bound for games with deterministic relabelling transducers is proven similarly to the one in Theorem 9.13 by reduction from the membership problem for linear bounded *deterministic* Turing machines. The simulation of a TM by a game works as in the proof of Theorem 9.8, as that proof also just requires a deterministic transducer. For linear bounded TMs, a relabelling transducer suffices, since no additional tape cells beyond those provided by the input are ever used.

Since all deterministic NWTs are also functional, the upper bound also applies to deterministic relabelling NWTs and the lower bound also to functional relabelling NWTs, which proves the claim. □

Theorem 9.16. *For any $k \geqslant 1$, for the class of games with functional relabelling transducers and bounded Call depth k,* JWin *is* NP*-complete.*

Proof. The upper bound for functional relabelling transducers uses the same nondeterministic algorithm as the one used for the upper bound of Theorem 9.15; the only difference is that, since the input game has *Call* depth k, the algorithm has to guess nondeterministically and verify at most $k \cdot |w|$ strategy choices and relabellings, each of which may be done nondeterminstically in polynomial time. This yields the desired NP upper bound.

We prove the lower bound for replay-free games with deterministic relabelling transducers by a reduction from the 3SAT problem: Given a propositional formula $\varphi = C_1 \wedge \ldots \wedge C_m$ over variables $x_1, \ldots, x_n$ where each clause C_j is a disjunction of exactly three literals, is there an assignment $\alpha : \{x_1, \ldots, x_n\} \to \{0, 1\}$ such that φ evaluates to 1 under α?

We construct from φ a game $G = (\Sigma, \Gamma, R, T)$ and string $w \in \mathrm{WF}(\Sigma)$ such that Juliet has a winning strategy on w in G if and only if φ is satisfiable.The basic idea behind the reduction is that the string w contains *variable substrings*, on which Juliet chooses an assignment α by using *Read* and *Call* moves, and a *clause substring*, which models the structure of φ and which gets rewritten after Juliet's choice of variable assignment into a form that allows a target DNWA to check whether α satisfies φ.

More concretely, the clause substring is $w_\varphi = w_{C_1} \cdots w_{C_m}$, where, for each $i \in [m]$, w_{C_i} is defined as

$$w_{C_i} = \langle C_i \rangle \langle \ell_{i,1} \rangle \langle / \ell_{i,1} \rangle \langle \ell_{i,2} \rangle \langle / \ell_{i,2} \rangle \langle \ell_{i,3} \rangle \langle / \ell_{i,3} \rangle \langle / C_i \rangle,$$

where for each $j \in [3]$ and $k \in [n]$,

$$\ell_{i,j} = \begin{cases} x_k, & \text{if } x_k \text{ is the } j\text{-th literal of } C_i \\ \overline{x_k}, & \text{if } \neg x_k \text{ is the } j\text{-th literal of } C_i. \end{cases}$$

The input string w is constructed inductively; we set $w = w_1$, where for each $i \in [n+1]$,

$$w_i = \begin{cases} \langle r\rangle\langle y_i^0\rangle\langle/y_i^0\rangle w_{i+1}\langle/r\rangle, & \text{for } i \in [n] \\ w_\varphi, & \text{for } i = n+1. \end{cases}$$

Play on w is supposed to proceed as follows: Juliet first goes through the variable substrings of the form $\langle y_i^0\rangle\langle/y_i^0\rangle$, choosing whether or not to call each $\langle/y_i^0\rangle$ in turn. On a *Call*, y_i^0 gets relabelled to y_i^1. In this way, Juliet chooses a variable assignment α, with a remaining y_i^0 interpreted as $\alpha(x_i) = 0$ and a rewritten y_i^1 interpreted as $\alpha(x_i) = 1$.

Afterwards, Juliet is supposed to call each $\langle/r\rangle$ in turn. On each such *Call*, the relabelling transducer T takes the valuation y_i^0 (or y_i^1 respectively) of the variable immediately following the corresponding $\langle r\rangle$ and rewrites all occurrences of x_i in w_φ into 0 and all $\overline{x_i}$ into 1 (or x_i into 1 and $\overline{x_i}$ into 0, respectively). After all literals in w_φ have been rewritten into 0 or 1, the target DNWA simply needs to check whether, for each $j \in [m]$, the substring enclosed in C_j tags contains at least one substring $\langle 1\rangle\langle/1\rangle$.

More formally, the game G uses the alphabet $\Sigma = \{x_i, \overline{x_i}, y_i^0, y_i^1 \mid i \in [n]\} \cup \{C_j \mid j \in [m]\} \cup \{r\}$ with function symbols $\Gamma = \{r\} \cup \{y_i^0 \mid i \in [n]\}$.

The relabelling transducer R behaves as follows:

- R rewrites each input of the form $\langle y_i^0\rangle\langle/y_i^0\rangle$ into $\langle y_i^1\rangle\langle/y_i^1\rangle$.
- On an input of the form $\langle r\rangle\langle\ell\rangle\langle/\ell\rangle v\langle/r\rangle$ (for $\ell \in \{y_i^0, y_i^1 \mid i \in [n]\}$), R memorises ℓ in its state and performs a relabelling on v:
  - If $\ell = y_i^0$ for some $i \in [n]$, then R relabels each x_i in v to 0 and each $\overline{x_i}$ to 1, and
  - If $\ell = y_i^1$ for some $i \in [n]$, then R relabels each x_i in v to 1 and each $\overline{x_i}$ to 0.

The target language T contains all strings w' of the form of w_1 as defined above, with the following modifications:

- w' does not contain any labels from $\{x_i, \overline{x_i} \mid i \in [n]\}$, and
- each $\langle C_j\rangle$ tag (for $j \in [m]$) is followed by at least one $\langle 1\rangle\langle/1\rangle$ substring before the corresponding $\langle/C_j\rangle$ tag.

The construction of G and w from φ is obviously possible in polynomial time. Furthermore, by the above consideration on the construction and verification of assignments for φ, it is easy to see that Juliet has a winning strategy on w in G if and only if φ is satisfiable. □

9.5. Other restrictions

The previous sections provided a complete decidability and complexity classification for JWin for the most obvious parameters – replay on the strategy side, and the degree of permitted insertion on the transducer side. Looking closely at lower bound proofs in these sections, however, yield some less obvious restrictions that might be examined in order to reduce complexity. In this section, we look at one such restriction gained from each of the previous sections – *depth-bounded NWT* for Section 9.2, games of *bounded Call width* for Section 9.3, and *write-once games* for Section 9.4, with the latter two actually yielding some (very restricted) cases where JWin is tractable.

Depth-bounded replacement

Looking again at the undecidability proof for Theorem 9.3, one can see that, for bounded replay, it hinges on the fact that Juliet can provide replacement strings of arbitrary depth (allowing for the simulation of a Turing machine for an arbitrary number of steps). Considering our practical motivation, it is rarely required that function calls in Active XML documents return arbitrarily deep trees. Therefore, we now investigate the impact of limiting replacement transducers' output depth.

For simplicity's sake, we assume depth-boundedness as a *semantic* restriction, i.e. we assert that all outputs in $R(w)$ produced by a depth-bounded replacement transducer R on a string w obey a given upper bound on their depth, without examining the decidability and complexity of determining whether or not a given transducer is depth-bounded.

We note that NWT with an output depth linear in the size of the input string are already strictly more expressive than ϵ-free NWT, so the lower bounds from Section 9.3 also hold for NWT with linear output depth. As these lower bounds are already quite high, we focus only on transducers whose output depth is bounded by a constant.

Definition 9.17. An NWT R is called *depth-bounded* if there is some constant $d \geqslant 0$ such that for any $w \in \mathrm{WF}(\Sigma)$ and any $w' \in R(w)$, the depth of w' is at most d.

Using depth-bounded NWT as replacement transducers places the complexity of the replay-free winning problem between those for general NWT and for ϵ-free NWT. The upper and lower bounds are proven similarly to those of Theorem 9.4, but use the fact that the stack size of a depth-bounded NWT on a fixed input is bounded by a constant.

Theorem 9.18. *For the class of replay-free games with depth-bounded NWT,* JWin *is* EXPSPACE*-complete.*

We prove the lower and upper bounds of this theorem as separate propositions.

Proposition 9.19. *For the class of replay-free games with depth-bounded NWT,* JWin *is* EXPSPACE*-hard.*

The proof of this lower bound once again follows the standard approach of reducing from a tiling problem, tasking Romeo with providing a tiling candidate and Juliet with finding errors in this candidate.

Proof. This lower bound proof uses the problem Exponential Corridor Tiling: Given a tiling instance consisting of a tile set U, vertical and horizontal constraints V, H, initial and final tile u_i, u_f and unary number n, is there a tiling of width $2^n - 1$ and arbitrary height?

From any input tiling instance, we construct a game $G = (\Sigma, \Gamma, R, T)$ with depth-bounded NWT R and an input word w such that Juliet has a winning strategy on w in G if and only if there exists *no* valid tiling of width $2^n - 1$, i.e. we reduce from the complement of Exponential Corridor Tiling. Since Exponential Corridor Tiling is EXPSPACE-complete, so is its complement.

The basic idea behind the construction of G is similar to that used in the proof of Proposition 9.5. The input string is

$$w = \langle r\rangle\langle o_n\rangle\langle e_n\rangle \cdots \langle o_1\rangle\langle e_1\rangle\langle s\rangle\langle /s\rangle\langle /e_1\rangle\langle /o_1\rangle \cdots \langle /e_n\rangle\langle /o_n\rangle\langle /r\rangle.$$

Juliet is first supposed to *Call* $\langle /s\rangle$, allowing Romeo to respond with the standard nested string encoding $\widehat{v}_0$ of a flat string $v_0 \in (U \cup \{\#\})^*$ which is supposed to encode a valid tiling t of width $2^n - 1$ in the standard way described in Section 2.5 (i.e. lines of t are concatenated, with # symbols as line separators). We can fix the transducer R in such a way that v_0 is always a concatenation of horizontally correct substrings (not necessarily of length $2^n - 1$) separated by # symbols. After $\widehat{v}_0$ is given, Juliet then plays *Call* on either e_1 or o_1, prompting Romeo to delete either all even-numbered or all odd-numbered positions in v_0 and yielding the standard nested string encoding $\widehat{v}_1$ of the resulting flat string v_1. Continuing this process further (i.e. sequentially calling exactly one of e_i or o_i for each $i \in [n]$) eventually yields a nested string $\widehat{v}_n$ which encodes what is supposed to be a single column in the tiling given by Romeo. This string is then checked for vertical correctness by the target DNWA like in the proof of Proposition 9.5. In this way, Juliet has a winning strategy on w in G if and only if every tiling candidate that can be provided by Romeo contains some (horizontal or encoding) error, i.e. if there is no valid tiling of the desired width. □

Similar to the proof of Theorem 9.4, the upper bound in Theorem 9.18 is proven by a reduction to (a variant of) the alternating iterated transduction problem: Given a string $w \in \mathrm{WF}(\Sigma)$, number k (given in unary), DNWA A, and pairs of *depth-bounded* NWTs $(T_{1,0}, T_{1,1}), \ldots, (T_{k,0}, T_{k,1})$, is there an $i_1 \in \{0,1\}$ such that for every $w_1 \in T_{1,i_1}(w)$ there exists $i_2 \in \{0,1\}$ such that for every $w_2 \in T_{2,i_2}(w_1)$ ... there exists $i_k \in \{0,1\}$ such that for every $w_k \in T_{k,i_k}(w_{k-1})$ it holds that $w_k \in L(A)$?

Proposition 9.20. *The alternating iterated transduction problem for depth-bounded NWT is in* EXPSPACE.

Proof. Basically, the exponential-space algorithm solving this problem follows the same idea as the 2-EXPTIME algorithm from the proof of Proposition 9.7; here, the bounded depth of input transducers ensures that the construction and type checking of transducers T_j (for each j) and T from that proof may be simulated on-the-fly by a co-nondeterministic exponential-space algorithm $\mathcal{A}$.

More specifically, it can be easily proven by induction that, in any run ρ of an NWT receiving as input a string of depth d_{in} and outputting a string of depth at most d_{out}, the sequence of hierarchical states in any configuration occurring in ρ has length at most $d_{\text{in}} + d_{\text{out}}$.

Since all input transducers are depth-bounded, we can assume without loss of generality that they have a common upper bound d on their output depth. Then, for each $j > 1$, each $T_{j,i}$ has an input *and* output depth bounded by d, since $T_{j,i}$ receives as its input the output of some $T_{j-1,i'}$. On the other hand, $T_{1,0}$ and $T_{1,1}$ also have an output depth bounded by d and an input depth bounded by $|w|$, which is also fixed for fixed input strings w. Similarly, the DNWA A receives as inputs only outputs of $T_{k,0}$ or $T_{k,1}$, so we may restrict our attention to configurations of A with a hierarchical state sequence of length at most d.

The idea behind the algorithm $\mathcal{A}$, then, is to traverse the input string w from left to right and, on each input tag, co-nondeterministically guess transducts for each combination of follow-up transducers and check that at least one resulting final transduct is in $L(A)$, thus simulating the transducer T from the proof of Proposition 9.7 and checking that each transduct from $T(w)$ is accepted by the NWA B from that proof.

The algorithm $\mathcal{A}$ is organised into k layers, with the j-th layer (for $j \in [k]$) simulating the workings of transducer T_j from the proof of Proposition 9.7, which takes as input 2^{j-1}-tuples of symbols from $\Sigma \cup \{\sqcup\}$ and outputs 2^j-tuples of such symbols by guessing transitions of both $T_{j,0}$ and $T_{j,1}$ on each component of the input tuple. The algorithm $\mathcal{A}$ does this simulation based on a single tuple of symbols for each layer (called that layer's *active input tuple*) – once layer $j-1$ produces an output tuple, this tuple becomes the active input for layer j, and $\mathcal{A}$ continues its simulation with layer j, producing an output to become the active input tuple for layer $j+1$ by either consuming the active input tuple of layer j or by an ϵ-transition (which leaves the active input tuple for layer j unchanged). Once the active input tuple of some layer j has been consumed, control passes back to layer $j-1$. The bottom layer k, instead of producing further active input tuples, directly simulates the effect of its output on 2^k modified copies of the DNWA A (i.e. simulates the NWA B from the proof of Proposition 9.7).

Some extra care has to be taken regarding deletions. The proof sketch so far (as well as the construction for Proposition 9.7) assumes that simulated transitions are synchronised in such a way that, when an input tuple is consumed, each component of that tuple produces two components of the output tuple. This assumption obviously does not hold in the presence of deleting transitions. For this reason, we allow active input tuples to contain *null positions* labelled $\perp$, which do not induce any transition on the corresponding transducers once the tuple is consumed.

In the sequel, as in the proof of Proposition 9.7, assume that all input transducers are in normal form as per Lemma 8.11, and for each $i \in \{0,1\}$ and $j \in [k]$, let $T'_{j,i}$ be an extension of $T_{j,i}$ to strings containing $\sqcup$ tags that ignores and reproduces these tags, and let A' be an analogous extension of A. We now describe the algorithm $\mathcal{A}$ in more detail.

For each j, the following information is stored for layer j:

- A 2^j-tuple of configurations of $T'_{j,0}$ (in odd positions) and $T'_{j,1}$ (in even positions), each consisting of a linear state and a sequence of at most $d + |w|$ (for $j = 1$) or $2d$ (for $j > 1$) hierarchical states. These configurations are initialised with the starting configuration of $T'_{j,0}$ or $T'_{j,1}$.
- A 2^{j-1}-tuple of symbols from $\langle\Sigma\rangle \cup \langle/\Sigma\rangle \cup \{\langle\sqcup\rangle, \langle/\sqcup\rangle, \bot\}$, where either all non-$\bot$ components are opening tags or all are closing tags (called the *active input tuple* of layer j). The non-$\bot$ components of each active input tuple give the next input symbols to be consumed by (2^{j-1} copies of) $T_{j,0}$ and $T_{j,1}$. The active input tuple of layer j is initialised to $\bot^{2^{j-1}}$

Additionally, $\mathcal{A}$ stores the following global information:

- A counter $\ell \in \{0, \ldots, |w|\}$ denoting the last position of the input string w that has been read, initialised to $\ell = 0$.
- A 2^k-tuple of configurations of A', each consisting of a single linear state and a sequence of up to d hierarchical states. These configurations are initialised with the starting configuration of A'.
- A layer counter $j \in \{0, \ldots, k\}$ denoting the current *active layer* being processed (with layer 0 corresponding to the input string itself). This counter is initialised to $j = 0$.

All of this information can obviously be stored in exponential space.
The algorithm $\mathcal{A}$ now proceeds as follows:

1. If $j = 0$...
   a) ...and $\ell < |w|$, then the input has not yet been completely read. In this case, $\mathcal{A}$ sets the active input (2^0-)tuple of layer 1 to the $(\ell + 1)$-th symbol of w, $\ell := \ell + 1$, and $j := 1$.
   b) ...and $\ell = |w|$, then the input has been processed completely, and $\mathcal{A}$ halts (with an acceptance condition to be detailed below).
2. If $0 < j < k$...
   a) ...and the active input tuple of layer j is *not* equal to $\bot^{2^{j-1}}$, then layer j still has some input to be processed. In this case, $\mathcal{A}$ guesses co-nondeterministically which of the following two steps to perform:
      i. $\mathcal{A}$ processes the input of layer j, i.e. for each non-$\bot$ position $i \in [2^{j-1}]$ of the active input tuple, $\mathcal{A}$ guesses a transition of $T'_{j,0}$ with the input symbol from that position starting at the configuration in position $2i - 1$ of the configuration tuple for layer j, and updates that configuration accordingly; similarly, $\mathcal{A}$ guesses a transition for $T'_{j,1}$ with position $2i$ of the configuration tuple. The outputs of these transitions are written to positions $2i - 1$ and $2i$ of a new 2^j-tuple t. For any $\bot$-position $i \in [2^{j-1}]$,

positions $2i-1$ and $2i$ of t are then filled with $\perp$ markers. Afterwards, the active input tuple of layer $j+1$ is set to t, the active input tuple of layer j is reset to $\perp^{2^{j-1}}$, and $j := j+1$.

ii. $\mathcal{A}$ performs an ϵ-transition. To that end, $\mathcal{A}$ guesses an index $i \in [2^j]$. If i is odd, $\mathcal{A}$ guesses an opening (resp. closing) ϵ-transition for $T'_{j,0}$ available in the configuration at position $(i+1)/2$ of the configuration tuple and updates that configuration accordingly, while configurations at all other positions remain unchanged. It then writes the output of that transition to position $(i+1)/2$ of a new 2^j-tuple t and fills all other positions of t with $\langle\sqcup\rangle$ (resp. $\langle/\sqcup\rangle$) symbols. If i is even, $\mathcal{A}$ proceeds analogously with $T'_{j,1}$ instead of $T'_{j,0}$ and $i/2$ instead of $(i+1)/2$. Finally, $\mathcal{A}$ sets the active input tuple of layer $j+1$ to t and $j := j+1$ *without* resetting the active input tuple of layer j.

b) ...and the active input tuple of layer j is $\perp^{2^{j-1}}$, i.e. layer j currently has no input waiting to be processed. In this case, $\mathcal{A}$ guesses co-nondeterministically which of the following two steps to perform:

i. $\mathcal{A}$ performs an ϵ-transition, as described under 2.a.ii.

ii. $\mathcal{A}$ ends its processing of layer j and sets $j := j-1$.

3. If $j = k$, then $\mathcal{A}$ basically proceeds as described under 2., with the only difference being that any time $\mathcal{A}$ has created an output tuple t of size 2^k, instead of setting $j := j+1$, $\mathcal{A}$ directly aggregates t onto the configuration 2^k-tuple for A', simulating, for each $i \in [2^k]$, a transition of A' starting from the i-th component of the configuration tuple, consuming the i-th component of t and storing the resulting configuration in the i-th position of the configuration tuple for A'.

Finally, once $\mathcal{A}$ halts, it accepts if at least one of the following conditions is met:

- One of the 2^k stored configurations for A' is accepting. In this case, there is some sequence of existential choices of transducers such that, for the universal choices made co-nondeterministically by $\mathcal{A}$, the final transduct is in $L(A)$.
- For some $j \in [k]$, the configuration tuple for layer j contains some non-accepting configuration. In this case, the co-nondeterministic choices taken by $\mathcal{A}$ have led to an incorrect transduction being performed, i.e. the corresponding run of $\mathcal{A}$ should not be counted against the acceptance condition.

To show correctness of $\mathcal{A}$, it needs to be proven that there exists a non-accepting run of $\mathcal{A}$ if and only if for all $i_1 \in \{0,1\}$ there exists $w_1 \in T_{1,i_1}(w)$ such that for all $i_2 \in \{0,1\}$ ... for all $i_k \in \{0,1\}$ there exists $w_k \in T_{k,i_k}(w_{k-1})$ with $w_k \notin L(A)$.

The proof for this is rather technical, but its basic idea is as follows: For the "only if" direction, we extract from an accepting run of $\mathcal{A}$ each "witness string" w_j inductively based on the sequence $i_1, \ldots, i_j$ of universal choices and prior witness strings $w_1, \ldots, w_{j-1}$ by taking the output string produced in component $i+1$ of layer j, where $i \in \{0, \ldots, 2^j -$

1} is the number represented by the binary encoding $i_1 \cdots i_j$. Using the construction of $\mathcal{A}$, it is then easy to see that $w_j \in T_{j,i_j}(w_{j-1})$, and that $w_k \notin L(A)$. Similarly, for the "if" direction we can construct a run of $\mathcal{A}$ from witness strings given universal choices of indices in $\{0, 1\}$, which is non-accepting because all transductions are performed correctly and, regardless of universal choices, the resulting final string w_k is not in $L(A)$.

Finally, as stated above, $\mathcal{A}$ requires only co-nondeterministic exponential space and can therefore be simulated by an EXPSPACE algorithm. □

Proof of Theorem 9.18. The lower bound was proven as Proposition 9.19. For the upper bound, we reduce JWin to the alternating iterated transduction problem for depth-bounded transducers using the same reduction as in the proof of Theorem 9.4.

Some care needs to be taken with this reduction, as the reduction from Theorem 9.4 constructs, from an input string w and a game G whose replacement transducer has output depth bounded by some constant d, an instance for the alternating iterated transduction problem whose transducers have a depth bound $d + |w|$; that is, the depth bound for the alternating iterated transduction problem actually depends on the size of the input. However, the proof of Proposition 9.20 shows that the algorithm given there still only takes exponential space, even if the depth bound of transducers depends on the input, since in this case, the algorithm keeps track of (exponentially many) configurations whose size is still at most polynomial in the input size. In this way, Proposition 9.20 still yields an EXPSPACE upper bound for JWin. □

Games with bounded Call width

The non-elementary lower bound in Theorem 9.9 follows from the fact that, in each string returned by Romeo, Juliet may play *Call* arbitrarily often. On a return string corresponding to a path of length n, Juliet may play *Call* on all n nodes bottom-up, with each such *Call* doubling the number of nodes below the called node, inducing a non-elementary blow-up.

To avoid this, we now examine games with bounded *Call width*, where, intuitively, Juliet may only play *Call* for a bounded number of times in each replacement string given by Romeo. Note that *Call* width is counted within each individual replacement string – so, in a game of *Call* depth 3 and *Call* width c, if Juliet plays *Call* on some position of the input string, she may then place up to c calls within the string returned by Romeo, and again up to c calls in *each* of the depth-2 replacement strings resulting from those calls.

More formally, the *Call* width of a play Π is the maximum number of times Juliet plays *Call* in any replacement string given by Romeo in Π. This definition extends naturally into that of *Call* width of a strategy. Note that *Call* width only applies to *replacement* strings, so Juliet may still call arbitrarily many positions of the *input* string, even for games with *Call* width 0. For this reason, replay-free strategies always have bounded *Call* width.

The proof of Theorem 9.8 shows that JWin remains undecidable for games with unbounded *Call* depth, even with *Call* width bounded by 1. For bounded replay, though,

the complexity of JWin collapses to that of the replay-free case if *Call* width is bounded.

Theorem 9.21. *For the class of games with ϵ-free NWT, Call depth bounded by $d \geqslant 1$ and Call width bounded by $k \geqslant 1$,* JWin *is* co-NEXPTIME*-complete.*

The basic proof idea for the upper bound in Theorem 9.21 is the same algorithm used for replay-free games with ϵ-free replacement NWT (Theorem 9.12) – using co-nondeterminism to guess replacement strings for Romeo and enumerating over all strategies for Juliet. As the proof shows, bounded *Call* width narrows down the search space sufficiently for this algorithm to work in co-nondeterministic exponential time.

Proof. As replay-free games always have bounded *Call* width, the lower bound carries over directly from Theorem 9.12.

The algorithm for the upper bound is almost the same as the one for Theorem 9.12, simply going through the input string from left to right, trying out all possible strategies for Juliet and guessing co-nondeterministically replacement strings for Romeo. The only difference is that, in this case, it also needs to track how much of the allowed *Call* width and depth in the current substring has been used up already.

As in the proof of Theorem 9.12, the correctness of this algorithm is obvious as it basically just simulates gameplay.

To prove that the running time and amount of non-determinism required by the algorithm is at most exponential, we again examine the maximum size of occurring strings and of the decision tree for Juliet for any fixed counter-strategy of Romeo, i.e. the recursion tree in any run of the algorithm.

For string sizes, we again note that, for any replacement NWT R and string w, all strings in $R(w)$ are of size at most $s \cdot |w|$ for some constant s only depending on R. Furthermore, any replacement substring v resulting from a *Call* move of depth $r < d$ allows for at most k further *Call* moves of depth $r + 1$ (each of which may be further played on with *Call* width k if $r + 1 < d$). Each of these depth-$(r + 1)$ substrings may be of size at most $s^k \cdot |v|$ (in case all k allowed calls go into re-transducing v in some fashion). In total, this means that increasing *Call* depth by 1 increases the size of occurring strings by a multiplicative factor exponential in k. Therefore, with *Call* depth d and *Call* width k, the final string after a play on some input string w is at most of size $|w| \cdot 2^{\mathcal{O}(dk)}$, and this size is also an upper bound on the size of each single transduct (and therefore also on the amount of non-determinism required any time a transduct is guessed).

Since each *Call* move of depth r enables at most k further calls of depth $r + 1$, and there are at most $|w|$ possible *Call* positions of depth 0 in any input string w, Juliet may play at most $c_{\max} \stackrel{\text{def}}{=} |w| \cdot k^d$ *Call* moves in any play on w.

To reach an upper bound on the number of positions Juliet has available to place calls on during an entire play, we use the above upper bound of $|w| \cdot 2^{\mathcal{O}(dk)}$ on the size of working strings and assume (as a generous estimate) that the *entire* string is replaced by a different string of length $|w| \cdot 2^{\mathcal{O}(dk)}$ every time Juliet plays *Call* on some position. Then, the total number $\ell_{\max}$ of positions Juliet can place calls on is bounded by the total number of positions that ever become available during the entire game, which is at

most the maximum string length times the maximum number of available *Call* moves, i.e. $\ell_{\max} = (|w| \cdot 2^{\mathcal{O}(dk)}) \cdot (|w| \cdot k^d) = |w| \cdot 2^{\mathcal{O}(dk)}$.

We can now give an upper bound on the size of the decision tree for JULIET for a given counter-strategy by ROMEO. Assume nodes of this tree to be labelled with positions that JULIET has to make a strategy decision on, and outgoing edges to be labelled either *Read* or *Call*. Then, each root-to-leaf path has length equal to at most $\ell_{\max}$ and contains at most $c_{\max}$ edges labelled *Call*, with all other edges being labelled *Read*. Since the sequence of *Read*- and *Call*-edges taken from the root forms a unique address in this tree, the number of different paths in the tree is at most the number of strings of length $\ell_{\max}$ over $\{Read, Call\}$ containing at most $c_{\max}$ occurrences of *Call*, which is in $\ell_{\max}^{\mathcal{O}(c_{\max})}$. Together with $\ell_{\max}$ being the maximum length of paths, this implies that the decision tree has at most $\ell_{\max}^{\mathcal{O}(c_{\max})} = |w| \cdot 2^{\mathcal{O}(|w| \cdot dk^{d+1})}$ nodes.

Since d and k are fixed, this means that the size of the recursion tree for the co-nondeterministic algorithm simulating play is at most exponential, and since each node of this recursion tree requires at most exponential computation time and non-determinism, the algorithm indeed runs in co-nondeterministic exponential time, as was to be proven. □

As mentioned above, bounded *Call* width does not affect JULIET's options for *Call* moves on the input string, as we generally want JULIET to be able to at least process *all* function symbols in the input. Dropping this requirement (i.e. bounding *Call width including input*) yields at least an exponential improvement in complexity.

Theorem 9.22. *For the class of games with ϵ-free NWT, Call depth bounded by d and Call width including input bounded by k,* JWIN *is*

(a) CO-NP*-complete for $d \geqslant 1$ and $k \geqslant 2$,*

(b) CO-NP*-complete for $d \geqslant 2$ and $k \geqslant 1$, and*

(c) in PTIME *for $d = k = 1$.*

Proof. The CO-NP upper bound uses almost exactly the same algorithm as Theorem 9.21, going through the input string from left to right, trying out all possible strategies for JULIET, guessing co-nondeterministically replacement strings for ROMEO and tracking *Call* depth and width along the way. The only difference to that algorithm lies in the fact that *Call* width is also tracked for the input string, which changes the analysis from the proof of Theorem 9.21 as follows.

With fixed bounds d on *Call* depth and k on *Call* width, the maximum number of *Call* moves that JULIET can play is $c_{\max} = k^d$, which is constant. With each *Call* move increasing the size of the current string by at most a multiplicative constant s (depending only on the replacement transducer), this means that the maximum size of strings is $\ell_{\max} = |w| \cdot s^{c_{\max}}$ for any input string w, which is linear in $|w|$. In analogy to the arguments from the proof of Theorem 9.21, this means that JULIET may perform calls on at most $c_{\max}$ out of $c_{\max} \cdot \ell_{\max} = \mathcal{O}(|w|)$ positions, implying that the decision tree has at most $\ell_{\max}^{\mathcal{O}(c_{\max})}$ paths of length $\ell_{\max}$ each, and therefore size $\mathcal{O}(|w|^c)$ for some constant

c. Since all strings that need to be processed or nondeterministically guessed at each node of the recursion tree are of polynomial length, this implies a co-nondeterministic polynomial time complexity.

The lower bounds in (a) and (b) follow by reductions from the problem CO-3-SAT: Given a propositional formula φ in conjunctive normal form with three literals per clause, is φ unsatisfiable? This problem is the complement of the well-known NP-complete problem 3-SAT, and therefore complete for CO-NP.

The reductions work as follows. Let $\varphi = C_1 \wedge \ldots \wedge C_m$ be a 3-CNF formula over variables $x_1, \ldots, x_n$ with clauses $C_1, \ldots, C_m$. We construct from φ a game $G = (\Sigma, \Gamma, R, T)$ and input string w such that ROMEO has a winning strategy on w in G if and only if φ is satisfiable; the basic idea behind the reduction is that JULIET'S first *Call* to the input string allows ROMEO to pick a variable assignment α for φ, while JULIET'S second *Call* is supposed to mark a clause that is not satisfied by α (if one exists). In this way, JULIET has a winning strategy if and only if every possible assignment that ROMEO can chose does not satisfy all of the clauses in φ. Using this idea, JULIET needs to perform exactly two *Call* moves; where and how these calls should be performed depends on whether the game is replay-free with *Call* width 2 (for (a)) or has *Call* depth 2 and *Call* width 1 (for (b)).

For both reductions, the input string w is of the form

$$w = \langle r\rangle\langle C_1\rangle \cdots \langle C_m\rangle\langle V\rangle\langle x_1\rangle\langle /x_1\rangle \cdots \langle x_n\rangle\langle /x_n\rangle\langle /V\rangle\langle /C_m\rangle \cdots \langle /C_1\rangle\langle /r\rangle,$$

i.e. its tree representation is a path with labels $r, C_1, \ldots, C_m, V$ (in top-down order), with the V-labelled node having as children leaves labelled $x_1, \ldots, x_n$.

For the reduction in (a), the set of function symbols are $\Gamma = \{V, C_1, \ldots, C_m\}$. JULIET'S first *Call* is supposed to be on $\langle /V\rangle$, which relabels V to V' and each x_i to either 0_i or 1_i (for $i \in [n]$), at ROMEO'S choice; afterwards, she calls exactly one $\langle /C_j\rangle$ (with $j \in [m]$), which relabels the called node to $\hat{C}_j$ and leaves the rest of the string unchanged. If JULIET plays *Call* on some $\langle /C_j\rangle$ without having called $\langle /V\rangle$ first, the replacement transducer rejects its input string and JULIET loses the game immediately. The target language accepts all strings containing exactly one node labelled with a symbol $\hat{C}_j \in \{\hat{C}_1, \ldots, \hat{C}_m\}$ such that the clause C_j is not satisfied by the variable assignment encoded below V.

For (b), the set of function symbols is $\Gamma = \{r, C'_1, \ldots C'_m\}$. JULIET'S first *Call* (the only one she can perform in the input string) is to $\langle /r\rangle$, which relabels r to r', each C_j to C'_j (for $j \in [m]$) and each x_i to either 0_i or 1_i (for $i \in [n]$), at ROMEO'S choice; afterwards, JULIET is supposed to call exactly one $\langle /C'_j\rangle$, relabelling it to $\hat{C}_j$. The target language accepts all strings rooted with r' that contain exactly one node labelled with a symbol $\hat{C}_j \in \{\hat{C}_1, \ldots, \hat{C}_m\}$ such that the clause C_j is not satisfied by the variable assignment encoded below V.

It is easy to see that both reductions can be computed in polynomial time and that in both cases, JULIET has a winning strategy in G if and only if φ is not satisfiable.

The PTIME upper bound in (c) is quite simple. Let $G = (\Sigma, \Gamma, R, T)$ be a game and w an input string; we now need to find out whether exactly one of the function symbols occurring in w can be called to yield a string in the target language. For any

fixed occurrence of a function symbol in w, it is easy to construct a NWT R' that simulates R on the substring rooted at the chosen function symbol and otherwise leaves w unchanged. A type checking test against the target DNWA can then determine in polynomial time (cf. Theorem 8.18 (b)) whether calling that function symbol leads to a string in the target language. Since there are less than $|w|$ occurrences of function symbols in w, polynomially many such type checking tests suffice to determine whether Juliet has a winning strategy in G on w that plays at most one *Call* move. □

Write-once games

As seen in Section 9.4, even in the very simple class of replay-free games with functional relabelling NWT replacement, we still fail to obtain a PTIME upper bound (assuming PTIME ≠ NP). Careful examination of lower bound proofs shows that the semantics for replay-free games still allows for a sort of "hidden replay": On a string of the form $\langle a\rangle\langle b\rangle v\langle/b\rangle\langle/a\rangle$, if Juliet plays *Call* first on $\langle/b\rangle$ then on $\langle/a\rangle$, the substring v undergoes *two* transductions – one from the *Call* to $\langle/b\rangle$, another from the *Call* to $\langle/a\rangle$. This makes it possible to perform any number d of transductions on a given string by enclosing it inside of d nested function symbols.

Excluding this hidden replay yields a very narrow restriction of context-free games, called *write-once* games. In these, no substring may be transduced more than once, i.e. Juliet may only play *Call* on any closing tag $\langle/a\rangle$ if the substring enclosed in it does not contain a substring on which Juliet has played *Call* before. Note that write-once games are always replay-free, but even weaker as far as Juliet's rewriting capabilities are concerned.

A slight adaptation of the proof of Theorem 9.14 shows that JWin remains PSPACE-hard for write-once games with arbitrary relabelling transducers; for functional (and deterministic) relabelling transducers, however, it is possible to prove tractability.

Theorem 9.23. *For the class of write-once games with functional relabelling transducers,* JWin *is in* PTIME.

Proof. The crucial insight for this proof is the fact that games with functional replacement transducers are essentially solitaire games for Juliet– the result of any *Call* on some substring is uniquely determined by that substring, with no choice for Romeo.

We utilise this fact by constructing from a given game $G = (\Sigma, \Gamma, R, T)$ a (generally non-functional) relabelling NWT R_J such that for each input string $w \in \mathrm{WF}(\Sigma)$, the image $R_J(w)$ is exactly the set of all strings that w can be rewritten into by some left-to-right sequence of *Read* and *Call* moves by Juliet. Checking for the existence of a winning strategy for Juliet on w then simply amounts to checking whether $R_J(w) \cap T \neq \varnothing$.

The NWT R_J is constructed from R by a simple modification: In its standard mode of operation, R_J simply reproduces its input. Before any opening tag, however, R_J may choose nondeterministically to start simulating R beginning with the next opening tag, rewrite the input substring up until the corresponding closing tag and then return to simply reproducing its input (or starting another simulation of R).

Now, for any $w, w' \in \mathrm{WF}(\Sigma)$ it holds that $w' \in T(w)$ if and only if there is a write-once strategy for Juliet rewriting w into w', which can be proven by a somewhat involved induction on the structure of w and w'. The two main insights required for this proof are the facts that (i) due to Juliet having complete information and R being functional, Juliet can make her decision whether to *Read* or *Call* already on *opening* tags instead of the corresponding closing tags, and (ii) since R_{J} is a relabelling transducer, the relabelling of a closing tag according to R_{J} is already determined at the corresponding opening tag, which enforces the corresponding rewriting for Juliet to be write-once.

The construction of R_{J} is obviously possible in polynomial time. Furthermore, given R_{J} and an input string w, by Corollary 8.13 and Lemma 8.14 we can construct in polynomial time a polynomial-size ϵ-NWA deciding $R_{\mathrm{J}}(w)$. Intersection nonemptiness with the target language T can then be checked for in polynomial time due to Lemma 8.4 and Theorem 8.5(a). □

9.6. Outlook and bibliographical remarks

This chapter presented an overview over complexity and decidability results for solving context-free games on nested words with transducer-based replacement. For (non-deleting) unrestricted NWT, ϵ-free NWT and relabelling NWT, a complete classification of decidability and complexity was given for games with no replay, bounded replay, and unbounded replay. Additionally, other restrictions arising from lower bound proofs in these "main" settings were also briefly examined.

The first main insight into context-free games provided in this chapter is that adding semantic dependencies between the input and output of function calls greatly complicates the winning problem; in fact, the winning problem with unbounded replay becomes undecidable as soon as the replacement mechanism is allowed to reproduce the called input and insert a single character. For bounded replay, allowing only a limited amount of insertion (such as given by ϵ-free NWT) leads to a non-elementary lower bound at least, and even for replay-free games, the complexity starts at NP-complete for games with functional relabelling transducers (which allow no insertion at all and basically cut Romeo out of the game) and go beyond exponential time with insertion added. While tractable cases exist, they limit the replacement mechanisms so much that the underlying game is effectively uninteresting.

The second main insight is that adding transducers as a replacement mechanism to context-free games opens up an enormous potential for variation beyond what we already saw for the regular replacement and validation cases presented in earlier chapters. As already hinted at in the previous chapter's outlook section, there is a plethora of different tree transducer models and, presumably, an equal amount of possible variations of nested word transducers, so the settings presented in this chapter can be considered as no more than just scratching the surface.

With the practical application in Active XML in mind, a logical next step would be investigating settings where target languages are given by practical schemas for XML, i.e. by the simple nested word automata examined in Chapters 5 and 6. However, since the

transducer model plays a much larger role in determining the complexity of the winning problem than in the regular replacement setting, this would presumably necessitate using a restricted model of NWTs whose domain and image can be described by practical schema languages; to the author, no such transducer model is known so far.

Bibliographical remarks. All results presented in this chapter are solely by the author and were previously published in [Sch16].

10. Conclusion

Context-free games, in their most basic form, are a natural extension of the word problem for context-free grammars into a two-player game; however, as this dissertation has shown, there exists a wide variety of generalisations and extensions beyond that basic form. This dissertation has provided a comprehensive guide to the state of the art in context-free games.

To that end, Chapters 3 and 4 gave an overview of context-free games on strings; Chapter 3 gave a complete complexity-theoretic classification by replay of the winning problem for Juliet with both finite and regular replacement languages, while Chapter 4 characterised the complexity of the problem of determining whether left-to-right strategies for Juliet capture all winning input strings in a given unconstrained game. Moreover, these two chapters introduced the basic proof techniques and algorithmic concepts used for context-free games without function call parameters: The abstraction of subgames into effects, the use of alternating automata, dual effects, and the protest technique as an important tool for lower bound reductions.

This work was then extended to context-free games on nested words in Chapter 6; since the original motivation for examining context-free games comes from Active XML, the nested word setting is much closer to the application than the string setting. It turned out that the proof techniques used for games on flat strings carry over more or less directly into the setting on nested words without function call parameters. While the complexity of the winning problem for Juliet with target languages represented by nested word automata is somewhat increased compared with the flat string setting, restricting target language representations to simple nested word automata (introduced in Chapter 5 to describe practical schema languages for XML) yields the same complexities for the winning problem as in the flat string case.

It should be noted, however, that already for games on flat strings, settings with a tractable winning problem are quite restricted, requiring either no replay at all, or bounded replay and explicitly enumerated finite replacement languages. Already the latter case is somewhat unrealistic for practical applications, as output specifications for service calls will usually be described as schemas of some sort.

This trend that game parameters have to be severely restricted in order to reach a tractable winning problem becomes even more pronounced once function call parameters enter the picture, as evidenced by our examination in Chapter 7 of replay-free games on nested words where function calls are subject to parameter validation. There, tractability can only be obtained by restricting games to a constant number of replacement DTDs; it is, however, doubtful whether this tractability result is of practical use, as the corresponding algorithm has a running time doubly exponential in the number of validation schemas used.

With semantic dependencies between function call parameters and results, modelled as variants of the nested word transducers introduced in Chapter 8, tractability requires almost prohibitively strict restrictions, along the lines of allowing Juliet at most a single function call in each well-nested subword of the input. On the other hand, undecidability results and non-elementary complexity lower bounds are quite easy to come by if replacement transducers allow for insertion of additional symbols and replay is allowed.

Parallel to this rising requirement of increasingly severe restrictions for tractability, we see a rising potential for variation and choice of parameters in the model of context-free games. While for games with regular replacement (Chapters 4 to 6), the basic parameters for context-free games are just replay, representation of target languages and (to a lesser degree) representation of replacement languages, parameter validation (Chapter 7) adds the number and concrete representation of validation schemas, and transducer-based replacement (Chapter 9) opens up an enormous potential for variation in the precise choice of the transducer model, along with other additional restrictions to strategies of Juliet.

Further directions

While this dissertation represents most of the state of the art in context-free games, there still exist several possible avenues for further research.

First, and possibly most interesting for the application in Active XML, is the question whether restrictions of the transducer-based replacement framework to practical schema languages might yield additional tractable restrictions. As the above summary suggests, it is unlikely that tractable cases with replay can be found in this way, but at the very least there might be some tractable cases that are of more interest to practical application than write-once games with functional transducers as presented in Chapter 9. However, a sensible restriction of transducer-based games to practical schema languages would probably require a transducer model whose input and output (i.e. domain and range) can easily be described using practical schema languages, which, to the best of the author's knowledge, still needs to be developed.

On the other hand, context-free games could be further extended beyond strings and trees, and into graphs, data words, or even more general (logical) structures. While the basic idea of context-free games extends to these more general structures easily enough, this extension is not without its challenges. For one thing, even the comparatively simple case of games with regular rewriting requires some notion of regular languages, which is not as canonically agreed upon for, say, graphs as it is for strings and trees (see e.g. [BS05; BJ06]). While monadic second order (MSO) logic might be a sensible candidate, it is not without its problems either, as already evaluating a first-order (FO) formula on graphs is generally PSPACE-complete [Sto74]; therefore, tractable cases are unlikely to be found using MSO logic. In addition to this problem, it remains to be seen whether the tools and techniques developed for games on strings and nested words (such as effects and alternating automata) can be used on more general structures.

Tangentially related to both the extension of context-free games to other structures

and the practical motivation of document rewriting in an uncertain environment, the last years have seen the rise of *Java Script Object Notation (JSON)*[IETF] as an easy-to-use formalism for web data exchange. So far, little theoretical work has been published on JSON (cf. [BRSV17]), but an extension of context-free games to JSON documents might be of interest to both theory and practice.

Finally, and somewhat related to the previous points, it would be interesting to see whether any additional applications for context-free games can be found. Considering the fact that context-free games are such a natural model, it is surprising that so far they have only found application in Active XML and, more recently, in program synthesis. Possibly, extensions to other logical structures might bring out connections to other fields of practical application, but so far, other uses have yet to be found.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[ABM08] Serge Abiteboul, Omar Benjelloun, and Tova Milo. "The Active XML project: an overview". In: *VLDB J.* 17.5 (2008), pp. 1019–1040.

[AM04] Rajeev Alur and P. Madhusudan. "Visibly pushdown languages". In: *STOC*. Ed. by László Babai. ACM, 2004, pp. 202–211.

[AM09] Rajeev Alur and P. Madhusudan. "Adding nesting structure to words". In: *J. ACM* 56.3 (2009).

[AMB05] Serge Abiteboul, Tova Milo, and Omar Benjelloun. "Regular rewriting of Active XML and unambiguity". In: *PODS*. 2005, pp. 295–303.

[AXML] ActiveXML. `http://webdam.inria.fr/axml/`.

[BCGR92] Samuel R. Buss, Stephen A. Cook, A. Gupta, and V. Ramachandran. "An Optimal Parallel Algorithm for Formula Evaluation". In: *SIAM J. Comput.* 21.4 (1992), pp. 755–780.

[BJ06] Dietmar Berwanger and David Janin. "Automata on Directed Graphs: Edge Versus Vertex Marking". In: *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*. Ed. by Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg. Vol. 4178. Lecture Notes in Computer Science. Springer, 2006, pp. 46–60.

[BKMW01] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. *Regular tree and regular hedge languages over unranked alphabets: Version 1, April 3, 2001*. Tech. rep. The Hongkong University of Science and Technology, 2001.

[BLS92] Allan Borodin, Nathan Linial, and Michael E. Saks. "An Optimal On-line Algorithm for Metrical Task System". In: *J. ACM* 39.4 (Oct. 1992), pp. 745–763.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[BPV04] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. "Incremental validation of XML documents". In: *ACM Trans. Database Syst.* 29.4 (2004), pp. 710–751.

[BRSV17] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. "JSON: data model, query languages and schema specification". In: *CoRR* abs/1701.02221 (2017).

[BS05] Franz-Josef Brandenburg and Konstantin Skodinis. "Finite graph automata for linear and boundary graph languages". In: *Theor. Comput. Sci.* 332.1-3 (2005), pp. 199–232.

[BS86] Gérard Berry and Ravi Sethi. "From Regular Expressions to Deterministic Automata". In: *Theor. Comput. Sci.* 48.3 (1986), pp. 117–126.

[BSSK13] Henrik Björklund, Martin Schuster, Thomas Schwentick, and Joscha Kulbatzki. "On optimum left-to-right strategies for active context-free games". In: *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. 2013, pp. 105–116.

[BW92] Anne Brüggemann-Klein and Derick Wood. "Deterministic Regular Languages". In: *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*. 1992, pp. 173–184.

[BW98] Anne Brüggemann-Klein and Derick Wood. "One-Unambiguous Regular Languages". In: *Inf. Comput.* 140.2 (1998), pp. 229–253.

[Ben03] Johan van Benthem. "Logic Games are Complete for Game Logics". In: *Studia Logica* 75.2 (2003), pp. 183–203.

[Boz07] Laura Bozzelli. "Alternating Automata and a Temporal Fixpoint Calculus for Visibly Pushdown Languages". In: *CONCUR- Concurrency Theory, 18th International Conference*. 2007, pp. 476–491.

[CKS81] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. "Alternation". In: *Journal of the ACM* 28.1 (1981), pp. 114–133.

[Chl86] Bogdan S. Chlebus. "Domino-Tiling Games". In: *J. Comput. Syst. Sci.* 32.3 (1986), pp. 374–392.

[Com+] Hubert Comon et al. *Tree Automata Techniques and Applications*. Available on: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th, 2007.

[Coo73] Stephen A. Cook. "A Hierarchy for Nondeterministic Time Complexity". In: *J. Comput. Syst. Sci.* 7.4 (1973), pp. 343–353.

[Cös15] Christian Cöster. "One-pass strategies for context-free games". MA thesis. Technische Universität Dortmund, 2015.

[Don70] John Doner. "Tree Acceptors and Some of Their Applications". In: *J. Comput. Syst. Sci.* 4.5 (1970), pp. 406–451.

[EB97] Peter van Emde Boas. "The convenience of tilings". In: *Complexity, Logic and Recursion*. Vol. 187. Lec. Notes in Pure and App. Math. Routledge, 1997, pp. 331–363.

[FJY90] Abdelaziz Fellah, Helmut Jürgensen, and Sheng Yu. "Constructions for alternating finite automata". In: *International journal of computer mathematics* 35.1-4 (1990), pp. 117–132.

[FRRST10] Emmanuel Filiot, Jean-François Raskin, Pierre-Alain Reynier, Frédéric Servais, and Jean-Marc Talbot. "Properties of Visibly Pushdown Transducers". In: *MFCS*. Ed. by Petr Hlinený and Antonín Kucera. Vol. 6281. Lecture Notes in Computer Science. Springer, 2010, pp. 355–367.

[GKR16] Valentin Goranko, Antti Kuusisto, and Raine Rönnholm. "Game-Theoretic Semantics for Alternating-Time Temporal Logic". In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, Singapore, May 9-13, 2016*. Ed. by Catholijn M. Jonker, Stacy Marsella, John Thangarajah, and Karl Tuyls. ACM, 2016, pp. 671–679.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadéniai Kiadó, Budapest, Hungary, 1984.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, eds. *Automata, Logics, and Infinite Games. A Guide to Current Research*. Springer, 2002.

[Gol77] Leslie M Goldschlager. "The monotone and planar circuit value problems are log space complete for P". In: *ACM SIGACT News* 9.2 (1977), pp. 25–29.

[HK11] Markus Holzer and Martin Kutrib. "Descriptional and computational complexity of finite automata - A survey". In: *Inf. Comput.* 209.3 (2011), pp. 456–470.

[HMM16] Lukás Holík, Roland Meyer, and Sebastian Muskalla. "Summaries for Context-Free Games". In: *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*. Ed. by Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen. Vol. 65. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 41:1–41:16.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)* Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

[IETF] Internet Engineering Task Force. *The JavaScript Object Notation (JSON) Data Interchange Format*. `https://tools.ietf.org/html/rfc7159`.

[JL76] Neil D. Jones and William T. Laaser. "Complete Problems for Deterministic Polynomial Time". In: *Theor. Comput. Sci.* 3.1 (1976), pp. 105–117.

[Kai09] Lukasz Kaiser. "Synthesis for Structure Rewriting Systems". In: *MFCS*. Ed. by Rastislav Královic and Damian Niwinski. Vol. 5734. Lecture Notes in Computer Science. Springer, 2009, pp. 415–426.

[Ken14] Krystian Kensy. "Kontextfreie Spiele mit beschränkter Rundenzahl". MA thesis. Technische Universität Dortmund, 2014.

[Kul10] Joscha Kulbatzki. "Active XML und Untersuchungen perfekter Rewriting-Strategien für kontexfreie Spiele auf Strings". MA thesis. Technische Universität Dortmund, 2010.

[MAABN05] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Frederic Dang Ngoc. "Exchanging intensional XML data". In: *ACM Trans. Database Syst.* 30.1 (2005), pp. 1–40.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. "Taxonomy of XML schema languages using formal language theory". In: *ACM Trans. Internet Techn.* 5.4 (2005), pp. 660–704.

[MMN17] Roland Meyer, Sebastian Muskalla, and Elisabeth Neumann. "Liveness Verification and Synthesis: New Algorithms for Recursive Programs". In: *CoRR* abs/1701.02947 (2017).

[MNS09] Wim Martens, Frank Neven, and Thomas Schwentick. "Complexity of Decision Problems for XML Schemas and Chain Regular Expressions". In: *SIAM J. Comput.* 39.4 (2009), pp. 1486–1530.

[MNSB06] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. "Expressiveness and complexity of XML Schema". In: *ACM Trans. Database Syst.* 31.3 (2006), pp. 770–813.

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. "On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract)". In: *STACS*. 1995, pp. 229–242.

[MSS06] Anca Muscholl, Thomas Schwentick, and Luc Segoufin. "Active Context-Free Games". In: *Theory Comput. Syst.* 39.1 (2006), pp. 237–276.

[Nev02] Frank Neven. "Automata, Logic, and XML". In: *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*. Ed. by Julian C. Bradfield. Vol. 2471. Lecture Notes in Computer Science. Springer, 2002, pp. 2–26.

[PP03] Marc Pauly and Rohit Parikh. "Game Logic - An Overview". In: *Studia Logica* 75.2 (2003), pp. 165–182.

[PV00] Yannis Papakonstantinou and Victor Vianu. "DTD Inference for Views of XML Data". In: *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*. Ed. by Victor Vianu and Georg Gottlob. ACM, 2000, pp. 35–46.

[RS08] Jean-François Raskin and Frédéric Servais. "Visibly Pushdown Transducers". In: *ICALP (2)*. Ed. by Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz. Vol. 5126. Lecture Notes in Computer Science. Springer, 2008, pp. 386–397.

[SS07] Luc Segoufin and Cristina Sirangelo. "Constant-Memory Validation of Streaming XML Documents Against DTDs". In: *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings*. Ed. by Thomas Schwentick and Dan Suciu. Vol. 4353. Lecture Notes in Computer Science. Springer, 2007, pp. 299–313.

[SS15] Martin Schuster and Thomas Schwentick. "Games for Active XML Revisited". In: *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*. 2015, pp. 60–75.

[SS16] Martin Schuster and Thomas Schwentick. "Games for Active XML Revisited". To be published in *Theory Comput Syst* (2016), available online first, doi: 10.1007/s00224-016-9682-4. 2016.

[SV02] Luc Segoufin and Victor Vianu. "Validating Streaming XML Documents". In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. Ed. by Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis. ACM, 2002, pp. 53–64.

[Sav70] Walter J. Savitch. "Relationships Between Nondeterministic and Deterministic Tape Complexities". In: *J. Comput. Syst. Sci.* 4.2 (1970), pp. 177–192.

[Sch07] Thomas Schwentick. "Automata for XML - A survey". In: *J. Comput. Syst. Sci.* 73.3 (2007), pp. 289–315.

[Sch12] Thomas Schwentick. "Foundations of XML Based on Logic and Automata: A Snapshot". In: *Foundations of Information and Knowledge Systems - 7th International Symposium, FoIKS 2012, Kiel, Germany, March 5-9, 2012. Proceedings*. Ed. by Thomas Lukasiewicz and Attila Sali. Vol. 7153. Lecture Notes in Computer Science. Springer, 2012, pp. 23–33.

[Sch16] Martin Schuster. "Transducer-Based Rewriting Games for Active XML". In: *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*. Ed. by Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier. Vol. 58. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 83:1–83:13.

[Sto74] Larry J. Stockmeyer. "The Complexity of Decision Problems in Automata Theory". Dissertation. Department of Electrical Engineering, MIT, 1974.

[TVY08] Alex Thomo, Srinivasan Venkatesh, and Ying Ying Ye. "Visibly Pushdown Transducers for Approximate Validation of Streaming XML". In: *Foundations of Information and Knowledge Systems, 5th International Symposium, FoIKS 2008, Pisa, Italy, February 11-15, 2008, Proceedings*. Ed. by Sven Hartmann and Gabriele Kern-Isberner. Vol. 4932. Lecture Notes in Computer Science. Springer, 2008, pp. 219–238.

[TW68] James W. Thatcher and Jesse B. Wright. "Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic". In: *Mathematical Systems Theory* 2.1 (1968), pp. 57–81.

[Tho90] Wolfgang Thomas. "Automata on Infinite Objects". In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. Elsevier and MIT Press, 1990, pp. 133–192.

[Vea97] Margus Veanes. *On computational complexity of basic decision problems of finite tree automata*. Tech. rep. UPMAIL Technical Report 133, Uppsala University, Computing Science Department, 1997.

[Wal01] Igor Walukiewicz. "Pushdown Processes: Games and Model-Checking". In: *Inf. Comput.* 164.2 (2001), pp. 234–263.

[Wal02] Johannes Waldmann. "Rewrite Games". In: *RTA*. Ed. by Sophie Tison. Vol. 2378. Lecture Notes in Computer Science. Springer, 2002, pp. 144–158.

[Yu97] Sheng Yu. "Regular languages". In: *Handbook of formal languages*. Springer, 1997, pp. 41–110.