Games for Active XML Revisited

Martin Schuster and Thomas Schwentick

TU Dortmund, Germany

Abstract

The paper studies the rewriting mechanisms for intensional documents in the Active XML framework, abstracted in the form of *active context-free games*. The *safe rewriting* problem studied in this paper is to decide whether the first player, JULIET, has a winning strategy for a given game and (nested) word; this corresponds to a successful rewriting strategy for a given intensional document. The paper examines several extensions to active context-free games.

The primary extension allows more expressive schemas (namely XML schemas and regular nested word languages) for both target and replacement languages and has the effect that games are played on nested words instead of (flat) words as in previous studies. Other extensions consider validation of input parameters of web services, and an alternative semantics based on insertion of service call results.

In general, the complexity of the safe rewriting problem is highly intractable (doubly exponential time), but the paper identifies interesting tractable cases.

1998 ACM Subject Classification F.2.m Miscellaneous, F.4.2 Grammars and Other Rewriting Systems, H.3.5 Online Information Services

Keywords and phrases Active XML, Computational Complexity, Nested Words, Rewriting Games, Semistructured Data

Digital Object Identifier 10.4230/LIPIcs.ICDT.2015.60

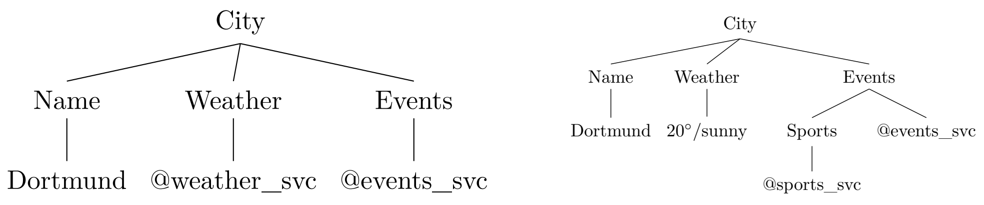
1 Introduction

Scientific context

This paper contributes to the theoretical foundations of intensional documents, in the framework of *Active XML* [1]. It studies game-based abstractions of the mechanism transforming intensional documents into documents of a desired form by calling web services. One form of such games has been introduced under the name *active context-free games* in [11] as an abstraction of a problem studied in [9].[1] The setting in [9] is as follows: an *Active XML* document is given, where some elements consist of functions representing web services that can be called. The goal is to rewrite the document by a series of web service calls into a document matching a given *target schema*.

Towards an intuition of Active XML document rewriting, consider the example in Figure 1 of an online local news site dynamically loading information about weather and local events (adapted from [9] and [11]). Figure 1a shows the initial Active XML document for such a site, containing *function nodes* which refer to a weather and an event service, respectively, instead of concrete weather and event data. After a single function call to each of these services has been materialised, the resulting document may look like the one depicted in Figure 1b. Note that the rewritten document now contains new function nodes; further rewriting might be

[1] Actually, the two notions were introduced in the respective conference papers.



(a) Example document before rewriting. **(b)** Same document after function calls.

Figure 1 Example of Active XML rewriting.

necessary to reach a document in a given target schema (which could, for instance, require that the document contains at least one indoor event if the weather is rainy).

Modelling this rewriting problem as a game follows the approach of dealing with uncertainty by playing a "game against nature": We model the process intended to rewrite a given document into a target schema by performing function calls as a player (JULIET). As her moves, she chooses which function nodes to call, and her goal is to reach a document in the target schema. Returns of function calls, on the other hand, are chosen (in accordance with some schema for each called service) by an antagonistic second player (ROMEO), whose goal is to foil JULIET. The question whether a given document can always be rewritten into the target schema may then be solved by deciding whether JULIET has a winning strategy. More specifically, given an input document, target schema and return schemas for function calls, there should exist a *safe rewriting* algorithm that always rewrites the input document into the target schema, no matter the concrete returns of function calls, if and only if JULIET has a winning strategy in the corresponding game.[2]

In [9], the target schema is represented by an XML document type definition (DTD). It was argued that, due to the restricted nature of DTDs, the problem can be reduced to a rewriting game on strings where, in each move a single symbol is replaced by a string, the set of allowed replacement strings for each symbol is a regular language and the target language is regular[3], as well.

In [11] the complexity of the problem to determine the winner in such games (mainly with finite replacement languages) was studied. Whereas this problem is undecidable in general, there are important cases in which it can be solved, particularly if JULIET chooses the symbols to be replaced in a left-to-right fashion. In and after [11, 9], research very much concentrated on games on strings (and thus on the setting with DTDs). Furthermore, to achieve tractability, a special emphasis was given to the restriction to *bounded* strategies, in which the recursion depth with respect to web service calls is bounded by some constant.

Our approach

The aim of this paper is to broaden the scope and extend the investigation of games for Active XML in several aspects. First of all, we consider stronger schema languages (compared to DTDs) such as XML Schema and Relax NG, due to their practical importance. To allow for this extension, our games are played on nested words [3].[4]

[2] It is hard to give a precise statement of *safe rewriting* that does not already involve games, but we hope that the general idea of this statement becomes sufficiently clear.
[3] More precisely, it should be given by a *deterministic* regular expression.
[4] More precisely: word encodings of nested words in the sense of [3].

Table 1 Summary of complexity results. All results are completeness results.

	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

Furthermore, we study the impact of the validation of input parameters for web service calls (partly considered already in [9]), and investigate an alternative semantics, where results of web service calls are inserted next to the node representing the web service, as opposed to replacing that node.

As we are particularly interested in the identification of tractable cases, we follow the previous line of research by concentrating on strategies in document order (left-to-right strategies) and by considering bounded strategies (*bounded replay*) and strategies in which no calls in results from previous web service calls are allowed (*no replay*). However, we also pinpoint the complexity of the general setting.

As a basic intuition for the concept of replay, consider again the online news site example from Figure 1, and assume that the schema for the event service's returns is (partially) given by @event_svc → (Sports|Movie)@event_svc, i.e. the event service allows for dynamic loading of additional results. A strategy with no replay would not be allowed to fetch any additional results in the situation of Figure 1b, while a strategy with bounded replay k (for some constant k) could load up to k more events after the first. A strategy with unbounded replay would be able to fetch an arbitrary number of results, but might lead to a rewriting process that does not terminate if unsuccessful.

Our contributions

Our complexity results with respect to stronger schema languages are summarised in Table 1. In the general setting, the complexity is very bad: doubly exponential time. However, there are tractable cases for XML Schema: replay-free strategies in general and strategies with bounded replay in the case of finite replacement languages (that is, when there are only finitely many possible answers, for each web service). It should be noted that the PSPACE-hardness result for the case with DTDs, bounded replay and infinite replacement languages indicates that the respective PTIME claim in [9] is wrong.

In the setting where web services come with an input schema that restricts the parameters of web service calls, we only study replay-free strategies. It turns out that this case is tractable if all schemas are specified by DTDs and the number of web services is bounded. On the other hand, if the desired document structure is specified by an XML Schema or the number of function symbols is unbounded, the task becomes PSPACE-hard.

For insertion-based semantics, we identify an undecidable setting and establish a correspondence with the standard "replacement" semantics, otherwise.

As a side result of independent interest, we show that the word problem for alternating nested word automata is PSPACE-complete.

Related work

We note that the results on flat strings in this paper do not directly follow from the results in [11], as [11] assumed target languages given by DFAs as opposed to *deterministic* regular expressions, which are integral to both DTDs and more expressive XML schema languages. However, the techniques from [11] can be adapted.

More related work for active context-free games than the papers mentioned so far is discussed in [11]. Further results on active context-free games in the "flat strings" setting can be found in [2, 4]. A different form of 2-player rewrite games are studied in [13]. More general *structure rewriting games* are defined in [7].

Organisation

We give basic definitions in Section 2. Games with regular schema languages (given by nested word automata) are studied in Section 3, games in which the schemas are given as DTDs or XML Schemas are investigated in Section 4. Validation of parameters and insertion of web service results are considered in Section 5. Due to space restrictions, most proofs are omitted here. An appendix containing the omitted proofs can be found in [12].

Acknowledgements

We would like to thank the anonymous reviewers for their insightful and constructive comments. We are grateful to Nils Vortmeier and Thomas Zeume for careful proof reading, and to Krystian Kensy for checking our proof of Proposition 18 (b) and for pinpointing the problems in the algorithm of [9] as part of his Master's thesis.

2 Preliminaries

For any natural number $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, \dots, n\}$. Where M is a (finite) set, $\mathcal{P}(M)$ denotes the powerset of M, i.e. the set of all subsets of M. For an alphabet Σ, we denote the set of finite strings over Σ by Σ^* and ϵ denotes the empty string.

Nested words

We use nested words[5] as an abstraction of XML documents [3]. For a finite alphabet Σ, $\langle\Sigma\rangle \overset{\text{def}}{=} \{\langle a\rangle \mid a \in \Sigma\}$ denotes the set of all *opening* Σ*-tags* and $\langle/\Sigma\rangle \overset{\text{def}}{=} \{\langle/a\rangle \mid a \in \Sigma\}$ 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 (informally) 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. A nested string w is *rooted*, if its corresponding forest is a tree. In a nested string $w = w_1 \dots w_n \in \mathrm{WF}(\Sigma)$, 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. To stress the distinction from nested strings 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 avoid these terms to avoid confusion with Read and Call moves used in context-free games (see below).

[5] Our definition of nested words corresponds to word encodings of well-matched nested words in [3].

Context-free games

A *context-free game on nested words (cfG)* $G = (\Sigma, \Gamma, R, T)$ consists[6] of a finite alphabet Σ, a set $\Gamma \subseteq \Sigma$ of *function symbols*, a *rule set* $R \subseteq \Gamma \times \mathrm{WF}(\Sigma)$ and a *target language* $T \subseteq \mathrm{WF}(\Sigma)$. We will only consider the case where T and, for each symbol $a \in \Gamma$, the set $R_a \overset{\text{def}}{=} \{u \mid (a, u) \in R\}$ is a non-empty regular nested word language, to be defined in the next subsection.

A play of G is played by two players, Juliet and Romeo, on a word $w \in \mathrm{WF}(\Sigma)$. In a nutshell, Juliet moves the focus along w in a left-to-right manner and decides, for every closing tag[7] $\langle/a\rangle$ whether she plays a *Read* or, in case $a \in \Gamma$, a *Call* move. In the latter case, Romeo then replaces the rooted word ending at the position of $\langle/a\rangle$ with some word $v \in R_a$ and the focus is set on the first symbol of v. In case of a Read move (or an opening tag) the focus just moves further on. Juliet wins a play if the word obtained at its end is in T.

Towards a formal definition, 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 the first symbol of v is *the current position.* A *winning configuration* for Juliet is a configuration $\kappa = (\mathrm{J}, u, \epsilon)$ with $u \in T$. The configuration $\kappa' = (p', u', v')$ 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 $y' \in R_a$ (Romeo plays y');[8]

The *initial configuration* of game G for string w is $\kappa_0(w) \overset{\text{def}}{=} (\mathrm{J}, \epsilon, w)$. A *play* of G is either an infinite sequence $\Pi = \kappa_0, \kappa_1, \ldots$ or a finite sequence $\Pi = \kappa_0, \kappa_1, \ldots, \kappa_k$ of configurations, where, for each $i > 0$, $\kappa_{i-1} \to \kappa_i$ and, in the finite case, κ_k has no successor configuration. In the latter case, Juliet *wins* the play if κ_k is of the form $(\mathrm{J}, u, \epsilon)$ with $u \in T$, in all other cases, Romeo wins.

Strategies

A *strategy* for player $p \in \{\mathrm{J}, \mathrm{R}\}$ maps prefixes $\kappa_0, \kappa_1, \ldots, \kappa_k$ of plays, where κ_k is a p-configuration, to allowed moves. We denote strategies for Juliet by $\sigma, \sigma', \sigma_1, \ldots$ and strategies for Romeo by $\tau, \tau', \tau_1, \ldots$

A strategy σ is *memoryless* if, for every prefix $\kappa_0, \kappa_1, \ldots, \kappa_k$ of a play, the selected move $\sigma(\kappa_0, \kappa_1, \ldots, \kappa_k)$ only depends on κ_k. As context-free games are reachability games we only need to consider memoryless games; see, e.g., [6].

▶ **Proposition 1.** *Let G be a context-free game, and w a string. Then either* Juliet *or* Romeo *has a winning strategy on w, which is actually memoryless.*

[6] Some of the following definitions are taken from [4].
[7] It is easy to see that the winning chances of the game do not change if we allow Juliet 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 she has full information.
[8] We note that a Call move on $\langle/a\rangle$ in a substring of the form $\langle a\rangle y\langle/a\rangle$ actually deletes the substring y along with the opening and closing a-tags. This is consistent with the AXML intuition of the subtree rooted at a function node getting replaced when the function node is called.

Therefore, in the following, strategies σ for JULIET map configurations κ to moves $\sigma(\kappa) \in \{\text{Call}, \text{Read}\}$ and strategies τ for ROMEO map configurations κ to moves $\tau(\kappa) \in \text{WF}(\Sigma)$.

For configurations κ, κ' and strategies σ, τ we write $\kappa \xrightarrow{\sigma,\tau} \kappa'$ if κ' is the unique successor configuration of κ determined by strategies σ and τ. Given an initial word w and strategies σ, τ the play[9] $\Pi(\sigma, \tau, w) \overset{\text{def}}{=} \kappa_0(w) \xrightarrow{\sigma,\tau} \kappa_1 \xrightarrow{\sigma,\tau} \cdots$ is uniquely determined. If $\Pi(\sigma, \tau, w)$ is finite, we denote the word represented by its final configuration by $\text{word}_G(w, \sigma, \tau)$.

A strategy σ for JULIET is *finite* on string w if the play $\Pi(\sigma, \tau, w)$ is finite for every strategy τ of ROMEO. It is a *winning strategy* on w if JULIET wins the play $\Pi(\sigma, \tau, w)$, for every τ of ROMEO. A strategy τ for ROMEO is a *winning strategy* for w if ROMEO wins $\Pi(\sigma, \tau, w)$, for every strategy σ of JULIET. We only consider finite strategies for JULIET, due to JULIET'S winning condition. We denote the set of all finite strategies for JULIET in the game G by $\text{STRAT}_\text{J}(G)$, and the set of all strategies for ROMEO by $\text{STRAT}_\text{R}(G)$.

The *Call depth* of a play Π is the maximum nesting depth of Call moves in Π, if this maximum exists. 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. For a strategy σ of JULIET and a string $w \in \text{WF}(\Sigma)$, the *Call depth* $\text{Depth}^G(\sigma, w)$ of σ on w is the maximum Call depth in any play $\Pi(\sigma, \tau, w)$. A strategy σ has *k-bounded Call depth* if $\text{Depth}^G(\sigma, w) \leq k$ for all $w \in \text{WF}(\Sigma)$. We denote by $\text{STRAT}^k_\text{J}(G)$ 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*. For technical reasons, we need to use Call depth for some formal proofs and definitions, but we will stick with the more intuitive concept of replay wherever possible.

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

Nested word automata

A *nested word automaton (NWA)* $A = (Q, \Sigma, \delta, q_0, F)$ [3] is basically a pushdown automaton which performs a push operation on every opening tag and a pop operation on every closing tag, and in which the pushdown symbols are just states. More formally, A consists of a set Q of *states*, an alphabet Σ, a *transition function* δ, an *initial state* $q_0 \in Q$ and a set $F \subseteq Q$ of *accepting states*. The function δ is the union of a function $(Q \times \langle \Sigma \rangle) \to \mathcal{P}(Q \times Q)$ and a function $(Q \times Q \times \langle /\Sigma \rangle) \to \mathcal{P}(Q)$.

A *configuration* κ of A is a tuple $(q, \alpha) \in Q \times Q^*$, with a *linear state* q and a sequence α of *hierarchical states*, reflecting the pushdown store. A *run of A on $w = w_1 \dots w_n \in WF(\Sigma)$* is a sequence $\kappa_0, \dots, \kappa_n$ 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 $w_i = \langle a \rangle$, $(q_i, p) \in \delta(q_{i-1}, \langle a \rangle)$ (for some $p \in Q$), and $\alpha_i = p\alpha_{i-1}$, or
- if $w_i = \langle /a \rangle$, $q_i \in \delta(q_{i-1}, p, \langle /a \rangle)$ (for some $p \in Q$), 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 $L(A) \subseteq \text{WF}(\Sigma)$ 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 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 $\delta(q, \langle a \rangle) = \{(q', p')\}$

[9] As the underlying game G will always be clear from the context, our notation does not mention G explicitly.

(and accordingly for $\delta(q, p, \langle/a\rangle)$), and $\delta^*(p, w) = q$ if q is the unique state, for which $(p, \epsilon) \overset{w}{\rightsquigarrow}_A (q, \epsilon)$.

An NWA is in *normal form* if every transition function $\delta(p, \langle a\rangle)$ only uses pairs of the form (q, p). Informally, when A reads an opening tag it always pushes its current state (before the opening tag) and therefore can see this state when it reads the corresponding closing tag. As in this case the hierarchical state is just the origin state p of the transition, we write $\delta(p, \langle a\rangle) = q$ as an abbreviation of $\delta(p, \langle a\rangle) = (q, p)$, for DNWAs in normal form.

▶ **Lemma 2.** *There is a polynomial-time algorithm that computes for every deterministic NWA an equivalent deterministic NWA in normal form.*

Algorithmic Problems

In this paper, we study the following algorithmic problem JWin($\mathcal{G}$) for various classes $\mathcal{G}$ of context-free games.

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

A class $\mathcal{G}$ of context-free games in 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 always discuss the impact of the replacement language representations separately. In our most general setting, investigated in Section 3, target languages are represented by deterministic nested word automata, and replacement languages by (not necessarily deterministic) nested word automata. We do not consider the representation of target languages by non-deterministic NWAs, as (1) already for DNWAs the complexity is very high in general, and (2) we can show that even in the replay-free case the complexity would become EXPTIME-complete. We usually denote the automata representing the target and replacement languages by $A(T)$ and $A(R_a)$, respectively.

In Section 4 we study the cases where T is given as an XML Schema or a DTD. In each setting, we consider the cases of unrestricted replay, bounded replay (Call depth k, for some k), and no replay (Call depth 1). 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.

If the class $\mathcal{G}$ of games is clear from the context, we often simply write JWin instead of JWin($\mathcal{G}$).

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. $|G| = |\Sigma| + |R| + |A(T)|$.

3 Games with regular target languages

We first consider our most general case, where target languages are given by DNWAs, replacement languages by NWAs and replay is unrestricted, because the algorithm that we develop for this case can be adapted (and sped up) for many of the more restricted cases. It is important to note that our results do not *rely* on the presentation of schemas as nested word automata. In fact, in Section 4, we will assume that the target schema is given as an XML

Schema or a DTD. However, for our algorithms nested word automata are handy to represent (linearisations of) regular tree languages and therefore in *this* section target languages are represented by NWAs. We emphasize that deterministic bottom-up tree automata can be translated into deterministic NWAs in polynomial time [3].

This generic algorithm works in two main stages for a given cfG G and word w. It first analyses the game G and aggregates all necessary information in a so-called *call effect* C. Then it uses C to decide whether JULIET has a winning strategy in the game G on w.

The call effect C only depends on G and contains, for every function symbol f and every state q of the $A(T)$, all possible effects of the subgame starting with a Call move of JULIET on some symbol $\langle /f \rangle$ on the target language T, under the assumption that the sub-computation of $A(T)$ on the word yielded by the game from $\langle /f \rangle$ starts in state q. More precisely, it summarises which sets S of states JULIET can enforce by some strategy σ, where each S is a set of states of $A(T)$ that ROMEO might enforce with a counter strategy against σ.

The first stage of the algorithm consists of an inductive computation in which successive approximations $C^1, C^2, \ldots$ of C are computed, where C^i is the restriction of C to strategies of JULIET of Call depth i. The size of call effects and the number of iterations are at most exponential in $|G|$. However, the first stage can not be performed in exponential time as a single iteration might take doubly exponential time in $|G|$. It turns out through our corresponding lower bound that single iterations can not be done faster.

At the end of the first stage, the algorithm computes an alternating NWA A_G (of exponential size) from C that decides the set $\mathrm{JWin}(G)$. In the second stage, A_G is evaluated on w, taking at most polynomial space in $|A_G|$ and $|w|$.

A restriction of games to bounded replay does not improve the general complexity of the problem, as this is dominated by the doubly exponential effort of a single iteration. However, for replay-free games, no iterations are needed, the initial call effect C^1 is of polynomial size and can easily be computed and therefore, in this case, the overall complexity is dominated by the second stage, yielding a polynomial-space algorithm.

Altogether we prove the following theorem in this section.

▶ **Theorem 3.** *For the class of unrestricted games* $\mathrm{JWIN}(\mathcal{G})$ *is*

(a) 2-EXPTIME-*complete with unbounded replay,*
(b) 2-EXPTIME-*complete with bounded replay, and*
(c) PSPACE-*complete without replay.*

The rest of this section gives a proof sketch for Theorem 3.

Before we describe the generic algorithm in more detail, we discuss the very natural and more direct approach by alternating algorithms, in which a strategy for JULIET is nondeterministically guessed and the possible moves of ROMEO are taken care of by universal branching. In our setting of context-free games, there are the following obstacles to this approach: (1) ROMEO can, in general, choose from an infinite number of (and thus arbitrarily long) strings in R_a, for the current a, and (2) it is not a priori clear that such algorithms terminate 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 basically 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 the remaining cases (in particular in those cases considered in this section), the algorithms use abstracted moves instead of the actual replacement moves of the game. The two stages that were sketched above, then come very naturally: first, the abstraction has to be computed, then it can be used for the actual alternating computation.

Our abstraction from actual cfGs is based on the simple observation that instead of knowing the final word $\text{word}_G(w, \sigma, \tau)$ that is reached in a play $\Pi(\sigma, \tau, w)$, it suffices to know whether $\delta^*(q_0, \text{word}_G(w, \sigma, \tau)) \in F$ to tell 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) can thus be summarised by $\text{states}_G(q_0, w, \sigma) \overset{\text{def}}{=} \{\delta^*(q_0, \text{word}_G(w, \sigma, \tau)) \mid \tau \in \text{STRAT}_\text{R}(G)\}$.

To this end, it will be particularly useful to study the (abstractions of) possible outcomes of *subgames* that start from a Call move on some tag $\langle/a\rangle$ until the focus moves to the symbol after $\langle/a\rangle$.

▶ **Definition 4.** For a cfG $G = (\Sigma, \Gamma, R, T)$ with a deterministic target NWA $A(T) = (Q, \Sigma, \delta, q_0, F)$, the *call effect* $\mathcal{C}[G] : \Gamma \times Q \to \mathcal{P}(\mathcal{P}(Q))$ is defined, for every $a \in \Gamma$, $q \in Q$, by

$$\mathcal{C}[G](a, q) \overset{\text{def}}{=} [\{\text{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma) \mid \sigma \in \text{STRAT}_{\text{J,Call}}(G)\}]_{\text{min}},$$

where $\text{STRAT}_{\text{J,Call}}(G)$ contains all strategies of JULIET that start by playing Read on $\langle a\rangle$ and Call on $\langle/a\rangle$, and the operator $[\cdot]_{\text{min}}$ removes all non-minimal sets from a set of sets.

We next describe how to compute $\mathcal{C}[G]$ from a given cfG G. As already mentioned, our algorithm follows a fixpoint-based approach. It computes inductively, for $k = 1, 2, \dots$ the call effect of the restricted game of maximum Call depth k. We show that the fixpoint reached by this process is the actual call effect $\mathcal{C}[G]$.

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

$$\mathcal{C}^k[G](a, q) \overset{\text{def}}{=} \left[\{\text{states}_G(q, \langle a\rangle\langle/a\rangle, \sigma) \mid \sigma \in \text{STRAT}^k_{\text{J,Call}}(G)\}\right]_{\text{min}}.$$

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

▶ **Lemma 5.** *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.

This just follows from the definitions, as ROMEO can choose any string from R_a.

We next describe how each $\mathcal{C}^{k+1}[G]$ can be computed from $\mathcal{C}^k[G]$. The algorithm uses alternating nested word automata (ANWAs) which we will now define.

An *alternating nested word automaton (ANWA)* $A = (Q, \Sigma, \delta, q_0, F)$ is defined like an NWA, except that the two parts of δ map $(Q \times \langle\Sigma\rangle)$ into $\mathcal{B}^+(Q \times Q)$ and $(Q \times Q \times \langle/\Sigma\rangle)$ into $\mathcal{B}^+(Q)$, respectively, 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 $\mathcal{B}^+(Q \times Q)$).

The semantics of ANWA is defined via *runs*, which require the notion of *tree domains*. A tree domain is a prefix-closed language $D \subseteq \mathbb{N}^*$ of words over $\mathbb{N}$ such that, if $wk \in D$ for some $w \in D, k \in \mathbb{N}$, then also $wj \in D$ for all $j < k$. 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 v with k children, then $w1, \dots, wk \in D$ address those children.

For any function $\lambda : D \to (Q \cup (Q \times Q))$ and node address $x \in D$, we denote by $\overline{\lambda(x)}$ the linear state component of $\lambda(x)$, i.e. if $\lambda(x) = q$ or $\lambda(x) = (q, p)$ for some $p, q \in Q$, then $\overline{\lambda(x)} = q$.

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

- if $w_{i+1} \in \langle\Sigma\rangle$, then $\{\lambda(x \cdot 1), \dots, \lambda(x \cdot \ell)\} \models \delta(\overline{\lambda(x)}, w_{i+1})$, and
- if $w_{i+1} \in \langle/\Sigma\rangle$ with associated opening tag w_j, and $\lambda(y) = (q, p)$ for some $p, q \in Q$ (where y is the prefix of x of length j), then $\{\lambda(x \cdot 1), \dots, \lambda(x \cdot \ell)\} \models \delta(\overline{\lambda(x)}, p, w_{i+1})$.

An ANWA A *accepts* a nested word w if there is a run (D, λ) over w such that $\lambda(x) \in F$, for every $x \in D$ of length $|w|$.

ANWAs are used twice in the generic algorithm, first, to inductively compute $\mathcal{C}^{k+1}[G]$ from $\mathcal{C}^k[G]$, second to actually decide JWin(G), given $\mathcal{C}[G]$. The following proposition will be crucial, in both cases.

▶ **Proposition 6.** *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 $L(A_{\mathcal{C}[G]}) = JWin(G)$.*

The computation of $\mathcal{C}^{k+1}[G]$ from $\mathcal{C}^k[G]$ involves a non-emptiness test for ANWAs, the second stage a test whether $w \in L(A_{\mathcal{C}[G]})$. Therefore, both of the following complexity results for ANWAs influence the complexity of our algorithms.

▶ **Proposition 7.**
(a) *Non-emptiness for ANWAs is* 2-EXPTIME-*complete.*
(b) *The membership problem for ANWAs is* PSPACE-*complete.*

Statement (a) follows immediately from the corresponding result for visibly pushdown automata in [5], statement (b) is new, to the best of our knowledge, and seems to be interesting in its own right. It is shown in [12].

Now we continue describing the ingredients of the first stage of the generic algorithm.

▶ **Lemma 8.** *Given a state $q \in Q$, an alphabet symbol $a \in \Gamma$, and $\mathcal{C}^k[G]$, for some $k \geq 1$, the call effect $\mathcal{C}^{k+1}[G](a, q)$ can be computed in doubly exponential time in $|G|$.*

By Lemmas 5 and 8, one can compute $\mathcal{C}^k[G]$ inductively, for every $k \geq 1$. By definition it holds, for every q and a, that $\mathcal{C}^k[G](a, q)$ is contained in the closure of $\mathcal{C}^{k+1}[G](q, a)$ under supersets. As there are $\leq 2^{|Q|}$ 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 $a \in \Sigma$, $q \in Q$:

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

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

▶ **Proposition 9.** *For every cfG G it holds: $\mathcal{C}^*[G] = \mathcal{C}[G]$.*

Now we can give a (high-level) proof for Theorem 3.

Proof of Theorem 3. We first justify the upper bounds. Let G be a cfG and w a word. By Lemma 5, $\mathcal{C}^1[G]$ can be computed in polynomial time from G. For the replay-free case, we can immediately construct an ANWA for JWin(G) and evaluate it on w, yielding a PSPACE upper bound by Proposition 7.

For (a) and (b), $\mathcal{C}[G]$ ($\mathcal{C}^k[G]$, respectively) can be computed in doubly exponential time, A_C can be computed in exponential time (in the size of G), and whether $w \in L(A_C)$ can then be tested in polynomial space in $|A_C|$ and $|w|$, that is, in at most exponential space in $|G|$ and $|w|$.

That these upper bounds can not be considerably improved, is stated in the following proposition, thereby completing the proof of Theorem 3. ◀

▶ **Proposition 10.** *For the class of unrestricted games* JWin *is*
(a) 2-EXPTIME-*hard with bounded replay, and*
(b) PSPACE-*hard with no replay.*

Claims (a) and (b) of Proposition 10 follow from the corresponding parts of Proposition 7; in the proof, we construct from an ANWA A a replay-free cfG simulating A on any input word w (yielding claim (b)) and explain how replay can be added to that game to find and verify a witness for the non-emptiness of A, if one exists (yielding claim (a)).

For finite (and explicitly given) replacement languages the complexity changes considerably in the cases with replay, but not in the replay-free case.

▶ **Proposition 11.** *For the class of unrestricted games with finite replacement languages,* JWin$(\mathcal{G})$ *is*
(a) EXPTIME-*complete with unbounded replay, and*
(b) PSPACE-*complete with bounded or without replay.*

The upper bound in (a) follows as for finite replacement languages $\mathcal{C}^{k+1}[G](a,q)$ can be computed from $\mathcal{C}^{k}[G](a,q)$ in polynomial space[10]. The PSPACE upper bound in (b) can then be achieved by the usual "recomputation technique" of space-bounded computations.

The lower bound in (a) already holds for flat words (see Theorem 4.3 in [11]). The lower bound in (b) follows as the proof of Proposition 10 only uses finite replacement languages.

As our algorithms generally construct ANWAs deciding JWin(G), the data complexity for JWin is in PSPACE for all cases considered in this section due to Proposition 7.

4 Games with XML Schema target languages

The results of Section 3 provide a solid foundation for our further studies, but the setting studied there suffers from two problems: (1) the complexities are far too high (at least for games with replay) and (2) the assumption that target and replacement languages are specified by (D)NWAs is not very realistic. In this section, we address both issues at the same time: when we require that target languages are specified by typical XML schema languages (DTD or XML Schema), we get considerably better complexities.

The better complexities basically all have the same reason: XML Schema target languages can be described by a restriction of nested word automata, which we call *simple* below. This restriction translates to the alternating NWAs corresponding to call effects. For simple ANWAs, however, the two basic algorithmic problems, Non-emptiness and Membership have dramatically better complexities: PSPACE and PTIME as opposed to 2-EXPTIME and PSPACE, respectively. We emphasise that, in accordance with the official standards, our definitions for DTDs and XML Schema require *deterministic* regular expressions.

Altogether, we prove the following complexity results.

▶ **Theorem 12.** *For classes of games with XML Schemas or DTDs, respectively,* JWin *is*

[10] It is worth noting that this upper bound even holds if the finite replacement language is not explicitly given, but represented by NWAs.

(a) EXPTIME-*complete for unbounded replay,*
(b) PSPACE-*complete for bounded replay, and*
(c) PTIME-*complete (under logspace-reductions) without replay.*

Here, the lower bounds are proven for DTDs, and the upper bounds for XML Schemas.

The lower bound in Theorem 12 (b) for the case of games with DTDs contradicts the statement of a PTIME algorithm in Section 4.3 of [9] (unless PTIME = PSPACE).[11]

Before we describe the proof of Theorem 12, we first define *single-type tree grammars* and *local tree grammars* as well-established abstractions of XML Schema and DTDs, respectively (see, e.g., [10]). However, we will refer to grammars of these types as XML Schemas and DTDs, respectively.

▶ **Definition 13.** 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 *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.

We omit the definition of the formal semantics of regular tree grammars. The nested word language $L(T)$ described by T is just the set of linearisations of trees of the tree language that is defined in the standard way.

We next define *simple* DNWAs, a restriction of DNWAs 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 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 [8], we could call such DNWAs *ancestor-sibling-based* but we prefer the term *simple* for simplicity.

▶ **Definition 14.** A deterministic NWA $A(T) = (Q, \Sigma, \delta, q_0, F)$ in normal form is *simple* (SNWA) if there exist a *type alphabet* Δ and *state set* P with $Q \subseteq P \times \Delta$, a *local acceptance function* $F_{\text{loc}} : \Sigma \to \mathcal{P}(Q)$, a *target state function* $t : Q \times \Sigma \to Q$ and a *failure state* $\perp \in Q \setminus F$, such that the following conditions are satisfied for every $a \in \Sigma$:

- for every $p, p' \in P, X \in \Delta$: $\delta((p, X), \langle a \rangle) = \delta((p', X), \langle a \rangle)$;
- for every $q \in F_{\text{loc}}(a)$: $\delta(q, p, \langle /a \rangle) = t(p, a)$;

[11] A close inspection of the construction in the proof in [9] reveals that the automaton constructed there does not deal correctly with the alternation between the choices of ROMEO and JULIET. More precisely, the automaton allows ROMEO to let the suffix of a replacement string depend on the choices of JULIET on its prefix.

- for every $q \in Q \setminus F_{\mathrm{loc}}(a)$: $\delta(q, p, \langle /a \rangle) = \bot$ and
- for every $q \in Q$: $\delta(\bot, \langle a \rangle) = \delta(\bot, q, \langle /a \rangle) = \bot$.
- for every $(p, X) \in Q$: $t((p, X), a) = (p', X)$ for some $p' \in P$.

A cfG is called *simple* if its target DNWA is simple.

▶ **Proposition 15.** *From every single-type tree grammar T, a simple DNWA A can be computed in polynomial time, such that $L(A) = L(T)$.*

The following adaptation of the notion of *simplicity* to ANWAs is a bit technical. It will guarantee however that the ANWAs obtained from simple games are simple and have reasonable complexity properties.

▶ **Definition 16.** An ANWA $A = (Q, \Sigma, \delta, q_0, F)$ with $Q \subseteq P \times \Delta$ (for some state set P and type alphabet Δ) is *simple* (SANWA), if it has the following two properties.

- *(Horizontal simplicity)* There are a *local acceptance function* $F_{\mathrm{loc}} : \Sigma \to \mathcal{P}(Q)$, 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 $q' \neq q_?$;
  - $\delta(q, q_?, \langle /a \rangle) = \begin{cases} \text{true}, & \text{if } q \in F_{\mathrm{loc}}(a) \\ \text{false}, & \text{if } q \notin F_{\mathrm{loc}}(a) \end{cases}$

  Furthermore, for each $(p, X) \in Q$ and $a \in \Sigma$, it holds that $t((p, X), a) = (p', X)$ for some $p' \in P$.
- *(Vertical Simplicity)* For each $X \in \Delta$ and $a \in \Sigma$, there is a $q \in Q$ such that for all $p \in P$ it holds that $\delta((p, X), \langle a \rangle) \in \mathcal{B}^+(\{q\} \times ((P \times \{X\}) \cup \{q_?\}))$.

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 also guarantee that SNWAs may also be interpreted as SANWAs.

▶ **Proposition 17.**

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

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

Proof of Theorem 12. The generic algorithm from the previous section can be adapted for simple cfGs, but with better complexity thanks to Proposition 17, to yield the upper bounds stated in Theorem 12.

More precisely, Proposition 17 (b) and Lemma 5 yield a polynomial time bound for replay-free games. Proposition 17 (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).[12]

[12] We actually use a slightly stronger result than Proposition 17 (a): deciding whether, for an NWA A_1 and a SANWA A_2, it holds $L(A_1) \cap L(A_2) \neq \emptyset$, is complete for PSPACE.

The upper bounds for games with unrestricted replay follows immediately and the upper bound for bounded replay can be shown similarly as in Proposition 11 (b).

The lower bounds are given by the following proposition. They mostly follow from careful adaptation of lower bound proofs of [11] for games on flat strings. ◀

▶ **Proposition 18.** *For the class of games with target languages specified by DTDs,* JWin *is*
(a) EXPTIME*-hard with unrestricted replay,*
(b) PSPACE*-hard with bounded replay, and*
(c) PTIME*-hard (under logspace-reductions) without replay*

For finite (and explicitly given) replacement languages we get feasibility even for bounded replay, but no improvement for unbounded replay.

▶ **Proposition 19.** *For the class of games with target languages specified by XML Schemas and explicitly enumerated finite replacement languages,* JWin *is*
(a) EXPTIME*-complete with unrestricted replay, and*
(b) PTIME*-complete (under logspace-reductions) with bounded replay or without replay.*
The same results hold for DTDs in place of XML Schemas.

Once again, as our algorithm generally computes a SANWA deciding JWin(G), the data complexity for JWin is in PTIME for all cases considered here, due to Proposition 17.

5 Validation of parameters and Insertion

In this section, we focus on two features that have not been addressed in the previous two sections: validation of the parameters of a function call with respect to a given schema, and a semantics which allows that returned trees do not *replace* their call nodes but are inserted next to them.

5.1 Validation of parameters

As pointed out in [9], in Active XML, parameters of function calls should be valid with respect to some schema. Transferred to the setting of cfGs this means that Juliet should only be able to play 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 $\langle /a\rangle$. Our definition of cfGs and the previous ones studied in the literature mostly ignore this aspect.[13] 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. 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 different validation DTDs is bounded by some constant.[14] It becomes intractable if the number of validation schemas can be unbounded and (already) with target and validation languages specified by XML Schemas, even with only one validation schema.

More precisely, we prove the following results.

[13] Actually, [9] takes validation into account but the precise way in which parameters are specified and tested is not explained in full detail.
[14] Note that this implies a polynomial-time data complexity for arbitrary replay-free games with DTD target, replacement and validation languages.

▶ **Theorem 20.** *For the class of games with validation with a bounded number of validation DTDs and target languages specified by DTDs,* JWin *is in* PTIME *without replay.*

The algorithm uses a bottom-up approach. The basic idea is that, starting from the leaves, at each level of the tree (that is for some node v and its leaf children) all relevant information about the game in the subtree t_v is computed with the help of flat replay-free games and aggregated in v. Then the children of v are discarded and the algorithm continues until only the root remains.

The following result shows that for slightly stronger games, parameter validation worsens the complexity.[15]

▶ **Theorem 21.** *For the class of games with validation,* JWin *(without replay) is*

(a) EXPTIME*-hard, if target and validation languages are specified by DNWAs (even with only one function symbol);*

(b) PSPACE*-hard, for games with only one function symbol, if the validation language is given by an XML schema, the target language by a DTD and a finite replacement language; and*

(c) PSPACE*-hard, for games with an unbounded number of validation DTDs and replacement and target languages specified by DTDs.*

Part (a) is proven by reduction from the intersection emptiness problem for DNWAs, while parts (b) and (c) use similar reductions from the problem of determining whether a quantified Boolean formula in disjunctive normal form is true.

Due to time constraints and as we are mainly interested in finding tractable cases, we have not looked for matching upper bounds.

5.2 Insertion rules

In our definition of Call moves, we define the successor configuration of a configuration $(\mathrm{R}, u\langle a\rangle v, \langle/a\rangle w)$ to be $(\mathrm{J}, u, v'w)$, that is, $\langle a\rangle v\langle/a\rangle$ is *replaced* by a string $v' \in R_a$. However, Active XML also offers an “append” option, where results of function calls are inserted as siblings after the calling function node (cf. [1]). 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 consider (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 (3) corresponds to the replay-free setting with replacement rules. Setting (1), however, gives Juliet a lot of power and makes $\mathrm{JWin}(\mathcal{G})$ undecidable.

▶ **Theorem 22.** *For the class of games with insertion semantics, target DNWAs and replacement NWAs,* JWin *is*

(a) *undecidable in general;*

(b) 2-EXPTIME*-complete for games with weak replay; and*

(c) *PSPACE-complete for games without replay.*

The proof idea for Theorem 22 is to simulate insertion-based games by replacement-based games and vice versa; part (a) additionally uses the undecidability of JWin for arbitrary (i.e. not necessarily left-to-right) strategies on games with flat strings, which was proven to be undecidable in [11].

[15] This is, of course not surprising. If any, the surprising result is Theorem 20.

6 Conclusion

The complexity of context-free games on nested words differs considerably from that on flat words (2-EXPTIME vs. EXPTIME), but there are still interesting tractable cases. One of the main insights of this paper is that the main tractable cases remain tractable if one allows XML Schema instead of DTDs for the specification of schemas.

Another result is that adding validation of input parameters can worsen the complexity, but tractability can be maintained by a careful choice of the setting. However, here the step from DTDs to XML Schema may considerably worsen the complexity.

Insertion semantics with unlimited replay yields undecidability.

We leave open some corresponding upper bounds in the setting with validation of input parameters. In future work, we plan to study the impact of parameters of function calls more thoroughly.

References

1 Serge Abiteboul, Omar Benjelloun, and Tova Milo. The Active XML project: an overview. *VLDB J.*, 17(5):1019–1040, 2008.

2 Serge Abiteboul, Tova Milo, and Omar Benjelloun. Regular rewriting of active XML and unambiguity. In *PODS*, pages 295–303, 2005.

3 Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

4 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*, pages 105–116, 2013.

5 Laura Bozzelli. Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. In *CONCUR- Concurrency Theory, 18th International Conference*, pages 476–491, 2007.

6 E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games. A Guide to Current Research.* Springer, 2002.

7 Lukasz Kaiser. Synthesis for structure rewriting systems. In Rastislav Královic and Damian Niwinski, editors, *MFCS*, volume 5734 of *Lecture Notes in Computer Science*, pages 415–426. Springer, 2009.

8 Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of XML schema. *ACM Trans. Database Syst.*, 31(3):770–813, 2006.

9 Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Frederic Dang Ngoc. Exchanging intensional XML data. *ACM Trans. Database Syst.*, 30(1):1–40, 2005.

10 Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

11 Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Active context-free games. *Theory Comput. Syst.*, 39(1):237–276, 2006.

12 Martin Schuster and Thomas Schwentick. Games for Active XML revisited. *CoRR*, abs/1412.5910, 2014. Available online at `http://arxiv.org/abs/1412.5910`.

13 Johannes Waldmann. Rewrite games. In Sophie Tison, editor, *RTA*, volume 2378 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 2002.