École Normale Supérieure de Cachan
Laboratoire Spécification et Vérification

PhD Thesis

Reasoning on Words and Trees with Data

On decidable automata on data words and data trees
in relation to satisfiability of LTL and XPath.

Diego Figueira

Advisors: Luc Segoufin, Stéphane Demri

December 2010, Cachan, France

Dedicado a mi hermano Santi.

CONTENTS

REASONING ON WORDS AND TREES WITH DATA

Summary A data word (resp. a data tree) is a finite word (resp. tree) whose every position carries a letter from a finite alphabet and a *datum* form an infinite domain. In this thesis we investigate automata and logics for data words and data trees with decidable reasoning problems: we focus on the emptiness problem in the case of automata, and the satisfiability problem in the case of logics.

On data words, we present a decidable extension of the model of alternating register automata studied by Demri and Lazić. Further, we show the decidability of the satisfiability problem for the linear-time temporal logic on data words $\mathsf{LTL}^{\downarrow}(\mathsf{X}, \mathsf{F}, \mathsf{U})$ (studied by Demri and Lazić) extended with quantification over data values. We also prove that the lower bounds of non-primitive recursiveness shown by Demri and Lazić for $\mathsf{LTL}^{\downarrow}(\mathsf{X}, \mathsf{F})$ carry over to $\mathsf{LTL}^{\downarrow}(\mathsf{F})$.

On data trees, we consider three decidable automata models with different characteristics. We first introduce the Downward Data automaton (DD automata). Its execution consists in a transduction of the finite labeling of the tree, and a verification of data properties for every subtree of the transduced tree. This model is closed under boolean operations, but the tests it can make on the order of the siblings is very limited. Its emptiness problem is in 2ExpTime. On the contrary, the other two automata models we introduce have an emptiness problem with a non-primitive recursive complexity, and are closed under intersection and union, but not complementation. They are both alternating automata with one register to store and compare data values. The automata class $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ extends the top-down automata ATRA of Jurdziński and Lazić. We exhibit similar decidable extensions as the one showed in the case of data words. This class can test for any regular tree language—in contrast to DD automata. Finally, we consider a bottom-up alternating tree automaton with one register (called BUDA). Although BUDA are one-way, they can test data properties by navigating the tree in both directions: upward and downward. In opposition to $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$, this automaton cannot test for properties on the sequence of siblings (like, for example, the order in which labels appear).

All these three models have connections with the logic XPath—a logic conceived for XML documents, which can be seen as data trees. Through the aforementioned automata we show that the satisfiability of three natural fragments of XPath are decidable. These fragments are: *downward* XPath, where navigation can only be done by child and descendant axes; *forward* XPath, where navigation also contains the next sibling axis and its transitive closure; and *vertical* XPath, whose navigation consists in the child, descendant, parent and ancestor axes. Whereas downward

XPath is ExpTime-complete, forward and vertical XPath have non-primitive recursive lower bounds.

Key words data word, data tree, data value, infinite alphabet, XML, LTL, XPath (downward, forward, vertical), alternating register automata, satisfiability, emptiness, decidable, non-primitive recursive, WSTS

RAISONNEMENT SUR MOTS ET ARBRES AVEC DONNÉES

Résumé Un mot de données (resp. un arbre de données) est un mot (resp. arbre) fini, dont chaque position est étiquetée avec une lettre d'un alphabet fini et une *donnée* d'un domaine infini. Dans cette thèse, nous étudions des automates et des logiques sur des mots et des arbres de données ayant des propriétés décidables: nous nous concentrons sur le problème du test du vide dans le cas des automates, et sur le problème de la satisfaisabilité dans le cas des logiques.

Sur les mots de données, nous présentons une extension décidable du modèle d'automate alternant avec registre étudié par Demri and Lazić. En outre, nous montrons la décidabilité du problème de satisfaisabilité pour la logique du temps linéaire sur les mots de données $\mathsf{LTL}^{\downarrow}(\mathsf{X},\mathsf{F},\mathsf{U})$ (étudié par Demri and Lazić) étendue avec une quantification sur des données. Nous montrons aussi que la borne inférieure de non-récursivité primitive montré par Demri and Lazić pour $\mathsf{LTL}^{\downarrow}(\mathsf{X},\mathsf{F})$ est déjà valable pour $\mathsf{LTL}^{\downarrow}(\mathsf{F})$.

Sur les arbres de données, nous considérons trois modèles décidables d'automates avec des caractéristiques différentes. Nous commençons par introduire l'automate avec donnée "downward" (automates DD). Son exécution consiste en une transduction ré-étiquetant la partie finie de l'étiquetage de l'arbre, et une vérification des propriétés des données de chaque sous-arbre de l'arbre résultant de la transduction. Ce modèle est clos par les opérations booléennes, mais les tests autorisés sur l'ordre des noeuds ayant le même père sont très limités. Son problème du vide est dans 2ExpTime. Au contraire, les deux autres modèles d'automates que nous introduisons ont un problème du vide avec une complexité non récursive primitive, et sont clos par intersection et union, mais par par complémentation. Ils ont tous les deux un contrôle alternant ainsi qu'un registre pour stocker et comparer les données. La classe des automates $\mathsf{ATRA}(\mathsf{guess},\mathsf{spread})$ généralise le modèle d'automate top-down ATRA de Jurdziński and Lazić. Nous introduisons des extensions décidables similaires à celles que nous avons étudiées dans le cas de mots de données. Cette classe d'automates généralise la notion de langage rationnel d'arbre, —contrairement aux automates DD. Enfin, nous considérons un modèle d'automate bottom-up avec un contrôle alternant et un registre (appelé BUDA). Bien que les BUDA soient bottom-up, ils peuvent tester des propriétés sur les données en navigant dans l'arbre dans les deux directions: vers le haut et vers le bas. Au contraire de $\mathsf{ATRA}(\mathsf{guess},\mathsf{spread})$, ce modèle d'automate ne peut pas tester de propriétés sur la séquence des noeuds ayant le même père (comme, par exemple, l'ordre dans lequel apparaissent leurs étiquettes).

Ces trois modèles d'automates ont des liens avec la logique XPath—une logique

conçue pour les documents XML, qui peuvent être vus comme des arbres de données. En utilisant les automates que nous avons mentionnés ci-dessus, nous montrons que la satisfaisabilité de trois fragments naturels de XPath sont décidables. Ces fragments sont: *downward* XPath, où la navigation ne peut se faire que via les axes child et descendant; *forward* XPath, où la navigation permet également les axes next sibling ainsi que sa clôture transitive, et *vertical* XPath, dont la navigation est limitée aux axes child, descendant, parent et ancestor. Alors que downward XPath est ExpTime-complet, les fragments forward et vertical de XPath ont une borne inférieure de non-récursivité primitive.

Mots clés mot de données, arbre de données, valeur des données, alphabet infini, XML, LTL, XPath (downward, forward, vertical), automate alternant avec registre, satisfaisabilité, test du vide, décidabilité, non-récursivité primitive, WSTS

MERCI !

I want to express my profound and sincere gratitude to my supervisor Luc Segoufin. For the trust he placed on me, for teaching me so many things and always giving me valuable advises and honest critics. I also thank wholeheartedly my co-supervisor Stéphane Demri. He always showed me the bright side of things, and was the first to encourage me to continue working with my ideas. I feel very lucky to have had two such superb supervisors!

I thank Georg Gottlob and Thomas Schwentick for giving me the honor of reviewing this dissertation. I also thank Ranko Lazić, Leonid Libkin and Carsten Lutz for accepting to be part of the jury of my defense.

Thanks to my family for always—even from 10,000 km away—being there for me. And to my friends from Argentina, France, Poland, and from all over the world.

Lastly and most importantly, thanks to Nadia, without whose stubborn love and patience nothing of this would have been possible.

1. INTRODUCTION

1.1 Motivation

Words and trees are amongst the most studied structures in computer science. In this thesis, we focus on words and trees that can contain elements from some infinite alphabet, like for example the set of integers, or the set of words over the alphabet $\{a, b\}$. These kind of structures are relevant to many areas.

In software verification, one may need to decide statically whether a program satisfies some given specification; and the necessity of dealing with infinite alphabets can arise from different angles. For example, in the presence of concurrency, we have an unbounded number of processes running, each one with its process identification, and we must verify properties specifying the interplay between these processes. Further, procedures may take parameters as input, and they can hence exchange data from some unbounded domain. Infinite alphabets can also emerge as a consequence of the use of recursive procedure calls, communication through FIFO channels, etc.

On the other hand, in a database context, infinite alphabets are a common occurrence. Let us dwell on static analysis tasks on XML documents and its query languages. In this context there are several pertinent problems serving static analysis. For example, there is the problem of *coherence*: is there a document in which a given query returns a non-empty result? The problem of *inclusion*: is it true that for any document, the result answered by one given query is contained in the result of another? And the problem of *equivalence*: do two given queries always return the same answers? In the database context, these questions are at the core of many static analysis tasks. For example, by answering the coherence problem one can decide whether the computation of a query on a database can be avoided because the query contains a contradiction; and by the equivalence problem if one query can be safely replaced by a simpler one. All these queries recurrently need to specify properties concerning not only the labels of the nodes, but also the actual data contained in the attributes.

Still in the context of databases, we can also regard logics that express data properties as specification languages. In verification of database-driven systems, we are provided with the specification of a system that interacts with a database, and we need to check whether it is possible to reach a state in which the database has some undesired property. In order to model the specification of the system as well as the property of the database—for example through an automaton or a logical formula—we typically need to take into account values from infinite domains.

Therefore, the study of formalisms to reason with words and trees that can carry elements from some infinite domain is relevant to all the aforementioned areas, and possibly more. To begin our study, first we need to fix once and for all the structure over which we intend to reason.

Data words and *data trees* are simple models that extend words and trees over finite alphabets. A data word is a word (*i.e.*, a finite sequence) where every position has a symbol from a finite alphabet (a *label*), and an element from some infinite domain (a *data value*). Similarly, a data tree is an unranked ordered finite tree, whose every node carries a label and a data value. By *unranked* we mean that every node has unboundedly many children, and by *ordered* that the children of a node are seen as an ordered list of subtrees, instead of a set or multiset. This model is in close relation to an XML document. Indeed, we will see that all the results we obtain on data trees can be translated into the XML setting.

To get familiar with these models, let us give some examples with possible *data properties* (*i.e.*, properties whose satisfaction rely on the model's data values) that one may be interested in verifying in a data word or a data tree.

Example 1.1. We have several processes (or execution threads of a process) running concurrently.[1] Each one of these has a process identifier (a number), and we model the history of execution of these processes. In the history, we record when the process i begins a task with an element (b, i), when it ended a task with (e, i), and when it reads (r, i) and writes (w, i) to the hard disk, during the task. To make things interesting, suppose that once in a while there is a maintenance shutdown of the disk (performed by a process with letter s). A possible history may be one like this one.

$$(\mathsf{b},1)(\mathsf{b},2)(\mathsf{r},1)(\mathsf{r},2)(\mathsf{w},1)(\mathsf{e},1)(\mathsf{b},1)(\mathsf{e},2)(\mathsf{e},1)(\mathsf{s},3)(\mathsf{b},4)(\mathsf{w},4)(\mathsf{e},4)(\mathsf{s},3)$$

In this example we see that process 1 performs two tasks, in the first one it reads and then writes, and in the second one it does not read nor write. Interleaved with these tasks process 2 also performs a reading task. Then the maintenance process 3 shutdowns the disk, etc.

Let us describe some possible properties one could want to verify over a history like this.

(W1) For every b there is a future e with the same data value.

(W2) Further, we can ask that for every data value, the data word restricted to that data value belongs to $\{\mathsf{s}, (\mathsf{b} \cdot \{\mathsf{r}, \mathsf{w}\}^* \cdot \mathsf{e})\}^*$.

(W3) Every time a task starts, it will end before the shutdown. That is, for every b there exists an e with the same data value, that occurs before the next s.

(W4) There exists a process whose first operation is a write.

[1] This example is inspired on an example recurrently used by Thomas Schwentick.

$$
\begin{aligned}
\textit{root} &\to \mathsf{category}\ \mathsf{stock} \\
\mathsf{stock} &\to (\mathsf{id})^* \\
\mathsf{category} &\to (\mathsf{category})^*\ (\mathsf{product})^*\ \mathsf{featured}? \\
\mathsf{product} &\to \mathsf{name}\ \mathsf{id}
\end{aligned}
$$

Fig. 1.1: A simple specification modeling the data trees of interest for our problem.

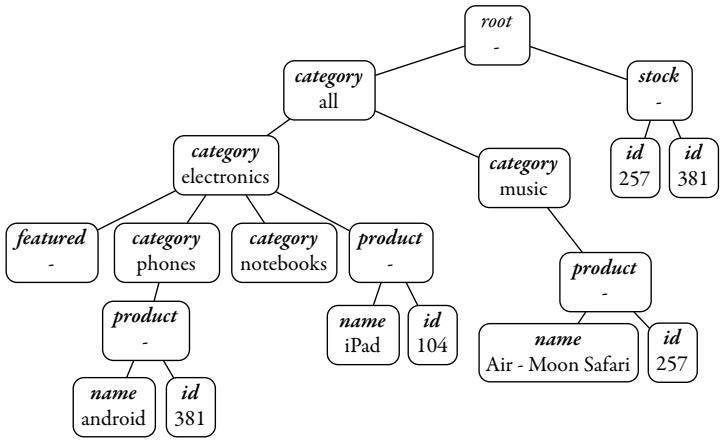

Fig. 1.2: An example of a data tree with information about the products sold by a web site.

(W5) Each time a s appears, all the processes occurred so far do not reappear in the future.

□

Example 1.2. Suppose we have a data tree containing all the products that are sold by a web site. Suppose that the tree is organized following the rules of the kind "if a node has label x, then the children have certain label" of Figure 1.1. These set of rules are read in the expected manner. For example, the root has two nodes, one with label category and one labeled stock. Each node labeled category has a child node labeled name, then a sequence of nodes labeled category, a sequence of nodes labeled product, and finally perhaps a node labeled featured. The idea is that the products are organized in a hierarchy of categories (described by the category child of the root), and also we have under stock a set of product identifiers that are on stock. Further, some categories may be "featured categories" of products, obtaining some special visibility in the site for all the products in the category and recursively in all subcategories. An example of a possible tree is shown in Figure 1.2.

Given a data tree that models such scenario, there might be a number of properties that we want to verify. In the spirit of depicting different sorts of

properties, consider the following.

(T1) It cannot happen that a category has the same name as a product.

(T2) All the product id under the category subtree are different. Also, all the product id under the stock node are different.

(T3) The name of a product allows to identify a product inside a category. In other words, all the product name in a category are different.

(T4) Every product in stock is categorized under some category.

(T5) All the categories have different names.

(T6) There is at least one product in stock that is in some descendant category of a featured category.

(T7) The same product cannot be in two categories in the ancestor-descendant relation.

□

These examples intend to show the variety of properties we may need to verify in an environment with data. They will become useful hereafter when comparing the expressiveness of formalisms. We consider two sorts of formalisms for specifying these properties: either by means of *logics*, or by means of accepting runs of *automata*. However, there is no decidable logic or class of automata that is able to express all these desirable properties and is closed under conjunction or intersection, neither in the case of trees nor in the case of words. At the same time, most of these independent properties can be expressed by some decidable formalism in the literature closed under intersection. The general outlook is that there are not many expressive formalisms that are decidable, and there is also a need for more general techniques to allow to treat data values to prove decidability results.

This thesis explores expressive logics and automata with decidable reasoning tasks. We give several new results on decidable logics on these data models, and we introduce new decidable automata models. The intention of this work is to devise new approaches and techniques to work with data values, and to give new insights on the limits of what is decidable on these models.

1.2 Contributions

The contribution of this thesis can be divided into two parts: data words and data trees.

Data words On data words, we focus on automata with one register and alternating control called ARA. This automata class is known to have a decidable emptiness problem (Demri and Lazić, 2009). We extend these automata with operators that add expressive power, while preserving decidability. These operators will be useful later to prove the decidability of the satisfiability problem for a logic. This extended class of automata is called ARA(guess, spread).

On the logical side, we center our attention on an extension of the linear-time temporal logic LTL(X, F, U) (with the next X, future F and until U modalities) on data words. This extension consists in having one *register* that allows to express data properties, and it is denoted by $\mathsf{LTL}^{\downarrow}(\mathsf{U},\mathsf{F},\mathsf{X})$. It contains a 'freeze' operator ($\downarrow$) to store the current datum in the register and a 'test' operator ($\uparrow$) to test—at any later position—that the current datum is equal to the stored one.

Demri and Lazić (2009) showed that $\mathsf{LTL}^{\downarrow}(\mathsf{U},\mathsf{F},\mathsf{X})$ is decidable, that $\mathsf{LTL}^{\downarrow}(\mathsf{F},\mathsf{X})$ has non-primitive recursive complexity, and that $\mathsf{LTL}^{\downarrow}(\mathsf{F},\mathsf{F}^{-1},\mathsf{X})$ is undecidable (where F^{-1} is the past modality). This thesis shows that the lower bounds are preserved for weak fragments, and that the decidability is also preserved for more expressive extensions. More precisely, we show that the fragment $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ is non-primitive recursive and that $\mathsf{LTL}^{\downarrow}(\mathsf{F},\mathsf{F}^{-1})$ is undecidable. On the other hand, we extend the decidability results by proving that $\mathsf{LTL}^{\downarrow}(\mathsf{U},\mathsf{F},\mathsf{X})$ can be equipped with restricted universal and existential quantification over data values, by a reduction to the ARA(guess, spread) emptiness problem.

Data trees We consider three decidable automata models with different characteristics that allow to express different sets of data properties.

We introduce an automata model that can make rich tests between distant positions of the subtree, but can only perform some limited tests over the order of the siblings. This model has a 2ExpTime emptiness problem, or a "modest" ExpTime complexity for some restricted subclass. We call this model the Downward Data automata (or DD automata for short).

As a second contribution, we extend the results of Jurdziński and Lazić (2008) on alternating top-down tree one register automata (ATRA), showing that similar extensions as the one showed in the case of data words can be decided. We call this class of automata ATRA(guess, spread).

Finally, we consider a *bottom-up* alternating tree automata with one register. Although the automaton is one way, it has features that allow to test data properties that can navigate the tree in both directions: upward and downward. We call this automaton model BUDA.

All these three models of automata have a decidable emptiness problem, and they all have connections with the logic XPath.

XPath is a logic for XML documents (which are essentially data trees). Expressions of this logic can navigate the tree by composing binary relations from a set of basic relations, that can contain the parent relation (noted $\uparrow$), child ($\downarrow$), ancestor ($\uparrow^*$), descendant ($\downarrow^*$), next sibling to the right ($\rightarrow$) or to the left ($\leftarrow$), and their

transitive closures ($\rightarrow^*$, $^*\leftarrow$). For example "$\uparrow[a]\uparrow\downarrow[b]$" defines the relation between two nodes x, y such that y is an uncle of x labeled b and x has a parent labeled a. Boolean tests are build by using these compound relations. An expression like $\langle\alpha\rangle$ (for α a relation) tests that there exists a node accessible with the relation α from the current node. Most importantly, a data test like $\langle\alpha = \beta\rangle$ (resp. $\langle\alpha \neq \beta\rangle$) tests that there are two nodes reachable from the current node with the relations α and β that have the same (resp. different) data value. We consider three natural fragments of XPath, according to which set of basic relations (usually called axes) we use: downward $\downarrow, \downarrow_*$; forward $\downarrow, \downarrow_*, \rightarrow, \rightarrow^*$; or vertical $\downarrow, \downarrow_*, \uparrow, \uparrow^*$. We call these fragments respectively the downward, forward and vertical fragments.

We prove that all three fragments—downward, forward and vertical XPath—have a decidable satisfiability problem. The first one is in ExpTime and the others have non-primitive recursive complexity. These bounds are optimal. The decidability proofs are in each case based in a reduction to the emptiness problem for one of the automata models we introduce here.

Finally, we mention that this work gives alternative, arguably simpler, decidability proofs for the known results from (Demri and Lazić, 2009; Jurdziński and Lazić, 2008). This simplification allows us to extend the decidability results with operators that add expressive power.

1.3 Related work

There are many works related with the investigation of this thesis. In this section we only mention some works that are very generally related to the general techniques and problems that we treat. We leave more precise comparisons with other works for future chapters, once we have better explained each of our formalism.

1.3.1 Data logics

There are many logics that can 'freeze' an element from some infinite domain to be able to compare it with other positions. These logics provide an operator that binds a variable with a part or feature of the model. In this sense, the freeze-logic most relevant to this thesis is the already mentioned $\mathsf{LTL}^{\downarrow}$ (Demri and Lazić, 2009) in which we can explicitly bind the current data value to a variable for later comparison.

This variable-binding mechanism is ubiquitous in many logical languages. For example, real-time logics study properties that are related to time. An instant of time t in this approach is a real number that can be bound to a variable x either implicitly (see *e.g.* Alur and Henzinger, 1994) or explicitly through clock variables (see *e.g.* Harel et al., 1990).

Also, other logics like Hybrid Logics (Goranko, 1996; Areces et al., 1999; Franceschet et al., 2003) contain a variable-binding mechanism similar to the freeze quantifier. The binding modality in these logics records the value of the current state.

In the area of artificial intelligence, there have also been works on logics over infinite alphabets, known as *Description Logics with concrete domains*. Paraphrasing Lutz (2003), Description Logics (DLs) are a family of logics designed for the representation of conceptual knowledge in artificial intelligence, and are closely related to Modal Logics (Blackburn et al., 2001). In some applications, it is important to equip DLs with expressive means that allow to describe "concrete qualities" of real-world objects such as their weight, temperature, and spatial extension. The standard approach is to augment DLs with so-called *concrete domains*, which consist of an infinite domain of data values, and a set of predicates over the domain.

1.3.2 Well-structured transition systems

In several chapters we make use of well-structured transition systems, that we tailor to our problems to obtain effective decision procedures.

The theory of well-structured transition systems (WSTS) is a development born in the field of verification of infinite state systems. Quoting Finkel and Schnoebelen (2001):

> "These are transition systems where the existence of a well-quasi-ordering over the infinite set of states ensures the termination of several algorithmic methods. WSTS's are an abstract generalization of several specific structures and they allow general decidability results [for many models.]"

These general techniques as we will use them in our work are the result of works by Finkel (1987, 1990); Abdulla, Čerāns, Jonsson, and Tsay (1996, 2000); Kouchnarenko and Schnoebelen (1997) and Finkel and Schnoebelen (2001).

WSTS serve to obtain decidability results in many fields (*cf.* Finkel and Schnoebelen, 2001, Part II) as in Petri nets, lossy and gainy systems—a prominent example will be shown in Section 2.4.5—, timed automata, string rewriting, process algebras, and possibly many more.

In this thesis we make use of some results on WSTS that will be explained in detail in Section 2.4. The notations and results that we use are based on (Finkel and Schnoebelen, 2001, § 5). We remark that these techniques have already been applied—in an indirect way—in the context of logics and automata on data words and data trees, in (Demri and Lazić, 2009) and (Jurdziński and Lazić, 2008). These works base their decidability results on a reduction to a restricted class of counter automata (*cf.* § 2.4.5), whose emptiness is shown to be decidable by WSTS arguments. In this thesis we use the same kind of arguments, the main difference is that in our strategy, we define an ad-hoc transition system that corresponds precisely to the run of our automaton, and we show that the transition system is well-structured. In short, we avoid making reductions to an intermediate formalism. This approach allows us to easily see what can be added to the automata without loosing the well-structuredness, and therefore the decidability. Also, WSTS have been used in a similar way to ours to show decidability of timed automata

(introduced by Alur and Dill, 1994) in (Lasota and Walukiewicz, 2008; Ouaknine and Worrell, 2004).

The chapters in which we make use of WSTS are: Chapters 3, 6, and 7.

1.4 Organization

This thesis has two parts. The first one dealing with *data words*, and the second one with *data trees*. In the opening Chapter 2 we introduce some necessary definitions and we explicit the results and techniques we will use from WSTS.

Part I: data words Chapter 3 presents all the results on data words. Section 3.3 contains the decidability of the $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ register automaton. Section 3.4 defines the logic $\mathsf{LTL}^{\downarrow}(\mathsf{U}, \mathsf{F}, \mathsf{X})$ on data words. Section 3.5 contains the decidability proof for a more expressive extension of $\mathsf{LTL}^{\downarrow}(\mathsf{U}, \mathsf{F}, \mathsf{X})$, and Section 3.6 shows the non-primitive recursiveness of $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ and the undecidability of $\mathsf{LTL}^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$.

Part II: data trees Chapter 4 is introductory to our results on data trees. It contains the main definitions of *data tree*, XML and XPath, and already contains some results that can be transferred from Part I.

The remaining chapters are identical in their organization. Each of them defines a novel model of automata, shows its decidability, and relates the automata with a fragment of XPath.

The DD automata are introduced in Chapter 5. This class of automata is shown to be decidable in Section 5.3. The decidability of the satisfiability problem for downward XPath is shown through a translation into this automata class in Section 5.4.

Chapter 6 introduces the automata class $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$. Decidability will follow from similar results from Chapter 3 in Section 6.3. Finally, we show that satisfiability of forward XPath is decidable by a reduction to the emptiness problem for these automata (Section 6.4.2).

Finally, in Chapter 7 we introduce the automata class BUDA. We show decidability through the theory of WSTS in Section 7.3. This automaton model is used to show decidability of vertical XPath in Section 7.4.

Chapter 8 concludes this thesis with some final comments and open questions.

2. PRELIMINARIES

In this chapter we introduce some basic notation and notions of languages, and in Section 2.4 we introduce some elements of the field of Well-structured transition systems that we will use throughout this thesis.

2.1 Notation

We first fix some basic notation. Let $\wp(C)$ denote the set of subsets of C, and $\wp_{<\infty}(C)$ be the set of *finite* subsets of C. Let $\mathbb{N} = \{0, 1, 2, \dots\}$, $\mathbb{N}_+ = \{1, 2, 3, \dots\}$, and let $[n] := \{1, \dots, n\}$ for any $n \in \mathbb{N}_+$. We fix once and for all $\mathbb{D}$ to be any infinite domain of data values; for simplicity in our examples we will consider $\mathbb{D} = \mathbb{N}$. In general we use letters $\mathbb{A}$, $\mathbb{B}$ for finite alphabets, the letter $\mathbb{D}$ for an infinite alphabet and the letters $\mathbb{E}$ and $\mathbb{F}$ for any kind of alphabet. By $\mathbb{E}^*$ we denote the set of finite sequences over $\mathbb{E}$, by $\mathbb{E}^+$ the set of finite sequences with at least one element over $\mathbb{E}$, and by $\mathbb{E}^\omega$ the set of infinite sequences over $\mathbb{E}$. We use '$\cdot$' as the concatenation operator between sequences. We write $|S|$ to denote the length of S (if S is a sequence), or the quantity of elements of S (if S is a set). We write $f : A \rightharpoonup B$ to denote a *partial* function from A to B. Given a function $f : A \to B$, and $a \in A$, $b \in B$, we define the function $f[a \mapsto b] : A \to B$ as $f[a \mapsto b](a') = f(a')$ for every $a' \in A$, $a' \neq a$ and $f[a \mapsto b](a) = b$. We use the symbol $\circ$ for the composition of functions, $(f \circ g)(x) = g(f(x))$.

2.2 Languages

We use the standard definition of **language**. Given an automaton $\mathscr{M}$ over data words (resp. over data trees), let $\mathcal{L}(\mathscr{M})$ denote the set of data words (resp. data trees) that have an accepting run of $\mathscr{M}$. We say that $\mathcal{L}(\mathscr{M})$ is the language of words (resp. trees) recognized by $\mathscr{M}$. We extend this definition to a class of automata $\mathscr{A}$: $\mathcal{L}(\mathscr{A}) = \{\mathcal{L}(\mathscr{M}) \mid \mathscr{M} \in \mathscr{A}\}$, obtaining a class of languages.

Equivalently, given a formula φ of a logic $\mathscr{L}$ over data words (resp. data trees) we denote by $\mathcal{L}(\varphi)$ the set of words (resp. trees) satisfying φ. This is also extended to a logic $\mathcal{L}(\mathscr{L}) = \{\mathcal{L}(\varphi) \mid \varphi \in \mathscr{L}\}$.

We say that a class of automata (resp. a logic) $\mathscr{A}$ is **at least as expressive as** another class (resp. logic) $\mathscr{B}$ iff $\mathcal{L}(\mathscr{B}) \subseteq \mathcal{L}(\mathscr{A})$. If additionally $\mathcal{L}(\mathscr{B}) \neq \mathcal{L}(\mathscr{A})$ we say that $\mathscr{A}$ is **more expressive** than $\mathscr{B}$.

We say that a class of automata $\mathscr{A}$ **captures** a logic $\mathscr{L}$ iff there exists a

translation $t : \mathscr{L} \to \mathscr{A}$ such that for every $\varphi \in \mathscr{L}$ and model (*i.e.*, a data tree or a data word) $\mathbf{m}$, we have that $\mathbf{m} \models \varphi$ if and only if $\mathbf{m} \in \mathcal{L}(t(\varphi))$.

2.3 Regular languages

In this thesis we make use of the several characterizations of regular languages over a finite alphabet $\mathbb{A}$. In particular, we use that a word language $\mathscr{L} \subseteq \mathbb{A}^*$ is **regular** iff it satisfies the following equivalent properties:

- there is a deterministic (or nondeterministic) finite automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathscr{L}$,
- it is described by a regular expression,
- there is a finite monoid $(M, \cdot)$ with a distinguished subset $S \subseteq M$, and a morphism $h : \mathbb{A}^* \to M$ such that $w \in \mathscr{L}$ iff $h(w) \in S$.

2.4 Well-structured transition systems

As already mentioned (§ 1.3.2), several decidability results included in this thesis, rely on techniques methods from the theory of well-quasi-orderings. The main results of Chapters 6 and 7 stem from interpreting the automaton's execution as an infinite state system with some good properties allowing to obtain an effective procedure for the emptiness problem. The decidability results rely on the existence of a well-quasi-ordering between states that is compatible with the transitions. These are examples of what is known in the literature as *well-structured transition systems*, or WSTS for short (see Finkel and Schnoebelen, 2001).

2.4.1 Basic definitions and results

We reproduce some standard definitions and known results of the theory of well-quasi-orderings that we make use of.

Definition 2.1. For a set S, we define $(S, \leq)$ to be a **well-quasi-order** (wqo) iff '$\leq$' $\subseteq S \times S$ is a relation that is reflexive, transitive and for every infinite sequence $w_1, w_2, \ldots \in S^\omega$ there are two indices $i < j$ such that $w_i \leq w_j$.

Dickson's Lemma. (Dickson, 1913) *Let $\leq_k \subseteq \mathbb{N}^k \times \mathbb{N}^k$ such that $(x_1, \ldots, x_k) \leq_k (y_1, \ldots, y_k)$ iff $x_i \leq y_i$ for all $i \in [k]$. For all $k \in \mathbb{N}$, $(\mathbb{N}^k, \leq_k)$ is a well-quasi-order.*

Definition 2.2. Given a quasi-order $(S, \leq)$ we define the **embedding order** as a relation $\sqsubseteq \subseteq S^* \times S^*$ such that $x_1 \cdots x_n \sqsubseteq y_1 \cdots y_m$ iff there exist $1 \leq i_1 < \cdots < i_n \leq m$ with $x_j \leq y_{i_j}$ for all $j \in [n]$.

Higman's Lemma. (Higman, 1952) *Let $(S, \leq)$ be a well-quasi-order. Let $\sqsubseteq \subseteq S^* \times S^*$ be the embedding order over $(S, \leq)$. Then, $(S^*, \sqsubseteq)$ is a well-quasi-order.*

Corollary 2.3. (of Higman's Lemma) *Let S be a* finite *alphabet. Let $\sqsubseteq$ be the subword relation over $S^* \times S^*$ (i.e., $x \sqsubseteq y$ if x is the result of removing some (possibly none) positions from y). Then, $(S^*, \sqsubseteq)$ is a well-quasi-order.*

Proof. It suffices to realize that $\sqsubseteq$ is indeed the embedding order over $(S, =)$, which is trivially a wqo since S is finite. □

Definition 2.4. Given a transition system $(S, \to)$, and $T \subseteq S$ we define $Succ(T) := \{a' \in S \mid \exists\, a \in T \text{ with } a \to a'\}$, and $Succ^*$ to its reflexive-transitive closure. Given a wqo $(S, \leq)$ and $T \subseteq S$, we define the *upward closure* of T as $\uparrow T := \{a \mid \exists\, a' \in T, a' \leq a\}$, and the *downward closure* as $\downarrow T := \{a \mid \exists\, a' \in T, a \leq a'\}$. We say that T is **downward-closed** (resp. **upward-closed**) iff $\downarrow T = T$ (resp. $\uparrow T = T$).

Definition 2.5. We say that a transition system $(S, \to)$ is **finitely branching** iff $Succ(\{a\})$ is finite for all $a \in S$. If $Succ(\{a\})$ is also effectively computable for all a, we say that $(S, \to)$ is **effective**.

2.4.2 Reflexive compatibility

Definition 2.6 (rdc)**.** A transition system $(S, \to)$ is **reflexive downward compatible** (or rdc for short) with respect to a wqo $(S, \leq)$ iff for every $a_1, a_2, a_1' \in S$ such that $a_1' \leq a_1$ and $a_1 \to a_2$, there exists $a_2' \in S$ such that $a_2' \leq a_2$ and either $a_1' \to a_2'$ or $a_1' = a_2'$.

Decidability will be shown as a consequence of the following propositions.

Proposition 2.7. (Finkel and Schnoebelen, 2001, Proposition 5.4) *If $(S, \leq)$ is a* wqo *and $(S, \to)$ a transition system such that (1) it is* rdc *, (2) it is effective, and (3) $\leq$ is decidable ; then for any finite $T \subseteq S$ it is possible to compute a finite set $U \subseteq S$ such that $\uparrow U = \uparrow Succ^*(T)$.*

Lemma 2.8. *Given $(S, \leq, \to)$ as in Proposition 2.7, a recursive downward-closed set $V \subseteq S$, and a finite set $T \subseteq S$. The problem of whether there exists $a \in T$ and $b \in V$ such that $a \to^* b$ is decidable.*

Proof. Applying Proposition 2.7, let $U \subseteq S$ finite, with $\uparrow U = \uparrow Succ^*(T)$. Since T is finite and V is recursive, we can test for every element of $b \in U$ if $b \in V$. On the one hand, if there is one such b, then by definition $b \in \uparrow Succ^*(T)$, or in other words $a \to^* a' \leq b$ for some $a \in T$, $a' \in Succ^*(a)$. But since V is downward-closed, $a' \in V$ and hence the property is true. On the other hand, if there is no such b in U, it means that there is no such b in $\uparrow U$ either, as V is downward-closed. This means that there is no such b in $Succ^*(T)$ and hence that the property is false. □

2.4.3 N-compatibility

The results and definitions just shown for reflexive compatibility (*i.e.*, compatibility in zero or one steps), can be easily extended to N-compatibility (*i.e.*, compatibility in at most N steps).

Given a binary relation $R \subseteq S \times S$ and $K \subseteq S$, let us write $R^{\leq n}(K)$ for $K \cup R(K) \cup R^2(K) \cup \cdots \cup R^n(K)$.

Definition 2.9. Given $N \in \mathbb{N}_+$, a transition system $(S, \rightarrow)$ is **N-downward compatible** with respect to a wqo $(S, \leq)$ iff for every $a_1, a_2, a_1' \in S$ such that $a_1' \leq a_1$ and $a_1 \rightarrow a_2$, there exists $a_2' \in S$ such that $a_2' \leq a_2$ and $a_1' \ (\rightarrow)^{\leq N} \ a_2'$.

Notice that rdc implies N-downwards-compatible for every N. We adapt the results contained in (Finkel and Schnoebelen, 2001, § 5) to extend the results of the previous section to the case of N-downwards-compatibility.

Suppose that $(S, \rightarrow)$ is a N-downwards-compatible with respect to a wqo $(S, \leq)$.

Lemma 2.10. *$\uparrow K \subseteq \uparrow K'$ implies $\uparrow Succ^{\leq 1}(K) \subseteq \uparrow Succ^{\leq N}(K')$.*

Proof. Assume $s \in \uparrow Succ^{\leq 1}(K)$. Then there exist $s_1 \in K$ and t_1 with $s_1 \ (\rightarrow)^{\leq 1} \ t_1 \leq s$. Because $\uparrow K \subseteq \uparrow K'$, there is a $s_2 \in K'$ with $s_2 \leq s_1$. Because $(S, \rightarrow, \leq)$ is N-downwards compatible, there exists a $s_2 \ (\rightarrow)^{\leq N} \ t_2$ with $t_2 \leq t_1$. Hence $t_2 \leq s$. Now $t_2 \in Succ^{\leq N}(K')$ entails $s \in \uparrow Succ^{\leq N}(K')$. □

Now assume $K_0 \subseteq S$ and define a sequence $K_0, K_1, \ldots$ of sets with $K_{n+1} := K_n \cup Succ(K_n)$. Let m be the first index such that $\uparrow K_m = \uparrow K_{m+N}$. Such an m must exist by (Finkel and Schnoebelen, 2001, Lemma 2.4).

Lemma 2.11. $\uparrow K_m = \uparrow \bigcup_{i \in \mathbb{N}} K_i = \uparrow Succ^*(K_0)$

Proof. The first equality is a consequence of Lemma 2.10. The $\subseteq$ is immediate and for the $\supseteq$, if there is an element in $(\uparrow \bigcup_{i \in \mathbb{N}} K_i) \setminus \uparrow K_m$, then there must be an element in $Succ(\uparrow K_m)$ but not in $\uparrow K_m$. Since $\uparrow\uparrow K_m \subseteq \uparrow K_m$, by Lemma 2.10, $\uparrow Succ^{\leq 1}(\uparrow K_m) \subseteq \uparrow Succ^{\leq N}(K_m) = \uparrow K_m$. Hence, there is no such element. The second equality follows from the definition of the K_i's. □

We then have the following proposition.

Proposition 2.12. *If $(S, \leq)$ is a* wqo *and $(S, \rightarrow)$ a transition system such that (1) it is N-*downwards compatible *for some fixed N, (2) it is effective, and (3) $\leq$ is decidable; then for any finite $T \subseteq S$ it is possible to compute a finite set $U \subseteq S$ such that $\uparrow U = \uparrow Succ^*(T)$.*

Proof. We take $K_0 = T$. The sequence $K_0, K_1, \ldots$ can be constructed effectively (each K_i is finite and $Succ$ is effective). The index m can be computed by computability of $\leq$. Finally, K_m is a computable finite basis of $\uparrow Succ^*(T)$. □

We then obtain the following Lemma, whose proof is immediate by following the same lines as the proof of Lemma 2.8.

Lemma 2.13. *Given $(S, \leq, \rightarrow)$ as in Proposition 2.12, a recursive downward-closed set $V \subseteq S$, and a finite set $T \subseteq S$. The problem of whether there exists $a \in T$ and $b \in V$ such that $a \rightarrow^* b$ is decidable.*

2.4.4 Powerset orderings

In Chapters 6 and 7 we will make use of orderings on *sets*. In this section we present some results that will become useful for showing that certain orders over sets are indeed wqo's.

Majoring ordering

Definition 2.14 ($\leq_\wp$)**.** Given an ordering $(S, \leq)$, we define the **majoring ordering** over $\leq$ as $(\wp_{<\infty}(S), \leq_\wp)$, where $\mathcal{S} \leq_\wp \mathcal{S}'$ iff for every $a \in \mathcal{S}$ there is $b \in \mathcal{S}'$ such that $a \leq b$.

Proposition 2.15. *If $(S, \leq)$ is a wqo, then the majoring order over $(S, \leq)$ is a wqo.*

Proof. The fact that this order is reflexive and transitive is immediate from the fact that $\leq$ is a wqo. The fact of being a *well*-quasi-order is a simple consequence of Higman's Lemma. Each finite set $\{a_1, \ldots, a_n\}$ can be seen as a sequence of elements $a_1, \ldots, a_n$, in any order. In this context, the embedding order is stricter than the majoring order. In other words, if $a_1, \ldots, a_n \sqsubseteq a'_1, \ldots, a'_m$, then $\{a_1, \ldots, a_n\} \leq_\wp \{a'_1, \ldots, a'_m\}$. By Higman's Lemma the embedding order over $(S, \leq)$ is a wqo, implying that the majoring order is as well. □

Proposition 2.16. *Let $\leq, \to_1 \subseteq S \times S$, $\leq_\wp, \to_2 \subseteq \wp_{<\infty}(S) \times \wp_{<\infty}(S)$ where $\leq_\wp$ is the* majoring order *over $(S, \leq)$ and $\to_2$ is such that if $\mathcal{S} \to_2 \mathcal{S}'$ then: $\mathcal{S} = \{a\} \cup \hat{\mathcal{S}}$, $\mathcal{S}' = \{b_1, \ldots, b_m\} \cup \hat{\mathcal{S}}$ with $a \to_1 b_i$ for every $i \in [m]$. Suppose that $(S, \leq)$ is a wqo which is rdc with respect to $\to_1$. Then, $(\wp_{<\infty}(S), \leq_\wp)$ is a wqo which is rdc with respect to $\to_2$.*

Proof. The fact that $(\wp_{<\infty}(S), \leq_\wp)$ is a wqo is given by Proposition 2.15. The rdc property with respect to $\to_2$ is straightforward. Suppose

$$\{a'\} \cup \mathcal{S}' \quad \leq_\wp \quad \{a\} \cup \mathcal{S} \quad \to_2 \quad \{b_1, \ldots, b_m\} \cup \mathcal{S}$$

with $\mathcal{S}' \leq_\wp \mathcal{S}$, $a' \leq a$, and $a \to_1 b_i$ for all $i \in [m]$. Since $a' \leq a$ and $\leq$ is rdc with $\to_1$, one possibility is that for each $a \to_1 b_i$ we can apply a $a' \to_1 b'_i$ with $b'_i \leq b_i$. In this case we obtain $\{a\} \cup \mathcal{S}' \to_2 \{b'_1, \ldots, b'_m\} \cup \mathcal{S}' \leq_\wp \{b_1, \ldots, b_m\} \cup \mathcal{S}$. The only case left to analyze is when, for some $a \to_1 b_i$ the compatibility is reflexive, that is, $a' \leq b_i$. But then we take a reflexive compatibility as well, since $\{a'\} \cup \mathcal{S}' \leq_\wp \{b_1, \ldots, b_m\} \cup \mathcal{S}$.

Finally, the case where a has no pre-image, $\mathcal{S}' \leq_\wp \{a\} \cup \mathcal{S}$ with $\mathcal{S}' \leq_\wp \mathcal{S}$, is reflexive compatible since in this case $\mathcal{S}' \leq_\wp \{b_1, \ldots, b_m\} \cup \mathcal{S}$. □

Minoring ordering

In Chapter 7 we use a finer grained notion of quasi-ordering, to be able to use the following result due to Jančar (1999).

Definition 2.17. The **Rado structure** is $(S_{\text{rado}}, \leq_{\text{rado}})$ with $S_{\text{rado}} = \{(i,j) \in \mathbb{N} \times \mathbb{N} \mid i < j\}$ and $(i_1, j_1) \leq_{\text{rado}} (i_2, j_2)$ iff

- $i_1 = i_2$ and $j_1 \leq j_2$, or
- $j_1 < i_2$.

Definition 2.18. A relation is a **ω^2-wqo** iff it is a wqo and does not contain an isomorphic copy of the Rado structure as an induced substurcture.

Definition 2.19 ($\leq_{min}$)**.** Given an ordering $(S, \leq)$, we define the **minoring ordering** over $\leq$ as $(\wp_{<\infty}(S), \leq_{min})$, where $\mathcal{S} \leq_{min} \mathcal{S}'$ iff for every $b \in \mathcal{S}'$ there is $a \in \mathcal{S}$ such that $a \leq b$.

Proposition 2.20. (Jančar, 1999, Theorem 1) *Given a quasi-order $(S, \leq)$, and $\leq_{min}$ the minoring ordering over $(S, \leq)$, then $(\wp_{<\infty}(S), \leq_{min})$ is a wqo iff $(S, \leq)$ is a ω^2-wqo.*

Proposition 2.21. (Marcone, 1994, Lemma 1.6) *If $(S, \leq_1)$ and $(S, \leq_2)$ are ω^2-wqo, then $(S, \leq_1 \cap \leq_2)$ is a ω^2-wqo.*

Proof. In fact Marcone proves a more general lemma that immediately implies this one. □

Proposition 2.22. *If $(S_1, \leq_1)$ and $(S_2, \leq_2)$ are ω^2-wqo, then $(S_1 \times S_2, \leq)$ is a ω^2-wqo, where $(a_1, a_2) \leq (b_1, b_2)$ iff $a_1 \leq_1 b_1$ and $a_2 \leq_2 b_2$.*

Proof. Let $(S_1 \times S_2, \leq_1^\dagger)$ be defined as $(a,b) \leq_1^\dagger (a',b')$ iff $a \leq_1 a'$. Similarly, let $(S_1 \times S_2, \leq_2^\dagger)$ be defined as $(a,b) \leq_2^\dagger (a',b')$ iff $b \leq_2 b'$. By means of contradiction, if $(S_1 \times S_2, \leq_1^\dagger)$ contains the Rado structure, then projecting on S_1 we obtain that $(S_1, \leq_1)$ contains the Rado structure, which cannot be since it is a ω^2-wqo. Thus, $(S_1 \times S_2, \leq_1^\dagger)$ and $(S_1 \times S_2, \leq_2^\dagger)$ are ω^2-wqo. In light of Proposition 2.21, $(S_1 \times S_2, \leq_1^\dagger \cap \leq_2^\dagger)$ is a ω^2-wqo. Since $\leq_1^\dagger \cap \leq_2^\dagger$ and $\leq$ are the same relation, we obtain that $(S_1 \times S_2, \leq)$ is a ω^2-wqo. □

Lemma 2.23. $(\mathbb{N}, \leq)$ *is a ω^2-wqo.*

Proof. $(\mathbb{N}, \leq)$ is a wqo, and since there are no incomparable elements it cannot contain the Rado structure. Indeed, every well-founded linear order is a ω^2-wqo. □

Corollary 2.24. *For all $k \geq 1$, $(\mathbb{N}^k, \leq_k)$ is a ω^2-wqo.*

Proof. A plain combination of Lemma 2.23 with $k-1$ applications of Proposition 2.22. □

2.4.5 An application: Incrementing Counter Automata

As a prominent example of a WSTS we introduce the class of *Incrementing Counter Automata*.

A *counter automaton* (CA) with zero testing is a tuple $\langle \mathbb{A}, Q, q_0, n, \delta, F \rangle$, where $\mathbb{A}$ is a finite alphabet, Q is a finite set of states, q_0 is the initial state, $n \in \mathbb{N}$ is the number of counters, $\delta \subseteq Q \times \mathbb{A} \times L \times Q$ is the transition relation over the instruction set $L = \{\texttt{inc}, \texttt{dec}, \texttt{ifzero}\} \times \{1, \ldots, n\}$, and $F \subseteq Q$ is the set of accepting states. A *counter valuation* is a function $v : \{1, \ldots, n\} \to \mathbb{N}$. An error-free run over $w \in \mathbb{A}^*$ is a finite sequence $\langle q_0, v_0 \rangle \overset{w_0,\ell_0}{\to} \langle q_1, v_1 \rangle \overset{w_1,\ell_1}{\to} \cdots$ observing the standard interpretation of the instructions $\ell_0, \ell_1, \cdots$ ($\langle \texttt{dec}, c \rangle$ can only be performed if c is nonzero), where $v_0, v_1, \ldots$ are counter valuations, v_0 assigns 0 to each counter, and $w = w_0 w_1 \ldots$ such that $w_i \in \mathbb{A} \cup \{\varepsilon\}$ for every i. A run is *accepting* iff it ends with an accepting state.

A *Minsky* CA has error-free runs. For these automata, already with only two counters, finitary emptiness is undecidable (Minsky, 1967). An *Incrementing* CA (from now on ICA) is defined as a Minsky CA except that its runs may contain errors that increase one or more counters non-deterministically. We write that two valuations are in the relation $v \leq v'$ iff, for every counter c, $v(c) \leq v'(c)$. Runs of Incrementing CA are defined by replacing the relation $\overset{a,\ell}{\to}$ by $\overset{a,\ell}{\rightsquigarrow}$, where $\langle p, u \rangle \overset{a,\ell}{\rightsquigarrow} \langle q, v \rangle$ iff there exist valuations u', v' such that $u \leq u'$, $v' \leq v$ and $\langle p, u' \rangle \overset{a,\ell}{\to} \langle q, v' \rangle$.

Example 2.25. As an example of the application of the previous results, let us show that the emptiness problem for ICA is decidable. By Dickson's Lemma, the order between valuations $\leq$ is a well-quasi-order. Further, if we extend the order over pairs of state-valuation by $(q, v) \leq (q', v')$ iff $q = q'$ and $v \leq v'$, we also obtain a well-quasi-order. It is easy to check that $\rightsquigarrow$ is rdc with respect to $\leq$ and that all the hypothesis of Proposition 2.7 are met. Further, since the set of accepting configurations (q, v) with q a final state are trivially downwards closed, we obtain by Lemma 2.8 that the emptiness problem is decidable. □

To complete the picture, we mention that the emptiness problem for ICA cannot be solved in primitive recursive time or space.

Proposition 2.26. *The emptiness problem for ICA is decidable with non-primitive recursive complexity.*

Proof idea. A different faulty variant of a Minsky machine is to have faulty *decrements* instead of faulty increments. This is known in the literature as the class of *Lossy Counter Automata* (LCA) (Mayr, 2003). LCA and ICA are equivalent as observed by Demri and Lazić (2009): for every ICA there is a LCA that accepts the reverse of the language, and viceversa. This translation consists in swapping the accepting states with the final states, and inverting the transition relation. If there is a run of one automaton over a word w, there is also a run of the other automaton over the reverse of w that preserves acceptance. Schnoebelen (2010)

shows that LCA have non-primitive recursive complexity, giving the lower bound of the proposition. □

Part I

WORDS

3. DATA WORDS

3.1 Introduction

In this chapter we investigate alternating register automata on data words in relation to logics. We define a one-way alternating automata model over data words, with one register for storing data and comparing them for equality, and two other operators (which we define later) that add more expressive power, thus expanding the results of Demri and Lazić (2009).

From the standpoint of register automata models, this chapter aims at two objectives: (1) simplifying the existent decidability proofs for the emptiness problem for alternating register automata; and (2) exhibiting decidable extensions for these models.

From the logical perspective, we work with the temporal logic LTL extended with one register for storing and comparing data values. It is denoted by $\mathsf{LTL}^{\downarrow}$ following the notation of Demri and Lazić. This logic contains a 'freeze' operator to store the current datum and a 'test' operator to test the current datum against the stored one. We show that (a) in the case of data words, satisfiability of $\mathsf{LTL}^{\downarrow}$ with *quantification* over data values is decidable; and (b) the satisfiability problem for very weak fragments of $\mathsf{LTL}^{\downarrow}$ with one register are non primitive recursive—markedly, these fragments have no 'next element' X modality.

Automata

The automata model we define is based on the ARA model (for Alternating Register Automata) of Demri and Lazić on data words. ARA are one-way automata with alternating control and one register to store data values for later comparison. This model was shown to have a decidable emptiness problem, through a non-trivial reduction to the emptiness problem for ICA.

In the present work, decidability of these models is shown by interpreting the semantics of the automaton in the theory of WSTS. The object of this alternative proof is twofold. On the one hand, we propose a direct, unified and self-contained proof of the main decidability results of Demri and Lazić. On the other hand, we further generalize these results. Our proof can be easily extended to show the decidability of the non-emptiness problem for two powerful extensions. These extensions consist in the following abilities: (a) the automaton can nondeterministically *guess* any data value of the domain and store it in the register; and (b) it can make a certain kind of universal quantification over the data values seen

along the run of the automaton, in particular over the data values seen so far. We name these extensions guess and spread respectively. These extensions can be added to the ARA model preserving decidability, although losing closure under complementation. We call the model of alternating tree register automata with these extensions by ARA(guess, spread). We demonstrate that these extensions are also decidable if the data domain is equipped with a linear order and the automata model is extended accordingly.

Temporal logics

ARA(guess, spread) over data words yield new decidability results on the satisfiability for some extensions of the temporal logic with one register denoted by $\mathsf{LTL}^{\downarrow}(\mathsf{U},\mathsf{X})$ in (Demri and Lazić, 2009). Our automata model captures an extension of this logic with quantification over data values, where we can express "*for all data values in the past, φ holds*", or "*there exists a data value in the future where φ holds*". Indeed, none of these two types of properties can be expressed in the formalism of Demri and Lazić. These quantifiers may be added to $\mathsf{LTL}^{\downarrow}(\mathsf{U},\mathsf{X})$ over data words without losing decidability. What is more, decidability is preserved if the data domain is equipped with a linear order that is accessible by the logic. However, adding the *dual* of any of these operators results in an undecidable logic.

Our second contribution on these logics is on lower bounds. Thanks to Demri and Lazić we know that $\mathsf{LTL}^{\downarrow}(\mathsf{F},\mathsf{X})$ is has a non-primitive recursive lower bound, and that $\mathsf{LTL}^{\downarrow}(\mathsf{F},\mathsf{F}^{-1},\mathsf{X})$ is undecidable. Here we show that these results carry over even in the absence of X. In other words we show—through a non-trivial encoding—that $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ is non-primitive recursive and $\mathsf{LTL}^{\downarrow}(\mathsf{F},\mathsf{F}^{-1})$ is undecidable. Also, we identify a fragment, called *simple*, that has a very simple navigation, and that has connections with the logic XPath, which will become useful in later chapters.

3.1.1 Contributions

From the standpoint of register automata models, our contributions can be summarized as follows.

- We exhibit a unified framework to show decidability for finitary emptiness problem for alternating register automata. This proof has the advantage of working both for data words and data trees practically unchanged. It is also a simplification of the existent decidability proofs.
- We exhibit a decidable extension for this model of automata, that we call ARA(guess, spread).

From the standpoint of logics, we show the following results.

- We prove that the temporal logic $\mathsf{LTL}^{\downarrow}(\mathsf{U},\mathsf{X})$ for data words extended with quantification over data values is decidable. This result is shown via a trans-

lation from the logic to the class of alternating register automata over data words $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$.

- We show that the temporal logic $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ is non primitive recursive and that $\mathsf{LTL}^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$ is undecidable. This is also shown for another fragment that contains a strict version of F but a more restrictive syntax.

3.1.2 Organization

In Section 3.3 we introduce our automata model $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$, and we show that the emptiness problem for these automata is decidable. This extends the results of Demri and Lazić (2009). To prove decidability, we adopt a different approach than the one taken by Demri and Lazić that enables us to show the decidability of some extensions, and to simplify the decidability proofs of the Chapter 6 in Part II of this thesis, which can be seen as a corollary of this proof. In Section 3.4 we introduce a temporal logic with registers. We show upper and lower bounds for fragments and extensions of this logic. In Section 3.5 we show that the extension of $\mathsf{LTL}^{\downarrow}(\mathsf{U}, \mathsf{X})$ with quantification over data values has a decidable satisfiability problem. This is done by a reduction to the emptiness problem of $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ automata. Finally, in Section 3.6 we prove that the restricted fragment $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ has non primitive recursive complexity, and we also investigate the problem with past modalities or two registers.

3.1.3 Data words

We conclude this section by describing precisely the models we are going to work with.

We consider a finite word over $\mathbb{E}$ as a function $\mathbf{w} : [n] \to \mathbb{E}$ for some $n \in \mathbb{N}_+$, and we define the set of words as $\mathit{Words}(\mathbb{E}) := \{[n] \to \mathbb{E} \mid n \in \mathbb{N}_+\}$. We write $\mathsf{pos}(\mathbf{w}) = \{1, \ldots, n\}$ to denote the set of **positions** (that is, the domain of $\mathbf{w}$). Given $\mathbf{w} \in \mathit{Words}(\mathbb{E})$ and $\mathbf{w}' \in \mathit{Words}(\mathbb{F})$ with $\mathsf{pos}(\mathbf{w}) = \mathsf{pos}(\mathbf{w}') = P$, we write $\mathbf{w} \otimes \mathbf{w}' \in \mathit{Words}(\mathbb{E} \times \mathbb{F})$ for the word such that $\mathsf{pos}(\mathbf{w} \otimes \mathbf{w}') = P$ and $(\mathbf{w} \otimes \mathbf{w}')(x) = (\mathbf{w}(x), \mathbf{w}'(x))$. A **data word** is a word of $\mathit{Words}(\mathbb{A} \times \mathbb{D})$, where $\mathbb{A}$ is a finite alphabet of letters and $\mathbb{D}$ is an infinite domain. We define the **word type** as a function $\mathsf{type}_{\mathbf{w}} : \mathsf{pos}(\mathbf{w}) \to \{\triangleright, \bar{\triangleright}\}$ that specifies whether a position has a next element or not. That is, $\mathsf{type}_{\mathbf{t}}(i) = \triangleright$ iff $(i+1) \in \mathsf{pos}(\mathbf{t})$.

3.2 Related work

The main decidability results presented here first appeared in (Figueira, 2010). Here we include the full proofs and the analysis for alternating register automata on *data words* (something that is out of the scope of (Figueira, 2010), that focuses on data trees) as well as its relation to temporal logics. Also, we show how to proceed

in the presence of a linear order, maintaining decidability. All the results on lower-bounds contained in Section 3.6 are a joint work with Luc Segoufin (Figueira and Segoufin, 2009).

Register Automata

The work of Kaminski and Francez (1994) was a precursor of numerous extensions of automata theory to the case of infinite alphabets. The cited work introduces the class of register automata[1], an extension of finite automata with several registers to store and compare data values. These automata have a decidable emptiness problem. Neven, Schwentick, and Vianu (2004) make a thorough study of expressiveness between pebble automata, register automata (one- or two-way, alternating or nondeterministic), and an extension of monadic second-order logic. The model of one-way alternating register automata ARA studied by Demri and Lazić (2009), can be seen as extensions of the register automata of Kaminski and Francez. The ARA model is extended with alternation on the one hand, but on the other hand restricted to use only one register. It then comes as no surprise that they have incomparable expressive power. In contrast to classical automata, alternating register automata are more expressive than nondeterministic, and two-way register automata are more expressive than one-way (Kaminski and Francez, 1994; Neven et al., 2004). As an example, the *nonces property* (stating that no two word positions have the same data value) or the (W1) property is expressible by an alternating automaton with one register, but not by a nondeterministic k register automaton. In this chapter we study $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ which is an extension of ARA. This automata class will be able to express properties like (W4) or (W5), that cannot be expressed by ARA (not even with a k register ARA).

Bouyer, Petit, and Thérien (2003) studied a decidable class of automata, inspired by the question of representation of timed languages, coining the term *data word*. They propose a class of data word languages admitting a characterization in terms of one-way register automata, an algebraic characterization and even a logical characterization. This is a decidable class of languages, although its expressive power is limited, since it cannot express, for example, the nonces property.

In (Figueira, Hofman, and Lasota, 2010b) we investigate the relation between timed automata and register automata. In particular, this work shows that most problems in one model can be reduced into similar problems in the other. Hence, works on timed automata are also relevant to the community working on data words.

First-order logic

The satisfiability of first-order logic with two variables and data equality tests is explored by Bojańczyk, Muscholl, Schwentick, Segoufin, and David (2006). The logic $\mathsf{FO}^2(\sim, <, +1)$ is shown to have a decidable satisfiability problem. This is

[1] Kaminski and Francez originally used the name *Finite-memory Automata*.

first-order logic where we have only two variables x and y (that we can however reuse), and three binary relations: $<$ is interpreted as the order of positions in the word, $+1$ denotes the relation of consecutive positions, and $\sim$ relates two positions that have the same data value. Further, any $\mathsf{FO}^2(\sim, <, +1)$ formula prefixed with an existential monadic second order quantification (noted $\mathsf{EMSO}^2(\sim, <, +1)$) is decidable. This logic can equivalently be seen as a very elegant automaton called *data automaton*. Björklund and Schwentick (2010) show that data automata capture register automata, and they introduce an equivalent but quite different formalism named *class memory automata*. Further, the decidability is extended to the logic $\mathsf{FO}^2(\sim, <, +1, \ldots, +n)$, where each $+k$ relation tests that the two positions are at distance k. However, this logic is incomparable in terms of expressiveness with respect to the automata and logics of this chapter. For example, in the context of Example 1.1, the property (W2) can be easily expressed in $\mathsf{FO}^2(\sim, <, +1)$ but not with the automata class $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ we introduce here, nor with any of the extensions of $\mathsf{LTL}^{\downarrow}(\mathsf{U}, \mathsf{X})$ we show decidable. In turn, the property (W3) can be expressed with the ARA class, or even with the weakest formalisms treated here $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ but it cannot be expressed with $\mathsf{EMSO}^2(\sim, <, +1)$. Concerning the complexity of the satisfiability of $\mathsf{FO}^2(\sim, <, +1)$ (or $\mathsf{EMSO}^2(\sim, <, +1)$), it is equivalent—modulo an ExpTime reduction—to reachability of VASS (short for Vector Addition System with States) which is ExpSpace-hard and not known to have any primitive recursive upper-bound. On the other hand, the complexity of the emptiness of $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ and all the $\mathsf{LTL}^{\downarrow}$ fragments treated here are decidable in non primitive recursive time, and this is also a lower bound since they can code the Ackermann function.

Adding order

In this context, we could consider that our domain is linearly-ordered $(\mathbb{D}, \prec)$. While the finite satisfiability of $\mathsf{FO}^2(<, +1, \sim, \prec)$ is undecidable (Bojańczyk et al., 2006), Schwentick and Zeume (2010) have recently shown that satisfiability of $\mathsf{FO}^2(\leq, \precsim)$ is in ExpSpace, where $x \precsim y$ iff the data value of x is less or equal than that of y. This fragment cannot test for local properties as it does not contain the $+1$ operator. Thus, in some sense this logic is related to $\mathsf{LTL}^{\downarrow}(\mathsf{F})$, although it is incomparable since "for every a there is a b with the same data value" is not expressible by $\mathsf{LTL}^{\downarrow}(\mathsf{F})$, and since (W3) still cannot be expressed by $\mathsf{FO}^2(\leq, \precsim)$. In a similar setting—but this time in the presence of a *discrete* linear order $(\mathbb{D}, \prec)$—, Manuel (2010) shows decidability of $\mathsf{FO}^2(+1, \tilde{+1})$, where $\tilde{+1}\,(x, y)$ holds if y has the successor data value of x. It is important to mention that in Sections 3.3.3 and 3.5.2 we will show decidable extensions of our results on register automata and $\mathsf{LTL}^{\downarrow}$ to formalisms that can access to a linear order over the data domain.

Manuel and Ramanujam (2009) study a model of automata over infinite alphabets called *Class Counting Automata* with an ExpSpace emptiness problem.[2]

[2] However, for this ExpSpace bound to hold, integers must be coded in *unary*.

This model of automata allows to count the multiplicity of data values along a word, where there is a counter for each data value seen that can count up to a constant. It can express a property like "there are at least k positions with the same data value" for some constant k, that cannot be expressed in the previous formalisms. In contrast, from another perspective this model has a limited expressive power, since the counters cannot be decremented. As in the previous work, a property that uses a simple navigation to relate data values like property (W1) cannot be expressed.

Multiple data values

Bouajjani, Habermehl, Jurski, and Sighireanu (2007) study a logic over models that extend data words: each element has a *vector* of data values. The formalism consists in a fragment of $\mathsf{FO}(<)$ to that allows to navigate the word on top of a theory on data values. The most relevant result with respect to our work is that given a decidable first order theory on data values, then, any Σ_2 first order formula using $<$ for navigation has a decidable satisfiability problem. This result implies the decidability of formalisms with powerful domain-specific operations. For example, if data values are numbers, we may consider using Presburger arithmetic. It is then immediate that in this setting we may express properties that cannot be expressed in any of the formalisms mentioned so far. However, the navigation to interrelate the data values among different positions of the words is severely limited. It can be seen that a property which is simple to express with formalisms $\mathsf{FO}^2(\sim, <)$, ARA, or even $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ as (W1) cannot be captured with a Σ_2 navigation.

Kara, Schwentick, and Zeume (2010) also investigated logics on a similar model that they name *multi-attributes data words*, where every position of a word is labeled with a finite set of pairs (a, d) where a is from some finite alphabet and d from some infinite domain. This work investigates a logic based on $\mathsf{LTL}^{\downarrow}_1$, with a special construct to access the different data values at a position. Markedly, this fragment includes past modalities, but cannot test for *inequality* of a data value with respect to the data value in the register. The novelty of this logic rests on the fact that it handles *multi*-attributes data words, since when restricted to data words it corresponds to the fragment "simple-LTL" (as defined by Demri and Lazić) extended with Until and Since, but restricted in the data tests it can perform. The satisfiability of simple-LTL is decidable since it can be effectively translated to $\mathsf{FO}^2(\sim, <, +1, \ldots, +n)$—in fact, it is equivalent.

Deutsch, Hull, Patrizi, and Vianu (2009) consider even a more general data model where every position is labeled by a state of a relational database, *i.e.*, by a set of relations over a fixed signature. Navigation between positions is performed through temporal operators, and first-order formulæ can be evaluated at states. Properties depending on values at different states can be stated by a prefix of universal quantification of data values. However, no property requiring a "$\forall\exists$" data quantification, like (W1), can be expressed in this formalism.

3.3 Alternating Register Automata

An **Alternating Register Automaton** (ARA) consists in a one-way automaton on data words with alternation and *one* register to store and test data. In (Demri and Lazić, 2009) it was shown that the finitary emptiness problem is decidable and non primitive recursive. Here, we consider an extension of ARA with two operators: spread and guess. We call this model ARA(spread, guess).

Definition 3.1. An alternating register automaton of ARA(spread, guess) is a tuple $\mathcal{M} = \langle \mathbb{A}, Q, q_I, \delta \rangle$ such that

- $\mathbb{A}$ is a finite alphabet;
- Q is a finite set of states;
- $q_I \in Q$ is the initial state; and
- $\delta : Q \to \Phi$ is the transition function, where Φ is defined by the grammar

$$a \mid \bar{a} \mid \odot? \mid \mathsf{store}(q) \mid \mathsf{eq} \mid \overline{\mathsf{eq}} \mid q \wedge q' \mid q \vee q' \mid \triangleright q \mid \mathsf{guess}(q) \mid \mathsf{spread}(q, q')$$

where $a \in \mathbb{A}, q, q' \in Q, \odot \in \{\triangleright, \bar{\triangleright}\}$.

This formalism without the guess and spread transitions is equivalent to the automata model of (Demri and Lazić, 2009) on finite data words, where $\triangleright$ is to move to the next position to the right on the data word, $\mathsf{store}(q)$ stores the current datum in the register and eq (resp. $\overline{\mathsf{eq}}$) tests that the current node's value is (resp. is not) equal to the stored. We call a state $q \in Q$ **moving** if $\delta(q) = \triangleright q'$ for some $q' \in Q$.

As this automaton is one-way, we define its semantics as a set of 'threads' for each node that progress synchronously. That is, all threads at a node move one step forward simultaneously and then perform some non-moving transitions independently. This is done for the sake of simplicity of the formalism, which simplifies both the presentation and the decidability proof.

Next we define a *configuration* and then we give a notion of a *run* over a data word $\mathbf{w}$. A **configuration** is a tuple $\langle i, \alpha, \gamma, \Delta \rangle$ that describes the partial state of the execution at position i. $i \in \mathsf{pos}(\mathbf{w})$ is the *position* in the data word $\mathbf{w}$, $\gamma = \mathbf{w}(i) \in \mathbb{A} \times \mathbb{D}$ is the current position's letter and datum, and $\alpha = \mathsf{type}_{\mathbf{w}}(i)$ is the *word type* of the position i. Finally, $\Delta \in \wp_{<\infty}(Q \times \mathbb{D})$ is a finite set of active **threads**, each thread (q, d) consisting in a state q and the value d stored in the register. We will always note the set of threads of a configuration with the symbol Δ, and we write $\Delta(d) = \{q \in Q \mid (q, d) \in \Delta\}$ for $d \in \mathbb{D}$, $\Delta(q) = \{d \in \mathbb{D} \mid (q, d) \in \Delta\}$ for $q \in Q$. By $\mathcal{C}_{\mathrm{ARA}}$ we denote the set of all configurations. Given a set of threads Δ we write $\mathit{data}(\Delta) := \{d \mid (q, d) \in \Delta\}$, and $\mathit{data}(\langle i, \alpha, (a, d), \Delta \rangle) := \{d\} \cup \mathit{data}(\Delta)$. We say that a configuration is **moving** if for every $(q, d) \in \Delta$, q is moving.

$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \{(q_j, d')\} \cup \Delta \rangle \quad \text{if } \delta(q) = q_1 \vee q_2, j \in \{1,2\} \tag{3.1}$$
$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \{(q_1, d'), (q_2, d')\} \cup \Delta \rangle \quad \text{if } \delta(q) = q_1 \wedge q_2 \tag{3.2}$$
$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \{(q', d)\} \cup \Delta \rangle \quad \text{if } \delta(q) = \mathsf{store}(q') \tag{3.3}$$
$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \Delta \rangle \quad \text{if } \delta(q) = \mathsf{eq} \text{ and } d = d' \tag{3.4}$$
$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \Delta \rangle \quad \text{if } \delta(q) = \overline{\mathsf{eq}} \text{ and } d \neq d' \tag{3.5}$$
$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \Delta \rangle \quad \text{if } \delta(q) = \beta? \text{ and } \beta \in \alpha \tag{3.6}$$
$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \Delta \rangle \quad \text{if } \delta(q) = b \text{ and } b = a \tag{3.7}$$
$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \Delta \rangle \quad \text{if } \delta(q) = \bar{b} \text{ and } b \neq a \tag{3.8}$$

Fig. 3.1: Definition of the transition relation $\to_\varepsilon \subseteq \mathcal{C}_{\text{ARA}} \times \mathcal{C}_{\text{ARA}}$, given a configuration $\rho = \langle i, \alpha, (a,d), \{(q,d')\} \cup \Delta \rangle$.

To define a run we first introduce three transition relations over *node configurations*: the *non-moving* relation $\to_\varepsilon$ and the *moving* relation $\to_\triangleright$. We start with $\to_\varepsilon$. If the transition corresponding to a thread is a $\mathsf{store}(q)$, the automaton *sets* the register with current data value and continues the execution of the thread with state q; if it is eq, the thread *accepts* (and in this case disappears from the configuration) if the current datum is *equal* to that of the register, otherwise the computation for that thread cannot continue. The reader can check that the rest of the cases defined in Figure 3.1 follow the intuition of an alternating automaton.

The cases that follow correspond to our extensions to the model of (Demri and Lazić, 2009). The guess instruction extends the model with the ability of storing *any* datum from the domain $\mathbb{D}$. Whenever $\delta(q) = \mathsf{guess}(q')$ is executed, a data value (nondeterministically chosen) is saved in the register.

$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \{(q', e)\} \cup \Delta \rangle \quad \text{if } \delta(q) = \mathsf{guess}(q'), e \in \mathbb{D} \tag{3.9}$$

Note that the store instruction may be simulated with the guess, eq and $\wedge$ instructions, while guess cannot be expressed by the ARA model.

The 'spread' instruction is an unconventional operator in the sense that it depends on the data of *all* threads in the current configuration with a certain state. Whenever $\delta(q) = \mathsf{spread}(q_2, q_1)$ is executed, a new thread with state q_1 and datum d is created for each thread $\langle q_2, d \rangle$ present in the configuration. With this operator we can code a universal quantification over all the ancestors' data values. We demand that this transition may only be applied if all other possible $\to_\varepsilon$ kind of transitions were already executed. In other words, only spread transitions or *moving* transitions are present in the configuration.

$$\rho \to_\varepsilon \langle i, \alpha, (a,d), \{\langle q_1, d \rangle \mid \langle q_2, d \rangle \in \Delta\} \cup \Delta \rangle \tag{3.10}$$

iff $\delta(q) = \mathsf{spread}(q_2, q_1)$ and for all $\langle \tilde{q}, \tilde{d} \rangle \in \Delta$ either $\delta(\tilde{q})$ is a spread or a moving instruction.

The $\to_{\triangleright}$ transition advances all threads of the node simultaneously, and is defined, for any type $\alpha' \in \{\triangleright, \bar{\triangleright}\}$ and symbol and with data value $\gamma' \in \mathbb{A} \times \mathbb{D}$,

$$\langle i, \triangleright, \gamma, \Delta\rangle \to_{\triangleright} \langle i+1, \alpha', \gamma', \Delta_{\triangleright}\rangle \tag{3.11}$$

iff

(i) for all $(q,d) \in \Delta$, $\delta(q)$ is moving; and

(ii) $\Delta_{\triangleright} = \{\langle q', d\rangle \mid (q,d) \in \Delta, \delta(q) = \triangleright\, q'\}$.

Finally, we define the transition between configurations as $\rightarrowtail := \to_{\triangleright} \cup \to_{\varepsilon}$.

A *run* over a data word $\mathbf{w} = \mathbf{a} \otimes \mathbf{d}$ is a non-empty sequence $\mathcal{C}_1 \rightarrowtail \cdots \rightarrowtail \mathcal{C}_n$ with $\mathcal{C}_1 = \langle 1, \alpha_0, \gamma_0, \Delta_0\rangle$ and $\Delta_0 = \{\langle q_I, \mathbf{d}(1)\rangle\}$ (*i.e.*, the thread consisting in the initial state with the first datum), such that for every $j \in [n]$ with $\mathcal{C}_j = \langle i, \alpha, \gamma, \Delta\rangle$: (1) $i \in \mathsf{pos}(\mathbf{w})$; (2) $\gamma = \mathbf{w}(i)$; and (3) $\alpha = \mathsf{type}_{\mathbf{w}}(i)$. We say that the run is *accepting* iff $\mathcal{C}_n = \langle i, \alpha, \gamma, \emptyset\rangle$ contains an empty set of threads. If for an automaton $\mathcal{M}$ we have that $\mathcal{L}(\mathcal{M}) \neq \emptyset$ we say that $\mathcal{M}$ is *nonempty*.

3.3.1 Properties

We show the following two statements, that are equivalent given the fact that the ARA model is closed under complement.

- the ARA(guess, spread) class is more expressive than ARA.
- $\mathcal{L}$(ARA(guess, spread)) is not closed under complementation.

Proposition 3.2 (Expressive power)**.**

(a) the ARA(guess) class is more expressive than ARA;

(b) the ARA(spread) class is more expressive than ARA.

Proof. Let $\mathbf{w} = \mathbf{a} \otimes \mathbf{d}$ be a data word. To prove (a), consider the property "*There exists a datum d and a position i with $\mathbf{d}(i) = d$, $\mathbf{a}(i) = a$, and there is no position j with $\mathbf{d}(j) = d$, $\mathbf{a}(j) = b$.*". This property can be easily expressed by ARA(guess). It suffices to guess the data value d and check two properties using alternation: that we can reach an element (a,d), and that for every element (b,d') we have that $d \neq d'$. We argue that this property cannot be expressed by the ARA model. Suppose ad absurdum that it is expressible. This means that its negation would also be expressible by ARA (since they are closed under complement). The negation of this property states

"*For every data value d, if there is an element (a,d) in the word, then there is an element (b,d).*" (P1)

With this kind of property one can code an accepting run of a Minsky machine, whose emptiness problem is undecidable. This would prove that ARA(guess) have

an undecidable emptiness problem, which is in contradiction with the decidability proof that we will give in Section 3.3.2. Let us see how the reduction works.

The emptiness problem for Minsky machine is known to be undecidable even with an alphabet consisting of one symbol, so we disregard the letters read by the automaton in the following description. Consider then a 2-counter alphabet-blind Minsky machine whose instructions are of the form (q, ℓ, q') with ℓ being the operation over the counters

$$\ell \in \{\texttt{inc}, \texttt{dec}, \texttt{ifzero}\} \times \{1, 2\},$$

and q, q' states from the automaton's set of states Q. A run on this automaton is a sequence of applications of transition rules, for example

$$(q_1, \texttt{inc}_1, q_2)\,(q_2, \texttt{inc}_2, q_3)\,(q_3, \texttt{inc}_1, q_2)\,(q_2, \texttt{dec}_1, q_1)\,(q_1, \texttt{dec}_1, q_2)\,(q_2, \texttt{ifzero}_1, q_3)$$

This run has an associated data word over the alphabet

$$Q \times \{\texttt{inc}, \texttt{dec}, \texttt{ifzero}\} \times \{1, 2\} \times Q,$$

where the corresponding data value of each instruction is used to match increments with decrements, for example,

$$\underset{1}{(q_1, \texttt{inc}_1, q_2)}\,\underset{2}{(q_2, \texttt{inc}_2, q_3)}\,\underset{3}{(q_3, \texttt{inc}_1, q_2)}\,\underset{2}{(q_2, \texttt{dec}_1, q_1)}\,\underset{1}{(q_1, \texttt{dec}_1, q_2)}\,\underset{4}{(q_2, \texttt{ifzero}_1, q_3)}.$$

Using the ARA model we can make sure that (i) *all increments have different data values and all decrements have different data values*; and (ii) *for every* $(\,, \texttt{inc}_i, \,)$ *element with data value d that occurs to the left of a* $(\,, \texttt{ifzero}_i, \,)$*, there must be a* $(\,, \texttt{dec}_i, \,)$ *element with data value d that occurs in between.* (Demri and Lazić, 2009) shows how to express these properties using ARA. However, properties (i) and (ii) are not enough to make sure that every prefix of the run ending in a $\texttt{ifzero}_i$ instruction must have as many increments as decrements of counter i. Indeed, there could be more decrements than increments—but not the opposite, thanks to (ii).

The missing condition to verify that the run is correct is: (iii) *for every decrement there exists a previous increment with the same data value.* In fact, we can see that property (P1) can express condition (iii): we only need to change a by a decrement transition in the coding, and b by an increment transition of the same counter. But then, assuming that property (P1) can be expressed by ARA(guess), the emptiness problem for ARA(guess) is undecidable. This is absurd, as the emptiness problem is decidable, as we will show later on in Theorem 3.5.

Using a similar reasoning as before, we show (b): the ARA(spread) class is more expressive than ARA. Consider the property "*there exists a position i labeled b such that $\mathbf{d}(i) \neq \mathbf{d}(j)$ for all $j < i$ with $\mathbf{a}(j) = a$.*" as depicted in Figure 3.2. Let us see how this can be coded into ARA(spread). Assuming q_0 is the initial state, the

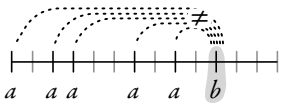

Fig. 3.2: A property not expressible in ARA.

transitions should reflect that every datum with label a seen along the run is saved with a state q_a, and that this state is in charge of propagating this datum. Then, we guess a position labeled with b and check that all these stored values under q_a are different from the current one. For succinctness we write the transitions as positive boolean combinations of the basic operations.

$$\delta(q_0) = (b \wedge \mathsf{spread}(q_a, q_1)) \ \vee \ \big((\bar{a} \vee \mathsf{store}(q_a)) \wedge \triangleright q_0\big),$$
$$\delta(q_1) = \overline{\mathsf{eq}}, \qquad \delta(q_a) = (\bar{\triangleright}? \vee \triangleright q_a).$$

This property cannot be expressed by the ARA model. Were it expressible, then its negation

> "*for every element b there exists a previous one labeled a with the same data value*" (P2)

would also be. Just as before we can use property (P2) to express condition (iii), and force that for every decrement in a coding of a Minsky machine there exists a corresponding previous increment. This leads to a contradiction by proving that the emptiness problem for ARA(spread) is undecidable. □

Corollary 3.3. *ARA(guess), ARA(spread) and ARA(guess, spread) are not closed under complementation.*

Proof. If they were closed under complement, then we could express some of the properties described in the proof of Proposition 3.2, resulting in an undecidable model, which is in contradiction with Theorem 3.5. □

We then have the following properties of the automata model.

Proposition 3.4 (Boolean operations)**.** *The class $\mathcal{L}(ARA(spread, guess))$ has the following properties:*

(i) it is closed under union,

(ii) it is closed under intersection,

(iii) it is not closed under complement.

Proof sketch. Items (i) and (ii) are straightforward if we notice that the first argument of spread ensures that this transition is always relative to the states of one of the automata being under intersection or union. Item (iii) follows from Corollary 3.3. □

3.3.2 Emptiness problem

This section is dedicated to show the following theorem.

Theorem 3.5. *The emptiness problem for* $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ *is decidable.*

As already mentioned, decidability for ARA was proved in (Demri and Lazić, 2009). Here we propose an alternative approach that simplifies the proof of decidability of the two extensions spread and guess.

The proof goes as follows. We will define a wqo $\precsim$ over $\mathcal{C}_{\text{ARA}}$ and show that $(\mathcal{C}_{\text{ARA}}, \precsim)$ is *rdc* with respect to $\rightarrowtail$ (Lemma 3.10). Note that strictly speaking $\rightarrowtail$ is an infinite-branching transition system as $\rightarrow_{\triangleright}$ may take *any* value from the infinite set $\mathbb{D}$, and $\rightarrow_{\varepsilon}$ can also guess any value. However, it can trivially be restricted to an effective *finitely* branching one. Then, by Proposition 2.7, $(\mathcal{C}_{\text{ARA}}, \rightarrowtail)$ has an effectively computable upward-closed reachability set, and this implies that the emptiness problem of $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ is decidable.

Since our model of automata only cares about equality or inequality of data values, it is convenient to work modulo renaming of data values.

Definition 3.6 ($\sim$)**.** We say that two configurations $\rho, \rho' \in \mathcal{C}_{\text{ARA}}$ are **equivalent** (notation $\rho \sim \rho'$) if there is a bijection $f : \mathit{data}(\rho) \to \mathit{data}(\rho')$ such that $f(\rho) = \rho'$, where $f(\rho)$ stands for the replacement of every data value d by $f(d)$ in ρ.

Definition 3.7 ($\preceq$)**.** We first define the relation $(\mathcal{C}_{\text{ARA}}, \preceq)$ such that

$$\langle i, \alpha, \gamma, \Delta \rangle \;\preceq\; \langle i', \alpha', \gamma', \Delta' \rangle$$

iff $\alpha = \alpha'$, $\gamma = \gamma'$, and $\Delta \subseteq \Delta'$.

Notice that by the definition above we are 'abstracting away' the information concerning the position i. We finally define $\precsim$ to be $\preceq$ modulo $\sim$.

Definition 3.8 ($\precsim$)**.** We define $\rho \precsim \rho'$ iff there is $\rho'' \sim \rho'$ with $\rho \preceq \rho'$.

The following lemma follows from the definitions.

Lemma 3.9. $(\mathcal{C}_{\text{ARA}}, \precsim)$ *is a well-quasi-order.*

Proof. The fact that $\precsim$ is a *quasi-order* (*i.e.*, reflexive and transitive) is immediate from its definition. To show that it is a *well*-quasi-order, suppose we have an infinite sequence of configurations $\rho_1\rho_2\rho_3\cdots$. It is easy to see that it contains an infinite subsequence $\tau_1\tau_2\tau_3\cdots$ such that all its elements are of the form $\langle i, \alpha_0, (a_0, d), \Delta \rangle$ with

- α_0 and a_0 fixed, and
- $\Delta(d) = \mathcal{C}_0$ fixed,

This is because we can see each of these elements as a finite *coloring*, and apply the pigeonhole principle on the infinite set $\{\rho_i\}_i$.

Consider then the function $g_\Delta : \wp(Q) \to \mathbb{N}$, such that $g_\Delta(\mathcal{S}) = |\{d \mid \mathcal{S} = \Delta(d)\}|$ (we can think of g_Δ as a tuple of $(\mathbb{N})^{|\wp(Q)|}$). Assume the relation $\precsim^\dagger$ defined as $\Delta \precsim^\dagger \Delta'$ iff $g_\Delta(\mathcal{S}) \leq g_{\Delta'}(\mathcal{S})$ for all $\mathcal{S}$. By Dickson's Lemma $\precsim^\dagger$ is a wqo, and then there are two $\tau_i = \langle i', \alpha_0, (s_0, d_i), \Delta_i\rangle$, $\tau_j = \langle j', \alpha_0, (s_0, d_j), \Delta_j\rangle$, $i < j$ such that $\Delta_i \precsim^\dagger \Delta_j$. For each $\mathcal{S} \subseteq Q$, there exists an injective mapping $f_\mathcal{S} : \{d \mid g_{\Delta_i}(d) = \mathcal{S}\} \to \{d \mid g_{\Delta_j}(d) = \mathcal{S}\}$, as the latter set is bigger than the former by $\precsim^\dagger$. We define the injection $f : \mathit{data}(\tau_i) \to \mathit{data}(\tau_j)$ as the (disjoint) union of all $f_\mathcal{S}$'s. The union is disjoint since for every data value d and set of threads Δ, there is a unique set $\mathcal{S}$ such that $d \in g_\Delta(\mathcal{S})$. We then have that $\tau_i \sim f(\tau_i) \preceq \tau_j$. Hence, $\tau_i \precsim \tau_j$. □

The core of this proof is centered in the following lemma.

Lemma 3.10. *$(\mathcal{C}_{\text{ARA}}, \rightarrowtail)$ is rdc with respect to $(\mathcal{C}_{\text{ARA}}, \precsim)$.*

Proof. We shall show that for all $\rho, \tau, \rho' \in \mathcal{C}_{\text{ARA}}$ such that $\rho \rightarrowtail \tau$ and $\rho' \precsim \rho$, there is τ' such that $\tau' \precsim \tau$ and either $\rho' \rightarrowtail \tau'$ or $\tau' = \rho'$. Since by definition of $\precsim$ we work modulo $\sim$, we can further assume that $\rho' \preceq \rho$ without any loss of generality. The proof is a simple case analysis of the definitions for $\rightarrowtail$. All cases are treated alike, here we present the most representative. Suppose first that $\rho \to_\varepsilon \tau$, then one of the definition conditions of $\to_\varepsilon$ must apply.

If Eq. (3.4) of the definition of $\to_\varepsilon$ (Fig. 3.1) applies, let

$$\rho = \langle i, \alpha, (a, d), \{(q, d)\} \cup \Delta\rangle \to_\varepsilon \tau = \langle i, \alpha, (a, d), \Delta\rangle$$

with $\delta(q) = \mathsf{eq}$. Let $\rho' = \langle i', \alpha, (a, d), \Delta'\rangle \preceq \rho$. If $(q, d) \in \Delta'$, we can then apply the same $\to_\varepsilon$-transition obtaining $\rho \succeq \rho' \to_\varepsilon \tau' \preceq \tau$. If there is no such (q, d), we can safely take $\rho' = \tau'$ and check that $\tau' \preceq \tau$.

If Eq. (3.3) applies, let

$$\rho = \langle i, \alpha, (a, d), \{(q, d')\} \cup \Delta\rangle \to_\varepsilon \tau = \langle i, \alpha, (a, d), \{\langle q', d\rangle\} \cup \Delta\rangle$$

with $\rho \to_\varepsilon \tau$ and $\delta(q) = \mathsf{store}(q')$. Again let $\rho' \preceq \rho$ containing $(q, d') \in \Delta'$. In this case we can apply the same $\to_\varepsilon$-transition arriving to τ' where $\tau' \preceq \tau$. Otherwise, if $(q, d') \notin \Delta'$, we take $\rho' = \tau'$ and then $\tau' \preceq \tau$.

If a guess is performed (Eq. (3.9)), let

$$\rho = \langle i, \alpha, (a, d), \{(q, d')\} \cup \Delta\rangle \to_\varepsilon \tau = \langle i, \alpha, (a, d), \{(q', e)\} \cup \Delta\rangle$$

with $\delta(q) = \mathsf{guess}(q')$. Let $\rho' = \langle i', \alpha, (a, d), \Delta'\rangle \preceq \rho$. Suppose there is $(q, d') \in \Delta'$, then we then take a guess transition from ρ' obtaining some τ' by guessing e and hence $\tau' \preceq \tau$. Otherwise, if $(q, d') \notin \Delta'$, we take $\tau' = \rho'$ and check that $\tau' \preceq \tau$.

Finally, if a spread is performed (Eq. (3.10)), let

$$\rho = \langle i, \alpha, \gamma, \{(q, d')\} \cup \Delta\rangle \to_\varepsilon \tau = \langle i, \alpha, \gamma, \{(q_1, d) \mid (q_2, d) \in \Delta\} \cup \Delta\rangle$$

with $\delta(q) = \mathsf{spread}(q_2, q_1)$. Let $\rho' = \langle i', \alpha, \gamma, \Delta' \rangle \preceq \rho$ and suppose there is $(q, d') \in \Delta'$ (otherwise $\tau' = \rho'$ works). We then take a spread instruction $\rho' \to_\varepsilon \tau'$ and see that $\tau' \preceq \tau$, because any (q_1, d') in τ' generated by the spread must come from (q_2, d') of ρ', and hence there is some (q_2, d') in ρ; now by the spread applied on ρ, (q_1, d') is in τ.

The remaining cases of $\to_\varepsilon$ are only easier.

Finally, there can be a 'moving' application of $\rightarrowtail$. Suppose that we have

$$\rho = \langle i, \triangleright, (a, d), \Delta \rangle \to_\triangleright \tau = \langle i + 1, \alpha_1, (a_1, d_1), \Delta_1 \rangle$$

where $\rho \to_\triangleright \tau$. Let $\rho' = \langle i', \triangleright, (a, d), \Delta' \rangle \preceq \rho$. If ρ' is such that $\rho' \preceq \tau$, the relation is trivially compatible. Otherwise, we shall prove that there is τ' such that $\rho' \rightarrowtail \tau'$ and $\tau' \precsim \tau$. Condition i of $\to_\triangleright$ (*i.e.*, that all states are moving) holds for ρ', because all the states present in ρ' are also in ρ (by definition of $\preceq$) where the condition must hold. Then, we can apply the $\to_\triangleright$ transition to ρ' and obtain τ' of the form $\langle i' + 1, \alpha_1, (a_1, d_1), \Delta_1' \rangle$. Notice that we are taking α_1, a_1 and d_1 exactly as in τ, and that Δ_1' is completely determined by the $\to_\triangleright$ transition from Δ'. We only need to check that $\tau' \preceq \tau$. Take any $(q, d') \in \Delta_1'$. There must be some $(q', d') \in \Delta'$ with $\delta(q') = \triangleright q$. Since $\Delta' \subseteq \Delta$, we also have $(q, d) \in \Delta_1$. Hence, $\Delta_1' \subseteq \Delta_1$ and then $\tau' \preceq \tau$. □

We just showed that $(\mathcal{C}_{\text{ARA}}, \rightarrowtail)$ is *rdc* with respect to $(\mathcal{C}_{\text{ARA}}, \precsim)$. The only missing ingredient to have decidability is the following, which is trivial.

Lemma 3.11. *The set of accepting configurations of $\mathcal{C}_{\text{ARA}}$ is downward closed with respect to $\precsim$.*

We write $\mathcal{C}_{\text{ARA}}/\!\sim$ to the set of configurations modulo $\sim$, by keeping one representative for every equivalence class. Note that the transition system $(\mathcal{C}_{\text{ARA}}/\!\sim, \rightarrowtail)$ is effective. This is just a consequence of the fact that the $\rightarrowtail$-image of any configuration has only a finite number of configurations modulo $\sim$, and representatives for every class are effectively computable. Hence, we have that $(\mathcal{C}_{\text{ARA}}/\!\sim, \precsim, \rightarrowtail)$ verify conditions (1) and (2) from Proposition 2.7. Finally, condition (3) holds as $(\mathcal{C}_{\text{ARA}}/\!\sim, \precsim)$ is a is a wqo (by Lemma 3.10) that is computable. We can then apply Lemma 2.8, obtaining that for a given $\mathcal{M} \in \mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$, testing wether there exists a final configuration τ and an element ρ in

$$\{\langle 1, \alpha, (a, d_0), \{(q_I, d_0)\} \rangle \mid \alpha \in \{\triangleright, \bar{\triangleright}\}, a \in \mathbb{A}\}$$

—for any fixed d_0—such that $\rho \sim \rho' \rightarrowtail^* \tau$ (for some ρ') is decidable. Thus, we can decide the emptiness problem and Theorem 3.5 follows.

3.3.3 Ordered data

We show here that the previous decidability result holds even if we add *order* to the data domain. Let $(\mathbb{D}, <)$ be a linear order, like for example the reals or the

natural numbers with the standard ordering. Let us replace the instructions eq and $\overline{\mathsf{eq}}$ with

$$\delta(q) := \ \ldots \ \mid \ \mathsf{test}(>) \ \mid \ \mathsf{test}(<) \ \mid \ \mathsf{test}(=) \ \mid \ \mathsf{test}(\neq)$$

and let us call this model of automata $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread}, <)$. The semantics is as expected. $\mathsf{test}(<)$ verifies that the data value of the current position is less than the data value in the register, $\mathsf{test}(>)$ that is greater, and $\mathsf{test}(=)$ (resp. $\mathsf{test}(\neq)$) that both are (resp. are not) equal. We modify accordingly $\rightarrow_\varepsilon$, for $\rho = \langle i, \alpha, (a, d), \{(q, d')\} \cup \Delta \rangle$.

$$\rho \rightarrow_\varepsilon \langle i, \alpha, (a, d), \Delta \rangle \quad \text{if } \delta(q) = \mathsf{test}(<) \text{ and } d < d' \tag{3.12}$$
$$\rho \rightarrow_\varepsilon \langle i, \alpha, (a, d), \Delta \rangle \quad \text{if } \delta(q) = \mathsf{test}(>) \text{ and } d > d' \tag{3.13}$$
$$\rho \rightarrow_\varepsilon \langle i, \alpha, (a, d), \Delta \rangle \quad \text{if } \delta(q) = \mathsf{test}(=) \text{ and } d = d' \tag{3.14}$$
$$\rho \rightarrow_\varepsilon \langle i, \alpha, (a, d), \Delta \rangle \quad \text{if } \delta(q) = \mathsf{test}(\neq) \text{ and } d \neq d' \tag{3.15}$$

All the remaining definitions are preserved. We can show that the emptiness problem for this extended model of automata is still decidable.

Theorem 3.12. *The finitary emptiness problem for $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread}, <)$ is decidable.*

As in the proof in the previous Section 3.3.2, we show that there is a wqo $\ll \subseteq \mathcal{C}_{\text{ARA}} \times \mathcal{C}_{\text{ARA}}$ that is rdc with respect to $\rightarrowtail$, such that the set of final states is $\ll$-downward closed. However, we need to be be more careful when showing that we can always work modulo an equivalence relation.

Definition 3.13. A function f is an **order-preserving bijection** on $D \subseteq \mathbb{D}$ iff it is a bijection on $\mathbb{D}$, and furthermore for every $\{d, d'\} \subseteq D$, if $d < d'$ then $f(d) < f(d')$.

The following Lemma is plain from the definition just seen.

Lemma 3.14. *Let $D \subseteq \mathbb{D}$, $|D| < \infty$. There exists an order-preserving bijection f on D such that*

- *for every $\{d, d'\} \subseteq D$ such that $d < d'$ then there exists $\tilde{d}$ such that $f(d) < \tilde{d} < f(d')$,*
- *for every $d \in D$ there exists $\tilde{d}$ such that $f(d) < \tilde{d}$, and there exists $\tilde{d}$ such that $\tilde{d} < f(d)$.*

Definition 3.15 ($\sim_{ord}$)**.** $\rho \sim_{ord} \rho'$ iff $f(\rho) = \rho'$ for some order-preserving bijection f on $data(\rho)$.

Remark 3.16. If $\rho \rightarrowtail \rho'$ then there exists $\hat{d} \in \mathbb{D}$ such that $\{\hat{d}\} \cup data(\rho) \subseteq data(\rho')$. This is a simple consequence of the definition of $\rightarrowtail$.

Let us define a version of $\rightarrowtail$ that works modulo $\sim_{ord}$, and let us call it $\rightarrowtail_{ord}$.

Definition 3.17. $\rho_1 \rightarrowtail_{ord} \rho_2$ iff $\rho_1' \rightarrowtail \rho_2'$ for some $\rho_1' \sim_{ord} \rho_1$ and $\rho_2' \sim_{ord} \rho_2$.

In the previous section, when we had that $\sim$ was simply a bijection and we could not test any linear order $<$, it was plain that we could work modulo $\sim$. However, here we are working modulo a more complex relation $\sim_{ord}$. In the next lemma we show that working with $\rightarrowtail$ or working with $\rightarrowtail_{ord}$ is equivalent.

Lemma 3.18. *If $\rho_0 \rightarrowtail_{ord} \cdots \rightarrowtail_{ord} \rho_n$, then $\rho_0' \rightarrowtail \cdots \rightarrowtail \rho_n'$, with $\rho_i' \sim_{ord} \rho_i$ for every i.*

Proof. The case $n = 1$ is trivial. If on the other hand we have $\rho_0 \rightarrowtail_{ord} \cdots \rightarrowtail_{ord} \rho_{n-1} \rightarrowtail_{ord} \rho_n$, then by inductive hypothesis we obtain $\rho_1' \rightarrowtail \cdots \rightarrowtail \rho_{n-1}'$ and $\rho_{n-1}'' \rightarrowtail \rho_n''$ with $\rho_i' = \rho_i$ for every i, $\rho_{n-1}'' \sim_{ord} \rho_{n-1} \sim_{ord} \rho_{n-1}'$, and $\rho_n'' \sim_{ord} \rho_n$. Let g be the witnessing bijection such that $g(\rho_{n-1}'') = \rho_{n-1}'$, and let us assume that $\{\hat{d}\} \cup \mathit{data}(\rho_{n-1}'') \subseteq \mathit{data}(\rho_n'')$ by Remark 3.16.

Let f be an order-preserving bijection on $\bigcup_{i \leq n-1} \mathit{data}(\rho_i')$ as in Lemma 3.14. We can then pick a data value $\tilde{d}$ such that

- for every $d > \hat{d}$ with $d \in \mathit{data}(\rho_{n-1}'')$, $f(g(d)) > \tilde{d}$, and
- for every $d < \hat{d}$ with $d \in \mathit{data}(\rho_{n-1}'')$, $f(g(d)) < \tilde{d}$.

Let $h := (g \circ f)[\hat{d} \mapsto \tilde{d}]$. We then have

- $h(\rho_{n-1}'') \rightarrowtail h(\rho_n'')$,
- $f(\rho_1') \rightarrowtail \cdots \rightarrowtail f(\rho_{n-1}')$,
- $f(\rho_{n-1}') = h(\rho_{n-1}'')$.

In other words there exist $\rho_0''' \rightarrowtail \rho_n'''$ with $\rho_i''' \sim_{ord} \rho_i$ for every i. □

Now that we proved that we can work modulo $\sim_{ord}$, we show that we can decide if we can reach an accepting configuration by means of $\rightarrowtail_{ord}$, by introducing some suitable ordering $\ll$ and showing the following lemmas.

Lemma 3.19. *$(\mathcal{C}_{\text{ARA}}, \ll)$ is a well-quasi-order.*

Lemma 3.20. *$(\mathcal{C}_{\text{ARA}}, \ll)$ is rdc with respect to $\rightarrowtail_{ord}$.*

Lemma 3.21. *The set of accepting configurations of $\mathcal{C}_{\text{ARA}}$ is downward closed with respect to $\ll$.*

We next define the order $\ll$ and show that the aforementioned lemmas are valid. In the same spirit as before, $\ll$ is defined as $\preceq$ modulo $\sim_{ord}$.

Definition 3.22 ($\ll$)**.** $\rho_1 \ll \rho_2$ iff $\rho_1' \preceq \rho_2'$ for some $\rho_1' \sim_{ord} \rho_1$ and $\rho_2' \sim_{ord} \rho_2$.

To prove Lemma 3.19, given a configuration $\rho = \langle i, \alpha, (a, d), \Delta \rangle$, with $data(\rho) = \{d_1 < \cdots < d_n\}$ we define

$$abs(d_i) = \Delta(d_i) \cup \{\star \mid d_i = d\} \subseteq Q \cup \{\star\}$$
$$abs(\rho) = abs(d_1), \ldots, abs(d_n) \in (\wp(Q \cup \{\star\}))^*$$

where $\star \notin Q$ is to denote that the data value is the one of the current element.

Proof of Lemma 3.19. This is a consequence of Higman's Lemma stated as in Corollary 2.3. As stated above, we can see each configuration $\rho = (i, \alpha, (a, d), \Delta)$ as a word over $(\wp(Q \cup \{\star\}))^*$. As shown in Lemma 3.9 if there is an infinite sequence, there is an infinite subsequence $\rho_1, \rho_2, \ldots$, with the same type α and letter a. Then for the infinite sequence $abs(\rho_1), abs(\rho_2), \ldots$, Corollary 2.3 tells us that there are $i < j$ such that $abs(\rho_i)$ is a substring of $abs(\rho_j)$. This implies that they are in the $\ll$ relation. □

Proof of Lemma 3.20. Note that although $\ll$ is a more restricted wqo, for all the non-moving cases in which the register is not modified (that is, all except guess, spread, and store), the $\rightarrow_\varepsilon$ transition continues to be rdc. This is because for any $\tau \ll \rho \rightarrow_\epsilon \rho'$, $\rho = \langle i, \alpha, \gamma, \Delta \rangle$ and $\rho' = \langle i', \alpha', \gamma', \Delta' \rangle$ are similar in the following sense. Firstly $data(\rho) = data(\rho')$, and moreover the only difference between ρ and ρ' is that Δ' is the result of removing some thread (q, d) from Δ and inserting another one (q', d) with the same data value d. This kind of operation is compatible, since τ can perform the same operation $\tau \rightarrow_\varepsilon \tau'$ on the data value d', supposing that d' is the preimage of d given by the $\ll$ ordering. In this case, $\tau' \ll \rho'$. Otherwise, if there is no preimage of d, then $\tau \ll \rho'$. The compatibility of spread is shown equivalently.

Regarding the store instruction, we see that the operation consists in removing some (q, d) from Δ and inserting some (q', d_0) with d_0 the root's datum. This is downwards compatible since τ can perform the same operation on its root data value, which is necessarily the preimage of d_0.

For the remaining two cases (guess and $\triangleright$) we rely on the premise that we we work modulo $\sim_{ord}$. The idea is that we can always assume that we have enough data values to choose from in between the existing ones. That is, for every pair of data values $d < d'$ in a configuration, there is always one in between. We can always assume this since otherwise we can apply a bijection as the one described by Lemma 3.14 to obtain this property. Thus, at each point where we need to guess a data value (as a consequence of a guess(q) or a $\triangleright q$ instruction) we will have no problem in performing a symmetric action, preserving the embedding relation.

More concretely, suppose the execution of a transition $\delta(q) = \mathsf{guess}(q')$ on a thread (q, d_j) of configuration ρ with $data(\rho) = \{d_1 < \cdots < d_n\}$ guesses a data value d with $d_i < d < d_{i+1}$. Then, for any configuration $\tau \ll \rho$ with $data(\tau) = \{e_1 < \cdots < e_m\}$ and the property just described, there must be an order-preserving injection $f : data(\tau) \to data(\rho)$ with $f(\tau) \preceq \rho$. If τ contains a thread $(q, e_{j'})$ with $f(e_{j'}) = d_j$ the operation is simulated by guessing a data value

e such that $e > e_\ell$ for all e_ℓ such that $f(e_\ell) \leq d_i$ and $e < e_k$ for all e_k such that $f(e_k) > d_i$. Such data value e exists as explained before. The rdc compatibility of a $\delta(q) = \triangleright q'$ instruction is shown in an analogous fashion. □

Proof of Lemma 3.21. Given that $\ll$ is a subset of $\precsim$, and that by Lemma 3.11 the set of accepting configurations is $\precsim$-downward closed, it follows that this set is also $\ll$-downward closed. □

Finally, we should note that $(\mathcal{C}_{\text{ARA}}/\sim_{ord}, \rightarrowtail_{ord})$ is also finitely branching and effective. As in the proof of Section 3.3.2, by Lemmas 3.19, 3.20 and 3.21 we have that all the conditions of Proposition 2.7 are met and by Lemma 2.8 we conclude that the finitary emptiness problem for $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread}, <)$ is decidable.

Remark 3.23. Notice that this proof works independently of the particular ordering of $(\mathbb{D}, <)$. It could be dense or discrete, contain accumulation points, be open or closed, etc. In some sense, this automata model is blind to these kind of properties. If there is an accepting run on $(\mathbb{D}, <)$ then there is an accepting run on $(\mathbb{D}, <')$ for any linear order $<'$.

Question 3.24. It is perhaps possible that these results can be extended to prove decidability when $(\mathbb{D}, <)$ is a *partial order*, this time making use of Kruskal's tree theorem (Kruskal, 1960) instead of Higman's Lemma. We leave this issue as an open question.

Remark 3.25 (constants). One can also extend this model with a finite number of constants $\{c_1, \ldots, c_n\} \subseteq \mathbb{D}$. In this case, we extend the transitions with the possibility of testing that the data value stored in the register is (or is not) equal to c_i, for every i. In the proof, it suffices to modify $\sim_{ord}$ to take into account every constant c_i. In this case we define that $\rho \sim_{ord} \tau$ iff $f(\rho) = \rho'$ for some order-preserving bijection f on $data(\rho) \cup \{c_1, \ldots, c_n\}$ such that $f(c_i) = c_i$ for every i. In this case Lemma 3.14 does not hold anymore, as there could be finitely many elements in between two constants. This is however an easily surmountable obstacle, by adapting Lemma 3.14 to work separately on the $n+1$ intervals defined by $c_1, \ldots, c_n$. For each interval, if it is infinite we will have a Lemma 3.14-like statement, and if it is finite we simply state that we can make space between any two data values of D contained in the interval as long as there is at least one element of the which is not in D.

3.3.4 Timed automata

Our investigation on register automata also yields new results on the class of timed automata. An *alternating 1-clock timed automaton* is an automaton that runs over *timed words*. A timed word is a finite sequence of *events*. Each event carries a symbol from a finite alphabet and a *timestamp* indicating the quantity of time elapsed from the first event of the word. A timed word can hence be seen as a data word over the rational numbers, whose data values are strictly increasing. The automaton has alternating control and contains one *clock* to measure the lapse of

time between two events (that is, the difference between the data of two positions of the data word). It can reset the clock, or test whether the clock contains a number equal, less or greater than a constant, from some finite set of constants. For more details on this automaton we refer the reader to (Alur and Dill, 1994).

Register automata over ordered domains have a strong connection with timed automata. The work in (Figueira et al., 2010b) shows that the problems of nonemptiness, language inclusion, language equivalence and universality are equivalent—modulo an ExpTime reduction—for timed automata and register automata over a linear order. That is, any of these problems for register automata can be reduced to the same problem on timed automata, preserving the number of registers equal to the number of clocks, and the mode of computation (nondeterministic, alternating). And in turn, any of these problems for timed automata can also be reduced to a similar problem on register automata over a linear order. We argue that this is also true when the automata are equipped with guess and spread.

Consider an extension of 1-clock alternating timed automata, with spread and guess, where

- the operator $\mathsf{spread}(q, q')$ works in the same way as for register automata, duplicating all threads with state q as threads with state q', and
- the $\mathsf{guess}(q)$ operator resets the clock to any value, non deterministically chosen, and continues the execution with state q.

The coding technique of (Figueira et al., 2010b) can be adapted to deal with the guessing of a clock (the spread operator being trivially compatible), and one can show the following statement.

Lemma 3.26. *The emptiness problem for alternating 1-clock timed automata extended with guess and spread reduces to the emptiness problem for the class ARA(guess, spread, <).*

Hence, by Lemma 3.26 cum Theorem 3.12 we obtain the following result.

Theorem 3.27. *The emptiness problem for alternating 1-clock timed automata extended with guess and spread is decidable.*

3.3.5 A note on complexity

Although the $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ and $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread}, <)$ classes have both non-primitive recursive complexity, we must remark nonetheless the latter have much higher complexity. While the former can be roughly bounded by the Ackermann function applied to the number of states, the complexity of $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread}, <)$ majorizes every multiply-recursive function (in particular, Ackermann's). In some sense this is a consequence of relying on Higman's Lemma instead of Dickson's Lemma for the termination arguments of our algorithm.

More precisely, it can be seen that the emptiness problem for $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread}, <)$ sits in the class $\mathfrak{F}_{\omega^\omega}$ in the Fast Growing Hierarchy (Löb and Wainer, 1970)—an extension of the Grzegorczyk Hierarchy for non-primitive recursive functions—by a reduction to timed one clock automata (Section 3.3.4). The emptiness problem for timed one clock automata can be at the same time reduced to that of Lossy Channel Machines (Abdulla et al., 2005), which are known to be 'complete' for this class, *i.e.* in $\mathfrak{F}_{\omega^\omega} \setminus \mathfrak{F}_{<\omega^\omega}$ (Chambart and Schnoebelen, 2008). However, the emptiness problem for $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ belongs to $\mathfrak{F}_\omega$ in the hierarchy. The lower bound follows by a reduction from Incrementing Counter Automata (Demri and Lazić, 2009), which are hard for $\mathfrak{F}_\omega$ (Schnoebelen, 2010; Figueira et al., 2010a). The upper bound is a consequence of using a saturation algorithm with a wqo that is the component-wise order of the coordinates of a vector of natural numbers in a controlled way. The proof that it belongs to $\mathfrak{F}_\omega$ goes similarly as for Incrementing Counter Automata (see Figueira et al., 2010a, §7.2).

3.4 LTL with register

The logic $\mathsf{LTL}^{\downarrow}_n(\mathcal{O})$ is a logic for data words that corresponds to the extension of the Linear Temporal Logic $\mathsf{LTL}(\mathcal{O})$ on data words, where $\mathcal{O}$ is a subset of the usual navigation modalities, for example $\{\mathsf{F}, \mathsf{U}, \mathsf{X}\}$. The logic is extended with the ability to use n different *registers* for storing a data value for later comparisons, and it was studied in (Demri and Lazić, 2009; Demri et al., 2005). The *freeze* operator $\downarrow_i \varphi$ permits to *store* the current datum in register i and continue the evaluation of the formula φ. The operator $\uparrow_i$ *tests* whether the current data value is equal to the one stored in register i. We use the usual navigation modalities: the next (X), future (F) and until (U) temporal operators, together with its inverse counterparts (X^{-1}, F^{-1}, U^{-1}).

As it was shown by Demri and Lazić (2009), $\mathsf{LTL}^{\downarrow}_1(\mathsf{U}, \mathsf{X})$ has a decidable satisfiability problem with non primitive recursive complexity. However, as soon as $n \geq 2$, satisfiability of $\mathsf{LTL}^{\downarrow}_n(\mathsf{U}, \mathsf{X})$ becomes undecidable. We write $\mathsf{LTL}^{\downarrow}(\mathcal{O})$ as short for $\mathsf{LTL}^{\downarrow}_1(\mathcal{O})$. In this section we focus on decidable upper bounds for extensions of $\mathsf{LTL}^{\downarrow}(\mathsf{X}, \mathsf{U})$, and on lower bounds for some fragments.

In Section 3.5, we show that $\mathsf{LTL}^{\downarrow}(\mathsf{U}, \mathsf{X})$ can be extended with quantification over data values, extending the previous decidability results. On the other hand, in Section 3.6 we investigate the lower bounds for very weak fragments. By the lower bounds given in (Demri and Lazić, 2009), the complexity of the satisfiability problem for $\mathsf{LTL}^{\downarrow}(\mathsf{U}, \mathsf{X})$ is non-primitive recursive. Here we show that $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ has non-primitive recursive complexity, and that $\mathsf{LTL}^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$ is undecidable. Further, if two registers are allowed, $\mathsf{LTL}^{\downarrow}_2(\mathsf{F})$ is also undecidable.

For simplicity we define the semantics of the logic with only one register. For the particular result of Section 3.6.2 that uses the two-register fragment, we will explain how to extend this semantics. Fix a finite alphabet $\mathbb{A}$. Sentences of $\mathsf{LTL}^{\downarrow}(\mathcal{O})$, where

$$
\begin{array}{rll}
(\mathbf{w}, i) \models^d a & \text{iff} & \mathbf{a}(i) = a \\
(\mathbf{w}, i) \models^d \uparrow & \text{iff} & d = \mathbf{d}(i) \\
(\mathbf{w}, i) \models^d \downarrow \varphi & \text{iff} & (\mathbf{w}, i) \models^{\mathbf{d}(i)} \varphi \\
(\mathbf{w}, i) \models^d \mathsf{U}(\varphi, \psi) & \text{iff} & \text{for some } i \leq j \in \mathsf{pos}(\mathbf{w}) \text{ and for all } i \leq k < j \\
 & & \text{we have } (\mathbf{w}, j) \models^d \varphi \text{ and } (\mathbf{w}, k) \models^d \psi \\
(\mathbf{w}, i) \models^d \mathsf{X}\varphi & \text{iff} & i + 1 \in \mathsf{pos}(\mathbf{w}) \text{ and } (\mathbf{w}, i + 1) \models^d \varphi
\end{array}
$$

Fig. 3.3: Semantics of $\mathsf{LTL}^{\downarrow}(\mathsf{X}, \mathsf{F}, \mathsf{U}, \mathsf{X}^{-1}, \mathsf{F}^{-1}, \mathsf{U}^{-1})$ for a data word $\mathbf{w} = \mathbf{a} \otimes \mathbf{d}$ and $i \in \mathsf{pos}(\mathbf{w})$. The interpretation of $\wedge$, $\vee$ and $\neg$ is standard.

$$
\begin{array}{rll}
(\mathbf{w}, i) \models^d \exists^{\downarrow}_{\geq} \varphi & \text{iff} & \text{there exists } i \leq j \in \mathsf{pos}(\mathbf{w}) \text{ such that } (\mathbf{w}, i) \models^{\mathbf{d}(j)} \varphi \\
(\mathbf{w}, i) \models^d \forall^{\downarrow}_{\leq} \varphi & \text{iff} & \text{for all } 1 \leq j \leq i \text{ we have } (\mathbf{w}, i) \models^{\mathbf{d}(j)} \varphi
\end{array}
$$

Fig. 3.4: Semantics of $\mathsf{LTL}^{\downarrow}(\mathcal{O}, \exists^{\downarrow}_{\geq}, \forall^{\downarrow}_{\leq})$ for a data word $\mathbf{w} = \mathbf{a} \otimes \mathbf{d}$ and $i \in \mathsf{pos}(\mathbf{w})$.

$\mathcal{O} \subseteq \{\mathsf{X}, \mathsf{F}, \mathsf{U}, \mathsf{X}^{-1}, \mathsf{F}^{-1}, \mathsf{U}^{-1}\}$ are defined as follows,

$$\varphi ::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid a \mid \uparrow \mid \downarrow \varphi \mid o\,\varphi \qquad (o \in \mathcal{O})$$

where a is a symbol from the finite alphabet $\mathbb{A}$.

Figure 3.3 shows the definition of the satisfaction relation $\models$. As usual, we define the *future* modality as $\mathsf{F}\varphi := \mathsf{U}(\varphi, \top) \vee \varphi$, and U^{-1}, F^{-1}, X^{-1} as the symmetric of U, F, X. For example, in this logic we can express properties like "*for every a element there is a future b element with the same data value*" as $\mathsf{G}(\neg a \vee \downarrow (\mathsf{F}(b \wedge \uparrow)))$. We say that φ satisfies $\mathbf{w} = \mathbf{a} \otimes \mathbf{d}$, written $\mathbf{w} \models \varphi$, if $\mathbf{w}, 1 \models^{\mathbf{d}(1)} \varphi$.

3.5 LTL: Upper bounds

We study an extension of this language with a restricted form of *quantification* over data values. We will actually add *two* sorts of quantification. On the one hand the $\forall^{\downarrow}_{\leq}$ and $\exists^{\downarrow}_{\leq}$ quantify universally or existentially over all the data values occurred *before* the current point of evaluation. Similarly, $\forall^{\downarrow}_{\geq}$ and $\exists^{\downarrow}_{\geq}$ quantify over the *future* elements on the data word. Hence, we can express "*for all data values in the past, φ holds*", or "*there exists a data value in the future where φ holds*", with φ a $\mathsf{LTL}^{\downarrow}(\mathsf{X}, \mathsf{F}, \mathsf{U})$ formula. Semantics are given in Figure 3.4.

For our convenience and without any loss of generality, we will work in Negated Normal Form (nnf), using $\bar{\mathsf{U}}$ to denote the dual operator of U, and similarly for $\bar{\mathsf{X}}$. We write $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathcal{O})$ for the fragment of $\mathsf{LTL}^{\downarrow}(\mathcal{O})$ where the nnf of every formula

uses only operators from $\mathcal{O}$. Let us write $\mathfrak{F} := \{\mathsf{U}, \bar{\mathsf{U}}, \mathsf{X}, \bar{\mathsf{X}}\}$, denoting that we have all the *forward* modalities. Notice that $\mathsf{LTL}^{\downarrow}(\mathsf{U}, \mathsf{X})$ is the same as $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F})$. We investigate the fragment $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \forall^{\downarrow}_{\leq}, \exists^{\downarrow}_{\geq})$, and show that it has a decidable satisfiability problem. This fragment is not closed under negation and is hence different from $\mathsf{LTL}^{\downarrow}(\mathfrak{F}, \forall^{\downarrow}_{\leq}, \exists^{\downarrow}_{\geq})$. Indeed, the latter logic has an undecidable satisfiability problem, as we will show.

3.5.1 Satisfiability problem

This section treats the satisfiability problem for $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \forall^{\downarrow}_{\leq}, \exists^{\downarrow}_{\geq})$. But first let us show that $\exists^{\downarrow}_{\leq}$ and $\forall^{\downarrow}_{\geq}$ result in an undecidable logic.

Theorem 3.28. *Let $\exists^{\downarrow}_{<}$ be the operator $\exists^{\downarrow}_{\leq}$ restricted only to the data values occurring strictly before the current point of evaluation. Then, on finite data words:*

(1) satisfiability of $\mathsf{LTL}^{\downarrow}_{nnf}(\mathsf{F}, \mathsf{G}, \exists^{\downarrow}_{<})$ is undecidable; and

(2) satisfiability of $\mathsf{LTL}^{\downarrow}_{nnf}(\mathsf{F}, \mathsf{G}, \forall^{\downarrow}_{\geq})$ is undecidable.

Proof. We prove (1) and (2) by reduction of the halting problem for Minsky machines. We show that these logics can code an accepting run of a 2-counter Minsky machine as in Proposition 3.2. Indeed, we show that the same kind of properties are expressible in this logic. To prove this, we build upon the proof of Theorem 3.38 showing that $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathsf{F}, \mathsf{G})$ can code conditions (i) and (ii) of the proof of Proposition 3.2. Here we show that both $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathsf{F}, \mathsf{G}, \exists^{\downarrow}_{<})$ and $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathsf{F}, \mathsf{G}, \forall^{\downarrow}_{\geq})$ can express condition (iii), ensuring that for every decrement ($\mathtt{dec}_i$) there is a previous increment ($\mathtt{inc}_i$) with the same data value. Let us see how to code this.

(1) The $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathsf{F}, \mathsf{G}, \exists^{\downarrow}_{<})$ formula

$$\mathsf{G}(\mathsf{dec}_i \rightarrow \exists^{\downarrow}_{<} \uparrow)$$

states that the data value of every decrement must *not* be new, and in the context of this coding this means that it must have been introduced by an increment instruction.

(2) The $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathsf{F}, \mathsf{G}, \forall^{\downarrow}_{\geq})$ formula

$$\forall^{\downarrow}_{\geq}(\mathsf{F}(\mathsf{dec}_i \wedge \uparrow) \rightarrow \mathsf{F}(\mathsf{inc}_i \wedge \uparrow))$$

evaluated at the first element of the data word expresses that for every data value: if there is a *decrement* with value d, then there is an *increment* with value d. It is then easy to ensure that they appear in the correct order (first the increment, then the decrement).

The addition of any of these conditions to the coding of the proof of Theorem 3.38. results in a coding of an n-counter Minsky machine, whose emptiness problem is undecidable. □

Corollary 3.29. *The satisfiability problem for* $\mathsf{LTL}^{\downarrow}_{nnf}(\mathfrak{F}, \exists^{\downarrow}_{\leq})$ *and* $\mathsf{LTL}^{\downarrow}_{nnf}(\mathfrak{F}, \forall^{\downarrow}_{\geq})$ *are undecidable.*

Proof. The property of item (1) in the proof of Theorem 3.28 can be equally coded in $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \exists^{\downarrow}_{\leq})$ as $\mathsf{G}(\mathsf{X}(\mathsf{dec}_i) \rightarrow \exists^{\downarrow}_{\leq}(\mathsf{X} \uparrow))$. The undecidability of $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \forall^{\downarrow}_{\geq})$ follows directly from Theorem 3.28, item (2). □

We now turn to our decidability result. We show that $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \forall^{\downarrow}_{\leq}, \exists^{\downarrow}_{\geq})$ has a decidable satisfiability problem by a translation to $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$.

The translation to code $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \forall^{\downarrow}_{\leq}, \exists^{\downarrow}_{\geq})$ into $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ is standard, and follows same lines as Demri and Lazić (2009)[3] (which at the same time follows the translation from $\mathsf{LTL}^{\downarrow}$ to alternating finite automata). We then obtain the following result.

Proposition 3.30. *ARA*(*guess*, *spread*) *captures* $\mathsf{LTL}^{\downarrow}_{nnf}(\mathfrak{F}, \exists^{\downarrow}_{\geq}, \forall^{\downarrow}_{\leq})$.

From Proposition 3.30 and Theorem 3.5 it will follow the main result, stated next.

Theorem 3.31. *The satisfiability problem for* $\mathsf{LTL}^{\downarrow}_{nnf}(\mathfrak{F}, \exists^{\downarrow}_{\geq}, \forall^{\downarrow}_{\leq})$ *is decidable.*

We now show how to make the translation to $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$ in order to obtain our decidability result.

Proof of Proposition 3.30. We show that for every formula $\varphi \in \mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \forall^{\downarrow}_{\leq}, \exists^{\downarrow}_{\geq})$ there exists an effectively computable $\mathsf{ARA}(\mathsf{spread}, \mathsf{guess})$ $\mathscr{M}$ such that for every data word $\mathbf{w}$,

$$\mathbf{w} \text{ satisfies } \varphi \quad \text{iff} \quad \mathscr{M} \text{ accepts } \mathbf{w}.$$

The translation for $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F})$ is like the one described in (Demri and Lazić, 2009) and presents no complications whatsoever. For the coding of the $\forall^{\downarrow}_{\leq}$ operator, we first make sure to maintain all the data values seen so far as *threads* of the configuration. We can do this easily as follows.

$$\delta(q_1) := \mathsf{store}(q_2) \qquad \delta(q_2) := (\bar{\triangleright}? \vee \triangleright q_1) \wedge q_{save} \qquad \delta(q_{save}) := \bar{\triangleright}? \vee \triangleright q_{save}$$

Now we can assume that at any point of the run, we maintain the data values of all the previous elements of the data word as threads (q_{save}, d). Note that these threads are maintained until the last element of the data word, at which point the test $\bar{\triangleright}?$ is satisfied and they are accepted. At the last element we cannot be sure to have the q_{save} threads with the data needed, but this is not a problem. In fact, a $\forall^{\downarrow}_{\leq}$ operator evaluated at the last element of a word can be simulated without using the $\forall^{\downarrow}_{\leq}$, as the last element is a distinguished one. That is, a formula $\forall^{\downarrow}_{\leq}(\bar{\mathsf{X}}\bot \wedge \varphi)$ results in the same automaton as the translation of $\mathsf{G}(\downarrow \mathsf{F}(\bar{\mathsf{X}}\bot \wedge \varphi))$. Then, for the

[3] Note that this logic already contains $\mathsf{LTL}^{\downarrow}(\mathsf{U}, \mathsf{X})$.

inner nodes we translate a formula $\forall^{\downarrow}_{\leq}(\mathsf{X}\top \wedge \varphi)$ as $\delta(q) := \rhd? \wedge \mathsf{spread}(q_{\text{save}}, q_\varphi)$, where q_φ codes the formula φ.

On the other hand, a formula like $\exists^{\downarrow}_{\geq}\varphi$ is simply translated as $\delta(q) = \mathsf{guess}(q')$ with $\delta(q') = q_\varphi \wedge q_{\mathsf{F}\uparrow}$, where $\delta(q_\varphi)$ is the translation of φ and $\delta(q_{\mathsf{F}\uparrow})$ is the translation of $\mathsf{F}\uparrow$. □

Moreover, we argue that these extensions add expressive power.

Proposition 3.32. *On finite data words:*

(i) The logic $\mathsf{LTL}^{\downarrow}_{nnf}(\mathfrak{F}, \forall^{\downarrow}_{\leq})$ *is more expressive than* $\mathsf{LTL}^{\downarrow}_{nnf}(\mathfrak{F})$*;*

(ii) The logic $\mathsf{LTL}^{\downarrow}_{nnf}(\mathfrak{F}, \exists^{\downarrow}_{\geq})$ *is more expressive than* $\mathsf{LTL}^{\downarrow}_{nnf}(\mathfrak{F})$*.*

Proof. This is a consequence of $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F})$ being closed under negation and Theorem 3.28. Ad absurdum, if one of these logics were as expressive as $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F})$, then it would be closed under negation, and then we could express conditions (1) or (2) of the proof of Theorem 3.28 and hence obtain that $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F})$ is undecidable. But this leads to a contradiction since by Theorem 3.31 $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F})$ is decidable. □

Remark 3.33. The translation of Proposition 3.30 is far from using all the expressive power of spread. In fact, we can consider a binary operator $\forall^{\downarrow}_{\leq}(\varphi, \psi)$ defined

$$\mathbf{w}, i \models^{d} \forall^{\downarrow}_{\leq}(\varphi, \psi) \quad \text{iff} \quad \text{for all } j \leq i \text{ such that } \mathbf{w}, j \models^{\mathbf{d}(j)} \psi, \text{ we have } \mathbf{w}, i \models^{\mathbf{d}(j)} \varphi.$$

with $\psi \in \mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F})$. This operator can be coded into $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$, using the same technique as in Proposition 3.30. The only difference is that instead of 'saving' every data value in q_{save}, we use several states $q_{\text{save}(\psi)}$. Intuitively, only the data values that verify the test $\downarrow \psi$ are stored in $q_{\text{save}(\psi)}$. Then, a formula $\forall^{\downarrow}_{\leq}(\varphi, \psi)$ is translated as $\mathsf{spread}(q_{\text{save}(\psi)}, q_\varphi)$.

3.5.2 Ordered data

If we consider a linear order over $\mathbb{D}$ as done in Section 3.3.3, we can consider $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \exists^{\downarrow}_{\geq}, \forall^{\downarrow}_{\leq})$ with richer tests

$$\varphi ::= \quad \uparrow_{>} \mid \uparrow_{<} \mid \dots$$

that access to the linear order and compare the data values for $=, <, >$. The semantics are extended accordingly, as in Figure 3.5. Let us call this logic *ord*-$\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \exists^{\downarrow}_{\geq}, \forall^{\downarrow}_{\leq})$. The translation from *ord*-$\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \exists^{\downarrow}_{\geq}, \forall^{\downarrow}_{\leq})$ into $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread}, <)$ as defined in Section 3.3.3 is straightforward. Thus we obtain the following result.

Proposition 3.34. *Finitary satisfiability problem for ord-*$\mathsf{LTL}^{\downarrow}_{nnf}(\mathfrak{F}, \exists^{\downarrow}_{\geq}, \forall^{\downarrow}_{\leq})$ *is decidable.*

$$(\mathbf{w}, i) \models^d \uparrow_> \quad \text{iff} \quad d > \mathbf{d}(i)$$
$$(\mathbf{w}, i) \models^d \uparrow_< \quad \text{iff} \quad d < \mathbf{d}(i)$$

Fig. 3.5: Semantics for the operators $\uparrow_>$, $\uparrow_<$ for a data word $\mathbf{w} = \mathbf{a} \otimes \mathbf{d}$ and $i \in \mathsf{pos}(\mathbf{w})$.

3.6 LTL: Lower bounds

We consider fragments of LTL with registers, where we can only perform navigation to a future or a past element (*i.e.*, no X, X^{-1} modalities). In this section we make an explicit distinction between the strict and non-strict future modalities. We write $\mathsf{F_s}$ to denote the strict version of F, which is defined as XF. Notice that although F can be defined in terms of $\mathsf{F_s}$, the converse does not hold. The fragments treated here are those whose operators are among F, F^{-1}, $\mathsf{F_s}$, $\mathsf{F_s}^{-1}$.

We prove that the satisfiability problem for $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ has non-primitive recursive complexity, and that $\mathsf{LTL}^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$ has an undecidable satisfiability problem. Further, we also show that if we have two registers, then $\mathsf{LTL}^{\downarrow}_2(\mathsf{F})$ is also undecidable. All these results are known to hold when we have the next modality X at our disposal by (Demri and Lazić, 2009). Removing the X modality is not at all trivial, and requires a coding trick that allows to weakly simulate the behavior of X with the use of data values.

All the results of the present Section are a joint work with Luc Segoufin and appeared in (Figueira and Segoufin, 2009). In Section 4.5 we will lift our results to trees and XPath. In order to do this it is convenient to introduce now a restriction of $\mathsf{LTL}^{\downarrow}$ such that with this restriction $\mathsf{LTL}^{\downarrow}$ corresponds to XPath.

Definition 3.35 ($\mathsf{sLTL}^{\downarrow}$)**.** A formula of $\mathsf{LTL}^{\downarrow}$ is said to be **simple** if (i) there is at most one occurrence of $\uparrow$ within the scope of each occurrence of $\downarrow$ and, (ii) there is no negation between an occurrence of $\uparrow$ and its matching $\downarrow$, except maybe immediately before $\uparrow$. We denote by $\mathsf{sLTL}^{\downarrow}$ the fragment of $\mathsf{LTL}^{\downarrow}$ containing only simple formulæ.[4]

The correspondence between $\mathsf{sLTL}^{\downarrow}$ and XPath will be made explicit in Proposition 4.2 of Section 4.5. We show that the satisfiability problem for $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$ has non-primitive recursive complexity, while for $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$ is undecidable.

All non-primitive recursive lower bounds are shown by reduction from the emptiness problem for the class of Incrementing Counter Automata introduced in Section 2.4.5. The undecidability results are shown by reduction from Counter Automata.

[4] This fragment has no connection with the *simple* fragment defined by Demri and Lazić (2009).

3.6.1 The case of simple-LTL

In this section we show that satisfiability of $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$ is non primitive recursive over data words. We then prove that satisfiability is undecidable for $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$. In the next section we will show that the logic $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ is also non primitive recursive and that $\mathsf{LTL}^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$ is undecidable. Although this fragment appears to be artificial, remember that it has a special interest since it will permit us to deduce lower bounds for several XPath fragments (Section 4.5).

Lower bound for $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$.

Theorem 3.36. *Satisfiability of* $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$ *on data words is non primitive recursive.*

Proof. We exhibit a PTime reduction from the non-emptiness of ICA to satisfiability of $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$. Let $C = \langle \mathbb{A}, Q, q_0, n, \delta, F \rangle$ be an ICA.
Let $L = \{(\mathtt{inc}_i)_{1\leq i\leq n}, (\mathtt{dec}_i)_{1\leq i\leq n}, (\mathtt{ifzero}_i)_{1\leq i\leq n}\}$, and $\hat{\mathbb{A}} = Q \times (\mathbb{A} \cup \{\varepsilon\}) \times L \times Q$. We construct a formula $\varphi_C \in \mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$ that is satisfied by a data word iff C accepts the word. We view a run of C of the form:

$$\langle q_0, v_0 \rangle \overset{a,\mathtt{inc}_i}{\rightsquigarrow} \langle q_1, v_1 \rangle \overset{b,\mathtt{dec}_j}{\rightsquigarrow} \langle q_2, v_2 \rangle \overset{b,\mathtt{ifzero}_i}{\rightsquigarrow} \langle q_3, v_3 \rangle \ \cdots$$

as a string in $\hat{\mathbb{A}}$:

$$\langle q_0, a, \mathtt{inc}_i, q_1 \rangle \langle q_1, b, \mathtt{dec}_j, q_2 \rangle \langle q_2, b, \mathtt{ifzero}_i, q_3 \rangle \ \cdots$$

The formula φ_C will force that any string that satisfies it codes a run of C. In order to do this, φ_C must ensure that:

(begin) the string starts with q_0,

(end) the string ends with a state of F,

(tran) every symbol of $\hat{\mathbb{A}}$ in the string corresponds to a transition of C,

(chain) the last component of a symbol of $\hat{\mathbb{A}}$ is equal to the first component of the next symbol,

(pair) for each i, every symbol that contains $\mathtt{inc}_i$ occurring in the string to the left of a symbol containing $\mathtt{ifzero}_i$, can be paired with a symbol containing $\mathtt{dec}_i$ and occurring in between the $\mathtt{inc}_i$ and the $\mathtt{ifzero}_i$.

Before continuing let us comment on the (pair) condition. If we were coding runs of a perfect Minsky CA (ie, with no incremental errors), to the left of any position containing a $\mathtt{ifzero}_i$, we would require a perfect matching between $\mathtt{inc}_i$ and $\mathtt{dec}_i$ operations in order to make sure that the value of the counter i is indeed zero at the position of the test. But as we are dealing with ICA, we only need to check that each $\mathtt{inc}_i$ has an associated $\mathtt{dec}_i$ to its right and before the test for zero,

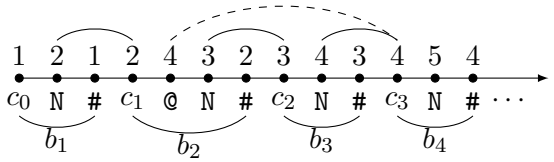

Fig. 3.6: Coding of an ICA run.

but we do not enforce the converse, that all $\texttt{dec}_i$ match an $\texttt{inc}_i$. This is fortunate because this would require past navigational operators.

The first difficulty comes from the fact that `(pair)` is not a regular relation. The pairing will be obtained using data values. The second difficulty is to enforce `(chain)` without having access to the string successor relation. In order to simulate the successor relation we add extra symbols to the alphabet, with suitable associated data values.

Let $\mathbb{A}' = \hat{\mathbb{A}} \cup \{\texttt{N}, \texttt{\#}, \texttt{@}\}$. The coding of a run consists in a succession of *blocks*. Each block is a sequence of 3 or 4 symbols, "c N #" or "c @ N #", with $c \in \hat{\mathbb{A}}$. The data value associated to the c and # symbols of a block is the same and uniquely determines the block: no two blocks may have the same data value. The data value associated with the symbol N of a block is the data value of the *next* block. If a block contains a symbol c that codes a $\texttt{inc}_i$ operation that is later in the string followed by a $\texttt{ifzero}_i$, then this block contains a symbol @ whose data value is the that of the block containing $\texttt{dec}_i$ that is paired with c.

For instance in the example of Figure 3.6 one can see four blocks b_1, b_2, b_3, b_4. Each of them starts with a symbol from $\hat{\mathbb{A}}$ coding a transition of the ICA and ends with a # with the same data value marking the end of the block. Inside the block, the data value of N is the same as the data value of the next block. The data value of @ corresponds to that of a future block. In this example c_1 must correspond to a $\texttt{inc}_i$ while c_3 to $\texttt{dec}_i$ and there must be a $\texttt{ifzero}_i$ somewhere to the remaining part of the word (say, b_5). Moreover c_2 can't be a $\texttt{ifzero}_i$ as otherwise the data value of the symbol @ would refer to a block to the left of c_2.

We now show that the coding depicted above can be enforced in $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$. By $\pi_1, \pi_2, \pi_3, \pi_4$ we denote the projection of each symbol of $\hat{\mathbb{A}}$ into its corresponding component. To simplify the presentation we use the following abbreviations:

$$\hat{\sigma} \equiv \bigvee_{c \in \hat{\mathbb{A}}} c \qquad \textsc{inc}(i) \equiv \bigvee_{c \in \hat{\mathbb{A}}, \pi_3(c) = \texttt{inc}_i} c \qquad \textsc{inc} \equiv \bigvee_i \textsc{inc}(i)$$

$$\textsc{dec}(i) \equiv \bigvee_{c \in \hat{\mathbb{A}}, \pi_3(c) = \texttt{dec}_i} c \qquad \textsc{last} \equiv \hat{\sigma} \wedge \neg \mathsf{F_s}\, \hat{\sigma} \qquad \textsc{iz}(i) \equiv \bigvee_{c \in \hat{\mathbb{A}}, \pi_3(c) = \texttt{ifzero}_i} c$$

The formula φ_C that we build is the conjunction of all the folowing formulæ.

Forcing the structure.

$\mathsf{G}(\textsc{last} \Rightarrow \neg \mathsf{F_s} \top)$: The string ends with the last transition,

$\bigwedge_{c \in \{\mathtt{N},\mathtt{@},\mathtt{\#}\}} \mathsf{G}(c \Rightarrow \neg(\downarrow \mathsf{F_s}(c \wedge \uparrow)))$: the data value associated to each `N`, `#` and `@` uniquely determines the occurrence of that symbol,

$\mathsf{G}\big((\hat{\sigma} \wedge \neg\textsc{last}) \Rightarrow (\downarrow \mathsf{F_s}(\mathtt{N} \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow)))\big)$: each occurrence of $\hat{\mathbb{A}}$ (except the last one) is in a block that contains a `N` and then a `#`,

$\bigwedge_i \mathsf{G}\big((\textsc{inc}(i) \wedge \mathsf{F_s}(\textsc{iz}(i))) \Rightarrow \downarrow \mathsf{F_s}(\mathtt{@} \wedge \mathsf{F_s}(\mathtt{N} \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow)))\big)$: every $\mathtt{inc}_i$ block to the left of a $\mathtt{ifzero}_i$ must have a `@` before the `N`,

$\mathsf{G}((\hat{\sigma} \wedge \neg\textsc{inc}) \Rightarrow \neg \downarrow \mathsf{F_s}(\mathtt{@} \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow)))$: only blocks `inc` are allowed to have a `@`,

$\bigwedge_{s \in \{\mathtt{N},\mathtt{@}\}} \mathsf{G}\big(\hat{\sigma} \Rightarrow \neg(\downarrow \mathsf{F_s}(s \wedge \mathsf{F_s}(s \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow))))\big)$: there is at most one occurrence of `N` and `@` in each block,

$\bigwedge_{s \in \hat{\mathbb{A}} \cup \{\mathtt{\#}\}} \mathsf{G}(\hat{\sigma} \Rightarrow \neg \downarrow \mathsf{F_s}(s \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow))$: there is exactly one symbol `#` and one symbol of $\hat{\mathbb{A}}$ per block,

$\mathsf{G}\big(\mathtt{N} \Rightarrow \downarrow \mathsf{F_s}(\mathtt{\#} \wedge \mathsf{F_s}(\hat{\sigma} \wedge \uparrow))\big)$: each symbol `N`'s datum points to a block to its right,

$\mathsf{G}(\mathtt{N} \Rightarrow \neg \downarrow \mathsf{F_s}(\mathtt{\#} \wedge \mathsf{F_s}(\mathtt{\#} \wedge \mathsf{F_s}(\hat{\sigma} \wedge \uparrow))))$: in fact `N` has has to point be the next block.

Once the structure has the expected shape, we can enforce the run as follows. All the formulæ below are based on the following trick. In a test of the form $\downarrow \mathsf{F_s}(\mathtt{N} \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow))$ which is typically performed at a position of symbol $\hat{\mathbb{A}}$, the last symbol `#` must have the same data value as the initial position. Hence, because of the structure above, both must be in the same block. Thus the middle symbol `N` must also be inside that block. From the structure we know that the data value of this `N` points to the next block. Therefore by replacing the test `N` by one of the form $\mathtt{N} \wedge (\downarrow \mathsf{F_s}(\uparrow \wedge s))$ we can transfer some finite information from the current block to the next one. This gives the desired successor relation.

Forcing a run.

(begin)
$$\bigvee_{c \in \hat{\mathbb{A}}, \pi_1(c) = q_0} c$$

(end)
$$\mathsf{F_s}\Big(\textsc{last} \wedge \bigvee_{c \in \hat{\mathbb{A}}, \pi_4(c) \in F} c\Big)$$

(tran) All elements used from $\hat{\mathbb{A}}$ correspond to valid transitions. Let $\hat{\mathbb{A}}^C$ be that set of transitions of C,
$$\mathsf{G}\Big(\bigwedge_{c \in \hat{\mathbb{A}} \setminus \hat{\mathbb{A}}^C} \neg c\Big)$$

(chain) For every $c \in \hat{\mathbb{A}}$,

$$\mathsf{G}\Big(c \Rightarrow \big(\downarrow \mathsf{F_s}\big(\mathtt{N} \wedge \mathsf{F_s}(\#\wedge \uparrow) \wedge \bigvee_{\substack{d\in\hat{\mathbb{A}},\\ \pi_4(c)=\pi_1(d)}} (\downarrow \mathsf{F_s}(d\wedge\uparrow))\big)\big)\Big)$$

(pair) We first make sure that the block of the @ of an $\mathtt{inc}_k$ is matched with a block of a $\mathtt{dec}_k$:

$$\bigwedge_k \mathsf{G}\Big(\textsc{inc}(k) \Rightarrow \Big(\neg\downarrow \mathsf{F_s}(@\wedge \mathsf{F_s}(\#\wedge\uparrow))\vee \downarrow \mathsf{F_s}\big(@\wedge\downarrow \mathsf{F_s}(\textsc{dec}(k)\wedge\uparrow)\wedge \mathsf{F_s}(\#\wedge\uparrow)\big)\Big)\Big)$$

Now, every $\mathtt{inc}_k$ block to the left of a $\mathtt{ifzero}_k$ block:

1. The block must contain an @ element:

$$\bigwedge_k \mathsf{G}\big(\textsc{inc}(k) \Rightarrow \big(\downarrow \mathsf{F_s}(@ \wedge \mathsf{F_s}(\#\wedge\uparrow)) \vee \neg\mathsf{F_s}(\textsc{iz}(k))\big)\big)$$

2. The data value of that @ element must point to a future block *before* any occurrence of $\mathtt{ifzero}_k$:

$$\bigwedge_k \mathsf{G}\big(\textsc{inc}(k) \Rightarrow \neg\big(\downarrow \mathsf{F_s}\big(@\wedge\downarrow \mathsf{F_s}(\textsc{iz}(k)\wedge \mathsf{F_s}\uparrow)\wedge \mathsf{F_s}(\#\wedge\uparrow)\big)\big)\big)$$

This concludes the construction of φ_C.

Correctness. Given an ICA C, the proof exhibited a formula $\varphi_C \in \mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$ that is satisfied by a data word iff C accepts the word. We show here that the construction is *correct*.

From data word to an accepting ICA run. Let w be a data word satisfying φ_C. Let $B_1B_2\cdots B_m$ be the decomposition of w into blocks as described above. Let $c_i = \langle q_i, a_i, \ell_i, q_{i+1}\rangle$ be the element of B_i in $\hat{\Sigma}$. We show that $w' = a_1a_2\cdots a_m$ is accepted by C. We construct a run of C on w' such that C is in state q_i before reading letter a_i of w' and execute the transitions c_i. By *(tran)* each one of these are valid transitions of the ICA, and *(chain)* ensures that the sequence is consistent. *(begin)* takes care of the initialization condition and *(end)* of the accepting condition. It remains to construct the valuation $v_1, \ldots, v_{m+1}$ of the counter at each step such that we have $\langle q_i, v_i\rangle \overset{a_i,\ell_i}{\rightsquigarrow} \langle q_{i+1}, v_{i+1}\rangle$ as desired.

As expected v_1 is the zero valuation. We construct the rest of the valuations by induction, simulating a perfect counter machine and introducing incrementing errors in a lazy way, whenever necessary. At step i, given a transition $\langle q_i, a_i, \ell_i, q_{i+1}\rangle$ and the current valuation v_i we set v_{i+1} such that $\langle q_i, v_i\rangle \overset{a_i,\ell_i}{\rightsquigarrow} \langle q_{i+1}, v_{i+1}\rangle$.

- If ℓ_i is $\texttt{inc}_k$, then $v_{i+1} := v_i[k \mapsto v_i(k)+1]$ and clearly $\langle q_i, v_i\rangle \overset{a_i,\ell_i}{\to} \langle q_{i+1}, v_{i+1}\rangle$.
- If ℓ_i is $\texttt{dec}_k$ then:
  - If $v_i(k) > 0$ then $v_{i+1} := v_i[k \mapsto v_i(k) - 1]$.
  - If $v_i(k) = 0$, then $v_{i+1} := v_i$.

  In both cases one can verify that $\langle q_i, v_i\rangle \overset{a_i,\ell_i}{\rightsquigarrow} \langle q_{i+1}, v_{i+1}\rangle$, the transition being with an incrementing error of 1 in the second case.
- If ℓ_i is $\texttt{ifzero}_k$, we set $v_{i+1} := v_i$. To ensure that $\langle q_i, v_i\rangle \overset{a_i,\ell_i}{\rightsquigarrow} \langle q_{i+1}, v_{i+1}\rangle$ we must show that necessarily $v_i(k) = 0$. This is a consequence of condition `pair`. This condition enforces that each $\texttt{inc}_k$ must be paired with a unique $\texttt{dec}_k$ to its right and before the $\texttt{ifzero}_k$ test. Hence the lazy strategy above ensures that each increment is decremented later on and before the zero test.

From an accepting ICA run to a data word. Given an accepting run of the ICA $\langle q_1, v_1\rangle \overset{a_1,\ell_1}{\rightsquigarrow} \langle q_2, v_2\rangle \overset{a_2,\ell_2}{\rightsquigarrow} \dots \overset{a_m,\ell_m}{\rightsquigarrow} \langle q_{m+1}, v_{m+1}\rangle$ we construct a data word w verifying φ_C as follows. Let c_i be $\langle q_i, a_i, \ell_i, q_{i+1}\rangle$. w consists of a concatenation of m blocks $B_1 B_2 \cdots B_m$ each one defined:

- If $i = m$, B_i is the word c_i with data value m.
- If $i < m$, and $\ell_i \in \{\texttt{dec}_k, \texttt{ifzero}_k\}$, then set B_i to $c_i\texttt{N\#}$ with respective data values $i, i+1, i$.
$$\langle c_i, i\rangle\langle \texttt{N}, i+1\rangle\langle \texttt{\#}, i\rangle$$
- If $i < m$ and ℓ_i is $\texttt{inc}_k$. Consider the minimal $d > i$ such that $v_{d+1}(k) = v_i(k)$ and set B_i to
  - c_i `@ N #` with respective data values $i, d, i+1, i$ if d exists
  - c_i `N #` with respective data values $i, i+1, i$ if not.

This construction assigns the unique data value i to each block B_i, and set the data value of each symbol `N` to the data value of the next block B_{i+1}. The constraints `(begin)`, `(end)`, `(tran)`, `(chain)` are obviously true. It remains to verify the constraints on `@`: they must have a unique data value, must point to a corresponding `dec` instruction that occur before any corresponding zero test, and no two `@` may point to the same `dec` instruction. Consider a position i such that ℓ_i is $\texttt{inc}_k$ with a subsequent zero test $\texttt{ifzero}_k$ at position $j > i$. Because the zero test is correct, the increment of counter k made at step i must be decremented at some position between i and j. Hence there is a $i < d < j$ such that $v_{d+1}(k) = v_i(k)$. By construction the data value of the `@` symbol of B_i is the minimal such d. By minimality, c_d must be a $\texttt{dec}_k$ instruction. Assume now that at position $i' < i$ there is also a $\texttt{inc}_k$ instruction. The data value of the `@` symbol of $B_{i'}$ cannot be

d. Because if this would be the case then $v_{i'}(k) = v_{d+1}(k) = v_i(k)$ and d would not be minimal for the position i'. Hence (pair) is also satisfied and $w \models \varphi_C$.

This concludes the proof that the finitary satisfiability problem for $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$ has non-primitive recursive complexity. □

Undecidability of $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$

We now consider $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$. The extra modality can be used to code the run of a (non faulty) Minsky CA.

Theorem 3.37. *Satisfiability of* $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$ *is undecidable.*

Proof. Consider a Minsky CA C. We revisit the proof of Theorem 3.36. It is easy to see that we can enforce the absence of faulty increments during the run by asking that every $\mathtt{dec}_i$ element is referenced by some previous $\mathtt{inc}_i$ block: $\bigwedge_i \mathsf{G}(\textsc{dec}(i) \Rightarrow \downarrow \mathsf{F_s}^{-}(@ \wedge \uparrow))$. We thus make sure that every dec is related to a corresponding inc. Hence, the coding is that of a *perfect* (non faulty) run. □

3.6.2 The case of LTL

In this section we lift the lower bounds of the previous section by considering the temporal operator F instead of $\mathsf{F_s}$. We can only do so by removing at the same time the restriction to simple formulæ. Hence the results of this section cannot be applied to XPath. Notice that $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ and $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$ are incomparable in terms of expressive power. Indeed, properties like $\downarrow \mathsf{F}(\mathsf{a} \wedge \bar{\uparrow} \wedge \mathsf{F}(\mathsf{b} \wedge \uparrow))$ cannot be expressed in $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$, while $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ cannot express that the model has at least two elements. We do not know whether $\mathsf{sLTL}^{\downarrow}(\mathsf{F})$, which is weaker than the two above mentioned logics, is already non primitive recursive. The results of this section improve the results of Demri and Lazić (2009) which show that satisfiability is non primitive recursive for $\mathsf{LTL}^{\downarrow}(\mathsf{X}, \mathsf{F})$ and undecidable for $\mathsf{LTL}^{\downarrow}(\mathsf{X}, \mathsf{F}, \mathsf{F}^{-1})$.

Theorem 3.38. *Over data words,*

(1) Satisfiability of $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ *is decidable and non primitive recursive.*

(2) Satisfiability of $\mathsf{LTL}^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$ *is undecidable.*

Proof. We concentrate first on Item (1), the proof of Item (2) being similar.

Item **(1).** Consider an ICA C and recall the coding of runs of C used in the proof of Theorem 3.36. In the construction of the formula φ_C, whenever we have "$s \wedge \mathsf{F_s}(s' \wedge \varphi)$" for some φ and $s \neq s'$ two different symbols, $\mathsf{F_s}$ can be equivalently replaced by the F temporal operator. This is the case in all formulæ except in three places:

(i) The formula saying that N should point to the next block contains $\# \wedge \mathsf{F_s}(\# \wedge \varphi)$. But from the structure that is enforced, this can equivalently be replaced by $\# \wedge \mathsf{F}(\mathtt{N} \wedge \mathsf{F}(\# \wedge \varphi))$.

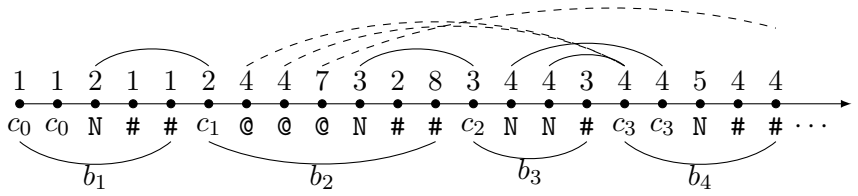

Fig. 3.7: $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ cannot avoid having repeated consecutive symbols.

(ii) To enforce that each block contains at most one occurrence of symbols in $\hat{\mathbb{A}} \cup \{\mathtt{N}, \mathtt{\#}, \mathtt{@}\}$.

(iii) To enforce that no two symbols in $\hat{\mathbb{A}} \cup \{\mathtt{N}, \mathtt{\#}, \mathtt{@}\}$ have the same data value.

In order to cope with (ii) and (iii), we use a slightly different coding for runs of C. This coding is the same as the one for the proof of Theorem 3.36 except that we allow succession of equal symbols, denoted as *group* in the sequel. Note that in a group two different occurrences of the same symbol in general may have different data values, as we can no longer enforce their distinctness. However, as we will see, we can enforce that the $\hat{\mathbb{A}}$ group of elements of a block have all *the same data value*.

Hence a *block* is now either a group of $c \in \hat{\mathbb{A}}$ followed by a group of `N` followed by a group of `#` or the same with a group of `@` in between. A coding of a run is depicted in Figure 3.7.

This structure is enforced by modifying the formulæ of the proof of Theorem 3.36 as follows.

(a) The formulæ that limit the number of occurrences of symbols in a block are replaced by formulæ *limiting the number of groups* in a block.

(b) The formulæ requiring that no two occurrences of a same symbol may have the same data values are replaced by formulæ requiring that no two occurrences of a same symbol *in different groups* have the same data value.

(c) In all other formulæ, F_s is replaced by F.

(d) Finally, we ensure that although we may have repeated symbols inside a block, all symbols from $\hat{\mathbb{A}}$ have the same data value.

$$\mathsf{G}(\hat{\sigma} \Rightarrow \neg \downarrow \mathsf{F}((\hat{\sigma} \vee \mathtt{\#}) \wedge \neg \uparrow \wedge \mathsf{F}(\mathtt{\#} \wedge \uparrow)))$$

Note that this implies that each `N` of a group must have the same data values as they all point to the next block. However there could still be `@` symbols with different data values as depicted in Figure 3.7.

The new sentences now imply, for instance, that:

- Every position from a group of $c \in \hat{\mathbb{A}}$ have the same data value which is later matched by an element of a group of `#`.
- Every position from a group of `N` has the same data value as a position $c \in \hat{\mathbb{A}}$ of the *next* block (and then, it has only *one* possible data value).
- Every position from a group `@` has the same data value as a position $c \in \hat{\mathbb{A}}$ of a block to its right. Note that the data values of two `@` of the same group may correspond to the data values of symbols in different blocks. This is basically the main conceptual difference with the previous proof.

The proof of correctness of the construction is left to the reader.

Item (2)**.** Consider a Minsky Counter Automaton C. We revisit the previous proof. We modify the coding of a run of C by also inserting a group of symbols `@` in each `dec` block. Using F^{-1}, we add formulæ ensuring that *every* block containing a `dec` instruction also contains a group of `@` symbols such that their data values refer to previous blocks containing a matching `inc` instruction. We also make sure that two `@` from different blocks have different data values. All this can be done by taking the F^{-1} counterpart of the formulæ we constructed with F.

We show that the new formula φ_C is satisfied by a data word iff C accept the word. Constructing a data word that satisfies φ_C from a word accepted by C is done as in the proof of Theorem 3.36.

For the other direction one needs to show that from a word that satisfies φ_C, the corresponding word with the obvious run and obvious valuations of the counter is accepting for C. One can verify that the new extra conditions do imply that no null counter is ever decreased and each zero test is correct, based on the fact that for every $\texttt{dec}_i$ there must be at least one $\texttt{inc}_i$ to its left, and for every $\texttt{inc}_i$ there must have a related $\texttt{dec}_i$ before any $\texttt{ifzero}_i$ test. □

Note that in the previous proof we used the fact that $\mathsf{LTL}^{\downarrow}(\mathsf{F})$, although it has only one register, can make (in)equality tests several times throughout a path (as used in the formula of item (d) in the proof), something that $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$ and XPath cannot do.

Two registers. When 2 registers are available the previous result can be adapted to code a (non faulty) Minsky CA with a strategy similar to (Demri and Lazić, 2009, Theorem 5.4). The semantics of $\mathsf{LTL}^{\downarrow}_2(\mathsf{F})$ extend those of $\mathsf{LTL}^{\downarrow}(\mathsf{F})$, by having two registers instead of just one. There are two store operators: $\downarrow_1$ and $\downarrow_2$, one for storing a data value in the first register, and the other in the second register. Similarly there are two test operators $\uparrow_1$ and $\uparrow_2$ for testing whether the current data value is equal to the first register, or to the second register. The semantics are extended in the obvious way. We then have the following statement.

Theorem 3.39. *Satisfiability of* $\mathsf{LTL}^{\downarrow}_2(\mathsf{F})$ *is undecidable over data words.*

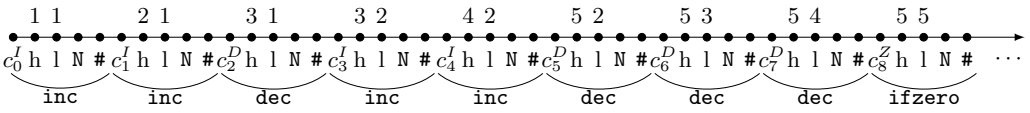

Fig. 3.8: Coding of a 1-counter machine run. Only data values of the `h` and `l` symbols are represented for the sake of clarity. Only one symbol per group is also represented. Elements depicted as c_i^I, c_i^D and c_i^Z correspond, respectively, to increment, decrement and zero testing symbols of $\hat{\Sigma}$.

Proof. We adapt the proof of part 1. of Theorem 3.38 using ideas already present in (Demri and Lazić, 2009; Lisitsa and Potapov, 2005) for coding Minsky CA with only forward temporal operators. Fix a two counter machine $C = \langle \Sigma, Q, q_0, 2, \delta, F \rangle$. The main idea is to modify the coding of Theorem 3.38 by adding in each block a symbol h_1 whose data value is supposed to be the one of the last $\mathtt{inc}_1$ that has previously occurred, a symbol l_1 whose data value is supposed to be the one of the last $\mathtt{inc}_1$ that has not been decreased yet and similarly with h_2 and l_2 for the second counter. A zero test of counter 1 can then only occur in a block where the data values of h_1 and l_1 match. The coding is depicted in Figure 3.8.

Based on these ideas, the formula φ_C that we construct makes sure that:

(i) The structure. We enforce a structure that is a sequence of blocks of the form "$\langle q, w, l, q' \rangle \mathrm{h}_1 \mathrm{l}_1 \mathrm{h}_2 \mathrm{l}_2 \mathtt{N\#}$", with the possibility that there are repeated consecutive symbols. This can be done in the same way we did before.

As for the previous codings, we demand that in each block, for every element in the first group of $\hat{\Sigma}$ symbols there is an element in the last group of symbols `#` with the same data value. This is enforce with a formula of $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ as before.

With the help of the second register we can now enforce that each group of symbols has the same data value, for all symbol s we have the formula:

$$\mathsf{G}\big(\hat{\sigma} \Rightarrow \downarrow_1 \neg \mathsf{F}\big(s \wedge \downarrow_2 \mathsf{F}\,(s \wedge \neg \uparrow_2 \wedge \mathsf{F}(\# \wedge \uparrow_1))\big)\big)$$

(ii) Each $\langle q, w, l, q' \rangle$ occurring in the string is in δ. For the first one $q = q_0$, and for the last one $q' \in F$. For any $\langle q, w, l, q' \rangle$ and $\langle q'', w', l', q''' \rangle$ in consecutive blocks, $q' = q''$. This can be checked just as in the previous codings.

(iii) In the initial block, the data values of h_1 and l_1 are the same and the data values of h_2 and l_2 are the same (we only show the formula for h_1 and l_1):

$$\downarrow_1 \neg \mathsf{F}\big(\mathrm{h}_1 \wedge \downarrow_2 \mathsf{F}(\mathrm{l}_1 \wedge \neg \uparrow_2 \wedge \mathsf{F}(\# \wedge \uparrow_1))\big)$$

(iv) In any block immediately after a `ifzero` instruction, the data values of h_1, l_1, h_2, l_2 are identical to those of the `ifzero` instruction. We only give the formula for $\mathtt{ifzero}_1$ instruction with the h_1 symbol.

$\neg\mathsf{F}\big(\textsc{iz}(1)\wedge\downarrow_1\mathsf{F}\big(\mathrm{h}_1\wedge\downarrow_2\mathsf{F}(\psi\wedge\mathsf{F}(\#\wedge\uparrow_1))\big)\big)$
where $\psi\equiv\mathtt{N}\wedge\downarrow_1\mathsf{F}(\mathrm{h}_1\wedge\neg\uparrow_2\wedge\mathsf{F}(\#\wedge\uparrow_1))$

(v) In any block immediately after an $\mathtt{inc}_1$ instruction, h_1 is not in the same class as any preceding h_1 (stated in the formula below) and l_1, h_2, l_2 are in the same class as in the previous block (this can be stated using a formula similar to the one for for item (iv)).

$\neg\mathsf{F}\big(\mathrm{h}_1\wedge\downarrow_1\mathsf{F}\big(\textsc{inc}(1)\wedge\downarrow_2\wedge\mathsf{F}(\varphi\wedge\mathsf{F}(\#\wedge\uparrow_2))\big)\big)$
where $\varphi\equiv\mathtt{N}\wedge\downarrow_2\mathsf{F}\big(\uparrow_2\wedge\mathsf{F}(\mathrm{h}_1\wedge\uparrow_1\wedge\mathsf{F}(\#\wedge\uparrow_2))\big)$

We have of course corresponding formulæ for blocks with $\mathtt{inc}_2$ instructions.

(vi) In any $\mathtt{dec}_1$ block, h_1 and l_1 have different data values.

$$\neg\mathsf{F}\big(\textsc{dec}(1)\wedge\downarrow_1\mathsf{F}\big(\mathrm{h}_1\wedge\downarrow_2\mathsf{F}(\mathrm{l}_1\wedge\uparrow_2\wedge\mathsf{F}(\#\wedge\uparrow_1))\big)\big)$$

We have a corresponding formula for blocks with a $\mathtt{dec}_2$ instruction.

(vii) In any $\mathtt{ifzero}_1$ block, h_1 and l_1 have the same data value. This can be expressed with a formula as above. We have a corresponding formula for $\mathtt{ifzero}_2$ blocks.

(viii) In any block immediately after a $\mathtt{dec}_1$ block B,

h_1 is in the same class as the previous h_1, and

l_1 is in the same class as the h_1 occurring in a block immediately after the rightmost block (occurring before B) with a h_1 that is in the same class as the l_1 of B.

h_2 and l_2 are in the same class as their corresponding symbol of B.

The first and third conditions can be expressed using formulæ similar to those above. The difficult part is to enforce the second condition. For this we introduce some macros:

$$\begin{aligned}\mathrm{InBlock}_i(\varphi)&\equiv\downarrow_i\mathsf{F}(\varphi\wedge\mathsf{F}(\#\wedge\uparrow_i))\\ \mathrm{NextBlock}_i(\varphi)&\equiv\downarrow_i\mathsf{F}(\mathtt{N}\wedge\mathsf{F}(\#\wedge\uparrow_i)\wedge\\ &\qquad\downarrow_i\mathsf{F}(\hat{\sigma}\wedge\varphi\wedge\mathsf{F}(\#\wedge\uparrow_i)))\\ \mathrm{InNextBlock}_i(\varphi)&\equiv\mathrm{NextBlock}_i(\mathrm{InBlock}_i(\varphi))\end{aligned}$$

The second condition can then be stated:

$$\neg\mathsf{F}\Big(\hat{\sigma}\wedge\downarrow_2\mathsf{F}\Big(\mathrm{h}_1\wedge\downarrow_1\mathsf{F}\big(\mathtt{N}\wedge\mathsf{F}(\#\wedge\uparrow_2)\wedge\downarrow_2\mathsf{F}(\hat{\sigma}\wedge\uparrow_2\wedge\mathsf{F}(\mathrm{h}_1\wedge\neg\uparrow_1\wedge\downarrow_2\varphi))\big)\Big)\Big)$$

where $\varphi\equiv\mathsf{F}\big(\textsc{dec}(i)\wedge\mathrm{InBlock}_2(\mathrm{l}_1\wedge\uparrow_1)\wedge\mathrm{InNextBlock}_1(\mathrm{l}_1\wedge\neg\uparrow_2)\big)$

We have a corresponding formula for blocks following a $\mathtt{dec}_2$ instruction. □

Logic	Complexity	Details
$\mathsf{LTL}^{\downarrow}(\mathsf{F})$	NPR, decidable	Thm. 3.38 & (Demri and Lazić, 2009)
$\mathsf{LTL}^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$	undecidable	Thm. 3.38
$\mathsf{LTL}^{\downarrow}_2(\mathsf{F})$	undecidable	Thm. 3.39
$\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$	NPR, decidable	Thm. 3.36 & (Demri and Lazić, 2009)
$\mathsf{sLTL}^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$	undecidable	Thm. 3.37

Fig. 3.9: Summary of results. NPR stands for a non-primitive recursive lower bound.

By (Demri and Lazić, 2009) it is known that satisfiability of $\mathsf{LTL}^{\downarrow}(\mathsf{X}, \mathsf{F})$ with infinite data words is undecidable. The proof of Theorem 3.38 can be extended to code runs of ICA over infinite data words, which is known to be undecidable, to show that this result already holds in the absence of X.

Theorem 3.40. *On infinite data words, the satisfiability problem of* $\mathsf{LTL}^{\downarrow}(\mathsf{F})$ *is undecidable.*

Proof. Consider an ICA C that runs over infinite words and has a Büchi accepting condition, and let us call it ICA^{ω}. It is known that emptiness of ICA^{ω}over infinite words is undecidable (see Demri and Lazić, 2009, Theorem 2.9). We then modify the proof of Theorem 3.38 in order to code runs of C over infinite data words. The only thing that needs to be changed in the original coding of the proof of Theorem 3.38 is `(end)` in order to reflect the accepting conditions of ICA^{ω}. This can be done by the following formula:

`(end')`

$$\mathsf{G}\left(\mathsf{F_s}\left(\bigvee_{\substack{c\in\hat{\Sigma},\\ \pi_1(c)\in F}} c\right)\right)$$

□

3.7 Discussion

It would be interesting to know whether the strictness of the axis $\mathsf{F_s}$ is necessary to obtain the non primitive recursiveness of $\mathsf{sLTL}^{\downarrow}(\mathsf{F})$. Note that the proof of Theorem 3.36 uses in an essential way the possibility to make (in)equality tests several times throughout a path. This is exactly what cannot be expressed in $\mathsf{sLTL}^{\downarrow}(\mathsf{F})$.

Question 3.41. Is satisfiability of $\mathsf{sLTL}^{\downarrow}(\mathsf{F})$ on data words primitive recursive?

In Figure 3.9 we summarize the main results and some of the consequences we have mentioned.

Part II

TREES

4. TREES WITH DATA

This chapter is introductory to our results on data trees of future chapters.

In this chapter we introduce the basic definitions for data trees, and the query language XPath. Along this thesis we will work on XPath over data trees, although it is originally a language for XML documents. We take this decision because data trees are a simpler formalism to work with. Notably, all the results we will present on XPath over data trees imply similar results over XML documents. How to transfer these results into similar results on XML documents will be included in each individual chapter.

Finally, Section 4.5 contains some lower bounds and undecidability results of XPath fragments, which are a consequence of the lower bound results on $\mathsf{LTL}^{\downarrow}$ obtained in Part I.

4.1 Preliminaries

We formally define what is a *data tree* and an XML *document*. Later in Section 4.2 we define XPath over these models.

4.1.1 Unranked ordered finite trees

We define $\mathit{Trees}(\mathbb{E})$, the set of finite, ordered and unranked trees over an alphabet $\mathbb{E}$. A **position** in the context of a tree is an element of $(\mathbb{N}_+)^*$. The root's position is the empty string and we note it 'ϵ'. The position of any other node in the tree is the concatenation of the position of its parent and the node's index in the ordered list of siblings. Along this work we write '$\cdot$' for the concatenation operator, and we use x, y, z, w, v as variables for positions, while i, j, k, l, m, n as variables for numbers. Thus, for example $x{\cdot}i$ is a position which is not the root, and that has x as parent position, and there are $i-1$ siblings to the left of $x{\cdot}i$.

Formally, we define $\mathsf{POS} \subseteq \wp_{<\infty}((\mathbb{N}_+)^*)$ as the set of sets of finite tree positions, such that: $X \in \mathsf{POS}$ iff (a) $X \subseteq (\mathbb{N}_+)^*, |X| < \infty$; (b) it is prefix-closed; and (c) if $n{\cdot}(i+1) \in X$ for $i \in \mathbb{N}_+$, then $n{\cdot}i \in X$. A tree is a mapping from a set of positions to letters of the alphabet

$$\mathit{Trees}(\mathbb{E}) := \{\mathbf{t} : P \to \mathbb{E} \mid P \in \mathsf{POS}\} \ .$$

Given a tree $\mathbf{t} \in \mathit{Trees}(\mathbb{E})$, $\mathsf{pos}(\mathbf{t})$ denotes the domain of $\mathbf{t}$, which consists of the set of positions of the tree, and $\mathsf{alph}(\mathbf{t}) = \mathbb{E}$ denotes the alphabet of the tree. From now on, we informally refer by 'node' to a position x together with the value $\mathbf{t}(x)$.

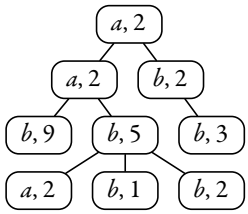

Fig. 4.1: A data tree.

We define the *ancestor* partial order $\preceq$ as the prefix relation $x \preceq x{\cdot}y$ for every $x{\cdot}y$, and the strict version $\prec$ as the strict prefix relation $x \prec x{\cdot}y$ for $|y| > 0$. Given a tree $\mathbf{t}$ and $x \in \mathsf{pos}(\mathbf{t})$, '$\mathbf{t}|_x$' denotes the subtree of $\mathbf{t}$ at position x. That is, $\mathbf{t}|_x : \{y \mid x{\cdot}y \in \mathsf{pos}(\mathbf{t})\} \to \mathsf{alph}(\mathbf{t})$ where $\mathbf{t}|_x(y) = \mathbf{t}(x{\cdot}y)$. In the context of a tree $\mathbf{t}$, a **siblinghood** is a maximal sequence of siblings. That is, a sequence of positions $x{\cdot}1, \ldots, x{\cdot}l \in \mathsf{pos}(\mathbf{t})$ such that $x{\cdot}(l+1) \notin \mathsf{pos}(\mathbf{t})$.

Given two trees $\mathbf{t}_1 \in \mathit{Trees}(\mathbb{E})$, $\mathbf{t}_2 \in \mathit{Trees}(\mathbb{F})$ such that $\mathsf{pos}(\mathbf{t}_1) = \mathsf{pos}(\mathbf{t}_2) = P$, we define $\mathbf{t}_1 \otimes \mathbf{t}_2 : P \to (\mathbb{E}{\times}\mathbb{F})$ as $(\mathbf{t}_1 \otimes \mathbf{t}_2)(x) = (\mathbf{t}_1(x), \mathbf{t}_2(x))$.

The set of **data trees** over a finite alphabet $\mathbb{A}$ and an infinite domain $\mathbb{D}$ is defined as $\mathit{Trees}(\mathbb{A}{\times}\mathbb{D})$. Note that every tree $\mathbf{t} \in \mathit{Trees}(\mathbb{A}{\times}\mathbb{D})$ can be decomposed into two trees $\mathbf{a} \in \mathit{Trees}(\mathbb{A})$ and $\mathbf{d} \in \mathit{Trees}(\mathbb{D})$ such that $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$. Figure 4.1 shows an example of a data tree. We define the **tree type** as a function $\mathsf{type}_{\mathbf{t}} : \mathsf{pos}(\mathbf{t}) \to \{\triangledown, \bar{\triangledown}\} \times \{\triangleright, \bar{\triangleright}\}$ that specifies whether a node has children and/or siblings to the right. That is, $\mathsf{type}_{\mathbf{t}}(x) := (a, b)$ where $a = \triangledown$ iff $x{\cdot}1 \in \mathsf{pos}(\mathbf{t})$, and where $b = \triangleright$ iff $x = x'{\cdot}i$ and $x'{\cdot}(i+1) \in \mathsf{pos}(\mathbf{t})$. The notation for the set of data values used in a data tree is

$$\mathit{data}(\mathbf{a} \otimes \mathbf{d}) := \{\mathbf{d}(x) \mid x \in \mathsf{pos}(\mathbf{d})\} \ .$$

With an abuse of notation we write $\mathit{data}(X)$ to denote all the elements of $\mathbb{D}$ contained in X, for whatever object X may be.

4.1.2 XML

An XML document can be seen as a tree whose every element has one label from a finite alphabet $\mathbb{A}$ and a set of data values indexed by some finite alphabet $\mathbb{B}$. This is called the set of *attributes* of the node. In other words, we consider that an XML document is an element of

$$\mathit{Trees}(\mathbb{A} \times \wp_{<\infty}(\mathbb{B} \times \mathbb{D}))$$

for $\mathbb{A}$ and $\mathbb{B}$ finite alphabets, and $\mathbb{D}$ some infinite domain.

Note that this model is more general than the class of data trees, since a data tree can be seen as an XML with a singleton set of attributes at each node. But at the same time, an XML document can also be coded as a data tree from $\mathit{Trees}((\mathbb{A} \cup \mathbb{B}) \times \mathbb{D})$, by coding the attributes of a node x labeled by $\mathbb{A}$, as leaves $x{\cdot}i$ labeled by $\mathbb{B}$. We will come back to this discussion in later chapters, verifying

in every case that all the results obtained on data trees can be transferred to the class of XML documents.

4.1.3 Types and dependencies

In the context of XML research, it is sometimes important to restrict the static analysis tasks to a subclass of 'valid' (in the context of the problem) XML documents. We next define some concepts in terms of data trees, since this is the model we will mostly work on along this thesis, but all these notions apply also to XML documents.

The subclasses of trees are generically called *document types*, and they define a subclass of data trees of particular interest. For example, in the context of a data tree modeling a library, we would be interested in documents where all nodes labeled book have exactly one child labeled ISBN and at least one child labeled author.

Given a document type T and a query Q defined in some language, we want to answer the question "Is there a document satisfying T such that Q answers a nonempty result?". Let us briefly examine some of the possibilities to define a type.

DTDs. A widely spread language for defining document types is the DTD (for Document Type Definition). DTDs allow to define a class of data trees, by taking into account the finite labeling of the tree. For the purpose of this thesis, it suffices to know that any DTD defines a regular language, although the converse is not true.

Functional dependencies. In addition to restrictions on the finite labeling of the tree, we can also demand restrictions on the data values.

- Key: In the example above, we could also ask that all the nodes labeled ISBN have different data value. The key constraint $key(a)$ can be stated as follows: "Every pair of different elements labeled by a contain different data values". Here, a is an attribute value.

- Restricted key: It is also sensible to ask that inside the subtree rooted at a book-node, all the authors have different data value, as it would make no sense to have two authors with the same name in the same book. The restricted key constraint $key(a, b)$ states "Every subtree with a root labeled by b verifies $key(a)$".

- Inclusion dependency: In the example just seen, suppose we have somewhere in the tree structure a list of recommended authors to show in the web page of the library. We could want as a rule, that there is at least one book of each of the featured authors. The inclusion dependency constraint $dep(a, b)$

states "For every node labeled a there exists a node labeled b with the same data value".

4.2 XPath on data trees

In this section we define XPath over data trees. Although all our results will be over data trees, in the next section we also define XPath over XML documents. This is done because in future chapters we will explain how all our results on XPath on data trees can be translated into similar results over XML documents.

4.2.1 Introduction

XPath is arguably the most widely used XML query language. It is implemented in XSLT and XQuery and it is used as a constituent part of several specification and update languages. XPath is fundamentally a general purpose language for addressing, searching, and matching pieces of an XML document. It is an open standard and constitutes a World Wide Web Consortium (W3C) Recommendation (Clark and DeRose, 1999), implemented in most languages and XML packages.

An important static analysis problem of a query language is that of optimization, which can reduce to the problem of query containment and query equivalence. In logics closed under boolean operators, these problems reduce to *satisfiability* checking: does a given query express some property? That is, is there a document where this query has a non-empty result? By answering this question we can decide at compile time whether the query contains a contradiction and thus the computation of the query on the document can be avoided, or if one query can be safely replaced by another one. Moreover, this problem becomes crucial for many applications on security, type checking transformations, and consistency of XML specifications.

Core-XPath (introduced by Gottlob et al. (2005)) is the fragment of XPath that captures all the navigational behavior of XPath. It has been well studied and its satisfiability problem is known to be decidable even in the presence of DTDs. The extension of this language with the possibility to make equality and inequality tests between attributes of elements in the XML document is named Core-Data-XPath in (Bojańczyk et al., 2009). The satisfiability problem for this logic is undecidable, as shown by Geerts and Fan (2005). It is then reasonable to study the interaction between different navigational fragments of XPath with equality tests to be able to find decidable and computationally well-behaved fragments.

4.2.2 Definition

We work with a simplification of XPath, stripped of its syntactic sugar. We consider fragments of XPath that correspond to the navigational part of XPath 1.0 with data equality and inequality. Let us give the formal definition of this logic. XPath is a two-sorted language, with *path* expressions (that we write α, β, γ) and *node*

expressions (φ, ψ, η). The fragment $\mathsf{XPath}(\mathcal{O}, =)$, with

$$\mathcal{O} \subseteq \{\downarrow, \downarrow_*, \downarrow_+, \uparrow, \uparrow^*, \uparrow^+, \rightarrow, \rightarrow^*, \rightarrow^+, \leftarrow, {}^*\leftarrow, {}^+\leftarrow\}$$

is defined by mutual recursion as follows:

$$\begin{array}{lll} \alpha, \beta \ ::= \ o \mid \alpha[\varphi] \mid [\varphi]\alpha \mid \alpha\beta \mid \alpha \cup \beta & & o \in \mathcal{O} \cup \{\varepsilon\} \ , \\ \varphi, \psi \ ::= \ a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle\alpha\rangle \mid \langle\alpha = \beta\rangle \mid \langle\alpha \neq \beta\rangle & & a \in \mathbb{A} \ . \end{array}$$

A *formula* of $\mathsf{XPath}(\mathcal{O}, =)$ is either a node expression or a path expression of the logic. $\mathsf{XPath}(\mathcal{O})$ is the fragment $\mathsf{XPath}(\mathcal{O}, =)$ without the node expressions of the form $\langle\alpha = \beta\rangle$ or $\langle\alpha \neq \beta\rangle$.

There have been efforts to extend this navigational core of XPath in order to have the full expressivity of MSO—for example by adding a least fix-point operator (*cf.* ten Cate, 2006, § 4.2)—but these logics generally lack clarity and simplicity. However, a form of recursion can be added by means of the Kleene star, which allows to take the transitive closure of any path expression. Although in general this is not enough to already have MSO—as shown by ten Cate and Segoufin (2008)—, it does give an intuitive language with counting ability. By $\mathsf{regXPath}(\mathcal{O}, =)$ we refer to the language where path expressions are extended

$$\alpha, \beta \ ::= \ o \mid \alpha[\varphi] \mid [\varphi]\alpha \mid \alpha\beta \mid \alpha \cup \beta \mid \alpha^* \quad o \in \mathcal{O}$$

by allowing the Kleene star on *any* path expression. Also, by $\mathsf{XPath}^{\varepsilon}(\mathcal{O}, =)$ (resp. $\mathsf{regXPath}^{\varepsilon}(\mathcal{O}, =)$) we denote the fragment of $\mathsf{XPath}(\mathcal{O}, =)$ (resp. of $\mathsf{regXPath}(\mathcal{O}, =)$) where node expressions are defined by

$$\varphi, \psi \ ::= \ a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle\alpha\rangle \mid \langle\varepsilon = \alpha\rangle \mid \langle\varepsilon \neq \alpha\rangle \qquad a \in \mathbb{A} \ .$$

That is, the data tests are performed between the data value of the *current* node and some other one accessed by α.

We formally define the semantics of XPath in Figure 4.2. As an example, if $\mathbf{t}$ is the data tree defined by Figure 4.1 on page 58,

$$[\![\langle\downarrow_*[\, b \wedge \langle\downarrow[b] \neq \downarrow[b]\rangle\,]\rangle]\!]^{\mathbf{t}} = \{\varepsilon, 1, 12\}.$$

Henceforward, we write $\mathbf{t} \models \varphi$ to denote $[\![\varphi]\!]^{\mathbf{t}} \neq \emptyset$. In this case we say that $\mathbf{t}$ 'satisfies' φ. We say that two formulæ φ, ψ of XPath are **equivalent** iff $[\![\varphi]\!]^{\mathbf{t}} = [\![\psi]\!]^{\mathbf{t}}$ for all data tree $\mathbf{t}$.

4.2.3 Fragments

We define several fragments of XPath. Each fragment is defined by the set of axes that the path expressions can use.

$$\begin{aligned}
[\![\rightarrow]\!]^{\mathbf{t}} &= \{(x{\cdot}i, x{\cdot}(i+1)) \mid x{\cdot}(i+1) \in \mathsf{pos}(\mathbf{t})\} & [\![\leftarrow]\!]^{\mathbf{t}} &= \{(x{\cdot}(i+1), x{\cdot}i) \mid x{\cdot}(i+1) \in \mathsf{pos}(\mathbf{t})\}\\
[\![\downarrow]\!]^{\mathbf{t}} &= \{(x, x{\cdot}i) \mid x{\cdot}i \in \mathsf{pos}(\mathbf{t})\} & [\![\uparrow]\!]^{\mathbf{t}} &= \{(x{\cdot}i, x) \mid x{\cdot}i \in \mathsf{pos}(\mathbf{t})\}\\
[\![\alpha^+]\!]^{\mathbf{t}} &= \text{the transitive closure of } [\![\alpha]\!]^{\mathbf{t}} & [\![\alpha^*]\!]^{\mathbf{t}} &= \text{the reflexive transitive closure of } [\![\alpha]\!]^{\mathbf{t}}\\
[\![\varepsilon]\!]^{\mathbf{t}} &= \{(x, x) \mid x \in \mathsf{pos}(\mathbf{t})\} & [\![\alpha\beta]\!]^{\mathbf{t}} &= \{(x, z) \mid \text{ there exists } y \text{ such that}\\
& & & \qquad (x, y) \in [\![\alpha]\!]^{\mathbf{t}}, (y, z) \in [\![\beta]\!]^{\mathbf{t}}\}\\
[\![\alpha \cup \beta]\!]^{\mathbf{t}} &= [\![\alpha]\!]^{\mathbf{t}} \cup [\![\beta]\!]^{\mathbf{t}} & & \\
[\![\alpha[\varphi]]\!]^{\mathbf{t}} &= \{(x, y) \in [\![\alpha]\!]^{\mathbf{t}} \mid y \in [\![\varphi]\!]^{\mathbf{t}}\} & [\![[\varphi]\alpha]\!]^{\mathbf{t}} &= \{(x, y) \in [\![\alpha]\!]^{\mathbf{t}} \mid x \in [\![\varphi]\!]^{\mathbf{t}}\}\\
[\![a]\!]^{\mathbf{t}} &= \{x \in \mathsf{pos}(\mathbf{t}) \mid \mathbf{a}(x) = a\} & [\![\langle\alpha\rangle]\!]^{\mathbf{t}} &= \{x \in \mathsf{pos}(\mathbf{t}) \mid \exists y.(x, y) \in [\![\alpha]\!]^{\mathbf{t}}\}\\
[\![\neg\varphi]\!]^{\mathbf{t}} &= \mathsf{pos}(\mathbf{t}) \setminus [\![\varphi]\!]^{\mathbf{t}} & [\![\varphi \wedge \psi]\!]^{\mathbf{t}} &= [\![\varphi]\!]^{\mathbf{t}} \cap [\![\psi]\!]^{\mathbf{t}}\\
[\![\langle\alpha{=}\beta\rangle]\!]^{\mathbf{t}} &= \{x \in \mathsf{pos}(\mathbf{t}) \mid \exists y,z\ (x, y) \in [\![\alpha]\!]^{\mathbf{t}}, & [\![\langle\alpha{\neq}\beta\rangle]\!]^{\mathbf{t}} &= \{x \in \mathsf{pos}(\mathbf{t}) \mid \exists y,z\ (x, y) \in [\![\alpha]\!]^{\mathbf{t}},\\
& \qquad (x, z) \in [\![\beta]\!]^{\mathbf{t}}, \mathbf{d}(y) = \mathbf{d}(z)\} & & \qquad (x, z) \in [\![\beta]\!]^{\mathbf{t}}, \mathbf{d}(y) \neq \mathbf{d}(z)\}
\end{aligned}$$

Fig. 4.2: Semantics of XPath for a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$.

Downward fragment We call the *downward* fragment of XPath to $\mathsf{XPath}(\downarrow_*, \downarrow, =)$. In fact, all our results will apply also to $\mathsf{regXPath}(\downarrow_*, \downarrow, =)$. In terms of expressivity, we see that $\mathsf{XPath}(\downarrow_*, =) \subsetneq \mathsf{XPath}(\downarrow_*, \downarrow, =) \subsetneq \mathsf{regXPath}(\downarrow, =) = \mathsf{regXPath}(\downarrow_*, \downarrow, =)$. In Chapter 5 we will show the decidability of the satisfiability problem for this logic and its fragments.

Forward fragment The *forward* fragment is an extension of the downward fragments with some horizontal navigation. In our notation, forward XPath is then $\mathsf{XPath}(\downarrow_*, \downarrow, \rightarrow^*, \rightarrow, =)$. We write $\mathfrak{F}$ for the forward operators $\{\downarrow_*, \downarrow, \rightarrow^*, \rightarrow\}$ and we hence refer to this fragment by $\mathsf{XPath}(\mathfrak{F}, =)$. In Chapter 6 we will prove decidability of the satisfiability problem for $\mathsf{XPath}(\mathfrak{F}, =)$ and $\mathsf{regXPath}(\mathfrak{F}, =)$.

Vertical fragment The *vertical* fragment $\mathsf{XPath}(\mathfrak{V}, =)$ is another extension of the downward fragment with upward axes, that is $\mathfrak{V} = \{\downarrow_*, \downarrow, \uparrow^*, \uparrow\}$. This fragment will be shown decidable in Chapter 7.

Horizontal fragment Later on in Section 4.5 we will show some lower bounds for several fragments of XPath running on data words. These lower bounds will then be transfered to fragments that contains horizontal or upwards axes, or that can force a model to be linear (*e.g.* $\mathsf{XPath}(\downarrow_*, \downarrow, \rightarrow, =)$).

4.2.4 Decision problems

We state the problems we will address, given a fragment $\mathscr{P}$ of XPath.

EQ-$\mathscr{P}$	Equivalence problem for $\mathscr{P}$
INPUT:	$\varphi, \psi \in \mathscr{P}$.
OUTPUT:	Is $[\![\varphi]\!]^{\mathbf{t}} = [\![\psi]\!]^{\mathbf{t}}$ for every data tree $\mathbf{t}$?

INC-$\mathscr{P}$	Inclusion problem for $\mathscr{P}$
INPUT:	$\varphi, \psi \in \mathscr{P}$.
OUTPUT:	Is $[\![\varphi]\!]^{\mathbf{t}} \subseteq [\![\psi]\!]^{\mathbf{t}}$ for every data tree $\mathbf{t}$?

SAT-$\mathscr{P}$	Satisfiability problem for $\mathscr{P}$
INPUT:	$\varphi \in \mathscr{P}$.
OUTPUT:	Is there a tree $\mathbf{t}$ such that $\mathbf{t} \models \varphi$?

Remark 4.1. All the logics we deal with are closed under boolean operations. Hence, the inclusion problem reduces to that of satisfiability. The inclusion problem for φ, ψ yields a 'yes' iff the satisfiability problem for $\varphi \wedge \neg\psi$ yields 'no'. Similarly, the equivalence problem reduces to that of satisfiability. Hence, since all the logics we deal with are closed under boolean operations, we will only focus in the satisfiability problem.

We are also interested in the following problem for $\mathscr{L}$ a class of tree regular languages and $\mathscr{P}$ a fragment of XPath.

SAT-$\mathscr{P} + \mathscr{L}$	Satisfiability problem for $\mathscr{P}$ under $\mathscr{L}$
INPUT:	$\varphi \in \mathscr{P}$ and $\mathcal{L} \in \mathscr{L}$.
OUTPUT:	Is there a tree $\mathbf{t} \in \mathcal{L}$ such that $\mathbf{t} \models \varphi$?

For the fragments treated in this thesis, these problems are equivalent to testing if there is a tree where the formula φ is satisfied at the *root*, since otherwise we can test $\downarrow_*[\varphi]$. Moreover, we can restrict ourselves to the case where φ is a *node expression*, as $[\![\alpha]\!] \neq \emptyset$ iff $[\![\langle\alpha\rangle]\!] \neq \emptyset$. We remind the reader that although we state the problem in terms of data trees, all our results hold on the class of all XML documents. Indeed, this is a consequence of considering an XML document as a data tree where the attributes are at leaf positions.

4.3 XPath on XML documents

As outlined before, XML documents may have multiple attributes with data values on each element, while data trees can only have one. Here we will show that every result we have stated in terms of data trees, also holds on the class of XML documents. Let us consider that the finite set of symbols is partitioned between the names for attributes and the symbols of the XML elements, $\mathbb{A}_{attr} \cup \mathbb{A}_{elem}$.

An XML document is hence a tree $\mathbf{a} \otimes \mathbf{d}$ where every position carries one label from $\mathbb{A}_{elem}$ and many data values indexed by $\mathbb{A}_{attr}$ that we call 'attributes', $\mathbf{a} : P \to \mathbb{A}_{elem}$, $\mathbf{d} : P \to \wp_{<\infty}(\mathbb{A}_{attr} \times \mathbb{D})$ for some $P \in \mathsf{POS}$.

We define XPath on XML documents as the extension where different attributes may be compared for (in)equality. Node expressions are defined

$$\varphi ::= a \mid \neg\varphi \mid \varphi \wedge \psi \mid \langle\alpha\rangle \mid \langle\alpha@attr1 \odot \beta@attr2\rangle$$

where $\odot \in \{=, \neq\}, a \in \mathbb{A}_{elem}$ and $attr1, attr2 \in \mathbb{A}_{attr}$. Let us call this logic attrXPath.

Although all the results of future chapters are stated in terms of data trees, we will show, for each of the fragments treated, that the same upper/lower bounds also hold for the corresponding fragments of attrXPath on XML documents.

4.4 Related work

4.4.1 Automata

Data automata Björklund and Bojańczyk (2007) extend the model of *class memory automata* (Björklund and Schwentick, 2010) over data words (equivalent to the data automata of Bojańczyk et al. (2006) mentioned in § 3.2) to work on bounded-depth data trees. That is, over unranked data trees whose every leaf is at a distance bounded by a constant. They show decidability of the emptiness problem through a reduction to priority multicounter automata, shown to be decidable by Reinhardt (2005).

Bojańczyk and Lasota (2010) introduce a powerful (and undecidable) automata model called *class automata* on data trees that captures data automata, XPath with the full set of axes, ATRA, and some other models. In fact, data automata and ATRA can be seen as simple restrictions on the semantics of class automata. Bojańczyk and Lasota also propose considering restrictions on the data tree model to obtain decidability results on class automata. Indeed this is an interesting line for future work.

Register Automata Extensions of (non)deterministic register automata on data trees (either bottom-up or top-down) have been studied by Kaminski and Tan (2008). Like in the data word setting, these automata can basically code existential data properties, like "there are two a's with the same data value" or "there exists a data value such that it is contained in all the a's" but not universal, like "all a's have different data value", or any of the properties of Example 1.2.

An alternating top-down extension but with only one register was introduced by Jurdziński and Lazić (2008), named ATRA. This model is decidable as soon as there is only one register, extending then the results on the similar automaton on words from Demri and Lazić (2009). It has a *forward* navigation over unranked trees: it walks the tree by moving either to the leftmost child or to the next sibling to the right. In future chapters we will introduce two automata that are also alternating and have one register. Since this class of automata is the most relevant to our work, we briefly discuss how this work relates to ours.

Firstly, the DD automaton of Chapter 5 is in some sense weaker than ATRA[1], but allows to have a decision procedure of 2ExpTime (or ExpTime for a restricted yet powerful fragment) instead of the non-primitive recursive lower bound of ATRA. On the other hand, the class of automata ATRA(guess, spread) of Chapter 6 is also top-down as the ATRA class, but contains some extra features that allow to make

[1] Strictly speaking, it is incomparable in expressive power, but a decidable extension of ARA can capture its behavior. Part of the necessary extension will be discussed in Section 6.4.5.

richer tests. Finally, while ATRA(guess, spread) can be seen as an extension of ATRA, the automata class BUDA introduced in Chapter 7 is certainly incomparable in expressive power with respect to all the previously discussed classes. Indeed the BUDA class is bottom-up instead of top-down, but still allows to test data properties of the subtrees achieving—in some sense—a two-way behavior. More precise comparisons with this work will be given in the chapters that follow.

4.4.2 Logics

XPath

Benedikt, Fan, and Geerts (2008) studied the satisfiability problem for many XPath fragments containing downwards and upwards axes, but no horizontal axes. Chapters 5 and 7 also contribute to this study, since we show decidability of XPath fragments with no horizontal axes. In our work we investigate two decidable fragments that were not studied by Benedikt et al. (2008): the downward fragment and the vertical fragment. Further, in Chapter 5 we improve some of the bounds on other downward fragments (with and without data values).

In (Geerts and Fan, 2005), several XPath fragments with horizontal axes are treated. However, this work does not have a direct relation to the forward fragment treated in Chapter 6. The only fragment with data tests and negation studied by Geerts and Fan (that is shown to be undecidable) is incomparable with the forward fragment.

Bojańczyk et al. (2009) shows the decidability of $\mathsf{XPath}^{\varepsilon}(\uparrow, \downarrow, \leftarrow, \rightarrow, =)$ with sibling and upward axes but restricted to local elements accessible by a 'one step' relation, and to data formulæ of the kind $\langle \varepsilon = \alpha \rangle$ (or $\neq$). However, most of the fragments we treat here disallow upward and sibling axes but allow the descendant $\downarrow_*$ axis and arbitrary $\langle \alpha = \alpha' \rangle$ data test expressions.

Finally, Jurdziński and Lazić (2008) showed that $\mathsf{XPath}^{\varepsilon}(\mathfrak{F}, =)$ is decidable, by translating formulæ of this fragment into ATRA automata. In Chapter 6 we will show that in fact the full forward fragment $\mathsf{XPath}(\mathfrak{F}, =)$ is decidable.

Two-variable logics

First-order logic with two variables and data equality tests is investigated by Bojańczyk et al. (2009). By $\mathsf{FO}^2(<, +1, \sim)$ we refer to first order logic with two variables. The symbol $<$ refers to two binary predicates: one for comparing the descendant/ancestor relationship of two nodes and one for the preceding/following relationship of two siblings. Similarly, $+1$ also refers to two binary predicates: one for comparing the parent/child relationship of two nodes and one for comparing the next/previous sibling relationship of two siblings. Finally, $\sim$ is for testing for data equality. Although in the absence of data values $\mathsf{FO}^2(<, +1)$ is expressive-equivalent to Core-XPath (*cf.* Marx, 2005), $\mathsf{FO}^2(<, +1, \sim)$ with data equality tests becomes incomparable with respect to all the data aware fragments treated here. Although the decidability of $\mathsf{FO}^2(<, +1, \sim)$ is hitherto unknown, it is shown that

$\mathsf{FO}^2(+1, \sim)$ is decidable, its complexity lying somewhere between NExpTime and 3NExpTime.

4.4.3 Other formalisms

Patterns

David (2008) studied tree patterns with data test. A data tree pattern in essentially an unranked tree with two kind of edges: descendant or child. It also allows to test for labels of nodes and, more importantly, contains constraints of equality and inequality between any pair of nodes of the pattern. When such a pattern can (or cannot) be injected into a data tree while preserving these relations we say that it is satisfied (or not satisfied) by the data tree. The cited work studies the satisfiability of boolean combinations of such patterns. This formalism, although incomparable with the aforementioned logics and automata, provides another means to test for properties. This problem can also be seen from a perspective of conjunctive conjunctive queries, and as such was treated by Björklund, Martens, and Schwentick (2008). Although in general the satisfiability problem for boolean combinations of tree patterns is undecidable (proven independently by Björklund et al. and David), some decidable fragments are identified in the cited works.

Tests for isomorphism

Consider automata that can perform isomorphism tests over an infinite set of structures. This is some sort of infinite alphabet with equality tests. Indeed, data values can be coded for example as subtrees with a distinct label. We mention some models of automata on trees with this special flavor of tests.

There have been studies on tree automata, where transitions can perform isomorphism tests. Either in the ranked case (Bogaert and Tison, 1992), unordered unranked case (Lugiez, 2003), or ordered unranked case (Löding and Wong, 2009). In the data tree setting, these results can be seen as automata that can make rich tests on the data values of the siblings of a tree, but that cannot relate the data values between parent and child.

On the other hand, isomorphism tests can be performed in a *global* manner. That is, we ask whether there exists an execution of an automaton over a tree, such that every pair of nodes labeled with a states q and p by the run verify that their induced subtrees are isomorphic (or non-isomorphic). Several variants of automata based on these kinds of tests are shown to be decidable in (Filiot et al., 2007, 2008; Barguñó et al., 2010). By coding data values as subtrees, it is possible to obtain some decidability results in the data trees setup. However, notice that the tests performed correspond to a quantification "exists... for all..." (indeed these automata are not closed under complementation). In some sense, it approximately corresponds to having a restricted $\mathsf{EMSO}^2(+1, \sim)$ formula which is

a conjunction of formulæ of the kind

$$\exists X_1, \ldots, X_n.(\psi \wedge \forall x, y.\xi(x, y) \rightarrow \eta(x, y)),$$

where ψ does not contain $\sim$; ξ is a boolean combination of test for labels; and η is either $x \sim y$ or $\neg(x \sim y)$.

4.5 Lower bounds of XPath

We can transfer lower-bound results from Section 3.6.1.[2] In order to do this, we have to show a key property: simple formulæ can be translated to XPath and back. The proof of this result is straightforward by induction on the formula.

Proposition 4.2. *Over data words,* $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s})$ *and* $\mathsf{XPath}(\rightarrow^+, =)$ *have the same expressive power. The same holds for* $\mathsf{sLTL}^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$ *and* $\mathsf{XPath}(^+\leftarrow, \rightarrow^+, =)$*. Moreover, in both cases, the transformation from* $\mathsf{sLTL}^{\downarrow}$ *to XPath takes polynomial time while it takes exponential time in the other direction.*

Proof sketch. The translation from $\mathsf{sLTL}^{\downarrow}$ to XPath is straightforward by induction. One then naturally obtains equality tests of the form $\langle \varepsilon = \alpha \rangle$ or $\langle \varepsilon \neq \alpha \rangle$ depending on whether the unique possible negation is present or not.

The other translation is also by induction. The non trivial part concerns the translation of node expressions of the form $\langle \alpha = \beta \rangle$. Over data words, one of the path expressions α or β must end first, say α. In this case the node expression can be decomposed into a disjunction of (exponentially many) expressions that first start with a path expression that *merge* α with the part β_1 of β that is common to with α, followed by a test of the form $\langle \varepsilon = \beta_2 \rangle$, where $\beta = \beta_1\beta_2$. Tests of the form $\langle \varepsilon = \beta_2 \rangle$ are the immediate to translate into $\mathsf{sLTL}^{\downarrow}$. □

The restriction on negations in the definition of $\mathsf{sLTL}^{\downarrow}$ corresponds to the fact that XPath path expressions are always positive: any path α is essentially a nesting of operators F with intermediate tests. We remark that there is a big difference between $\mathsf{XPath}(\rightarrow^+, =)$ over data words and $\mathsf{XPath}(\downarrow_+, =)$ over data trees. Indeed $\mathsf{XPath}(\downarrow_+, =)$ is closed under bisimulation and hence it cannot assume that the tree is a vertical path. As the string structure was essential in the proof of Theorem 3.36, the non primitive recursiveness of $\mathsf{XPath}(\rightarrow^+, =)$ over data words does not lift to $\mathsf{XPath}(\downarrow_+, =)$ over data trees. Actually, satisfiability of $\mathsf{XPath}(\downarrow_+, =)$ is ExpTime-complete as we will see in Chapter 5. However, if one considers the logic $\mathsf{XPath}(\downarrow_+, \rightarrow, =)$ then the axis $\rightarrow$ can be used to enforce a vertical path $\neg\langle\rightarrow\rangle \wedge \neg\langle\downarrow_+\rightarrow\rangle$ and therefore it follows from Theorem 3.36 and Proposition 4.2 that:

Corollary 4.3. *Satisfiability of* $\mathsf{XPath}(\downarrow_+, \rightarrow, =)$ *on data trees is at least non primitive recursive.*

[2] These results are included in (Figueira and Segoufin, 2009).

Corollary 4.4. *Satisfiability of* $\mathsf{XPath}(\downarrow_+, =)$ *in the presence of DTDs is at least non primitive recursive.*

In Chapter 6 we will show that these two fragments are indeed decidable.

Similarly, in $\mathsf{XPath}(\downarrow_+, \uparrow^+, =)$ one can simulate a string by going down to a leaf using $\downarrow_+$ and then use the path from that leaf to the root as a string using $\uparrow^+$.

Corollary 4.5. *Satisfiability of* $\mathsf{XPath}(\uparrow^+, =)$ *on data trees is decidable and non-primitive recursive.*

Proof. The decidability from the fact that any $\mathsf{XPath}(\uparrow^+)$ formula can be easily translated into an ARA automaton, since the formula essentially reads a data *word*. □

It would be interesting to know whether the strictness of the axis $\rightarrow^+$ is necessary in the above two results. This boils down to know whether $\mathsf{sLTL}^{\downarrow}(\mathsf{F})$ is already not primitive recursive over data words. Note that the proof of Theorem 3.38 uses in an essential way the possibility to make (in)equality tests several times throughout a path. This is exactly what cannot be expressed in $\mathsf{sLTL}^{\downarrow}(\mathsf{F})$.

Question 4.6. Is satisfiability of $\mathsf{sLTL}^{\downarrow}(\mathsf{F})$ primitive recursive over data words?

We conclude with some simple consequences of Theorem 3.38 and Proposition 4.2:

Corollary 4.7. *Satisfiability of* $\mathsf{XPath}({}^+\!\leftarrow, \rightarrow^+, =)$ *and of* $\mathsf{XPath}(\downarrow_+, \uparrow^+, \rightarrow, =)$ *over data trees is undecidable.*

Corollary 4.8. *Satisfiability of* $\mathsf{XPath}(\rightarrow^+, \downarrow, \uparrow, =)$ *is undecidable.*

Proof. This is similar to the proof of Theorem 3.37 with a slight difference. Consider that the coding of the run of the counter machine is done at the first level of the tree (*i.e.*, at distance 1 from the root). Then, the property to ensure that every decrement has a corresponding increment is now:

$$\bigwedge_i \neg\langle\downarrow[\mathrm{DEC}(i) \wedge \neg\langle\varepsilon = \uparrow\downarrow[@]\rangle]\rangle \ .$$ □

4.5.1 Summary of results

In the table below we summarize the results and some of the consequences we have mentioned.

Logic	Complexity	Details
$\mathsf{XPath}(\downarrow_+, \rightarrow, =)$	non primitive recursive (decidability shown in Chapter 6)	Corollary 4.3
$\mathsf{XPath}(\downarrow_+, \uparrow^+, =)$	non primitive recursive (decidability shown in Chapter 7)	Corollary 4.5
$\mathsf{XPath}(\downarrow_+, \uparrow^+, \rightarrow, =)$	undecidable	Corollary 4.7
$\mathsf{XPath}(\rightarrow^+, \downarrow, \uparrow, =)$	undecidable	Corollary 4.8

5. DOWNWARD NAVIGATION

We introduce a decidable automaton model for data trees and we investigate the satisfiability problem for downward-XPath. We prove that this problem is decidable, precisely ExpTime-complete. These bounds also hold when path expressions allow closure under the Kleene star operator. To obtain these results, we introduce a Downward Data automata model (DD automata) over trees with data, which has a decidable emptiness problem. Satisfiability of downward-XPath can be reduced to the emptiness problem of DD automata and hence its decidability follows. Although downward-XPath does not include any horizontal axis, DD automata are more expressive and can perform certain (restricted) horizontal tests. Thus, we show that the satisfiability remains in ExpTime even in the presence of the regular constraints expressible by DD automata. However, the same problem in the presence of any regular constraint is known to have a non-primitive recursive complexity (Corollary 4.4). Finally, we give the exact complexity of the satisfiability problem for several fragments of downward-XPath.

5.1 Introduction

One of the main contributions of this chapter is that the satisfiability problem for $\mathsf{XPath}(\downarrow_*, \downarrow, =)$ is decidable. That is, the fragment with equality and inequality tests of attributes' values, the $\downarrow_*$ axis that can access descendant nodes at any depth and the $\downarrow$ axis to access child elements. Actually, we prove a stronger result, showing the decidability of the satisfiability of $\mathsf{regXPath}(\downarrow, =)$, which is the extension with the Kleene star operator. We nail down the precise complexity showing an ExpTime decision procedure, as $\mathsf{XPath}(\downarrow, \downarrow_*)$ is already ExpTime-hard as shown by Benedikt, Fan, and Geerts (2008). In order to do this, we introduce the class of Downward Data automata (DD automata). We show that any $\mathsf{regXPath}(\downarrow, =)$ formula can be effectively translated to an equivalent DD automata. This automata model has a 2ExpTime emptiness problem, but can be shown to be decidable in ExpTime when restricted to the sub-class of automata needed to capture $\mathsf{regXPath}(\downarrow, =)$. In this way we obtain an ExpTime procedure for the satisfiability of $\mathsf{regXPath}(\downarrow, =)$.

In fact, DD automata are more expressive than $\mathsf{regXPath}(\downarrow, =)$. For example, although $\mathsf{regXPath}(\downarrow, =)$ does not include any horizontal axis, DD automata can test for certain horizontal properties. This model can express, for instance, that the sequence of children of the root is recognized by the regular expression $(a\,b\,c)^*$. It will then follow that the satisfiability problem for $\mathsf{regXPath}(\downarrow, =)$ under the regular

constraints that can be expressed by DD automata remains decidable in ExpTime. These regular constraints are a especially well behaved class of regular properties, since the satisfiability problem for $\mathsf{regXPath}(\downarrow, =)$ under any regular language is known to have non-primitive recursive complexity (Corollary 4.4), even when only transitive axes are allowed.

On the other hand, we prove that the fragment $\mathsf{XPath}(\downarrow_*, =)$ without the $\downarrow$ axis is ExpTime-hard, even for a restricted fragment of $\mathsf{XPath}(\downarrow_*, =)$ without unions of path expressions. This reduction can only be done by using data equality tests, as the corresponding fragment $\mathsf{XPath}(\downarrow_*)$ without unions is shown to be PSpace-complete. We thus prove that the satisfiability problem for $\mathsf{XPath}(\downarrow_*, =)$, $\mathsf{XPath}(\downarrow_*, \downarrow, =)$ and $\mathsf{regXPath}(\downarrow, =)$ are all ExpTime-complete. Additionally, we present a natural fragment of $\mathsf{XPath}(\downarrow_*, =)$ that is PSpace-complete. We complete the picture showing that satisfiability for $\mathsf{XPath}(\downarrow, =)$ is also PSpace-complete. Altogether, we establish the precise complexity for all downward fragments of XPath with and without data tests (*cf.* Fig. 5.14 on page 120).

5.1.1 Related work

The main results of this chapter first appeared in the conference paper (Figueira, 2009). The cited work does not contain a full proof of this result, but only the main ideas due to a space limitation. Here we give a detailed proof by a reduction to a powerful class of automata, and to extend some results. Although the main XPath results of (Figueira, 2009) are the same as of the present work, the underlying automata model is completely different. Here we adopt a different strategy to show the decidability. There are basically two reasons to do this. Firstly, the automaton introduced here is simpler than the one of (Figueira, 2009): it does not require a nested definition between two different kind of automata as the one introduced in (Figueira, 2009). And secondly, it is more general: it can express data properties that cannot be expressed in the model of (Figueira, 2009), and it can test the data tree to have some (weak) regular properties on the sequence of children of a node. This last reason enables us to have a decidability procedure for the satisfiability of downward-XPath under a subclass of regular properties, something that was out of the scope of the work (Figueira, 2009).

Benedikt et al. (2008) studied the satisfiability problem for many XPath logics, mostly fragments without negation or without data equality tests. Also, the fragment $\mathsf{XPath}(\downarrow, =)$ is proved to be in NExpTime. We improve this result by providing an optimal PSpace upper bound. It is also known that $\mathsf{XPath}(\downarrow)$ is already PSpace-hard[1], and in this work we give a matching upper bound showing PSpace-completeness. Furthermore, Marx (2004) proves that $\mathsf{XPath}(\downarrow, \downarrow_*)$ is ExpTime-complete. In this work we prove that this complexity is preserved in the presence of data values and even under closure with Kleene star. We also consider

[1] This is a consequence of $\mathsf{XPath}(\downarrow)$ being able to code any formula from the normal modal logic K, which enjoys the finite- and tree-model properties (Blackburn et al., 2001), and is PSpace-complete (Ladner, 1977).

a fragment that is not mentioned in (Benedikt et al., 2008): $\mathsf{XPath}(\downarrow_*, =)$ and show that $\mathsf{XPath}(\downarrow_*)$ is PSPACE-complete while $\mathsf{XPath}(\downarrow_*, =)$ is EXPTIME-complete. In this case, data tests make a real difference in complexity.

In Chapter 6 the fragment $\mathsf{XPath}(\mathfrak{F}, =)$ (*i.e.*, forward XPath) is studied. In the cited chapter, the full set of downward and rightward axes are allowed, while the fragments treated here only allow the downward axis. It can hence express properties like (T3), or (T2) that cannot be expressed with the downward fragment. The forward fragment is shown to have a decidable satisfiability problem at the expense of a huge rise in complexity: from EXPTIME to non-primitive recursive. However, in the present chapter all the fragments considered are below EXPTIME, and meeting these elementary upper bounds requires an altogether different approach from the one taken in Chapter 6.

Regular properties on branches

One important object of a data tree is that of a 'branch': a succession of nodes that starts at the root and goes downward, ending at any node of the tree. Note that we consider the possibility that a branch may end at an inner node and not necessarily at a leaf. Given two positions $x \prec y$ in a branch, we define $\mathrm{str}(x, y)$ as the sequence of labels from the finite alphabet contained between x and y, including x and y. All the power of the automata model we will define relies on the ability to test data properties at distant positions of the tree. These positions are end points of branches whose string belongs to a certain regular language. We next define the execution of a nondeterministic finite automaton over a branch of a data tree.

NFAs $\mathcal{A} = (\mathbb{A}, Q, q_1, \delta, F)$are defined as usual, that is, $\mathbb{A}$ is a finite alphabet, Q is a finite set of states, q_1 is the initial state, $\delta \subseteq Q \times \mathbb{A} \times Q$ is the transition function, and $F \subseteq Q$ is the final set of states. As usual, $L(\mathcal{A})$denotes the set of strings accepted by $\mathcal{A}$. We write $(q, a, q') \in \mathcal{A}$ to denote that there is a transition from q to q' in $\mathcal{A}$ by reading a.

Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{b}$ be a data tree in $\mathit{Trees}(\mathbb{A}\times\mathbb{D})$. Given a NFA $\mathcal{A}$ over the alphabet $\mathbb{A}$, we consider the execution of $\mathcal{A}$ on the string contained between a position $x \in \mathsf{pos}(\mathbf{t})$ and a descendant position $x{\cdot}y \in \mathsf{pos}(\mathbf{t})$. For a state q, let $\mathcal{A}[q]$ be identical to $\mathcal{A}$ with the exception that now q is the initial state.

We note a 'one-step' of the execution as

$$q \xrightarrow[x]{\mathcal{A}} q'$$

if $(q, \mathbf{a}(x), q')$ is a transition of $\mathcal{A}$, the data tree being implicit in the notation. And we write

$$q \xrightarrow[x, x{\cdot}y]{\mathcal{A}} q' \quad \text{iff} \quad \begin{cases} q \xrightarrow[x]{\mathcal{A}} q' & \text{if } y = \epsilon; \\ q \xrightarrow[x]{\mathcal{A}} p_1 \xrightarrow[x\cdot i_1]{\mathcal{A}} p_2 \xrightarrow[x\cdot i_1\cdot i_2]{\mathcal{A}} \cdots \xrightarrow[x\cdot i_1\cdots i_n]{\mathcal{A}} p_{n+1} = q' & \text{if } y = i_1 \cdots i_n. \end{cases}$$

That is, if we can reach the configuration by reading the letters between x and $x{\cdot}y$ in a descending way. Note that the automaton's run includes the starting and ending labels. Hence, all runs execute at least one transition.

We fix a notation for the data values of those positions selected by some run of $\mathcal{A}$. For a state q of $\mathcal{A}$ and a tree $\mathbf{t}$, we write $[\![\mathcal{A}, q]\!]^{\mathbf{t}}$ to denote the set of data values of all positions x that can be reached starting at the root with state q and ending at x with a final state from F,

$$[\![\mathcal{A}, q]\!]^{\mathbf{t}} = \{\mathbf{d}(x) \mid x \in \mathsf{pos}(\mathbf{t}), \mathrm{str}(\epsilon, x) \in L(\mathcal{A}[q])\} \ .$$

We write $[\![\mathcal{A}]\!]^{\mathbf{t}}$ for $[\![\mathcal{A}, q_1]\!]^{\mathbf{t}}$, where q_1 is the initial state of $\mathcal{A}$.

5.2 Automata model

In this section we define an automata model that runs over data trees. We show that this model has a decidable emptiness problem in Section 5.3. In Section 5.4.1 we will show that $\mathsf{XPath}(\downarrow_*, \downarrow, =)$ formulæ can be effectively translated to an equivalent DD automaton, thus obtaining a decidability procedure for its satisfiability problem.

Our model, called *Downward Data automaton* (or simply *DD automaton*), has an execution that consists of two steps: (1) the execution of a transducer, and (2) the verification of data properties of the transduced tree.

For a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$, the first step consists in the translation of $\mathbf{a}$ into another tree $\mathbf{b}$. This is done using a nondeterministic letter-to-letter transducer over unranked trees. We adopt a more detailed definition, where the transducer explicitly has as a parameter the class $\mathscr{C}$ of regular properties that it can test over a siblinghood at each transition. If we take this parameter to be the set of all regular properties, this automaton is a standard transducer over unranked trees. However, the emptiness problem for Downward Data automata has a very high complexity lower bound unless we restrict $\mathscr{C}$ to be a suitable subclass of regular languages . This class, defined as the set of *extensible* languages, will be introduced in the sequel (Definition 5.7).

In the second step, for every subtree of the transduced tree $\mathbf{b} \otimes \mathbf{d}$, a property on the data values of the tree is verified. The letter at the root of the subtree under inspection determines the property to verify. The properties are boolean combinations of tests verifying the existence of data values shared by nodes in the subtree, hanging from branches satisfying some regular expression.

A brief comment on notation. In the definition of these automata there will be two sorts of sets of states, namely the states corresponding to the run of the transducer, and the states corresponding to the run of the verifier. To avoid confusion we consistently write $\dot{Q}, \dot{q}, \dot{q}', \ldots$ as symbols for the states of the transducer, and $Q, q, q', \ldots$ for the states of the verifier.

Transducer

Let $\mathscr{C}$ be a subclass of regular languages $\mathscr{C} \subseteq REG$. We define a bottom-up unranked tree transducer. This definition is parametrized by $\mathscr{C}$ in the following sense: At every transition, the automaton can test the siblinghood for membership in some regular language that must be in the class $\mathscr{C}$.

Definition 5.1. A **$\mathscr{C}$-transducer** defines a relation between trees with the same set of positions. Given a transducer $\mathcal{R}$, we use the same symbol $\mathcal{R}$ to denote the relation of the pairs of trees accepted by $\mathcal{R}$ with a slight abuse of notation, *i.e.*, $\mathcal{R} \subseteq \mathit{Trees}(\mathbb{A}\times\mathbb{B})$. The idea is that $\mathbf{a} \in \mathit{Trees}(\mathbb{A})$ and $\mathbf{b} \in \mathit{Trees}(\mathbb{B})$ are related by $\mathcal{R}$ if $\mathbf{a} \otimes \mathbf{b} \in \mathcal{R}$. Thus, in order to be related by $\mathcal{R}$, the trees $\mathbf{a}$ and $\mathbf{b}$ need to have the same set of positions. This relation is defined as the set of accepted trees of a nondeterministic bottom-up unranked transducer represented as a tuple $\langle \mathscr{C}, \mathbb{A}, \mathbb{B}, \dot{Q}, \dot{Q}_F, \delta \rangle$ where

- $\mathbb{A}$ and $\mathbb{B}$ are finite alphabets of letters,
- $\dot{Q}$ is a finite set of states,
- $\dot{Q}_F \subseteq \dot{Q}$ is the set of final states,
- $\mathscr{C} \subseteq REG(\dot{Q})$ is a class of regular languages over the alphabet $\dot{Q}$,
- $\delta \subseteq \dot{Q} \times \mathbb{A} \times \mathbb{B} \times \mathscr{C}$ is a finite set of transitions.

We define $\mathbf{a} \otimes \mathbf{b} \in \mathcal{R}$ iff $\mathbf{a} \in \mathit{Trees}(\mathbb{A})$, $\mathbf{b} \in \mathit{Trees}(\mathbb{B})$ with $\mathsf{pos}(\mathbf{a}) = \mathsf{pos}(\mathbf{b}) = P$, and there exists a states assigment $\rho : P \to \dot{Q}$, such that

- for every leaf x, there exists $\mathcal{L} \in \mathscr{C}$ with $\epsilon \in \mathcal{L}$ and $(\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L}) \in \delta$,
- for every siblinghood $x{\cdot}1, \ldots, x{\cdot}l$, there is a language $\mathcal{L} \in \mathscr{C}$ such that $(\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L}) \in \delta$ and $\rho(x{\cdot}1){\cdot}\cdots{\cdot}\rho(x{\cdot}l) \in \mathcal{L}$.

We call ρ a **run** of $\mathcal{R}$ on $\mathbf{a} \otimes \mathbf{b}$. We say that the run is **accepting** iff $\rho(\epsilon) \in \dot{Q}_F$.

We now turn to the second step.

Verifier

Definition 5.2. A **verifier** $\mathcal{V} \subseteq \mathit{Trees}(\mathbb{B}\times\mathbb{D})$ defines a set of data trees that are valid with respect to some data properties. It is a tuple $\langle \mathcal{A}_1, \ldots, \mathcal{A}_{\mathsf{K}}, \mathsf{v} \rangle$ of K NFA over the alphabet $\mathbb{B}$, namely $\mathcal{A}_1, \ldots, \mathcal{A}_{\mathsf{K}}$, and a function $\mathsf{v} : \mathbb{B} \to \Phi$ mapping letters of the alphabet to formulæ expressing data properties. The idea is that every subtree $\mathbf{t}|_x$ of the original tree $\mathbf{t}$ must verify a property expressed by the formula $\mathsf{v}(\mathbf{b}(x))$. A typical property that we can test at a subtree is the existence of two positions with the same data value, such that one is reachable by going downward through a branch whose labelling is in some regular language $\mathcal{L}_1$, and the other by some other branch with labelling in $\mathcal{L}_2$.

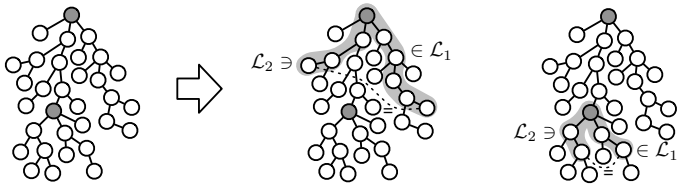

Fig. 5.1: If $\mathcal{L}_1, \mathcal{L}_2$ are the languages recognized by $\mathcal{A}_1, \mathcal{A}_2$, and the marked positions are labeled by a, then $\mathsf{v}(a)$ is verified in both subtrees.

The properties of Φ are a subset of closed first-order formulæ with no quantifier alternation (that is, no $\forall\exists$ or $\exists\forall$ patterns allowed), and K unary relations, one for every automaton $\mathcal{A}_i$, namely

$$D_1, \ldots, D_{\mathsf{K}} \ .$$

Given a set *Vars* of variables, Φ contains all the formulæ ϕ defined by the grammar

$$\begin{aligned} \phi &::= \neg\,\phi \ \mid\ \phi \wedge \phi \ \mid\ \phi \vee \phi \ \mid\ \exists \bar{v}.\psi \\ \psi &::= \psi \wedge \psi \ \mid\ \psi \vee \psi \ \mid\ v = v' \ \mid\ v \neq v' \ \mid\ D_i(v) \end{aligned}$$

where $\bar{v}$ stands for a set of variables from *Vars*, $v, v' \in$ *Vars*, $i \in [\mathsf{K}]$, and we restrict ϕ to have no free variables. The variables are interpreted over the set of data values, and the D_i's as sets of values reachable by the automata $\mathcal{A}_i$'s.

Given a data tree $\mathbf{t}$, let $\mathcal{I}_{\mathbf{t}}$ be the first-order interpretation where each unary relation D_i is interpreted as $[\![\mathcal{A}_i]\!]^{\mathbf{t}}$, and the interpretation of the domain is $\mathbb{D}$. We say that that φ is verified in $\mathbf{t}$ if φ is true under the interpretation $\mathcal{I}_{\mathbf{t}}$. A verifier $\langle \mathcal{A}_1, \ldots, \mathcal{A}_{\mathsf{K}}, \mathsf{v}\rangle$ *accepts* a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d} \in$ *Trees*$(\mathbb{B}\times\mathbb{D})$ iff for every position $x \in \mathsf{pos}(\mathbf{t})$ the formula $\mathsf{v}(\mathbf{a}(x))$ is verified in the subtree $\mathbf{t}|_x$.

Example 5.3. Suppose $\mathsf{v}(a) = \exists\, x_1, x_2\ .\ D_1(x_1) \wedge D_2(x_2) \wedge x_1 \neq x_2$, for a letter a. This means that for every a-rooted subtree we can find two branches in the regular languages recognized by $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively leading to two nodes whose data values are different as depicted in Figure 5.1. □

Definition 5.4. A $\mathscr{C}$**-Downward Data automaton** ($\mathscr{C}$-DD for short) is a pair $(\mathcal{R}, \mathcal{V})$ made of a $\mathscr{C}$-transducer $\mathcal{R} \subseteq$ *Trees*$(\mathbb{A}\times\mathbb{B})$ and a verifier $\mathcal{V} \subseteq$ *Trees*$(\mathbb{B}\times\mathbb{D})$. A data tree $\mathbf{a} \otimes \mathbf{d}$ is accepted by $(\mathcal{R}, \mathcal{V})$ iff there exists $\mathbf{b} \in$ *Trees*$(\mathbb{B})$ such that $\mathbf{a}\otimes\mathbf{b} \in \mathcal{R}$ and $\mathbf{b}\otimes\mathbf{d} \in \mathcal{V}$, as depicted in Figure 5.2. When we want to make explicit that the witnessing tree for the acceptance of $\mathbf{a} \otimes \mathbf{d}$ is $\mathbf{b}$, we will say equivalently that $\mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ is accepted by $(\mathcal{R}, \mathcal{V})$. Also, we say that $\mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ *has a run* if there is a $\mathcal{R}$ run on $\mathbf{a} \otimes \mathbf{b}$ where $\mathbf{b} \otimes \mathbf{d} \in \mathcal{V}$.

We now give some closure properties of DD automata.

We say that a class of languages $\mathscr{C}$ is closed under **componentwise product**, if for every two languages $\mathcal{L}_1, \mathcal{L}_2$ of $\mathscr{C}$ and the language $\mathcal{L}_1 \times_c \mathcal{L}_2 = \{(a_1, b_1)\cdots(a_n, b_n) \mid a_1 \cdots a_n \in \mathcal{L}_1,\ b_1 \cdots b_n \in \mathcal{L}_2\}$ is in $\mathscr{C}$. Note that if $\mathcal{L}_1, \mathcal{L}_2$ are regular, $\mathcal{L}_1 \times_c \mathcal{L}_2$ is regular.

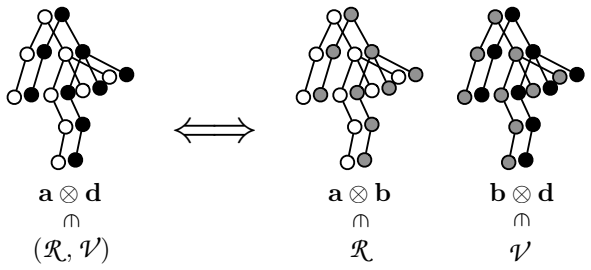

Fig. 5.2: Acceptance condition of a DD automaton $(\mathcal{R}, \mathcal{V})$.

Proposition 5.5. *Given a class $\mathscr{C}$ of languages closed under componentwise product, the class of languages recognized by $\mathscr{C}$-DD automata is closed under intersection.*

Proof. First observe that the set of properties Φ of any verifier is closed under conjunction and disjunction, since it is closed under the operators $\wedge$, $\vee$.

Suppose we have two DD automata $(\mathcal{R}^1, \mathcal{V}^1)$ and $(\mathcal{R}^2, \mathcal{V}^2)$, $\mathcal{R}^i \subseteq \mathit{Trees}(\mathbb{A}\times\mathbb{B}_i)$, $\mathcal{V}^i \subseteq \mathit{Trees}(\mathbb{B}_i\times\mathbb{D})$. We build the intersection automaton $(\mathcal{R}, \mathcal{V})$ where $\mathcal{R} \subseteq \mathit{Trees}(\mathbb{A}\times(\mathbb{B}_1\times\mathbb{B}_2))$ is a transducer that tags each position with a pair of letters, such that $\mathbf{a}\otimes\mathbf{b}_1\otimes\mathbf{b}_2 \in \mathcal{R}$ iff $\mathbf{a}\otimes\mathbf{b}_1 \in \mathcal{R}^1$ and $\mathbf{a}\otimes\mathbf{b}_2 \in \mathcal{R}^2$. This can be done since $\mathscr{C}$ is closed under componentwise product, as follows. We define the state space as $\dot{Q}^1 \times \dot{Q}^2$ and the final states as $\dot{Q}^1_F \times \dot{Q}^2_F$. For every transition $(\dot{q}_1, a, b_1, \mathcal{L}_1)$ of $\mathcal{R}^1$ and $(\dot{q}_2, a, b_2, \mathcal{L}_2)$ of $\mathcal{R}^2$ we have a transition $((\dot{q}_1, \dot{q}_2), a, (b_1, b_2), \mathcal{L}_1 \times_c \mathcal{L}_2)$ in $\mathcal{R}$.

On the other hand we build $\mathcal{V} \subseteq \mathit{Trees}((\mathbb{B}_1\times\mathbb{B}_2)\times\mathbb{D})$ such that $\mathbf{b}_1\otimes\mathbf{b}_2\otimes\mathbf{d} \in \mathcal{V}$ iff $\mathbf{b}_1 \otimes \mathbf{d} \in \mathcal{V}^1$ and $\mathbf{b}_2 \otimes \mathbf{d} \in \mathcal{V}^2$. This can be done since Φ is closed under conjunction. □

To obtain closure under complementation we need some extra hypothesis. Given an alphabet $\mathbb{A}$ and a letter $a \in \mathbb{A}$, we call the **membership language of** a to the language $\{w{\cdot}a{\cdot}w' \mid w, w' \in \mathbb{A}^*\}$. We say that a class $\mathscr{C} \subseteq \mathit{REG}$ is **closed under inverse homomorphisms** iff for every language $\mathcal{L} \in \mathscr{C}$ over an alphabet $\mathbb{A}$ and for every homomorphism $h : \mathbb{B} \to \mathbb{A}$ there is a language $\mathcal{L}' \in \mathscr{C}$ over $\mathbb{B}$ such that $\mathcal{L}' = \{w \in \mathbb{B}^* \mid h(w) \in \mathcal{L}\}$.

Proposition 5.6. *Let $\mathscr{C}$ a class of languages closed under all boolean operations, inverse homomorphisms and containing all membership languages. The class of languages recognized by $\mathscr{C}$-DD automata is closed under complementation.*

Proof. Let $(\mathcal{R}, \mathcal{V})$ with $\mathcal{R} \subseteq \mathit{Trees}(\mathbb{A}\times\mathbb{B})$, $\mathcal{V} \subseteq \mathit{Trees}(\mathbb{B}\times\mathbb{D})$, where $\dot{Q}$ is the set of states of $\mathcal{R}$. We build $(\mathcal{R}^c, \mathcal{V}^c)$ the complement of $(\mathcal{R}, \mathcal{V})$. We define $\mathcal{R}^c \subseteq \mathit{Trees}(\mathbb{A}\times\mathbb{B}')$ with $\mathbb{B}' = \wp(Z)$ where $Z = \dot{Q}\times\mathbb{B}\times\{\mathit{below}, \mathit{notyet}, \mathit{here}\}$. Every node x of the tree $\mathbf{t}$ is labeled by a set of tuples $(\dot{q}, b, i) \in Z$ describing, for each of the possible partial runs of $\mathcal{R}$ on $\mathbf{t}|_x$, whether there is a node that does *not* verify a property demanded by $\mathcal{V}$. More precisely, $\dot{q}$ is the state of the run at the root, b refers to the output letter of the root, and $i \in \{\mathit{below}, \mathit{notyet}, \mathit{here}\}$ is to keep

track of whether in the currently described run there is a node falsifying a property demanded by $\mathcal{V}$. The value *notyet* denotes that all the nodes in the subtree $\mathbf{t}|_x$ for this run verify the properties of $\mathcal{V}$; *below* that there is a node different from x that does not verify the property of $\mathcal{V}$, and *here* that in the current run x does not verify the property of $\mathcal{V}$. Formally, $\mathcal{R}^c$ tags every node x of the tree $\mathbf{t}$ with the set of tuples $(\dot{q}, b, i) \in Z$ such that there is a partial run of $\mathcal{R}$ on $\mathbf{t}|_x$ where

- the state of the root is $\dot{q}$,
- the output label of the root is b, and
- $i = \mathit{below}$ iff there is a node of $\mathbf{t}|_x$ different from the root with output label $(\dot{q}', b', i')$ such that $i' = \mathit{here}$.

Further, $\mathcal{R}^c$ checks that the output label of the root contains only tuples of the form $(\dot{q}, b, i)$ where either $i = \mathit{below}$, $i = \mathit{here}$ or $\dot{q}$ is not final in $\mathcal{R}$. This ensures that all possible accepting runs of $\mathcal{R}$ lead to trees that falsify some demand of $\mathcal{V}$ at some node.

Notice that if there is a node different from the root with $i' = \mathit{here}$ then $i = \mathit{below}$, and if there is no node different to the root with $i' = \mathit{here}$, then i may be *here* or *notyet*. Also, notice that the state at the leaves has $i = \mathit{here}$ or $i = \mathit{notyet}$ (*i.e.*, no *below*).

Assuming we can build $\mathcal{R}^c$ with the described behavior, we define $\mathcal{V}^c \subseteq \mathit{Trees}(\mathbb{B}' \times \mathbb{D})$ with $\mathsf{v}^c : \mathbb{B}' \to \Phi$ its formulæ assignment, as

$$\mathsf{v}^c(\dot{q}, b, i) = \begin{cases} \mathit{true} & \text{if } i \in \{\mathit{below}, \mathit{notyet}\} \\ \neg\, \mathsf{v}(b) & \text{if } i = \mathit{here}. \end{cases}$$

It follows that if a data tree $\mathbf{t}$ is accepted by $(\mathcal{R}^c, \mathcal{V}^c)$, then any run of $\mathcal{R}$ on $\mathbf{t}$ produces an output such that falsifies $\mathcal{V}$ at some node, and then $\mathbf{t}$ is not accepted by $(\mathcal{R}, \mathcal{V})$. In turn, if $\mathbf{t}$ is accepted by $(\mathcal{R}, \mathcal{V})$, there must be an accepting run of $\mathcal{R}$ whose output verifies $\mathcal{V}$. In other words, there must be a tuple $(\dot{q}, b, \mathit{notyet})$ with $\dot{q}$ an accepting state in the root of every output of $\mathcal{R}$ applied to $\mathbf{t}$, but this cannot be since $\mathcal{R}^c$ does not accept such trees.

Now let us describe with more care how $\mathcal{R}^c$ is built from $\mathcal{R}$. Let $\mathcal{L}_1, \ldots, \mathcal{L}_s$ be all the languages from $\mathscr{C}$ used in the transitions of $\mathcal{R}$. For every subset S of these languages consider a language $\mathcal{L}_S$ that tests that a word belongs to all the languages of S, and does not belong to any other language. It follows that $\mathcal{L}_S \in \mathscr{C}$, by closure under intersection and complementation.

Let us define the state space of $\mathcal{R}^c$ as $\mathbb{B}'$. Given language $\mathcal{L}$ over $\dot{Q}$ (the state space of $\mathcal{R}$), we can build a similar language $\mathcal{L}'$ over $\mathbb{B}'$ that tests that we can pick one tuple for each element of the word in such a way that when we project the first component, the word belongs to $\mathcal{L}$. If $\mathcal{L}$ is represented as a regular expression, this boils down to replacing every atomic expression $\dot{q}$ by a big disjunction of all the elements of $\mathbb{B}'$ containing a tuple with state $\dot{q}$. It follows that $\mathcal{L}' \in \mathscr{C}$

since it is closed under inverse homomorphisms. Note that we can further check that witnessing word of tuples contains an element $(\dot{q}, b, i)$ with $i = \mathit{notyet}$ (or $i = \mathit{below}$). This is a consequence of having the membership languages for each tuple with *notyet* and *below*.

Then, for every $A \subseteq \{\mathcal{L}_1, \ldots, \mathcal{L}_s, \mathit{notyet}, \mathit{below}\}$ consider a language $\mathcal{L}_A$ that tests

- that there is a word over $\mathbb{B}'$ and a tuple for every element such that the projection on $\dot{Q}$ is accepted by all languages of S and rejected by all others;
- if $\mathit{notyet} \in A$ that there is an element of the word containing a tuple with *notyet*;
- and if $\mathit{below} \in A$ that there is an element of the word containing a tuple with *below*.

Now, $\mathcal{R}^c$ is built with a set of rules of the form $(B, a, B, \mathcal{L}_A)$ where $A \subseteq \{\mathcal{L}_1, \ldots, \mathcal{L}_s, \mathit{here}, \mathit{below}\}$, $a \in \mathbb{A}$, and $B \in \mathbb{B}'$ is such that

for every $\dot{q} \in \dot{Q}$, $b \in \mathbb{B}$ and $\mathcal{L} \in A$ such that $(\dot{q}, a, b, \mathcal{L}) \in \delta$ then either

- $\mathit{below} \notin S$ and there is $i \in \{\mathit{here}, \mathit{notyet}\}$ where $(\dot{q}, b, i) \in B$, or
- $\mathit{below} \in S$ and $(\dot{q}, b, \mathit{below}) \in B$.

Thus, *below* means that there was a *here* in the subtree, and *here* means that it is in that precise point that a formula is falsified.

Finally, the accepting states are those whose every tuple with an accepting state contains no *notyet* flag. □

The emptiness problem for $\mathscr{C}$-Downward Data automata has a non-primitive recursive complexity if we do not impose any restriction to $\mathscr{C}$. The non-primitive recursive lower bound can be seen as a consequence of the fact that DD automata

- can force the model to be linear (i.e., that all nodes have at most one child), and
- can capture any downward XPath formula, as we will see in Section 5.4.1.

It is known that the satisfiability problem for downward XPath on non-branching data trees (called *data word*) is non-primitive recursive (see e.g. (Figueira and Segoufin, 2009)). Although the emptiness problem for DD automata without any restriction is not discussed here, we believe that it is decidable. Probably, it can be shown to be decidable via the theory of well quasi orderings, by a similar technique as the one used in (Figueira, 2010) in the context of downward alternating register automata.

Nevertheless, when $\mathscr{C}$ is restricted to have some good properties, emptiness of DD automata can be tested in 2ExpTime. Further, if the verifier is such that

the number of occurrences of the relations D_i inside every quantified subformula is bounded by a constant[2], we achieve an ExpTime decision procedure. We will show that downward XPath formulæ can be translated in PTime into equivalent DD automata. These automata are restricted to a class of languages $\mathscr{C}$ with good properties, and such that the number of occurrences of the D_i are always bounded by 2. Then, the decidability in ExpTime of the satisfiability problem for downward-XPath will follow. In the next section we define which is the necessary property that $\mathscr{C}$ must have in order to obtain the aforementioned upper bounds.

Extensibility of languages

To have a low complexity in the testing for emptiness of $\mathscr{C}$-DD automata, we need to weaken the kind of regular properties that the transducer can verify. That is, we need to restrict $\mathscr{C}$. The idea is that if a word is in a language, then an *extension* of the word where more occurrences of each letter may occur must also be in the language. Let us formally define this notion.

Let $\mathbb{A} = \{a_1, \ldots, a_n\}$ be an alphabet and p be the Parikh image function. That is, the function $p : \mathbb{A}^* \to \mathbb{N}^n$ that associates each word with a vector that counts the number of appearances of each letter, $p(w)(i) = |\{j \mid w(j) = a_i\}|$, where $w(j)$ denotes the jth letter of the string w, starting at 1.

Definition 5.7. We say that a regular language $\mathcal{L}$ is **extensible** if for every $m \in \mathbb{N}$ and word $w \in \mathcal{L}$ there exists another word $w' \in \mathcal{L}$ such that for any coordinate $i \in [n]$,

$$\text{if } p(w)(i) \neq 0 \text{ then } p(w')(i) \geq m, \text{ and}$$
$$\text{if } p(w)(i) = 0 \text{ then } p(w')(i) = 0.$$

In this case we say that w' is an m**-extension** of w. We write $\mathscr{E}$ for the class of all extensible regular languages. Likewise we define a *tree* language to be extensible if it can be defined by an unranked tree automaton whose transitions only use languages from $\mathscr{E}$ to test properties on the siblinghoods. Note that this is a restriction of the *horizontal* tests, and that an extensible tree language can still test for any regular property along a branch. By $\mathscr{E}_{tree}$ we denote the set of extensible tree languages.

Let us give some examples of extensible classes of languages. As a first example, consider the following class of languages.

$\exists$-*class* $:= \{\mathcal{L}_{\mathbb{A}_1,\ldots,\mathbb{A}_n,\mathbb{B}} \mid \mathbb{A}_1, \ldots, \mathbb{A}_n \text{ and } \mathbb{B} \text{ are a finite alphabets, } n \in \mathbb{N}\}$, where $\mathcal{L}_{\mathbb{A}_1,\ldots,\mathbb{A}_n,\mathbb{B}} := \{w \in (\cup_i \mathbb{A}_i \cup \mathbb{B})^* \mid w \text{ contains at least one letter from each } \mathbb{A}_i\}$

Note that $\exists$-*class* is a class of extensible languages, it is closed under all boolean operations, inverse homomorphisms and contains all membership languages. Therefore, the class of data tree languages accepted by $\exists$-*class*-DD automata is closed

[2] For example, $(\exists v.D_1(v) \wedge D_2(v)) \wedge \neg(\exists v.D_2(v) \wedge D_3(v))$ is bounded by 2, since there are at most 2 D_i's used in each quantified subformula.

under complementation. If we also close $\exists$-*class* under componentwise product, we still obtain an extensible language. In this case, the class of languages accepted by DD automata is closed by all boolean operaitons.

Also note that the class of $\exists$-*class*-transducers are those that can only test for the existence or non existence of children with a certain label, but they do not test any condition on the horizontal ordering or on the number of appearances of elements in the siblinghoods. In fact, they can mostly test for *vertical* properties along branches.

As another example, we define REG_* the class of *star*-regular languages.

$$REG_* := \{\mathcal{L} \in REG \mid \mathcal{L} \text{ is defined by a regular expression whose every symbol is in the scope of at least one } *\}$$

Thus, for example '$((a \mid b)\, c^* d)^*$' is in REG_* while '$a^* b$' is not. The class REG_* is trivially extensible.

The property of extensibility is trivially closed under union, but not necessarily under intersection or complementation.

Proposition 5.8. *Given two extensible languages $\mathcal{L}_1, \mathcal{L}_2$, $\mathcal{L}_1 \cup \mathcal{L}_2$ is also extensible.*

As a counter-example for the intersection, note that $(ab)^* \cap a^* b^* = \{\epsilon, ab\}$ which is not extensible, and for the complementation note also that under the singleton alphabet $\mathbb{A} = \{a\}$, $(a^+ a)^c = \{\epsilon, a\}$, which is not extensible.

In the sequel, we will show decidability of the emptiness problem for $\mathscr{E}$-DD automata.

5.3 The emptiness problem

Throughout this section, we fix the transducer to be an $\mathscr{E}$-transducer. The main objective of this section is to prove the following theorem.

Theorem 5.9. *The emptiness problem for $\mathscr{E}$-Downward Data automata can be decided in* 2ExpTime.

Sketch of the proof

The proof of the main theorem is divided into four parts. In the first part (Section 5.3.1) we define some decoration or marking of the nodes of a data trees that in some sense witnesses the acceptance of a run of a DD automaton. These decorations of the tree are the main structures with which we will work with in our proof.

The second part (Section 5.3.2) is dedicated to proving two properties. The first property states that if a DD automaton is nonempty, it accepts a tree decorated with some guidance system that marks the paths to be covered in order to verify the properties imposed by the verifier. In some sense, it decorates the tree as

in Figure 5.1, avoiding having two paths going through the same node. The guidance system is called *certificate* and this property is called *admissibility of correct certificates*. The second property states that if a DD automaton is nonempty, then it accepts a tree whose data values are in a certain normal form, as follows. Every pair of subtrees rooted at two different children of a node, have a disjoint set of data values, with the exception of some polynomially bounded many (this is called the *disjoint values property*).

The third part (Section 5.3.3) is centered around proving that DD automata have the exponential width model property. That is, if a DD automaton has a nonempty language, then it accepts a tree whose width is exponentially bounded in the size of the automaton.

In the fourth part (Section 5.3.4) we give the algorithm for testing emptiness of DD automata, which is based on the bound on the width and two other properties: the disjoint values property and the admissibility of correct certificates.

Parameters

We first fix some parameters that we will need in the complexity analysis. We fix, once and for all, that the verifier $\mathcal{V}$ contains K NFA: $\mathcal{A}_1, \ldots, \mathcal{A}_{\mathsf{K}}$. We write Aut to denote the se of these automata, $Aut = \{\mathcal{A}_1, \ldots, \mathcal{A}_{\mathsf{K}}\}$. We assume without any loss of generality that all $\mathcal{A}_1, \ldots, \mathcal{A}_{\mathsf{K}}$ share the same set of states $\mathcal{Q} := \{q_1, \ldots, q_{\mathsf{N}}\}$, and have q_1 as initial state. Also, for each i we write $\mathcal{Q}_F^{\mathcal{A}_i}$ for the set of final states of $\mathcal{A}_i$. By $|\mathcal{A}|$ we denote the number of transitions of $\mathcal{A}$, and Aut stands for $|\mathcal{A}_1| + \cdots + |\mathcal{A}_{\mathsf{K}}|$. Let *Vars* be the set of variables used by the formulæ of the verifier, and let $|\mathit{Vars}| = \mathsf{V}$. We write R for the maximum number of relations admitted under a quantification. In other words, for any formula of the verifier and for any quantified subformula $\exists \bar{x}.\psi$, there are at most R different relations used in ψ from the K available. The worst case would be when $\mathsf{R} = \mathsf{K}$, as in the formula $\mathsf{v}(b) = \exists x.D_1(x) \wedge \cdots \wedge D_{\mathsf{K}}(x)$ for some $b \in \mathbb{B}$. This is an important parameter, since we will later see that the subclass of DD automata with R fixed has an ExpTime emptiness problem, while the general class has a 2ExpTime emptiness problem. In Section 5.4.1 we will argue that downward XPath expressions can be translated into DD automata where $\mathsf{R} = 2$, and from this fact it will follow that its satisfiability is in ExpTime.

Finally, we fix $\dot{Q}$ to denote the set of states of the transducer. In addition, we use NFA to represent the regular language $\mathcal{L}$ for every transition $(\dot{q}, a, b, \mathcal{L}) \in \delta$. We will usually use the symbol $\mathcal{B}$ to denote such an automaton. We will assume without any loss of generality that all automata used in the transitions of the transducer share the same set of states $\tilde{Q} = \{\tilde{q}_0, \tilde{q}_1, \ldots\}$ and that all automata have the same initial state $\tilde{q}_0$. $|\mathcal{R}|$ stands for the number of transitions of $\mathcal{R}$, and we assume that $|\dot{Q}|$ is at most $|\mathcal{R}|$. By $|\mathcal{V}|$ we denote $\mathsf{K} + \mathsf{N} + \mathsf{R} + \mathsf{V} + \mathsf{Aut}$. Summing up, our complexity analysis will be based on the parameters: $\mathsf{K}, \mathsf{N}, \mathsf{R}, \mathsf{V}, |\dot{Q}|, |\tilde{Q}|, |\mathcal{R}|, \mathsf{Aut}, |\mathcal{V}|$.

5.3.1 Decorations of trees

In order to bound the width of the tree, we need a more fine grained notion of runs of a transducer. We label the nodes of the tree with the run of the automaton recognizing the regular language on the siblinghood used at the transition of the transducer's run. This will enable us to state a pumping argument on siblinghoods of the data tree.

Detailed run

Consider, for any extensible language $\mathcal{L} \in \mathscr{E}$ over $\dot{Q}$, a NFA $\mathcal{B}_\mathcal{L}$ with initial state $\tilde{q}_0$. Remember that every $\mathcal{B}_\mathcal{L}$ used in the transducer uses the same set $\tilde{Q}$ of states.

Definition 5.10. A **detailed run of a transducer** $\mathcal{R}$ on a tree $\mathbf{a}\otimes\mathbf{b} : P \to \mathbb{A}\times\mathbb{B}$ consists of a 3-uple (τ, ρ, ρ_h).

- ρ is a run, that in this context we call a *vertical run*.
- τ is a function
$$\tau : P \to \delta$$
specifying which choice of transitions are needed for the run ρ. We call τ a *transition assignment*. It verifies, for every position x with l children,
$$\tau(x) = (\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L}) \text{ where } \rho(x{\cdot}1)\cdots\rho(x{\cdot}l) \in \mathcal{L}.$$
We abbreviate $\mathcal{B}_x$ to denote the NFA $\mathcal{B}_\mathcal{L}$ of the regular language $\mathcal{L}$ defined in the transition $\tau(x)$ of the transducer's run.
- Finally, ρ_h is an assignment from the set of positions P to the set of states of the automaton $\mathcal{B}$ that corresponds to the regular language that needs to be checked in order to apply the transition.
$$\rho_h : P \to \tilde{Q}\ .$$
We call ρ_h the *horizontal run*. It verifies the following conditions (in short, that it is a run of a NFA on the siblinghood).
  1. For every leftmost sibling $x{\cdot}1 \in P$, $(\tilde{q}_0, \rho(x{\cdot}1), \rho_h(x{\cdot}1))$ is a transition of $\mathcal{B}_x$.
  2. For every pair of consecutive siblings $x{\cdot}i, x{\cdot}(i+1) \in P$,
  $$(\rho_h(x{\cdot}i), \rho(x{\cdot}(i+1)), \rho_h(x{\cdot}(i+1)))$$
  is a transition of $\mathcal{B}_x$.
  3. For every rightmost sibling $x{\cdot}l$, $\rho_h(x{\cdot}l)$ is a final state of $\mathcal{B}_x$.

This completes the definition of a detailed run.

We also need to be able to precisely describe the behavior of the *data values* at a position of a data tree.

Description of data

Next we introduce sets of data simultaneously accessible by R automata (remember that this is the maximum number of simultaneous relations D_i's used by the verifier $\mathcal{V}$). For each one of these sets of data values, we preserve *at most* V elements (this is a bound on the maximum number of variables used by $\mathcal{V}$).

Observe that the verifier can test properties of the set of the cardinality of the sets $[\![\mathcal{A}_i]\!]$, or a boolean combination of them. More precisely, theh verifier can only test the (in)existence of a number of data values that are in some *intersection* set $[\![\mathcal{A}_{i_1}]\!] \cap \cdots \cap [\![\mathcal{A}_{i_t}]\!]$. Here, $\mathcal{A}_{i_1}, \ldots, \mathcal{A}_{i_t}$ range over Aut, and $t \leq \mathsf{R}$. The tests boil down to checking whether

- the intersection contains *at least* $1, 2, \ldots, \mathsf{V}$ different elements, or
- the intersection contains *at most* $0, 1, \ldots, \mathsf{V}-1$ different elements.

Note that we cannot test, for example, that the set contains *exactly* V elements, since we would need a formula with $\mathsf{V}+1$ variables. We annotate the tree with this information. Since later we will need to check that this information is consistent between a parent position and its children, we need to also consider the states of the automata $\mathcal{A}_i$. Next, we define the set of intersections of at most R relations $[\![\mathcal{A}_{i_1}, q_{j_1}]\!] \cap \cdots \cap [\![\mathcal{A}_{i_\mathsf{R}}, q_{j_\mathsf{R}}]\!]$.

$$\mathit{Inters} := \wp_{\leq \mathsf{R}}(\mathit{Aut} \times \mathcal{Q}) \ ,$$

where $\wp_{\leq \mathsf{R}}$ denotes the set of subsets of at most R elements. The following holds by definition.

$$|\mathit{Inters}| \leq (\mathsf{K}.\mathsf{N})^{\mathsf{R}} \tag{5.1}$$

Given a data tree $\mathbf{t}$ and an intersection we extend the $[\![\ ]\!]^{\mathbf{t}}$ notation by taking the intersection of the R sets. That is,

$$[\![I]\!]^{\mathbf{t}} = \bigcap \{ [\![\mathcal{A}, q]\!]^{\mathbf{t}} \mid (\mathcal{A}, q) \in I \} \ , \quad \text{for } I \in \mathit{Inters}.$$

Definition 5.11. The **data profile** is a function that assigns the number of elements present at each $I \in \mathit{Inters}$ to every data tree as follows.

$$\begin{aligned} \textit{d-profile} &: \mathit{Trees}(\mathbb{A} \times \mathbb{D}) \to (\mathit{Inters} \to [0..\mathsf{V}]) \\ \textit{d-profile}(\mathbf{t}) &= \{ I \mapsto \min(|[\![I]\!]^{\mathbf{t}}|, \mathsf{V}) \mid I \in \mathit{Inters} \} \end{aligned}$$

Observe that since V is the number of variables in Φ, it is enough to count up to V.

Note that a tree's profile carries sufficient information to evaluate at the root any formula $\varphi \in \Phi$ used by the verifier. We write $f \models \varphi$ if f is a function $f : \mathit{Inters} \to [0..\mathsf{V}]$ and φ is a formula of the verifier $\mathsf{v}(b) = \varphi$ such that φ holds in

$$f \models \psi \wedge \psi' \text{ iff } f \models \psi \text{ and } f \models \psi'$$
$$f \models \neg\psi \text{ iff } f \not\models \psi$$
$$f \models \exists v_1, \ldots, v_t.\psi \text{ iff } I, g, h \models \psi \text{ for some } I \in \mathit{Inters},\ g : \{v_1, \ldots, v_t\} \to \{1, \ldots, t\}$$
$$\text{and } h : \{1, \ldots, t\} \to \wp(I) \text{ s.t. } |h^{-1}(I')| \leq f(I') \text{ for all } I' \subseteq I$$

$$I, g, h \models \psi \wedge \psi' \text{ iff } I, g, h \models \psi \text{ and } I, g, h \models \psi' \qquad I, g, h \models v = v' \text{ iff } g(v) = g(v')$$
$$I, g, h \models \psi \vee \psi' \text{ iff } I, g, h \models \psi \text{ or } I, g, h \models \psi' \qquad I, g, h \models v \neq v' \text{ iff } g(v) \neq g(v')$$
$$I, g, h \models D_i(v) \text{ iff } (\mathcal{A}_i, q_1) \in h(g(v))$$

Fig. 5.3: The relation $f \models \varphi$ given $f : \mathit{Inters} \to [0..\mathsf{V}]$.

a tree $\mathbf{t}$ if $f = \mathit{d\text{-}profile}(t)$. We formally define this relation in Figure 5.3. Thus, for any data tree $\mathbf{t}$, position x, and $\varphi \in \Phi$, $\mathit{d\text{-}profile}(\mathbf{t}|_x) \models \varphi$ iff φ holds at $\mathbf{t}|_x$.

A profile summarizes information about a position in terms of data values in its subtrees. In addition, we also define the description of a data value in terms of the different ways by which it can be obtained.

Definition 5.12. The **description** of a data value is the set of states of the automata that can access the data value and it is defined as

$$\mathrm{desc}_{\mathbf{t}}(d) := \{(\mathcal{A}, q) \in \mathit{Aut} \times \mathcal{Q} \mid d \in [\![\mathcal{A}, q]\!]^{\mathbf{t}}\} \in \mathit{Descriptions}, \text{ where}$$
$$\mathit{Descriptions} := \wp(\mathit{Aut} \times \mathcal{Q}).$$

Certificates

For any intersection I, we want to keep track of *which* data values are in $[\![I]\!]$, and *how to access* them in the subtree in order to verify that they belong to every $[\![\mathcal{A}, q]\!]$ in I. How do we decorate the tree in order to have this information at all times? Suppose $\mathbf{t}$ is a data tree and x a position in it. We will develop some branch marking system. For $d \in [\![I]\!]^{\mathbf{t}}$, we mark several downward paths starting in x and ending at a lower positions $y \succeq x$ with $\mathbf{d}(y) = d$. We do this in such a way that for every $(\mathcal{A}, q) \in I$ there is a marked path between x and y such that $\mathbf{d}(y) = d$ and $q \xrightarrow[x,y]{\mathcal{A}} q_f$ with q_f a final state. We will mark every element of this path with the data value 'd' to which it leads. We call this marking a *certificate*. However, markings of paths should not overlap. That we can always have such non-overlapping certificates is not obvious, and it will be the matter of Section 5.3.2.

A **certificate** of a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$ is a partial assignment $\kappa : \mathsf{pos}(\mathbf{t}) \rightharpoonup \mathit{data}(\mathbf{t})$. If κ is undefined for a position x, we write $\kappa(x) = \perp$, and we extend the $\mathrm{desc}_{\mathbf{t}}$ function with $\mathrm{desc}_{\mathbf{t}}(\perp) := \emptyset$ for convenience in the proofs. A certificate κ is said to be *correct* if it has the properties of being *valid* and *inductive* that we will define next.

The **validity** property for a position x ensures that the certificate takes into account all the necessary data values to witness every intersection. That is, that for every intersection I the data values of $[\![I]\!]^{\mathbf{t}|_x}$ are contained either in $\kappa(x)$ or in some $\kappa(x{\cdot}i)$ for a child position $x{\cdot}i$ of x. Since the verifier has only V variables, it is actually sufficient to verify the existence of certificates for up to V data values from $[\![I]\!]^{\mathbf{t}|_x}$. This property, when combined with the inductive property results in each of these data values having a path of certificates that witness each of the elements of I.

Definition 5.13. Given $I \in \mathit{Inters}$, $D \subseteq \mathbb{D}$, $C \subseteq \mathbb{D}\times\mathit{Descriptions}$, and given $d_{cert}, b_{curr}, d_{curr} \in \mathbb{D}$, then

$$\mathit{valid}(\, D,\, I,\, (b_{curr}, d_{curr}, d_{cert}),\, C\,)$$

holds if there are $k = \min(\mathsf{V}, |D|)$ different data values $d_1, \ldots, d_k \in D$ such that for every variable $i \in [k]$ and $(\mathcal{A}, q) \in I$, there exists $(q, b_{curr}, q') \in \mathcal{A}$ and

- $q' \in \mathcal{Q}_F^{\mathcal{A}}$ and $d_{curr} = d_{cert} = d_i$, or
- there is $(d_i, \mathit{Desc}) \in C$, with $(\mathcal{A}, q') \in \mathit{Desc}$.

The **inductivity** property states that for every position x such that $\kappa(x) = d \neq \bot$, if d is in some $[\![\mathcal{A}, q]\!]$, then there must exist a child position $x{\cdot}i$ with certificate d such that d is in $[\![\mathcal{A}, q']\!]^{\mathbf{t}|_{x\cdot i}}$ for some q' in the transition relation of $\mathcal{A}$.

Definition 5.14. Given $\mathit{Desc}_{cert} \in \mathit{Descriptions}$, then

$$\mathit{inductive}(\, \mathit{Desc}_{cert},\, (b_{curr}, d_{curr}, d_{cert}),\, C\,)$$

holds iff for every $(\mathcal{A}, q) \in \mathit{Desc}_{cert}$, there is $(q, b_{curr}, q') \in \mathcal{A}$ such that

- $q' \in \mathcal{Q}_F^{\mathcal{A}}$ and $d_{cert} = d_{curr}$, or
- there is $(d_{curr}, \mathit{Desc}) \in C$ with $(\mathcal{A}, q') \in \mathit{Desc}$.

Definition 5.15. A certificate κ is **correct** if for every position $x \in \mathsf{pos}(t)$ there exists a subset of children $\mathcal{C}_x \subseteq \{x{\cdot}i \mid x{\cdot}i \in \mathsf{pos}(\mathbf{t})\}$ such that

$$\mathit{inductive}(\, \mathrm{desc}_{\mathbf{t}|_x}(\kappa(x)),\, (\mathbf{b}(x), \mathbf{d}(x), \kappa(x)),\, \hat{\mathcal{C}}_x\,)$$

holds, where $\hat{\mathcal{C}}_x = \{(\kappa(y), \mathrm{desc}_{\mathbf{t}|_y}(\kappa(y))) \mid y \in \mathcal{C}_x\}$; and for every intersection $I \in \mathit{Inters}$ the valid property holds,

$$\mathit{valid}(\, [\![I]\!]^{\mathbf{t}|_x},\, I,\, (\mathbf{b}(x), \mathbf{d}(x), \kappa(x)),\, \hat{\mathcal{C}}_x\,)\ .$$

In this context we say that $\hat{\mathcal{C}}_x$ is a valid and inductive subset of children positions of x.

Take any x and $(\mathcal{A}, q) \in I \in \mathit{Inters}$ with $k = \min(\mathsf{V}, [\![I]\!]^{\mathbf{t}|_x})$. The correctness condition implies that there are $d_1, \ldots, d_k$ different data values and $x_1, \ldots, x_k \in \mathsf{pos}(\mathbf{t})$ below x such that for all i: $\mathbf{d}(x_i) = d_i$; for all $x \prec y \preceq x_i$, $\kappa(y) = d_i$; and $q \xrightarrow[x,x_i]{\mathcal{A}} q_f$ for some $q_f \in \mathcal{Q}_F^{\mathcal{A}}$. Note that not every data tree accepted by a DD has a correct certificate. (Think for instance in a tree with branching width 1, in which in order to verify the root's property, two different data values are needed.) Indeed, admissibility of correct certificates is a property shared only by *some* of the trees recognized by a DD. However, in Section 5.3.2 we will show that every nonempty DD accepts a tree which admits a correct certificate. Even more, we show that it accepts a tree where additionally the data values have a particular property, that we define as the *disjoint values property*.

5.3.2 Correct certificates and disjoint values

This section is dedicated to proving two central properties that are essential to obtain a decidability procedure for the DD automata emptiness problem. These properties state that every nonempty DD automaton accepts a tree that (1) admits a correct certificate, and (2) has the disjoint values property —a property that we will define in Section 5.3.2.

Firstly, in Section 5.3.2 we attack the question of wether we can always assume that we have a *correct* certificate, which is a property that is not shared by all runs. The next section is devoted to showing that for any tree $\mathbf{t}$ accepted by a DD automaton there exists a transformation of this tree $\mathbf{t}'$ obtained by duplicating some subtrees, that is also accepted by the automaton.

Secondly, Section 5.3.2 treats the question of wether we can always assume that the run and certificate satisfy the disjoint values property. We will show that given a correctly certified data tree, we can always rearrange the data values in order to meet this property, while preserving the run and the certificate.

We define the function $\hat{\kappa}$ by

$$\hat{\kappa}(x) := \{\kappa(x{\cdot}i) \mid x{\cdot}i \in \mathsf{pos}(\mathbf{t})\} \cup \{\kappa(x)\},$$

that is, $\hat{\kappa}$ assigns to every position x, the set of data values of the certificates of the children of x, as well as of x.

We devote the next two sections to show that the following theorem holds.

Theorem 5.16. *For every $\mathscr{E}$-DD automaton $(\mathcal{R}, \mathcal{V})$ that accepts a non-empty language, there is a tree $\mathbf{t}$ and a correct certificate of $\mathbf{t}$ with the disjoint values property such that $\mathbf{t}$ is accepted by $(\mathcal{R}, \mathcal{V})$.*

Correct certificates

In this section we exploit the particular property of *extensibility* (Definition 5.7) that we imposed to the regular languages used by $\mathscr{E}$-transducers.

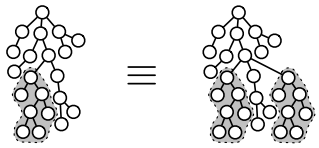

Fig. 5.4: Subtree replication.

As in previous sections, we have that every verifier we consider has a maximum number of relations inside a quantified subformula bounded by R, maximum number of variables bounded by V, the number of automata is K, and the number of states in any automaton is bounded by N.

Our first observation is that if we duplicate a subtree of a data tree as in Figure 5.4, the values of the certificates do not change, and either both trees are accepted or both rejected by any verifier. The following easy proposition is given without a proof.

Proposition 5.17. *Given a data tree* $\mathbf{t}$*, and given a position* $x{\cdot}i \in \mathsf{pos}(\mathbf{t})$ *consider the last index* l *such that* $x{\cdot}l \in \mathsf{pos}(\mathbf{t})$*. Let* $\mathbf{t}'$ *be the follwing data tree that results from duplicating the subtree* $\mathbf{t}|_{x\cdot i}$ *in* $\mathbf{t}$*. We define* $\mathbf{t}' := (\mathbf{t} \circ f_x)$*, where* f_x *is the following surjective function.*

$$f_x : \mathsf{pos}(\mathbf{t}) \cup \{x{\cdot}(l+1){\cdot}y \mid x{\cdot}i{\cdot}y \in \mathsf{pos}(\mathbf{t})\} \to \mathsf{pos}(\mathbf{t})$$

$$f_x(z) = \begin{cases} z & \text{if } z \not\geq x{\cdot}(l+1) \\ x{\cdot}i{\cdot}y & \text{if } z = x{\cdot}(l+1){\cdot}y \end{cases}$$

Then, for every $z \in \mathsf{pos}(\mathbf{t}')$*, and* $d \in \mathbb{D}$*,* $desc_{\mathbf{t}'|_z}(d) = desc_{\mathbf{t}|_{f_x(z)}}(d)$*.*

This can be also extended to $\mathcal{R}$-transducer runs, always by replicating subtrees. Here, the *extensibility* of the regular languages of $\mathcal{R}$ will be of utmost importance.

Proposition 5.18. *If we have*

- *a* $\mathscr{E}$*-DD automaton* $(\mathcal{R}, \mathcal{V})$*,*
- *a data tree* $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$*,*
- *a position* $x \in \mathsf{pos}(\mathbf{t})$ *with* $l = \#children(\mathbf{t}, x)$*,*
- *a run* ρ *of* $\mathcal{R}$ *on* $\mathbf{t}$
- *a regular language* $\mathcal{L}$ *such that* $(\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L}) \in \delta$ *and* $\rho(x{\cdot}1) \cdots \rho(x{\cdot}l) \in \mathcal{L}$
- *an extension* $\dot{q}_1 \cdots \dot{q}_n \in \mathcal{L}$ *of* $\rho(x{\cdot}1) \cdots \rho(x{\cdot}l)$ *(where of course* $n > l$*), and any* surjective *function* $h_x : [n] \to [l]$ *such that* $\dot{q}_i = \rho(x{\cdot}h_x(i))$ *for every* $i \in [n]$*,*

- *a data tree defined as* $\mathbf{t}' := (\mathbf{t} \circ f_x)$*, whose set of positions is*

$$P = \{x{\cdot}j{\cdot}y \mid x{\cdot}h_x(j){\cdot}y \in \mathsf{pos}(\mathbf{t})\} \cup \{z \mid z \in \mathsf{pos}(\mathbf{t}), x \not\prec z\}$$

and where f_x *is as follows.*

$$f_x : P \to \mathsf{pos}(\mathbf{t})$$
$$f_x(y) = \begin{cases} y & \text{if } x \not\prec y \\ x{\cdot}h_x(j){\cdot}y & \text{if } y = x{\cdot}j{\cdot}y \end{cases}$$

Then:

(1) For every position $y \in \mathsf{pos}(\mathbf{t}')$ *and data value* d*,* $desc_{\mathbf{t}'|_y}(d) = desc_{\mathbf{t}|_{f_x(y)}}(d)$*.*

(2) $\rho \circ f_x$ *is a run of* $\mathcal{R}$ *on* $\mathbf{t}'$*.*

Proof. We first show (1). Note that $\mathbf{t}'$ is the result of adding some subtrees $\mathbf{t}|_{x\cdot i_1}, \ldots, \mathbf{t}|_{x\cdot i_\ell}$, where $\ell = \#children(\mathbf{t}', x) - \#children(\mathbf{t}, x)$, plus some possible reordering of the siblinghood. Let $\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_\ell$ be data trees such that $\mathbf{t}_0 = \mathbf{t}$, and $\mathbf{t}_j$ with $j > 0$ is the result of adding $\mathbf{t}|_{x\cdot i_j}$ as a last child of x in $\mathbf{t}_{j-1}$. Observe that we can apply Proposition 5.17 to each of the pair of trees $(\mathbf{t}_0, \mathbf{t}_1), (\mathbf{t}_1, \mathbf{t}_2), \ldots, (\mathbf{t}_{\ell-1}, \mathbf{t}_\ell)$ obtaining that for every position $y \in \mathsf{pos}(\mathbf{t}_j)$ and $j \in [\ell]$,

$$\mathrm{desc}_{\mathbf{t}_j|_y}(d) = \mathrm{desc}_{\mathbf{t}_{j-1}|_{g_x^j(y)}}(d)$$

where $g_x^j : \mathsf{pos}(\mathbf{t}_j) \to \mathsf{pos}(\mathbf{t}_{j-1})$ is the surjective function given by Proposition 5.17. Then, the function $g_x = g_x^1 \circ \cdots \circ g_x^\ell$ is surjective onto $\mathsf{pos}(\mathbf{t})$, and for every position $y \in \mathsf{pos}(\mathbf{t}_\ell)$, $\mathrm{desc}_{\mathbf{t}_\ell|_y}(d) = \mathrm{desc}_{\mathbf{t}|_{g_x(y)}}(d)$. Now, note that $\mathbf{t}_\ell$ and $\mathbf{t}'$ (and equivalently f_x and g_x) differ only in the order of the subtrees of position x. Since, by definition, desc is invariant under reordering of siblings, it follows that (1) holds.

To show (2), first consider any position of the form $x{\cdot}i \in \mathsf{pos}(\mathbf{t}')$, and observe that $\mathbf{t}'|_{x\cdot i}$ has a run $(\rho \circ f_x)|_{x\cdot h_x(i)}$. This is a consequence of $\mathbf{t}'|_{x\cdot i}$ and $\mathbf{t}|_{x\cdot h_x(i)}$ being identical, and the automaton being bottom-up. Secondly, we prove that $\mathbf{t}'|_x$ has a run $(\rho \circ f_x)|_x$. This is because $\rho(x{\cdot}h_x(1)) \cdots \rho(x{\cdot}h_x(n)) \in \mathcal{L}$ by hypothesis and hence $(\rho \circ f_x)|_x(1) \cdots (\rho \circ f_x)|_x(n) \in \mathcal{L}$, where $n = \#children(\mathbf{t}', x)$. We can then apply the transition $(\rho(x), \mathbf{a}(x), \mathbf{b}(x), \mathcal{L})$ obtaining $\rho(x)$ at the root. Finally, since x has the same state in ρ and $\rho \circ f_x$, and the trees $\mathbf{t}$ and $\mathbf{t}'$ are isomorphic except for perhaps the subtree rooted on x, it follows that $\rho \circ f_x$ is a run on $\mathbf{t}'$. □

Remark 5.19. Note that Proposition 5.18 implies that if $\mathbf{t}$ is accepted by a verifier, then $\mathbf{t}'$ is also accepted, and idem for the transducer.

Now we can show that we can always restrict to *correct* certificates.

Proposition 5.20. *Every nonempty* $\mathscr{E}$*-DD automaton accepts some data tree with a* correct *certificate.*

Proof. Let us fix a DD automaton $(\mathcal{R}, \mathcal{V})$. We will show the following statement.

Claim 5.20.1. Given a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$, a run ρ on $\mathbf{t}$, and a data value $e \in \mathit{data}(\mathbf{t}) \cup \{\bot\}$ such that $\mathbf{t}$ is accepted by $\mathcal{V}$, there exists another tree $\mathbf{t}'$ with the same height as $\mathbf{t}$, with run ρ' and a *correct* certificate κ' such that $\kappa'(\epsilon) = e$, $\rho'(\epsilon) = \rho(\epsilon)$ and $\mathbf{t}'$ is accepted by $\mathcal{V}$ according to ρ'. Further, for every $d \in \mathbb{D}$, $\mathrm{desc}_{\mathbf{t}}(d) = \mathrm{desc}_{\mathbf{t}'}(d)$.

Since $(\mathcal{R}, \mathcal{V})$ is nonempty, there exists a data tree $\mathbf{t}$ with an accepting run ρ. If we take $e = \bot$, by Claim 5.20.1 we obtain a data tree $\mathbf{t}'$ with an accepting run ρ', and a correct certificate κ', proving the statement of the proposition.

of Claim 5.20.1. We proceed by induction on the height of $\mathbf{t}$.

The base case consists in showing that the property holds for a tree $\mathbf{t}$ of height 0 consisting of only one position: ϵ. In this case it suffices to define $\mathbf{t}' = \mathbf{t}$, $\kappa'(\epsilon) = e$ and $\rho' = \rho$, and all the properties are trivially met.

For the inductive case, suppose that for all trees of height at most h the statement holds. Let $\mathbf{t}$ be of height $h + 1$ with a run ρ and let $e \in \mathit{data}(\mathbf{t}) \cup \{\bot\}$. We need to provide a tree $\mathbf{t}'$ run ρ' and certificate κ' with the desired properties.

The basic idea is to build $\mathbf{t}'$ by replicating some subtrees of $\mathbf{t}$ to allow to have sufficently non-overlaping paths to generate the correct certificate. To build such tree and certificate, it is useful to have a function that, given a data value d, a NFA $\mathcal{A}$ and a state q, returns a position x with data value d such that $\mathrm{str}(\epsilon, x) \in L(\mathcal{A}[q])$. If there is no such witnessing position, (*i.e.*, if $d \notin [\![\mathcal{A}, q]\!]^{\mathbf{t}}$) it returns an undefined value. We denote by $\mathsf{posWitness}$ any such function. It has the next property, for every data value d, automaton $\mathcal{A}$ and state q.

- $\mathsf{posWitness}(d, \mathcal{A}, q) \in \{y \in \mathsf{pos}(\mathbf{t}) \mid \mathrm{str}(\epsilon, y) \in L(\mathcal{A}[q]),\ \mathbf{d}(y) = d\}$, or
- $\mathsf{posWitness}(d, \mathcal{A}, q) = \bot$ if $\{y \in \mathsf{pos}(\mathbf{t}) \mid \mathrm{str}(\epsilon, y) \in L(\mathcal{A}[q]),\ \mathbf{d}(y) = d\} = \emptyset$.

Remember that the certificate we build κ' need to have the data value e at the root. In order to verify the *inductivity* condition at the root, we will collect every witness position for the data value e. In some sense we want to gather information about which subtrees $\mathbf{t}|_j$ need to have a certificate with data value e.

$$A = \{(e, j) \mid j \preceq \mathsf{posWitness}(e, \mathcal{A}, q), \mathcal{A} \in \mathit{Aut}, q \in \mathcal{Q}, j \in \mathsf{pos}(\mathbf{t})\}$$

In order to build a *valid* certificate at the root, we also must take into account the subtrees $\mathbf{t}|_j$ that are necessary to verify the formulæ for every intersection I.

$$\mathcal{C}_I = \{(d, j) \mid d \in [\![I]\!]^{\mathbf{t}}, j \preceq \mathsf{posWitness}(d, \mathcal{A}, q), (\mathcal{A}, q) \in I, j \in \mathsf{pos}(\mathbf{t})\}$$

We must then witness all the elements of $E = A \cup \bigcup\{\mathcal{C}_I \mid I \in \mathit{Inters}\}$. We build a data tree $\mathbf{t}''$, which is the result of duplicating some of the subtrees of $\mathbf{t}$ so as to have enough space to fit all the necessary witness certificates required by E. We then need at most $|E|$ replications of trees. This is achieved by the

extensibility of the languages corresponding to the transitions of ρ. Let us consider $\dot{q}_1, \ldots, \dot{q}_n$ to be an $|E|$-extension of $\rho_i(1)\cdot\cdots\cdot\rho_i(l) \in \mathcal{L}$ for a language $\mathcal{L}$ such that $(\rho(\epsilon), \mathbf{a}(\epsilon), \mathbf{b}(\epsilon), \mathcal{L}) \in \delta$. We define f_ϵ as in Proposition 5.18, and we define $\mathbf{t}'' = \mathbf{t} \circ f_\epsilon$, $\rho'' = \rho \circ f_\epsilon$. It follows that ρ'' is a run on $\mathbf{t}''$ and by Remark 5.19 $\mathbf{t}''$ is accepted by $\mathcal{V}$.

To define κ', we must make sure that for each element of E there is a corresponding certificate in some child of the root of $\mathbf{t}''$. Of course, there cannot be two certificates in the same child, so we need a way to choose distinct children for distinct elements of E. Let g be a function that chooses, for each element of E, which subtree to use, $g : E \to \{j \mid j \in \mathsf{pos}(\mathbf{t}'')\}$ such that g is injective and $f_\epsilon(g(d, j{\cdot}y)) = j$ for every $(d, j{\cdot}y) \in E$.

We are in a good shape to define a certificate κ' for $\mathbf{t}''$ that satisfies the correctness property at the root. But we need κ' to be correct also at all subtrees of $\mathbf{t}''$. This is given by the inductive hypothesis.

For every subtree $\mathbf{t}''|_i$ of $\mathbf{t}''$ we apply the inductive hypothesis with the data value d_0 such that $g^{-1}(i) = (d_0, y)$ for some y, or with $\perp$ otherwise. We thus obtain a tree $\mathbf{t}'_i$ with a correct certificate κ'_i and run ρ'_i. Finally we build the following tree $\mathbf{t}'$, certificate κ', and run ρ' satisfying all the properties.

$$\mathbf{t}'(x) = \begin{cases} \mathbf{t}(x) & \text{if } x = \epsilon \\ \mathbf{t}'_i(y) & \text{if } x = i{\cdot}y,\ i \in \mathsf{pos}(\mathbf{t}''),\ y \in \mathsf{pos}(\mathbf{t}'_i) \end{cases}$$

$$\kappa'(x) = \begin{cases} e & \text{if } x = \epsilon \\ \kappa'_i(y) & \text{if } x = i{\cdot}y,\ i \in \mathsf{pos}(\mathbf{t}''),\ y \in \mathsf{pos}(\mathbf{t}'_i) \end{cases}$$

$$\rho'(x) = \begin{cases} \rho(x) & \text{if } x = \epsilon \\ \rho'_i(y) & \text{if } x = i{\cdot}y,\ i \in \mathsf{pos}(\mathbf{t}''),\ y \in \mathsf{pos}(\mathbf{t}'_i) \end{cases}$$

ρ' is a run on $\mathbf{t}'$ since it is composed of runs ρ'_i from the subtrees and at the root it satisfies the transition $(\rho'(\epsilon), \mathbf{a}(\epsilon), \mathbf{b}(\epsilon), \mathcal{L}) \in \delta$. κ' is a correct certificate, since the κ_i's are correct certificates for all the subtrees, and κ verifies the inductive and valid conditions at the root. The description of any data value d at the root was not altered since we only applied Proposition 5.18 and the inductive hypothesis that preserve the descriptions. Finally, we can see that $\mathcal{V}$ accepts $\mathbf{t}'$ since it accepts each of its subtrees, and also has the property of the root, since no descriptions of data values were modified. □

□

Disjoint values

We introduce a property concerning the data values of the tree. The idea is that given two disjoint subtrees $\mathbf{t}|_x$, $\mathbf{t}|_y$ with $x \not\geq y$, $y \not\geq x$, the only data values they can share, if any, are those of the certificates of their roots $\kappa(x), \kappa(y)$, or those of some of their children $\kappa(x{\cdot}i), \kappa(y{\cdot}j)$. Remember that these last ones constitute all

the witness data that are necessary to verify the profile at x and y. All other data values can be assumed to occur in only one of the two subtrees. Here we show that for every nonempty DD automaton there is always a tree that can be certified in such a way that this property holds. Next we formalize the *disjoint values property*, which will be an essential property in order to prove our main decidability result of Theorem 5.9.

Definition 5.21. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$ be a data tree recognized by a DD automaton $(\mathcal{R}, \mathcal{V})$. Let κ be a correct certificate. We say that it has the **disjoint values** property if, for every pair of incomparable positions $x, y \in \mathsf{pos}(\mathbf{a} \otimes \mathbf{d})$,

$$data(\mathbf{t}|_x) \cap data(\mathbf{t}|_y) \quad \subseteq \quad \hat{\kappa}(x) \cap \hat{\kappa}(y) \ .$$

We show here that we can always assume the model to have the disjoint values property. The idea is that once we have a correct certificate κ over a data tree $\mathbf{t}$, we know that at any inner node $x \in \mathsf{pos}(\mathbf{t})$, all the important data values to verify *d-profile*$(\mathbf{t}|_x)$ shared between subtrees $\{\mathbf{t}|_{x\cdot i} \mid x\cdot i \in \mathsf{pos}\}$, are those contained in the certificates of the children $\kappa(x\cdot i)$ or $\kappa(x)$. That is, for every x, the only necessary data values to verify its profile (or an ancestor's profile) is in $\hat{\kappa}(x)$. We can then ensure that these are the only data values that may be shared by any two $\mathbf{t}|_{x\cdot i}$, $\mathbf{t}|_{x\cdot j}$.

We now state the important proposition of this section.

Proposition 5.22 ((Disjointness))**.** *Given*

- *a data tree* $\mathbf{t}$ *with a correct certificate* κ*, and* $x \in \mathsf{pos}(\mathbf{t})$*,*
- *a data value* $d_x \in data(\mathbf{t}|_x) \setminus \hat{\kappa}(x)$*,* $d'_x \notin data(\mathbf{t})$*,*
- *the bijective function* $f : \mathbb{D} \to \mathbb{D}$ *such that* $f(d) = d$ *for all* $d \in data(\mathbf{t}) \setminus \{d_x\}$ *and* $f(d_x) = d'_x$*,*
- *the data tree* $\mathbf{t}'$ *with* $\mathsf{pos}(\mathbf{t}') = \mathsf{pos}(\mathbf{t})$ *and a certificate* κ'*, defined as follows.*

$$\mathbf{t}'(y) = \begin{cases} (\mathbf{a}(y), (f \circ \mathbf{d})(y)) & \text{if } y \succeq x \\ \mathbf{t}(y) & \text{otherwise} \end{cases} \qquad \kappa'(y) = \begin{cases} (f \circ \kappa)(y) & \text{if } y \succeq x \\ \kappa(y) & \text{otherwise} \end{cases}$$

Then, for every position $y \in \mathsf{pos}(\mathbf{t})$*,* *d-profile*$(\mathbf{t}|_y) =$ *d-profile*$(\mathbf{t}'|_y)$*, and* κ' *is a correct certificate for* $\mathbf{t}'$*.*

The following trivial lemma will be useful in the sequel.

Lemma 5.23. *Given a data tree* $\mathbf{t}$*,* $I, I' \in$ *Inters, and* $x, y \in \mathsf{pos}(\mathbf{t})$*,* $x \preceq y$*, such that for every* $(\mathcal{A}, q) \in I$ *there is* $(\mathcal{A}, q') \in I'$ *such that* $q \xrightarrow[x,y]{\mathcal{A}} q'$*, we have that* $[\![I']\!]^{\mathbf{t}|_{y\cdot i}} \subseteq [\![I]\!]^{\mathbf{t}|_x}$ *for every* i*.*

Proof. Given $(\mathcal{A}, q') \in I'$, $(\mathcal{A}, q) \in I$ as in the Lemma, and $d \in [\![\mathcal{A}, q']\!]^{\mathbf{t}|_{y\cdot i}}$, we show that $d \in [\![\mathcal{A}, q]\!]^{\mathbf{t}|_x}$. Let z be such that $\mathbf{d}(z) = d$ and $q' \xrightarrow[y\cdot i,z]{\mathcal{A}} q_f$ with $q_f \in \mathcal{Q}_F^{\mathcal{A}}$. By hypothesis, $q \xrightarrow[x,y]{\mathcal{A}} q'$. Hence, $q \xrightarrow[x,y]{\mathcal{A}} q' \xrightarrow[y\cdot i,z]{\mathcal{A}} q_f$, and then $d \in [\![\mathcal{A}, q]\!]^{\mathbf{t}|_x}$.

Now suppose $d \in [\![I']\!]^{\mathbf{t}|_{y\cdot i}}$. Since for all $(\mathcal{A}, q) \in I$ there exists $(\mathcal{A}, q') \in I'$ as before, with $d \in [\![\mathcal{A}, q']\!]^{\mathbf{t}|_{y\cdot i}}$, we can apply the same reasoning, obtaining that $d \in [\![I]\!]^{\mathbf{t}|_x}$. Thus, $[\![I']\!]^{\mathbf{t}|_{y\cdot i}} \subseteq [\![I]\!]^{\mathbf{t}|_x}$. □

of Proposition 5.22. First, suppose that y is such that

(1) $y \succeq x$, or

(2) $x \not\succeq y \not\succeq x$.

Then, $\kappa'|_y$ is a correct certificate for $\mathbf{t}'|_y$. This is because in the case (2), $\mathbf{t}|_y = \mathbf{t}'|_y$ and $\kappa|_y = \kappa'|_y$, and in the case (1), $\mathbf{t}'|_y$ is the result of applying the data bijection f to $\mathbf{t}|_y$, and $\kappa'|_y = f \circ \kappa|_y$.

Suppose then that $y \prec x$. We show that for any $k \leq \mathsf{V}$,

(a) if $|[\![I]\!]^{\mathbf{t}|_y}| \geq k$, then there are k distinct data values $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}|_y} \cap [\![I]\!]^{\mathbf{t}'|_y}$, and

(b) if $|[\![I]\!]^{\mathbf{t}'|_y}| \geq k$, then there are k distinct data values $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}|_y} \cap [\![I]\!]^{\mathbf{t}'|_y}$.

Note that (a) and (b) imply that the profiles of y are the same in $\mathbf{t}$ and $\mathbf{t}'$.

For (a), suppose $|[\![I]\!]^{\mathbf{t}|_y}| \geq k$. By correctness of κ there are k distinct data values $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}|_y}$ and k distinct positions $y\cdot i_1, \ldots, y\cdot i_k \in \mathsf{pos}(\mathbf{t})$ such that there are k downward paths starting at $y\cdot i_1, \ldots, y\cdot i_k$ with respective certificates $d_1, \ldots, d_k$. By the choice of d_x, if $d_x \in \{d_1, \ldots, d_k\}$, it cannot be that any witness position of $d_x \in [\![I]\!]^{\mathbf{t}|_y}$ lies within $\mathbf{t}|_x$, as it would mean that $d_x \in \hat{\kappa}(x)$. Thus, $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}'|_y}$.

For (b), assume there are k distinct data values $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}'|_y}$. If none of these is d'_x, then the certificate κ (which is the same as κ' for these values) shows the paths to witness these data values in $\mathbf{t}|_y$ and then $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}|_y}$. If $d'_x \in [\![I]\!]^{\mathbf{t}'|_y}$, as d'_x only appears in $\mathbf{t}|_x$, there must be an intersection I' such that $d'_x \in [\![I']\!]^{\mathbf{t}'|_x}$, and for every $(\mathcal{A}, q) \in I$ there is $(\mathcal{A}, q') \in I'$ with $q \xrightarrow[y,x']{\mathcal{A}} q'$, for $x = x'\cdot i$. Hence, $f^{-1}(d'_x) = d_x \in [\![I']\!]^{\mathbf{t}|_x}$. Since $d_x \notin \hat{\kappa}(x)$ by hypothesis, this means (by correctness of κ) that d_x is not in any 'small' intersection, and thus $|[\![I']\!]^{\mathbf{t}|_x}| > \mathsf{V}$, which, by Lemma 5.23, implies that $|[\![I]\!]^{\mathbf{t}|_y}| > \mathsf{V}$. Since for all other data values $d'_x \neq d \in [\![I]\!]^{\mathbf{t}'_y}$ we have that $d \in [\![I]\!]^{\mathbf{t}|_y}$, there must be k data values $d_1, \ldots, d_k \in [\![I]\!]^{\mathbf{t}|_y} \cap [\![I]\!]^{\mathbf{t}'|_y}$.

It follows that the certificate κ' is also *correct* for all positions y of $\mathbf{t}'$, since all the intersections I with $|[\![I]\!]^{\mathbf{t}|_y}| \leq \mathsf{V}$ coincide in $\mathbf{t}$ and $\mathbf{t}'$. □

By applying repeatedly Proposition 5.22 we obtain the following result.

Corollary 5.24. *For any data tree* $\mathbf{t} = \mathbf{b} \otimes \mathbf{d}$ *with correct certificate* κ*, there exists another data tree* $\mathbf{t}' = \mathbf{b} \otimes \mathbf{d}'$ *with correct certificate* κ' *such that for every position* $x \in \mathsf{pos}(\mathbf{t})$*,* $d\text{-}profile(\mathbf{t}|_x) = d\text{-}profile(\mathbf{t}'|_x)$ *and* $\mathbf{t}'$ *has the disjoint values property.*

Putting together Proposition 5.20 with Corollary 5.24 we hence verify Theorem 5.16 which was the objective of the current Section 5.3.2.

Now we have all the main ingredients to state the horizontal pumping argument. In the next section we show that every $\mathscr{E}$-DD automaton accepts a data tree whose width is bounded by a fixed function on the size of the automaton, or it does not accept any data tree at all.

5.3.3 Horizontal pumping

We first bound, for any x, the size of the set of children $\mathcal{C}_x$ necessary for any correct certificate.

Lemma 5.25. *For every correct certificate on a data tree* $\mathbf{t}$ *and every position* $x \in \mathsf{pos}(\mathbf{t})$ *there exists a subset* $\mathcal{C}_x$ *of children of* x *that is valid and inductive with respect to the certificate, and* $|\mathcal{C}_x| \leq \mathsf{L}$*, with* $\mathsf{L} = (\mathsf{K}.\mathsf{N})^{\mathsf{R}}.p(\mathsf{V}, \mathsf{R}, \mathsf{K}, \mathsf{N})$ *for some fixed polynomial* p*.*

Proof. We show that $\mathsf{L} := (\mathsf{K}.\mathsf{N})^{\mathsf{R}}.(\mathsf{V}.\mathsf{R}) + \mathsf{K}.\mathsf{N}$ is a good upper-bound. This is a direct consequence of the cardinality of *Inters* and the number of possible pairs $(\mathcal{A}, q)$. To verify the validity condition as in Definition 5.13, for every $I \in Inters$ out of at most $(\mathsf{K}.\mathsf{N})^{\mathsf{R}}$, we might need to add up to $\mathsf{V}.\mathsf{R}$ certificates in the children to witness I: one for each of the V data values and each $(\mathcal{A}, q) \in I$. On the other hand, to verify the inductivity condition on $\kappa(x)$, we need at most one extra child witness for each automaton and state in $\mathrm{desc}_{\mathbf{t}|_x}(\kappa(x))$, which can have no more than $\mathsf{K}.\mathsf{N}$ elements. □

Lemma 5.26 ((Horizontal pumping))**.** *Let*

- $\mathbf{t}$ *be a data tree accepted by a DD automaton* $(\mathcal{R}, \mathcal{V})$*,*
- (τ, ρ, ρ_h) *be a detailed* $\mathcal{R}$*-run and* κ *a correct certificate,*
- $x \in \mathsf{pos}(\mathbf{t})$ *and* $\mathcal{C}_x \subseteq \{x{\cdot}1, \ldots, x{\cdot}\#children(\mathbf{t}, x)\}$ *be a valid and inductive subset of children positions of* x*,*
- $x{\cdot}i, x{\cdot}(i+1), \ldots, x{\cdot}(i+j)$ *be a set of consecutive siblings with the following properties:*
  - $\rho_h(x{\cdot}i) = \rho_h(x{\cdot}j)$*,*
  - *none of the positions* $x{\cdot}(i+1), \ldots, x{\cdot}(i+j)$ *is in* $\mathcal{C}_x$*.*

Then the data tree $\mathbf{t}'$ *resulting from the deletion of* $\mathbf{t}|_{x\cdot(i+1)}, \ldots, \mathbf{t}|_{x\cdot(i+j)}$

$$\mathbf{t}' = \mathbf{t} \circ f, \qquad f(y) = \begin{cases} y & \text{if } y \in \mathsf{pos}(\mathbf{t}) \text{ and } \forall t : y \not\geq x\cdot(i+t) \\ x\cdot(i+t)\cdot z & \text{if } x\cdot(i+j+t)\cdot z \in \mathsf{pos}(\mathbf{t}) \end{cases}$$

is also accepted, with the detailed run $(\tau \circ f, \rho \circ f, \rho_h \circ f)$ *and correct certificate* $\kappa \circ f$ *resulting by the deletion of the positions* $x\cdot(i+1), \ldots, x\cdot(i+j)$.

Proof. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ and $l = \#\mathit{children}(\mathbf{t}, x)$. The run

$$\rho_h(x\cdot 1) \cdots \rho_h(x\cdot i)\rho_h(i+j+1) \cdots \rho_h(l)$$

continues to be an accepting run for the NFA $\mathcal{B}_x$ corresponding to $\tau(x)$ on the string $\rho(x\cdot 1) \cdots \rho(x\cdot i)\rho(i+j+1) \cdots \rho(l)$, and hence $\tau(x)$ correctly labels the position with a valid transition. On the other hand, κ continues to be correct since all the necessary elements of $\mathcal{C}_x$ are preserved after the pruning. □

Corollary 5.27. *If there exists a data tree recognized by a DD* $(\mathcal{R}, \mathcal{V})$ *with a correct certificate, then there also exists a tree with bounded width which is also recognized by* $(\mathcal{R}, \mathcal{V})$ *with a correct certificate. The bound is* $(\mathsf{K}.\mathsf{N})^{\mathsf{R}}.p(\mathsf{V}, \mathsf{R}, \mathsf{K}, \mathsf{N}, |\tilde{Q}|)$ *for some polynomial* p.

Proof. We show that it can be bounded by

$$\begin{aligned} \mathsf{W} &:= (|\tilde{Q}| - 1).(\mathsf{L} + 1) \quad \text{which by definition of } \mathsf{L} \\ &= (|\tilde{Q}| - 1).((\mathsf{K}.\mathsf{N})^{\mathsf{R}}.p'(\mathsf{V}, \mathsf{R}, \mathsf{K}, \mathsf{N}) + 1) \\ &= (\mathsf{K}.\mathsf{N})^{\mathsf{R}}.p(\mathsf{V}, \mathsf{R}, \mathsf{K}, \mathsf{N}, |\tilde{Q}|) \end{aligned} \tag{5.2}$$

for some polynomials p, p'. Given a maximal sequence of siblings $x\cdot 1, \ldots, x\cdot l$, we already established in Lemma 5.25 that a valid and inductive subset $\mathcal{C}_x$ needs no more than L elements to preserve the correctness of the certificate. Hence, there can be at most $\mathsf{L}+1$ zones of consecutive positions not in $\mathcal{C}_x$, and each one of them must have at most $|\tilde{Q}| - 1$ elements. Should it have more, then there are at least two repeating elements with the same horizontal configuration by the pigeonhole principle, and we can then apply Lemma 5.26 and remove some of the siblings. □

Remark 5.28. The bound of Corollary 5.27 also holds for trees with the disjoint values property, since this property is preserved when a subtree is removed.

We just showed how we can bound the width of a tree with a correct certificate. Unfortunately, bounding the height of the tree is not as simple. Here we will need to make use of the properties showed in Section 5.3.2.

If a run is such that it can be decorated with a correct certificate that has the disjoint values property, we can show that the acceptance or not of the tree can be decided by inspecting only some *local* conditions between every inner node and its children. This will be the object of the next section, where we will prove that these local properties can be tested in 2ExpTime.

5.3.4 The emptiness algorithm

In this section we show how to label each node of the tree with some finite information (that we call *tree configuration*). We do this in such a way that testing whether a data tree is accepted or not by a DD automaton amounts to verifying a local property between a node's configuration and the configurations of its children, for every node of the tree. The configuration depends solely on the certificate of the root and its children, and on the state of the run of the transducer. There is a doubly exponential number in the size of the automaton of such configurations. This fact, together with the bounded width of the tree we showed in Corollary 5.27, leads to a decision procedure to test for emptiness. The algorithm runs in 2ExpTime considering R as a parameter, or ExpTime if R is taken as a constant.

By Theorem 5.16, we assume for the rest of this section that we are always working with a data tree $\mathbf{t}$ equipped with: an accepting run ρ of $\mathcal{R}$ on $\mathbf{t}$, and a correct certificate κ on $\mathbf{t}$, under the disjoint values property.

In the next section we define the configurations that we associate to each node. We will show an algorithm to test if there is a tree with an accepting configuration at the root. This algorithm heavily relies on the fact that the tree is ranked, given by Corollary 5.27. The correctness of this algorithm will be a consequence of the disjoint values property and the admisibility of correct certificates shown in Section 5.3.2.

Configurations

Next, we define a configuration for a data tree. The idea is that each position x of a data tree $\mathbf{t}$ is associated with the configuration for $\mathbf{t}|_x$. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$. We describe what a configuration looks like for a A **tree configuration** contains the state of the transducer's run, the data value of the root, the root's certificate, the children's certificates, and the description of all the data values of the certificates it contains. In the following definition remember that $\hat{\kappa}(x)$ denotes the set consisting of $\kappa(x)$ and all $\kappa(x')$ for x' a children of x, and that $\rightharpoonup$ defines partial functions.

$$TConfigs = \dot{Q} \times \mathbb{D} \times \mathbb{D} \times \wp_{\leq \mathsf{W}}(\mathbb{D}) \times (\mathbb{D} \rightharpoonup \wp(Aut \times \mathcal{Q}))$$
$$tconfig(\mathbf{t}, \rho, \kappa) = \big(\rho(\epsilon),\, \mathbf{d}(\epsilon),\, \kappa(\epsilon),\, \hat{\kappa}(\epsilon),\, \{d \mapsto \mathrm{desc}_{\mathbf{t}}(d) \mid d \in \hat{\kappa}(\epsilon)\} \big)$$

Although the configurations contain data values, it is not important to know the concrete data values. We are only interested in the classes of equivalence modulo equality contained in the configuration. This is sensible, since the model of automata presented can only test for data equality or inequality. Later on, we will see that this means that we can substitute $\mathbb{D}$ with a *finite* alphabet.

The objective is to prove that if we are given a tree with a run and certificate, we can deduce the configuration of the root by inspecting only the configurations of the immediate subtrees. And vice versa, if we are given a forest of trees with their respective runs and configurations, and a configuration that is compatible

with them (in a sense that will be described below), we can then build a witness data tree with the configuration in question.

Given a DD automaton $(\mathcal{R}, \mathcal{V})$, what conditions on the configurations do we need to check? To abstract these conditions we define an entailment relation that checks whether the root configuration can be deduced from the configurations of the children

$$\vdash \subseteq (\mathit{TConfigs})^* \times \mathit{TConfigs}\ .$$

We give the conditions for $M \vdash (\dot{q}_0, d_0, c_0, C_0, \alpha_0)$ to hold, with $(\dot{q}_0, d_0, c_0, C_0, \alpha_0)$ the configuration of the root and $M = (\dot{q}_1, d_1, c_1, C_1, \alpha_1) \cdots (\dot{q}_m, d_m, c_m, C_m, \alpha_m)$ the configurations of the immediate subtrees.

We define that $M \vdash (\dot{q}_0, d_0, c_0, C_0, \alpha_0)$ holds if the following conditions are satisfied. The conditions, although lengthy, are straightforward. They are necessary and sufficient to have a tree where $(\dot{q}_0, d_0, c_0, C_0, \alpha_0)$ is the configuration of the root, and M are the configurations of the children of the root. They can be informally described as follows: (i) $\vdash$ is consistent with the run of the transducer; (ii) c_0 is a fresh data value (*i.e.*, a data value that is not in M), or a data value equal to some c_i; (iii) every data value c_i is contained in C_0; (iv) the c_i's have the *validity* property for every intersection; (v) c_0 and the c_i's have the *inductivity* property; (vi) α_0 is obtained from the description at the children configuration α_i's, by applying all possible transitions from any of the automata; (vii) the root satisfies the verifier's formula. In the next definition, we use a function

$$f : (\mathit{TConfigs})^* \to (\mathbb{B}\times\mathbb{D}) \to \mathbb{D} \to \wp(\mathit{Aut}\times\mathcal{Q})$$

such that $f(M)(b, d_0)(d) = A$ if A is the data description of d, deduced from the root's letter and datum (b, d_0), and the configurations of the children positions M.

$$\begin{aligned} f(M)(b, d_0)(d) = \{(\mathcal{A}, q) \mid i \in [m], d \in C_i, (\mathcal{A}, q') \in \alpha_i(d), (q, b, q') \in \mathcal{A}\} \cup \\ \{(\mathcal{A}, q) \mid d = d_0, (q, b, q') \in \mathcal{A}, q' \in \mathcal{Q}_F^{\mathcal{A}}\} \end{aligned}$$

We also use

$$h : (\mathit{TConfigs})^* \to (\mathbb{B}\times\mathbb{D}) \to \mathit{Inters} \to \wp(\mathbb{D})$$

where $h(M)(b, d_0)(d)$ is the set of all data values in M that satisfy the intersection I at the root.

$$h(M)(b, d_0)(I) = \{d \mid d \in \{d_0\} \cup C_1 \cup \cdots \cup C_m,\ I \subseteq f(M)(b, d_0)(d)\}$$

The conditions are as follows.

(i) There exists $(\dot{q}_0, a, b, \mathcal{L}) \in \delta$ with $\dot{q}_1 \cdots \dot{q}_m \in \mathcal{L}$. For the remaining conditions let us fix $f' = f(M)(b, d_0)$ and $h' = h(M)(b, d_0)$.

(ii) Either $c_0 = c_i$ for some $i \in [m]$, or $c_0 = d_0$ otherwise.

(iii) $C_0 = \{c_0, c_1, \ldots, c_m\}$.

(iv) For every $I \in \mathit{Inters}$, the validity condition (Definition 5.13) holds,

$$\mathit{valid}(\hbar'(I), I, (b, d_0, c_0), \cup_{i\in[m]}\{(c_i, \alpha_i(c_i))\}) \ .$$

(v) The inductive condition (Definition 5.14) holds,

$$\mathit{inductive}(\alpha_0(c_0), (b, d_0, c_0), \cup_{i\in[m]}\{(c_i, \alpha_i(c_i))\}) \ .$$

(vi) $\alpha_0 = \{d \mapsto f'(d) \mid d \in C_0\}$.

(vii) $\{I \mapsto |\hbar'(I)|_{\leq \mathsf{v}}\} \models \mathsf{v}(b)$, where v is the verifier's mapping, and $\models$ is as defined in Figure 5.3 on page 83.

We remark that in the above definition we do not exclude the case where $M = \epsilon$. In fact, this case corresponds to the configurations of the leaves.

Correctness of $\vdash$

We verify that this is indeed enough to have a decision procedure. Below, the *soundness* Proposition 5.29 states that, given a sequence of trees with their respective configurations, for any configuration entailed from these we can find a tree that witnesses this configuration. On the other hand, the *completeness* Proposition 5.32 states that for any tree, the configurations of the immediate subtrees entail the configuration of the tree.

Proposition 5.29 ((soundness))**.** *Given m data trees with $\mathcal{R}$-runs, correct certificates and the disjoint values property*

$$\mathbf{t}_1, \rho_1, \kappa_1, \ldots, \mathbf{t}_m, \rho_m, \kappa_m$$

with $\mathbf{t}_i = \mathbf{a}_i \otimes \mathbf{b}_i \otimes \mathbf{d}_i \in \mathit{Trees}(\mathbb{A}\times\mathbb{B}\times\mathbb{D})$ such that ρ_i is a run for $\mathbf{a}_i \otimes \mathbf{b}_i$, $\mathcal{V}$ accepts $\mathbf{b}_i \otimes \mathbf{d}_i$, and

$$\forall i \neq j \quad \mathit{data}(\mathbf{t}_i) \cap \mathit{data}(\mathbf{t}_j) \ \subseteq \ \hat{\kappa}_i(\epsilon) \cap \hat{\kappa}_j(\epsilon) \ . \tag{5.3}$$

If

$$M = (\dot{q}_1, d_1, c_1, C_1, \alpha_1) \cdots (\dot{q}_m, d_m, c_m, C_m, \alpha_m) \vdash T$$

with $(\dot{q}_i, d_i, c_i, C_i, \alpha_i) = \mathit{tconfig}(\mathbf{t}_i, \rho_i, \kappa_i)$ for every i, then there exists a tree $\mathbf{t}_0 \in \mathit{Trees}(\mathbb{A}\times\mathbb{B}\times\mathbb{D})$ with run ρ_0 and correct certificate κ_0 with the disjoint values property, such that it is accepted by $\mathcal{V}$, and

$$\mathit{tconfig}(\mathbf{t}_0, \rho_0, \kappa_0) = T.$$

Before going into the details of the proof, we need to state some necessary lemmas (5.30, 5.31). Take any tree $\mathbf{t}_0$ with root (a, b, d_0) and subtrees $\mathbf{t}_1, \ldots, \mathbf{t}_m$, such that

$$\text{either} \quad d_0 \in C_1 \cup \cdots \cup C_m \quad \text{or} \quad d_0 \notin \cup_{i\in[m]} \mathit{data}(\mathbf{t}_i) \ . \tag{5.4}$$

Let us fix $f' = f(M)(b, d_0)$ and $\hbar' = \hbar(M)(b, d_0)$.

Lemma 5.30. *For any $d \in C_1 \cup \cdots \cup C_m$, $f'(d) = desc_{\mathbf{t}_0}(d)$.*

Proof. The fact that $d \in C_1 \cup \cdots \cup C_m$ implies that $d \in \hat{\kappa}_i(\epsilon)$ for some i. Then, by hypothesis (5.3), if $d \in data(\mathbf{t}_j)$ for some $j \in [m]$, $j \neq i$, then $d \in \hat{\kappa}_j(\epsilon)$. This means that we have a complete description of d at every element of M: for every $j \in [m]$ either $\alpha_j(d) = \mathrm{desc}_{\mathbf{t}_j}(d)$, or $\alpha_j(d) = \bot$ and $\mathrm{desc}_{\mathbf{t}_j}(d) = \emptyset$. Hence, by definition of f' we have: $f'(d) = \mathrm{desc}_{\mathbf{t}_0}(d)$. □

Lemma 5.31. *For any intersection I and $k \leq \mathsf{V}$, $|[\![I]\!]^{\mathbf{t}_0}| \geq k$ iff $|\hbar'(I)| \geq k$.*

Proof. The right-to-left implication is a direct consequence of the definition of $\hbar'(I)$. For the left-to-right implication, suppose that $|[\![I]\!]^{\mathbf{t}_0}| \geq k$. Take $d \in [\![I]\!]^{\mathbf{t}_0}$. We argue that if d appears in some $\hat{\kappa}_i(\epsilon)$, then it is in $\hbar'(I)$. This is because all the subtrees $\mathbf{t}_i$ with $d \in data(\mathbf{t}_i)$ must have $d \in C_i$, as a consequence of the hypothesis (5.3). Hence we have a *complete* description of d at every subtree, and f' yields the correct description of d at the root by Lemma 5.30, which implies that $d \in \hbar'(I)$ by definition of $\hbar$.

Suppose on the other hand that d is not in $\hat{\kappa}_i(\epsilon)$ for any i. Since every κ_i is correct, this means that d does not appear in any small intersection $[\![I']\!]$ of fewer than V data values. Hence, in particular $|[\![I]\!]^{\mathbf{t}_0}| \geq \mathsf{V}$. By hypothesis (5.3), we know that d can appear in at most one tree $\mathbf{t}_i$, or at the root. We consider both cases.

If d appears only at the root, it means that $d = d_0$, and for every $(\mathcal{A}, q) \in I$ there is $q' \in \mathcal{Q}_F^{\mathcal{A}}$ with $(q, b, q') \in \mathcal{A}$ and this is taken into account by the definition of $\hbar'(I)$. Hence, $d \in \hbar'(I)$.

If d appears only in $\mathbf{t}_i$, this means that there is an intersection I' such that $d \in [\![I']\!]^{\mathbf{t}_i}$ and for every $(\mathcal{A}, q) \in I$ there is $(\mathcal{A}, q') \in I'$ with $(q, b, q') \in \mathcal{A}$. Note that if $|[\![I']\!]^{\mathbf{t}_i}| < \mathsf{V}$ then $\hat{\kappa}_i(\epsilon) \supseteq [\![I']\!]^{\mathbf{t}_i}$. Since we had assumed that $d \notin \hat{\kappa}_i(\epsilon)$, we have that $|[\![I']\!]^{\mathbf{t}_i}| \geq \mathsf{V}$. Then, there are V data values $d_1, \ldots, d_{\mathsf{V}} \in \hat{\kappa}_i(\epsilon) \cap [\![I']\!]^{\mathbf{t}_i}$. By Lemma 5.23, $[\![I']\!]^{\mathbf{t}_i} \subseteq [\![I]\!]^{\mathbf{t}_0}$ and thus $d_1, \ldots, d_{\mathsf{V}} \in [\![I]\!]^{\mathbf{t}}$. This means that $d_1, \ldots, d_{\mathsf{V}}$ are described in M and hence that $d_1, \ldots, d_{\mathsf{V}} \in \hbar'(I)$ and $|\hbar'(I)| \geq \mathsf{V}$. Finally, note that d cannot appear in some $\mathbf{t}_i$ *and* at the root by hypothesis (5.4). □

of Proposition 5.29. We show that there exists $(a, b, d_{cert}) \in \mathbb{A} \times \mathbb{B} \times \mathbb{D}$ such that the tree with (a, b, d_{cert}) at the root and $\mathbf{t}_1, \ldots, \mathbf{t}_m$ as immediate subtrees has a correct certificate κ_0 with the disjoint values property, and a $\mathcal{R}$-run ρ_0, such that the configuration of the tree is T.

We first define the run. Let $T = (\dot{q}_0, d_0, d_{cert}, C_0, \alpha_0)$. As we remarked before, we are only interested in the classes of equivalence of $T, T_1, \ldots, T_m$ modulo renaming of data values. So that we can always assume that d_0 is such that hypothesis (5.4) holds: if $d_0 \notin C_1 \cup \cdots \cup C_m$, we simply assume that d_0 is any value not contained in $data(\mathbf{t}_1) \cup \cdots \cup data(\mathbf{t}_m)$.

By condition (i) of the entailment definition, there is some transition $(\dot{q}_0, a, b, \mathcal{L})$ of $\mathcal{R}$ such that $\rho_1(\epsilon) \cdots \rho_m(\epsilon) \in \mathcal{L}$. Let $\mathbf{t}_0 \in Trees(\mathbb{A} \times \mathbb{B} \times \mathbb{D})$, defined by $\mathbf{t}_0(\epsilon) = (a, b, d_0)$ and $\mathbf{t}_0(i \cdot x) = \mathbf{t}_i(x)$ for $i \in [m]$. Hence the run ρ_0 defined as $\rho_0(\epsilon) = \dot{q}_0$, $\rho_0(i \cdot x) = \rho_i(x)$ is a valid $\mathcal{R}$-run on $\mathbf{t}_0$.

We now show that T is indeed a configuration that corresponds to $\mathbf{t}_0$. By condition (ii) either $d_{cert} = d_0$ for some fresh data value (i.e., not appearing in any subtree, by hypothesis (5.4)), or $d_{cert} = \kappa_i(\epsilon)$ for some $i \in [m]$. In the first case we trivially have

$$\mathrm{desc}_{\mathbf{t}_0}(d) = \{(\mathcal{A}, q) \mid (q, b, q') \in \mathcal{A}, q' \in \mathcal{Q}_F^{\mathcal{A}}\} = \alpha_0(d)$$

by definition of α_0 (condition (vi)) and definition of $f'(d_{cert})$. In the second case, we have by Lemma 5.30 that $f'(d_{cert})$ correctly yields the description of d_{cert} at the root and we verify $\mathrm{desc}_{\mathbf{t}_0}(d_{cert}) = \alpha_0(d_{cert})$. By condition (iii) and using the same reasoning as before, $\kappa_1(\epsilon) \cup \cdots \cup \kappa_m(\epsilon) = C_0$ and for all $d \in C_0$, $\alpha_0(d) = \mathrm{desc}_{\mathbf{t}_0}(d)$. Then, we have that $\mathit{dconfig}(\mathbf{t}_0, \rho_0, \kappa_0) = T$.

We define the certificate κ_0 as $\kappa_0(\epsilon) = d_{cert}$ and $\kappa_0(i \cdot x) = \kappa_i(x)$. From hypothesis (5.3) and the fact that $\mathbf{t}_i, \kappa_i$ have the disjoint values property for every $i \in [m]$, we deduce that $\mathbf{t}_0, \kappa_0$ also has the disjoint values property.

We show that κ_0 is a *correct* certificate for $\mathbf{t}_0$. For the *validity* condition we first have by Lemma 5.31 that the sets $\hbar'(I)$ are good approximations[3] of $[\![I]\!]^{\mathbf{t}_0}$. This, together with condition (iv) and Definition 5.13 gives us that κ_0 must be *valid*, while the *inductivity* property is a consequence of Definition 5.14 and condition (v).

Finally, since $\hbar'(I)$ has the same cardinality up to V elements as $[\![I]\!]^{\mathbf{t}_0}$, $\{I \mapsto |\hbar'|_{\leq \mathsf{V}}\} \models \varphi$ iff φ holds at $\mathbf{t}_0$ for any $\varphi \in \Phi$ used in $\mathcal{V}$. Then, by condition (vii), $\mathbf{t}_0$ is accepted by $\mathcal{V}$. □

Proposition 5.32 ((completeness))**.** *Given a data tree* $\mathbf{t} \in \mathit{Trees}(\mathbb{A} \times \mathbb{B} \times \mathbb{D})$ *with correct certificate* κ*, run* ρ *and the disjoint values property accepted by* $\mathcal{V}$*, then*

$$\mathit{tconfig}(\mathbf{t}|_1, \rho|_1, \kappa|_1) \cdots \mathit{tconfig}(\mathbf{t}|_m, \rho|_m, \kappa|_m) \vdash \mathit{tconfig}(\mathbf{t}, \rho, \kappa)$$

for m *the maximum index such that* $m \in \mathsf{pos}(\mathbf{t})$*.*

Proof. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$. We verify conditions (i) through (vii). Condition (i) is trivially true as ρ is a run on $\mathbf{t}$. Condition (ii) holds, since the $\kappa(\epsilon)$ can be either equal to $\mathbf{d}(\epsilon)$ or equal to some child certificate $\kappa(i)$ as a consequence of κ being a correct certificate. Condition (iii) holds because we have all the certificates from the child configurations. The correctness of the descriptions of condition (vi) for the data values $\{\kappa(1), \ldots, \kappa(m)\}$ is based on the disjoint values property. As a consequence of this property, we have that for every data value $\kappa(i)$ and every $j \in [m]$, if $\kappa(i) \in \mathit{data}(\mathbf{t}|_j)$, then $\kappa(i) \in \hat{\kappa}(j)$. This means that we have a complete description of $\kappa(i)$ for every subtree $\mathbf{t}|_j$. For the case of the root's certificate $\kappa(\epsilon)$ we have two cases. If $\kappa(\epsilon)$ equals some $\kappa(i)$, then we use the same argument as before. Otherwise, we use the *inductivity* of the certificate κ, knowing that by Definition 5.14 if $\kappa(\epsilon)$ is in some $[\![\mathcal{A}, q]\!]^{\mathbf{t}}$, there must be a path of certificates with value $\kappa(\epsilon)$. So, the fact that there are no $\kappa(i) = \kappa(\epsilon)$ means that $\mathrm{desc}_{\mathbf{t}}(\kappa(\epsilon))$ can

[3] They coincide in $\min([\![I]\!]^{\mathbf{t}_0}, \mathsf{V})$ elements.

be completely witnessed locally by inspecting only the root. Then, the description obtained by f' contained in $tconfig(\mathbf{t}, \rho, \kappa)$ is correct. Since condition (vi) holds and $\mathbf{b} \otimes \mathbf{d}$ satisfies $\mathsf{v}(\mathbf{b}(\epsilon))$, then it verifies condition (vii). Finally, conditions (iv) and (v) are consequences of the validity and inductivity properties of κ respectively. □

Remark 5.33. Observe that Corollary 5.27 with Remark 5.28 gives us a bound on the width of a recognized tree with a correct run and the disjoint values property. We can thus restrict ourselves to relations $T_1 \cdots T_t \vdash T$ with $t \leq \mathsf{W}$ from now on. (Remember that W is the bound on the width of the tree given by Corollary 5.27.)

Also, note that the concrete data values of the configuration are not important and can be abstracted away, as soon as they allow to test the conditions of the entailment $\vdash$. For $T, T' \in TConfigs$, let us write $T \sim T'$ if there is a bijection of data values $f : \mathbb{D} \to \mathbb{D}$ such that $f(T) = T'$, where $f(T)$ stands for the replacement of every datum d by $f(d)$ in T. In every configuration there are at most $\mathsf{W}+2$ data values (the root's data value, the root's certificate, and at most W corresponding to the certificates of the children). Then, by Remark 5.33, a '$\vdash$' test involves not more than $\mathsf{W}+1$ configurations and hence we only need at most $(\mathsf{W}+1).(\mathsf{W}+2)$ different data values. Let us define $TConfigs'$ to be $TConfigs$ where instead of having $\mathbb{D}$ as data domain, we have $\{1, \ldots, (\mathsf{W}+1).(\mathsf{W}+2)\}$. Let

$$\mathbb{D}' := \{1, \ldots, (\mathsf{W}+1).(\mathsf{W}+2)\} \tag{5.5}$$

and then let $TConfigs'$ be defined in terms of the restricted set of data values $\mathbb{D}'$,

$$TConfigs' = \dot{Q} \times \mathbb{D}' \times \mathbb{D}' \times \wp_{\leq \mathsf{W}}(\mathbb{D}') \times (\mathbb{D}' \rightharpoonup \wp(Aut \times \mathcal{Q})) \ . \tag{5.6}$$

We then have the following obvious lemma.

Lemma 5.34. *For every $T, T_1, \ldots, T_n \in TConfigs$ such that $T_1 \cdots T_n \vdash T$, there exist $T', T'_1, \ldots, T'_n \in TConfigs'$ with $T \sim T'$ and $T_i \sim T'_i$ for all i, such that $T'_1 \cdots T'_n \vdash T'$.*

This means that, since we are only interested in the tree configurations that can be reached by $\vdash$ modulo isomorphism of data values, we can simply use the tree configurations of $TConfigs'$. These are doubly exponential in R, or singly exponential if R is fixed.

Lemma 5.35. *The number of elements in $TConfigs'$ is exponential in $|\mathcal{R}|$ and $|\mathcal{V}|$ if R is a constant, or doubly exponential otherwise.*

Proof. Now we have at most $|\mathbb{D}'|$ data values, and the function of each configuration is restricted to the data values (at most W) of the configuration. We then have the following bounds by definition of $TConfigs'$ (5.6).

$$|TConfigs'| \ \leq \ |\dot{Q}|.|\mathbb{D}'|.2^{\mathsf{K.N}}.(|\mathbb{D}'|.2^{\mathsf{K.N}})^{\mathsf{W}}$$

$|\mathbb{D}'|$ is polynomial in W, and we then have

$$|\mathit{TConfigs}'| \leq |\dot{Q}|.(p(\mathsf{W}))^{q(\mathsf{W})}.2^{\mathsf{K.N}.r(\mathsf{W})} \tag{5.7}$$

for some polynomials p, q, r. W is exponential only in R, and if R is constant, W is polynomial in $|\mathcal{R}|$ and $|\mathcal{V}|$ by definition (5.2). Thus, the lemma follows. □

As the first step towards an upper bound, we observe that the $\vdash$ relation on $\mathit{TConfigs}'$ can be checked in polynomial time in $\mathsf{Aut}, \mathsf{W}, |\mathcal{R}|$.

Lemma 5.36. *Given $T, T_1, \ldots, T_n \in \mathit{TConfigs}'$ with $n \leq \mathsf{W}$, $T_1 \cdots T_n \vdash T$ can be tested in time $p(\mathsf{Aut}, \mathsf{W}, |\mathcal{R}|)$ for some polynomial p.*

Proof. Condition (i) can be tested in polynomial time in $|\mathcal{R}|$, $|\tilde{Q}|$ and n, where $|\tilde{Q}|$ is polynomial in W. The function $f(T_1 \cdots T_n)(b, d_0)(d)$ can be computed in polynomial time in Aut and $|T_1| + \cdots + |T_n|$ (which is polynomial in W). It follows that the description function α of condition (vi) can be checked in polynomial time in W, since it consists in at most $\mathsf{W}+1$ applications of $f(M)(b, d_0)(d)$. Conditions (ii) and (iii) are easily checked in polynomial time in $|T|, |T_1|, \ldots, |T_n|$. Condition (v) can be tested in polynomial time in W and Aut by the above reasons. For condition (iv), we see that $h(M)(b, d_0)(I)$ can be built in time polynomial, and that for every possible intersection I and value of $h(M)(b, d_0)(I)$ a polynomial condition must be checked, hence, since $|\mathit{Inters}|$ is polynomial in W, testing condition (iv) remains polynomial. □

We now show an algorithm to test whether a tree configuration can be reached by the entailment relation $\vdash$.

Theorem 5.37. *The emptiness problem for DD automata is in* 2ExpTime. *It can be tested in time*

$$(\mathsf{Aut}, |\mathcal{R}|.\mathsf{V.R.K.N})^{p(|\tilde{Q}|,\mathsf{V},\mathsf{R}).r(\mathsf{K.N})^{s(\mathsf{R})}}$$

for p, r and s polynomials.

Proof. We consider a standard reachability algorithm by saturation. We start with an initial empty set of configurations $C_0 = \emptyset$, and we iterate to make it grow to entailed configurations until, after at most $|\mathit{TConfigs}'|$ iterations, the set stabilizes. We then test if some of the reachable tree configurations contains a final state.

The set of initial configurations is $C_0 = \emptyset$. At iteration $i+1$, for every possible $T_0 \in \mathit{TConfigs}'$ we test the following conditions

- $T_0 \notin C_i$
- There exists a (possibly empty) sequence $T_1, \ldots, T_t$ with $t \leq \mathsf{W}$ such that
  - $T_1, \ldots, T_t \in C_i$
  - $T_1, \ldots, T_t \vdash T_0$

and we define $C_{i+1} := C_i \cup C'$, for C' the set of all configurations T_0 satisfying the above conditions. If C' is empty, we stop and return the subset C_i of $TConfigs'$ of $(\vdash)$-reachable configurations.

This algorithm clearly gives as a result the set of configurations of all the accepted trees. For every iteration we might need to perform $|TConfigs'|^{\mathsf{W}+1}$ tests for the $\vdash$ conditions, each one demanding $p(\mathsf{Aut}, \mathsf{W}, |\mathcal{R}|)$ for a polynomial p by Lemma 5.36. Finally, the loop can only be executed $|TConfigs'|$ times. We then have that the total time consumed is

$$|TConfigs'|^{\mathsf{W}+2}.p(\mathsf{Aut}, \mathsf{W}, |\mathcal{R}|) \ .$$

By the inequation (5.7) of Lemma 5.35, and since W is exponential only in R by (5.2), we have that the emptiness problem is bounded by

$$(\mathsf{Aut}.|\mathcal{R}|.\mathsf{V}.\mathsf{R}.\mathsf{K}.\mathsf{N})^{p(|\tilde{Q}|,\mathsf{V},\mathsf{R}).r(\mathsf{K}.\mathsf{N})^{s(\mathsf{R})}}$$

for p, r, s polynomials, and we hence have a 2ExpTime decision procedure. We just proved that the problem of whether a DD automaton accepts a tree $\mathbf{t}$ with a correct certificate with the disjoint values properties, can be tested in 2ExpTime. Then, by Theorem 5.16 the result follows. □

Note that the theorem above implies the Main Theorem 5.9. From the previous proof, we have that the complexity is doubly exponential only in R.

Corollary 5.38. *If* R *is fixed, emptiness of DD automata is in* ExpTime*. It can be tested in time bounded by*

$$(\mathsf{Aut}.|\mathcal{R}|.\mathsf{V}.\mathsf{K}.\mathsf{N})^{p(|\tilde{Q}|,\mathsf{V},\mathsf{K},\mathsf{N})}$$

for some polynomial p*.*

Note that the height of a $\vdash$ derivation is directly related to the height of the tree. Hence, for trees with a fixed height, we can take advantage of this fact by performing an on-the-fly algorithm.

Definition 5.39. We define the **height-h emptiness problem** as follows.

	Height-h emptiness problem
Input:	A DD automaton $(\mathcal{R}, \mathcal{V})$ and a number $h \in \mathbb{N}$.
Output:	Is there a data tree $\mathbf{t}$ of height at most h such that $\mathbf{t}$ is accepted by $(\mathcal{R}, \mathcal{V})$?

For the next theorem let us assume that h is coded in unary.

Theorem 5.40. *If* R *is fixed, the height-h emptiness problem of DD automata is in* PSpace*.*

Proof. Given the previous algoritm of Theorem 5.37, note that all the configurations corresponding to derivations of height i are present once the ith iteration has been executed. Also, note that the height of the derivation and the height of the tree are related in this sense: any tree of height h can be witnessed by a derivation of height h. We can then build a top-down NPSPACE algorithm as follows.

Let us denote by $g(T, l)$ the following procedure for $T \in \mathit{TConfigs}'$, $l \in [0..h]$. First, $g(T, 0)$ yields 'ok' iff $\epsilon \vdash T$, otherwise it fails. $g(T, l+1)$ performs the following tasks. Guesses the tree configurations $T_1, \ldots, T_n$ with $0 \leq n \leq \mathsf{W}$. Checks $T_1 \cdots T_n \vdash T$ in PSPACE, and recursively tests that $g(T_1, l), \ldots, g(T_n, l)$ succeed. Consider now the main algorithm that guesses a root configuration $T \in \mathit{TConfigs}'$ and checks both that it contains a final state $\dot{q} \in \dot{Q}_F$ and that $g(T, h)$ succeeds. This algorithm correctly answers whether there exists a derivation of height at most h that has T at the root.

Notice that the space needed to store the data description function α of a configuration T is bounded by $sp(\alpha) = (\mathsf{W}+1).\log(|\mathbb{D}'|).\mathsf{K}.\mathsf{N}$, for $\mathbb{D}'$ as defined in (5.5). Then the space needed to store a tree configuration is

$$\begin{aligned} sp(T) &= \log(|\dot{Q}|) + \log(|\mathbb{D}'|) + sp(\alpha) \\ &\leq \log(|\dot{Q}|) + p(|\tilde{Q}|, \mathsf{K}, \mathsf{N}, \mathsf{V}) \end{aligned}$$

for some polynomial p. This is a consequence of (5.5) and (5.2), and the fact that R is fixed. If we perform a DFS evaluation strategy of g we only need to store simultaneously at most $(\mathsf{W}+1).h$ configurations and hence the algorithm takes a space polynomial in the size of the automata $\mathcal{R}, \mathcal{V}$ considering R is fixed. It is then immediate that this is a NPSPACE procedure for any configuration and l.

Thus, as NPSPACE = PSPACE the theorem follows. □

Remark 5.41. If $sp(\vdash)$ is the space needed to check $T_1 \cdots T_n \vdash T$, $n \leq \mathsf{W}$, then the algorithm of Theorem 5.40 uses an amount of space bounded by

$$sp(\vdash) + p(h, \log(|\dot{Q}|), |\tilde{Q}|, \mathsf{K}, \mathsf{N}, \mathsf{V})$$

for some polynomial p.

The purpose of the remark above for discriminating the space needed to perform the entailment condition $sp(\vdash)$ will become clear in Section 5.4.2.

5.4 Satisfiability of downward XPath

The first part of this section is consecrated to the proof of decidability of the satisfiability problem for $\mathsf{regXPath}(\downarrow, =)$, the language with the child relation and the Kleene star over path expressions. In later subsections we consider the satisfiability problem of several fragments of this logic, with or without data tests.

5.4.1 Regular-downward XPath

The proof of satisfiability for $\mathsf{regXPath}(\downarrow,=)$ goes by reduction to the emptiness problem of DD automata. Before embarking in the reduction, we need to fix some standard terminology.

Definition 5.42. We say that a set of formulæ S is **closed under subformulæ** if, for every $\varphi \in S$, and ψ a subformula of φ, then $\psi \in S$. S is **closed under simple negations** if, for every $\varphi \in S$ either it is of the form $\neg\psi$, or $\neg\varphi \in S$. We note $S^{\neg}$ to the minimal superset of S closed under subformulæ and simple negations.

A **locally consistent set** over S is a maximal subset of $H \subseteq S$ that satisfies the following conditions:

- For all $\neg\varphi \in S$: $\neg\varphi \in H$ iff $\varphi \notin H$.
- For all $\varphi \wedge \psi \in S$: $\varphi \in H$ and $\psi \in H$ iff $\varphi \wedge \psi \in H$.
- For all $\varphi \vee \psi \in S$: $\varphi \in H$ or $\psi \in H$ iff $\varphi \vee \psi \in H$.

For the rest of the section, we consider the parameters of the DD automata (K, N, R, V, $|\dot{Q}|$, $|\tilde{Q}|$, $|\mathcal{R}|$, Aut, $|\mathcal{V}|$) as defined in Section 5.3.

Theorem 5.43. *Given a formula $\eta \in \mathsf{regXPath}(\downarrow,=)$, a DD automaton $(\mathcal{R},\mathcal{V})$ can be effectively built, such that for any data tree $\mathbf{t}$: $\mathbf{t} \models \eta$ iff $(\mathcal{R},\mathcal{V})$ accepts $\mathbf{t}$.*

Proof. Let η be a formula of $\mathsf{regXPath}(\downarrow,=)$. We build the DD automaton $(\mathcal{R},\mathcal{V})$, where $\mathcal{R}$ tags each node with those sub-node expressions of η that hold at each node, and $\mathcal{V}$ checks that all the data and path expressions are verified.

Let $\mathsf{nsub}(\eta) = \{\gamma \mid \gamma \text{ a node expression in } \mathsf{sub}(\eta)\}$, where $\mathsf{sub}(\eta)$ is the set of subformulæ of η. Let $\mathbb{B}$ be the set of all locally consistent sets over $\{\eta\}^{\neg}$. Let us build $\mathcal{R}$ in such a way that at each step it chooses nondeterministically one element from $\mathbb{B}$ consistent with the current label and outputs it. That is, if $a \in \mathbb{A}$ is the letter of the current position then it outputs any element $b \in \mathbb{B}$ such that $\neg a \notin b$.[4] Note that the transducer only needs one (final) state (*i.e.*, $\dot{Q} = \{\dot{q}\}$) to reflect this behavior. Further, the only regular language used in all its transitions is $\{\dot{q}\}^*$. We can represent $\{\dot{q}\}^*$ with a NFA with a singleton set of states $\tilde{Q}$.

For every path expression α in $\mathsf{sub}(\eta)$, let $\mathcal{A}_\alpha$ be a NFA over the alphabet $\mathbb{B}$ that recognizes α. For example, if $\alpha = [\varphi]\!][\psi]$, then $\mathcal{A}_\alpha$ recognizes $\{b\,b' \mid b,b' \in \mathbb{B}, \varphi \in b, \psi \in b'\}$. It can be built in polynomial time in $|\alpha|$ and $|\mathbb{B}|$ (but note that $|\mathbb{B}|$ is exponential in $|\eta|$). We define the verifier $\mathcal{V}$ to contain the set $Aut = \{\mathcal{A}_\alpha \mid \alpha \text{ path expression of } \mathsf{sub}(\eta)\}$ of automata. The mapping v is defined, for every element $b \in \mathbb{B}$, as the formula that tests all the path formulæ in b.

$$\mathsf{v}(b) = \bigwedge_{\substack{(\neg)\langle\alpha\odot\beta\rangle\in b \\ \odot\in\{=,\neq\}}} (\neg)\,\exists v,v' \,.\, \big(v \odot v' \wedge D_\alpha(v) \wedge D_\beta(v')\big) \quad \wedge \quad \bigwedge_{(\neg)\langle\alpha\rangle\in b} (\neg)\,\exists v \,.\, D_\alpha(v)$$

[4] Note that if $a \notin \mathsf{sub}(\eta)$, then $\neg a \notin b$ and $a \notin b$ for all $b \in \mathbb{B}$.

Intuitively, the only purpose of the transducer $\mathcal{R}$ is to *guess* which subformulæ of η are true and which are false. The real work is done by the verifier $\mathcal{V}$, checking that every formula was correctly guessed.

Suppose that $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ is accepted by $(\mathcal{R}, \mathcal{V})$, *i.e.*, $\mathbf{a} \otimes \mathbf{b} \in \mathcal{R}$, $\mathbf{b} \otimes \mathbf{d} \in \mathcal{V}$. For every position $x \in \mathsf{pos}(\mathbf{t})$ and subformula $\psi \in \mathsf{sub}(\eta)$, we show that $\psi \in \mathbf{b}(x)$ if and only if $x \in [\![\psi]\!]^{\mathbf{a}\otimes\mathbf{d}}$. We proceed by induction on the size of ψ. The base case is when ψ is a test for a label. This is immediate by the definition of $\mathcal{R}$, which preserves the label: $\mathbf{a}(x) \in \mathbf{b}(x)$ if $\mathbf{a}(x) \in \mathsf{sub}(\eta)$. Suppose now that $\psi = \langle \alpha \odot \beta \rangle$ with $\odot \in \{=, \neq\}$. By definition of $\mathsf{v}(\mathbf{b}(x))$, $\mathcal{V}$ verifies that there are data values $d \odot d'$ such that $d \in [\![\mathcal{A}_\alpha]\!]^{\mathbf{b}\otimes\mathbf{d}|_x}$ and $d' \in [\![\mathcal{A}_\beta]\!]^{\mathbf{b}\otimes\mathbf{d}|_x}$. This means that there is a path that starts at x and ends at some position $x{\cdot}y$ that satisfies α, in the sense that whenever a node expression ξ has to hold in a position $x{\cdot}x'$ (where $x' \preceq y$), $\mathcal{A}_\alpha$ verifies that $\xi \in \mathbf{b}(x{\cdot}x')$. By inductive hypothesis, we obtain that $x' \in [\![\xi]\!]^{\mathbf{a}\otimes\mathbf{d}}$. Hence, $(x, x{\cdot}y) \in [\![\alpha]\!]^{\mathbf{a}\otimes\mathbf{d}}$, where $d = \mathbf{d}(x{\cdot}y)$. Applying the same reasoning for β and d', we obtain that $x \in [\![\langle \alpha \odot \beta \rangle]\!]^{\mathbf{a}\otimes\mathbf{d}}$. The case where $\psi = \langle \alpha \rangle$ is only easier. Finally, if ψ is a boolean combination, we simply need to apply the inductive hypothesis and the rules of locally consistent sets.

Suppose now that $\mathbf{t} \models \eta$ for some data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$. We show that $\mathbf{a} \otimes \mathbf{d}$ is accepted by the automaton that results from the translation above. Let $\mathbf{b}(x) = \{\psi \in \{\eta\}^\neg \mid x \in [\![\psi]\!]^{\mathbf{t}}\}$ for every $x \in \mathsf{pos}(\mathbf{t})$. It is easy to check that $\mathbf{b}(x)$ is a locally consistent set and hence that $\mathbf{b}(x) \in \mathbb{B}$. Note that $\mathbf{a} \otimes \mathbf{b}$ is accepted by the transducer $\mathcal{R}$ since $\neg\mathbf{a}(x) \notin \mathbf{b}(x)$. The verifier accepts $\mathbf{b} \otimes \mathbf{d}$ since every test performed at a position x corresponds basically to testing $\mathbf{b}(x) = \{\psi \in \{\eta\}^\neg \mid x \in [\![\psi]\!]^{\mathbf{t}}\}$ by using the labels of the descendant positions. For every $\psi \in \{\eta\}^\neg$ the following can be shown by induction on the size of ψ. If $\psi \in \mathbf{b}(x)$ (in other words, if $x \in [\![\psi]\!]^{\mathbf{t}}$), and ψ is of the form $(\neg)\langle \alpha \odot \beta \rangle$ or $(\neg)\langle \alpha \rangle$, then the formula of the verifier ψ' is true in $\mathbf{t}|_x$, where ψ' is the translation of ψ according to $\mathsf{v}(b)$. This proves that $\mathbf{t}$ is accepted by $(\mathcal{R}, \mathcal{V})$. □

The result above already gives us a decidability procedure for the satisfiability problem. Let us analyse its upper bound.

Corollary 5.44. *The translation of Theorem 5.43 yields an automaton that uses at most* $\mathsf{R} = 2$ *relations per existential clause and at most* $\mathsf{V} = 2$ *variables. It can be built in exponential time in* η *such that* $\mathsf{K} + \mathsf{N} \leq p(|\eta|)$ *for some polynomial* p*. The sets of states* $\dot{Q}$ *and* $\tilde{Q}$ *have only one state.*

Proof. The fact that $\mathsf{R} = 2$ and $\mathsf{V} = 2$ is immediate from the definition of $\mathcal{V}$. As already discussed, the transducer only uses one state, $|\dot{Q}| = 1$, and that it only needs to use transitions $(\dot{q}, a, b, \mathcal{L})$ with $\mathcal{L} = (\dot{Q})^*$. $\mathcal{L}$ is represented with a NFA with a set of states $|\tilde{Q}| = 1$.

The translation we presented is not polynomial. Indeed there is an exponential number of transitions both in $\mathcal{R}$ and in the automata $\mathcal{A}_\alpha$ of $\mathcal{V}$, but it contains a *polynomial* number of automata $\mathcal{A}_\alpha$. Further, each $\mathcal{A}_\alpha$ has a number of states

polynomial in $|\alpha|$, and a number of transitions polynomial in $|\alpha|$ and $|\mathbb{B}|$. This last one is singly exponential in $|\varphi|$. □

From Theorem 5.43 and Corollary 5.44 we conclude that the satisfiability problem for the full fragment is in ExpTime.

Theorem 5.45 ((main result))**.** *SAT-*$\mathsf{regXPath}(\downarrow,=)$ *is in* ExpTime*.*

Proof. Given a formula $\eta \in \mathsf{regXPath}(\downarrow,=)$ we build, in exponential time, a $\mathscr{E}$-DD automaton $(\mathcal{R},\mathcal{V})$ as in Theorem 5.43. As remarked in Corollary 5.44, $\mathcal{R}$ is such that the set of states $\dot{Q}$ and $\tilde{Q}$ are fixed. $\mathcal{V}$ is such that K and N are polynomial in $|\eta|$, $\mathsf{R}=2$, and $\mathsf{V}=2$.

We run the emptiness algorithm of Theorem 5.37 on $(\mathcal{R},\mathcal{V})$. Since R is fixed, by Corollary 5.38, the time consumed by this algorithm is bounded by

$$(\mathsf{Aut}.|\mathcal{R}|.\mathsf{V}.\mathsf{K}.\mathsf{N})^{p(|\tilde{Q}|,\mathsf{V},\mathsf{K},\mathsf{N})}$$

for some polynomial p. Notice that all the variables in the exponent, namely $|\tilde{Q}|,\mathsf{V},\mathsf{K},\mathsf{N}$, are bounded by a polynomial in $|\eta|$. The remaining variables can be at most exponential in $|\eta|$. Hence, we have an exponential time algorithm for testing the satisfiability of a formula $\eta \in \mathsf{regXPath}(\downarrow,=)$. □

Lower bound

We next prove the ExpTime-hardness of satisfiability for $\mathsf{XPath}(\downarrow_*,=)$. Remarkably, this logic cannot express a *one step* down in the tree as it does not possess the $\downarrow$ axis, and this will be the major obstacle in the coding.

Theorem 5.46. *SAT-*$\mathsf{XPath}(\downarrow_*,=)$ *is* ExpTime*-hard.*

Proof. The proof is by reduction from the *two-player corridor tiling game*. An instance of this game consists of a set T of tiles $T=\{T_1,\ldots,T_s\}$, a special winning tile T_s, the sequence of initial tiles $\{T_1^0,\ldots,T_n^0\} \subseteq T$, and the horizontal and vertical tiling relations $H,V \subseteq T\times T$. The game is played on an $n\times\mathbb{N}$ board where the initial configuration of the first row is given by $T_1^0\cdots T_n^0$. At any moment during the game any pair of horizontally consecutive tiles must be in the relation H and every pair of vertically consecutive tiles in the relation V. The game is played by two players: *Abelard* and *Eloise*. Each player takes turn in placing a tile of his or her choice, filling the board from left to right, from bottom to top, always respecting the horizontal and vertical constraints H and V. *Eloise* is the first to play, and she wins if during the game the winning tile T_s is placed on the board, or if *Abelard* cannot place any tile. Otherwise, if the game continues indefinitely or if *Eloise* cannot place any tile, the game is won by *Abelard*. A *partial* game is a game that may not have finished. We assume that the tile T_s can only appear as the last placed element in the board (and in this case the game is finished). It is known that deciding whether *Eloise* has a winning strategy is ExpTime-complete.

Abstract representation of a winning strategy It is easy to see that in this game *Eloise* has a winning strategy if, and only if, she has a strategy to win before the row s^n of the board is reached (s is the number of tiles). (Otherwise, there would be a repeated configuration in two different rows, and the game could be shrunk to one with less than s^n rows.) Hence, we represent a partial game as a string of length at most $s^n \cdot n$, containing the plays on the board from left to right, bottom to top, respecting the constraints H and V. We represent a winning strategy for *Eloise* as a tree of nodes labeled with tiles such that

- the root contains T_1^0,
- every node at even depth (*e.g.*, the root) contains as children every tile T such that the path of tiles (from the root to the current node) appended with T is a partial game,
- every node at odd depth with a tile $T \neq T_s$ contains at least one children,
- all the maximal paths of the tree represent winning games for *Eloise*.

Concrete representation of a winning strategy We must now produce a property of $\mathsf{XPath}(\downarrow_*, =)$ such that any tree that satisfies it represents a winning strategy for *Eloise*. We use the coding of a winning strategy as presented before, extended with some extra nodes and labels, which are necessary to make sure that every path contains at most $n \cdot s^n$ tiles, and that the nodes verify the H and V restrictions. For simplicity, we assume that n is an even number, and hence that all positions in odd columns are played by *Eloise* and the others by *Abelard*. (A similar coding strategy can be used when n is odd.)

Our alphabet consists of

- the symbols $I_1 \dots I_n$ that indicate the current column of the corridor,
- the symbols $b_0 \dots b_m$ where $m = \lceil (n+1) \cdot \log(s) \rceil$ that act as *bits* to count from 0 to s^n (it is enough that they count *at least* up to s^n),
- the symbols $T_1 \dots T_s$ to code the tile placed at each play,
- a symbol # to separate rows, and an extra symbol \$ whose role will be explained later.

Inside a path, each block of nodes between two consecutive occurrences of # codes the evolution of the game for a particular row. Each node labeled I_i has a tile associated, coded as a descendant node with the same data value containing some label T_j as label. For example, in Figure 5.5, the first column I_1 of the current row is associated to the tile T_3, because $\langle T_3, 1\rangle$ is a descendant of $\langle I_1, 1\rangle$ with the same data value. Similarly, each occurrence of # is associated to a number, which is the number of the current row. This number is coded by the b_i elements with

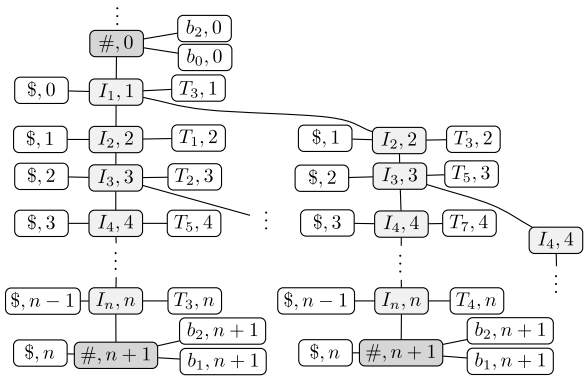

Fig. 5.5: Part of the model coding all the plays of row 5, which is between the #-element associated to 5 (101 in binary), and the element # with number 6 (110).

the same data value. In the example, $\langle \#, 0 \rangle$ is associated to the bits b_0 and b_2 that give the binary number 101.

Finally, the symbol \$ is used to delimit the region where the next element of the coding must appear, this will be our way of thinking the *next step* of the coding. This is to move from one position of the board to the next one, from the last position to the # delimiter, and from the # to the first position of the next row. Intuitively, between an element I_i, $i < n$, and the element \$ with the same data value, only I_{i+1} may appear. (There may be, however, more than one node with label I_i before the appearance of I_{i+1}, or more than one node I_{i+1}. This corresponds o the fact that we have a reflexive-transitive relation $\downarrow_*$.) This mechanism of coding a very relaxed 'one step' is the building block of our coding. The idea is that since the logic lacks the $\downarrow$ axis, we need to restrict the appearance of the next move of the game to a limited fragment of the tree. By means of this element \$, we can state, for example, that whenever we are in a I_2 element, then in this restricted portion I_3 must appear with $\langle \varepsilon = \downarrow_*[I_3]\downarrow_*[\$] \rangle$. In a similar way we can demand that there is a prefix of I_2 elements, after which *all* elements have label I_3, until the occurrence of the label \$, as we will see later.

In Figure 5.5 we show an example of a possible extract of the tree between the # associated to the number 5 until the next # associated to the number 6. The coding forces the tree to have branching as it contains all possible answers of *Abelard* at even positions.

Properties of the tree coding a winning strategy Since the properties expressed by our logic cannot avoid having repeated consecutives labels along a path, the coding will handle *groups* of nodes with the same label. Let us call an *a-group* to a maximal connected segment of a path that has the label a. The fact that there could be a sequence of elements with equal label does not cause any problem to the coding. In some sense it is redundant information in the coding.

Following the intuition given before, we spell out the concrete properties of the

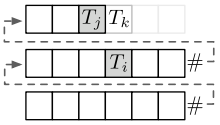

Fig. 5.6: Every legal move T_k of *Abelard* is played.

data tree that encodes a winning strategy for *Eloise*.

(1) For every i, we demand that there are no data values shared by different I_i-groups along a path. Likewise for all T_i and #-groups.

(2) The nodes labeled by \$ are leaves, in the sense that no other symbol may appear as descendant.

(3) Every I_i has its corresponding \$, *i.e.*, it has a descendant labeled \$ with the same data value.

(4) Every I_i has a next element, unless it contains the winning tile. That is, there is a I_{i+1} between I_i and its corresponding \$ if $i < n$ (and similarly, a # after I_1, and a I_1 after #).

(5) Each I_i has a unique tile: there is a descendant with equal data value and a tile T as label, and all descendants with equal data value carrying a tile, have the tile T.

(6) Every I_i inside a step along a branch must have the same tile.

(7) Between I_i ($i < n$) and its corresponding \$ there can only appear an I_{i+1}-group. (The same applies for the labels I_n and #, and for # and I_1.)

(8) The tiles match horizontally: the tiles corresponding to any two consecutive nodes labeled I_i and I_{i+1} are in the H relation. Likewise, the tiles match vertically: the tiles corresponding to any two nodes labeled I_i separated by exactly one #-group are in the V relation.

(9) All the elements corresponding to the *first* row have tiles $T_1^0 \cdots T_n^0$.

(10) Every legal move of *Abelard* is taken into account. For every node I_i with i being odd such that there is a tile T_k that can be played in the next position (according to H, V and the tiles already placed, as in Figure 5.6), then it must appear in the next position. Note that this forces a branching in the tree.

(11) The data value of a # element is associated to a counter. The least significant bit corresponds to b_0. A bit 1 at a bit position i is coded as the presence of a descendant node with the same data value labeled b_i. The counter starts in 0, and along a path, each time a #-group appears, the counter increments by one.

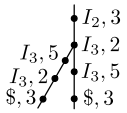

Fig. 5.7: Repeated elements in the coding.

(12) There is no # element that has all the b_i bits in 1. Because that would mean that *Eloise* was not able to put a T_s tile in less than s^n rounds.

As already mentioned, we do not avoid having more than one element before the \$. As shown in Figure 5.7, there may be consecutive repetitions of the same label along a path, or subtrees that are duplicated, but this does not spoil the coding. We are actually forcing properties for *all* branches and all possible extra elements that the tree may contain. Any extra element or branching induces more copies of winning strategies for *Eloise*.

For every data tree with these properties, one can replace all consecutive appearances of the same label along a path by only one appearance, and strip off all the nodes containing labels which are not tiles. This gives us a winning strategy. Conversely, given a winning strategy for *Eloise*, we can add data values to the tree and the necessary nodes to transform it into a data tree that satisfies all the aforementioned properties.

Enforcing the properties in $\mathsf{XPath}(\downarrow_*, =)$ We first define some predicates that we will use throughout the coding. $\mathsf{s}^k_\sigma(\varphi)$ evaluates φ at a node at k-steps (with our way of coding a step as we have seen before) from the current point of evaluation, given that the current symbol is σ. For this purpose we first define $next(I_i) := I_{i+1}$ (if $i < n$), $next(I_n) := \#$ and $next(\#) := I_1$. Hence, we define for $a \in \{\#, I_1, \ldots, I_n\}$,

$$\mathsf{s}^0_a(\varphi) := a \wedge \varphi \qquad \mathsf{s}^{k+1}_a(\varphi) := a \wedge \langle \varepsilon = \downarrow_*[\mathsf{s}^k_{next(a)}(\varphi)]\downarrow_*[\$]\rangle$$

Similarly, t_i checks that the tile of the current node I corresponds to T_i,

$$\mathsf{t}_i := \langle \varepsilon = \downarrow_*[T_i]\rangle$$

bit_i checks that the i-bit of the counter's binary encoding of a #-node is one (1),

$$bit_i := \langle \varepsilon = \downarrow_*[b_i]\rangle$$

and G forces a property to hold at all nodes of the tree.

$$\mathsf{G}(\varphi) := \neg\langle\downarrow_*[\neg\varphi]\rangle$$

We now exhibit the XPath formulæ for each of the properties just seen.

(1) For every I_i: $\neg\downarrow_*[I_i \wedge \langle \varepsilon = \downarrow_*[\neg I_i]\downarrow_*[I_i]\rangle]$. Likewise for T_i and #.

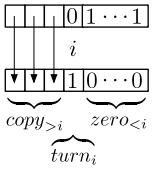

Fig. 5.8: A bitwise increment of the counter.

(2) $\mathsf{G}(\$ \to \neg\langle\downarrow_*[\neg\$]\rangle)$.

(3) For every I_i: $\mathsf{G}(I_i \to \langle\varepsilon = \downarrow_*[\$]\rangle)$.

(4) $\mathsf{G}(I_i \wedge \neg\mathsf{t}_s \to \mathsf{s}^1_{I_i}(\top)) \wedge \mathsf{G}(\# \to \mathsf{s}^1_{\#}(\top))$.

(5) For every I_i and $\ell \neq j$: $\mathsf{G}(\neg(\mathsf{t}_\ell \wedge \mathsf{t}_j))$. For every i: $\mathsf{G}(I_i \to \bigvee_j \mathsf{t}_j)$.

(6) For every $i < n$ and $j \neq k$, $\mathsf{G}(I_i \to \neg\langle\varepsilon = \downarrow_*[I_{i+1} \wedge \mathsf{t}_j]\downarrow_*[I_{i+1} \wedge \mathsf{t}_k]\downarrow_*[\$]\rangle)$. (And a similar condition for # and I_1.)

(7) For every $i < n$ and $a \notin \{I_i, I_{i+1}, \$\}$, $\mathsf{G}(I_i \to \neg\langle\varepsilon = \downarrow_*[a]\downarrow_*[\$]\rangle)$, $\mathsf{G}(I_i \to \neg\langle\varepsilon = \downarrow_*[I_{i+1}]\downarrow_*[I_i]\downarrow_*[\$]\rangle)$, and $\mathsf{G}(I_i \to \neg\langle\varepsilon = \downarrow_*[\$]\downarrow_*[I_i \vee I_{i+1}]\downarrow_*[\$]\rangle)$.

(8) The tiles match horizontally: for every k and T_i, T_j such that $(T_i, T_j) \notin H$, $\neg\langle\downarrow_*[I_k \wedge \mathsf{t}_i \wedge \mathsf{s}^1_{I_k}(\mathsf{t}_j)]\rangle$. The tiles match vertically: for every k and T_i, T_j such that $(T_i, T_j) \notin V$, $\neg\langle\downarrow_*[I_k \wedge \mathsf{t}_i \wedge \mathsf{s}^{n+1}_{I_k}(\mathsf{t}_j)]\rangle$.

(9) For all $i \in [1..n]$ and tile $T_j = T^0_i$, $\mathsf{s}^i_{\#}(\mathsf{t}_j)$ must hold at the root.

(10) For every T_i, T_j, T_k such that $(T_i, T_k) \in V$ and $(T_j, T_k) \in H$:

$$\neg\big\langle\downarrow_*\big[I_{2\ell} \wedge \mathsf{t}_i \wedge \mathsf{s}^n_{I_{2\ell}}\big(I_{2\ell-1} \wedge \mathsf{t}_j \wedge \neg\mathsf{s}^1_{I_{2\ell-1}}(\mathsf{t}_k)\big)\big]\big\rangle$$

(11) It is easy to code that the first # is all-zero. The increment of the counter between two # is coded as in Figure 5.8, by $\mathsf{G}(\# \wedge \mathit{flip}(i) \to \mathit{zero}_{<i} \wedge \mathit{turn}_i \wedge \mathit{copy}_{>i})$, where

$$\mathit{flip}(i) := \neg\mathit{bit}_i \wedge \bigwedge_{j<i} \mathit{bit}_j \quad \mathit{zero}_{<i} := \bigwedge_{j<i} \neg\mathsf{s}^{n+1}_{\#}(\mathit{bit}_j) \quad \mathit{turn}_i := \neg\mathsf{s}^{n+1}_{\#}(\neg\mathit{bit}_i)$$

$$\mathit{copy}_{>i} := \bigwedge_{j>i} (\mathit{bit}_j \wedge \neg\mathsf{s}^{n+1}_{\#}(\neg\mathit{bit}_j)) \vee (\neg\mathit{bit}_j \wedge \neg\mathsf{s}^{n+1}_{\#}(\mathit{bit}_j))$$

(12) $\mathsf{G}(\# \to \neg\bigwedge_i \mathit{bit}_i)$

This completes the coding. It is easy to see that all the formulæ have polynomial size on s and n, and that they express the previous properties. Hence, *Eloise* has a winning strategy in the two-player corridor tiling game iff the conjunction of the formulæ just described is satisfiable. Notice that the above reduction does not use path unions, and this means that even $\mathsf{XPath}(\downarrow_*, =)$ stripped of path unions is ExpTime-hard. □

5.4.2 PSpace fragments

We now turn to some other downward fragments of XPath. We complete the picture by analysing the complexity of all the possible combinations of downward axes in the presence or absence of data values tests. We first introduce a basic definition that we use throughout the section.

Definition 5.47. We say that the logic $\mathscr{P}$ has the **poly-depth model property** if there exists a polynomial p such that for every formula $\varphi \in \mathscr{P}$, φ is satisfiable if and only if φ is satisfied by a data tree of height at most $p(|\varphi|)$.

We can now prove the following statement that we will later use to show PSPACE-completeness for $\mathsf{XPath}(\downarrow, =)$.

Proposition 5.48. *Every fragment $\mathscr{P}$ of $\mathsf{regXPath}(\downarrow, =)$ that has the poly-depth model property is in* PSPACE.

Proof. Suppose that if a formula $\eta \in \mathscr{P}$ is satisfied by a data tree, then it is satisfied by a tree of height $h \leq p(|\eta|)$, where p is a fixed polynomial that does not depend on η.

We make use of the translation of Theorem 5.43, but we avoid storingthe transition relations of $\mathcal{R}$ and of the automata Aut of $\mathcal{V}$ explicitly. Instead, we use two facts. First, that testing whether a set S of subformulæ of η is a locally consistent set uses polynomial space in $|\eta|$, and these sets can then be enumerated in polynomial space. And second, that $\mathsf{v}(b)$ can be built in polynomial space, given a locally consistent set b. Thus, the space $sp(\vdash)$ needed for checking $T_1 \cdots T_n \vdash T$ is polynomial.

Hence, by Theorem 5.40 with Remark 5.41 we have an emptiness algorithm that uses space $p(h, \mathsf{K}, \mathsf{N}) + sp(\vdash)$ for some polynomial p. This is a consequence of $|\dot{Q}|$, V, R and $|\tilde{Q}|$ being constant, and $h, \mathsf{K}, \mathsf{N}$ being polynomial in $|\eta|$ by Corollary 5.44. □

We use the result above to prove the following proposition.

Proposition 5.49. *SAT-*$\mathsf{XPath}(\downarrow, =)$ *is* PSPACE*-complete.*

Proof. Benedikt et al. (2008) show that $\mathsf{XPath}(\downarrow, =)$ is PSPACE-hard and NEXPTIME-easy. Here we show a PSPACE upper bound by proving the poly-depth model property. Note that if η is satisfiable in $\mathbf{t}$, then it is satisfiable in $\mathbf{t} \restriction n$ where n is the maximum number of nested $\downarrow$ in η[5], and $\mathbf{t} \restriction n$ is the submodel of $\mathbf{t}$ consisting of all the nodes that are at distance at most n from the root: $\mathbf{t} \restriction n = \{x \mapsto \mathbf{t}(x) \mid x \in \mathsf{pos}(\mathbf{t}), |x| \leq n\}$. Hence, by Proposition 5.48, $\mathsf{XPath}(\downarrow, =)$ is in PSPACE. □

This concludes our analysis of downward fragments of XPath with data tests. Summing up, we showed that the satisfiability problem for the logics $\mathsf{regXPath}(\downarrow, =)$,

[5] For example, the maximum number of nexted $\downarrow$ in the expression $\langle\downarrow[a \wedge \langle\downarrow\downarrow[b]\rangle]\downarrow[\langle\downarrow\downarrow[a]\rangle]\rangle$ is 4.

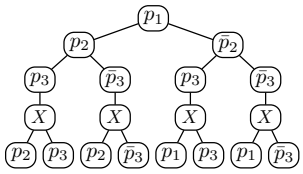

Fig. 5.9: A coding of the satisfiability of a formula $\exists p_1 \forall p_2, p_3.(p_1 \vee p_2) \wedge (p_3 \vee \neg p_3 \vee \neg p_2)$.

$\mathsf{XPath}(\downarrow, \downarrow_*, =)$ and $\mathsf{XPath}(\downarrow_*, =)$ is ExpTime-complete, while PSpace-complete for $\mathsf{XPath}(\downarrow, =)$. For the sake of completeness, we now turn to downward fragments where no data tests are available.

Corollary 5.50. *SAT-XPath*$(\downarrow)$ *is* PSpace*-complete.*

Proof. The lower bound by (Benedikt et al., 2008, Theorem 5.1) and the upper bound by Proposition 5.49. □

Proposition 5.51. *XPath*$(\downarrow_*)$ *is* PSpace*-hard.*

Proof. The proof goes by reduction from an instance of the QBF (Quantified Boolean Formula (Garey and Johnson, 1979)) validity problem to SAT-$\mathsf{XPath}(\downarrow_*)$.

Let

$$\varphi = Q_1 p_1 \dots Q_n p_n.\psi$$

where p_i are propositional variables (pairwise distinct), $Q_i \in \{\forall, \exists\}$ and ψ is a formula of the propositional calculus in CNF.

The idea is to force a model in which every branch contains a full valuation for the variables $p_1, \dots, p_n$. The tree's alphabet is $\mathbb{A} = \{p_1, \dots, p_n, \bar{p}_1, \dots, \bar{p}_n, X\}$, and every branch lists a valuation in order, that is, first there is a node with a label in $\{p_1, \bar{p}_1\}$, then another in $\{p_2, \bar{p}_2\}$, etc. The label X simply marks the ending of a valuation in a branch. After this marking we build the tree that satisfies ψ by choosing a witness atom for every disjunctive clause. Finally, we check that there are no inconsistencies with respect to its valuation. For example, in Figure 5.9 we depict a possible tree that is forced by a formula.

Let v_i be the formula that specifies that the node is a valuation for the propositional variable p_i, $v_i := p_i \vee \bar{p}_i$. We specify formulæ $f_1, \dots, f_n$ depending on $Q_1, \dots, Q_n$, where G is defined as in the proof of Theorem 5.46.

- If $Q_1 = \forall$, then $f_1 = \langle\downarrow_*[p_1]\rangle \wedge \langle\downarrow_*[\bar{p}_1]\rangle$.
  If $Q_1 = \exists$, then $f_1 = \langle\downarrow_*[p_1]\rangle \vee \langle\downarrow_*[\bar{p}_1]\rangle$.
- If $Q_i = \forall$ for $i > 1$, then $f_i = \mathsf{G}(v_{i-1} \to \langle\downarrow_*[p_i]\rangle \wedge \langle\downarrow_*[\bar{p}_i]\rangle)$.
  If $Q_i = \exists$, then $f_i = \mathsf{G}(v_{i-1} \to \langle\downarrow_*[p_i]\rangle \vee \langle\downarrow_*[\bar{p}_i]\rangle)$.
- φ_X forces that the label X always appears once the valuation for all propositions has been defined.

$$\varphi_X = \neg\langle\downarrow_*[v_1]\downarrow_*[v_2] \cdots \downarrow_*[v_{n-1}]\downarrow_*[v_n \wedge \neg\langle\downarrow_*[X]\rangle]\rangle$$

- For all X we build a formula for $\psi = C_1 \wedge \ldots \wedge C_l$ where $C_i = t_1 \vee \ldots \vee t_m$ and each t_j is either p_k or $\bar{p}_k$ for some k. That is,

$$\tau = \bigwedge_{C_i} \bigvee_{t \in C_i} \langle \downarrow_* [t] \rangle$$

and this must hold for all X-valued nodes,

$$\varphi_\psi = \mathsf{G}(X \to \tau) \ .$$

- Finally, we must check that there are no inconsistencies between the p_i along a branch.

$$\varphi_{inc} = \bigwedge_{i=1}^{n} \neg\langle \downarrow_* [p_i] \downarrow_* [\bar{p}_i] \rangle \wedge \neg\langle \downarrow_* [\bar{p}_i] \downarrow_* [p_i] \rangle$$

The final formula is then

$$\varphi_F = \bigwedge_{i=1}^{n} f_i \wedge \varphi_X \wedge \varphi_\psi \wedge \varphi_{inc} \ .$$

Suppose that φ is valid. We can build a tree that satisfies φ_F as follows. Let's assume that $Q_1 = \exists$ (the other case is similar). Note that $\exists p_i Q_{i+1} p_{i+1} \cdots Q_n p_n . \psi$ is valid if, and only if, there is a valuation $v(p_i)$ for p_i such that

$$Q_{i+1} p_{i+1} \cdots Q_n p_n . \psi[p_i \mapsto v(p_i)]$$

is valid, (where this is interpreted as the result of replacing p_i by true or false in the formula according to $v(p_i)$). Conversely, $\forall p_i Q_{i+1} p_{i+1} \cdots Q_n p_n . \psi$ is valid if, and only if, both $Q_{i+1} p_{i+1} \cdots Q_n p_n . \psi[p_i \mapsto 1]$ and $Q_{i+1} p_{i+1} \cdots Q_n p_n . \psi[p_i \mapsto 0]$ are valid. Now, we build the tree as follows. We first take the witnessing valuation of p_1 for the validity of φ and put it as the root. Then we iterate adding leaves in the tree until obtaining a tree whose every leaf is at depth $n-1$ in the following way. Suppose we are adding the children of a leaf at depth i with $i < n-1$. Let v be the valuation of $p_1 \ldots p_i$ that corresponds to the labels that appear in the path that leads to the current leaf. Consider φ' as the result of replacing every p_j by true or false corresponding to $v(p_j)$ for every $j \leq i$ in $Q_{i+1} p_{i+1} \cdots Q_n p_n \psi$, and note that φ' must be valid. If $Q_{i+1} = \exists$ we just add one child with the witnessing valuation $v(p_{i+1})$ that makes $\varphi'[p_{i+1} \mapsto v(p_{i+1})]$ valid. If $Q_{i+1} = \forall$ we add *two* children with labels p_{i+1} and $\bar{p}_{i+1}$. Once we do this with all the quantifiers, we append a node with label X to all leaves, and put the witness of the satisfaction of ψ which must be a choice between the valuations that occurred along the path, which gives us a tree whose every leaf is at depth $n+1$, as in Figure 5.9. Note that it cannot be that there is a p_i and $\bar{p}_i$ along a path. Indeed, this tree satisfies φ_F.

On the other hand, if φ_F is satisfied by a tree, then we can produce witness for every $\exists$ quantifier of the formula, always making sure that the valuation chosen for

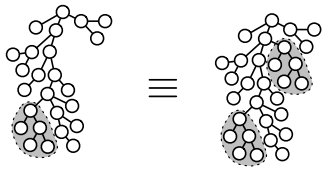

Fig. 5.10: The subtree copy property.

one propositional variable p_i will not change, since we disallow having p_i nad $\bar{p}_i$ in the same path. This shows that φ is in fact valid.

Then we have that φ is QBF-valid if and only if φ_F is satisfiable. Note that this reduction does not use path unions. Then, this lower bound holds even in the absence of path unions. □

Now we focus in finding an upper bound for SAT-$\mathsf{XPath}(\downarrow_*)$. We prove that it is in PSPACE by the poly-depth model property. But before doing that, we need to introduce an important property of this logic. The **subtree copy** property states that if a tree satisfies some $\mathsf{XPath}(\downarrow_*)$ property at the root, then the tree where some subtree was copied at a higher position of the tree (as depicted in Figure 5.10) also satisfies the property.

Lemma 5.52 ((subtree copy))**.** *Given a tree* $\mathbf{t}$*, and given two positions* $x, y \in \mathsf{pos}(\mathbf{t})$ *with* $x \prec y$*, consider the last index* l *such that* $x{\cdot}l \in \mathsf{pos}(\mathbf{t})$*. Let* $\mathbf{t}'$ *be defined as follows.*

$$\mathsf{pos}(\mathbf{t}') = \mathsf{pos}(\mathbf{t}) \cup \{x{\cdot}(l+1){\cdot}z \mid y{\cdot}z \in \mathsf{pos}(\mathbf{t})\}$$

$$\mathbf{t}'(z) = \begin{cases} \mathbf{t}(z) & \text{if } z \in \mathsf{pos}(\mathbf{t}) \\ \mathbf{t}(y{\cdot}w) & \text{if } z = x{\cdot}(l+1){\cdot}w \end{cases}$$

Then, for every $w \in \mathsf{pos}(\mathbf{t})$ *and* $\varphi \in \mathsf{XPath}(\downarrow_*)$*,* $\mathbf{t}|_w \models \varphi$ *iff* $\mathbf{t}'|_w \models \varphi$*. The shapes of* $\mathbf{t}$ *and* $\mathbf{t}'$ *are illustrated in Figure 5.10.*

Proof. All path formulæ that hold at the root of $\mathbf{t}$, hold also in $\mathbf{t}'$ as it is an *extension* of the tree. On the other hand, any path formula that is satisfied at the root by a succession of nodes in a branch in $\mathbf{t}'$, can also be found in $\mathbf{t}'$, as we only count with the $\downarrow_*$ axis. This is true not only for the root but for any position of $\mathbf{t}$. In other words, the logic $\mathsf{XPath}(\downarrow_*)$ is closed under *subtree copy.* □

Note that the preceding Lemma 5.52 is a stronger property than that of Proposition 5.17, but this one holds only for $\mathsf{XPath}(\downarrow_*)$, a logic with no data tests or $\downarrow$ axis. Having stated the subtree copy property, we can now show the following proposition.

Proposition 5.53. *SAT-*$\mathsf{XPath}(\downarrow_*)$ *is in* PSPACE*.*

$$
\begin{aligned}
nseq : \mathsf{XPath}(\downarrow_*) &\to \wp\big(\big(\wp(\mathsf{XPath}(\downarrow_*))\big)^*\big) \\
nseq(\alpha \cup \beta) &= nseq(\alpha) \cup nseq(\beta) \qquad & nseq([\psi]) &= \{\{\psi\}\} \\
nseq(\alpha\beta) &= \{S_1{\cdot}(A_1 \cup A_2){\cdot}S_2 \mid & nseq(\varepsilon) &= \{\emptyset\} \\
&\qquad S_1{\cdot}A_1 \in nseq(\alpha), A_2{\cdot}S_2 \in nseq(\beta)\} & nseq(\downarrow_*) &= \{\emptyset{\cdot}\emptyset\}
\end{aligned}
$$

Fig. 5.11: Given a path expression α, $nseq(\alpha)$ is the set of possible sequence of node tests that a witnessing branch must satisfy.

Proof. We prove that $\varphi \in \mathsf{XPath}(\downarrow_*)$ is satisfiable iff it is satisfied by a tree of height bounded by $|\varphi|^2$.

For the proof of this statement we first define, for a path expression, the set of possible sequences of node tests that it must satisfy, that we note with '*nseq*' (Fig. 5.11). The idea is that if for instance $\{\psi, \varphi\}{\cdot}\{\varphi\}{\cdot}\{\psi, \eta\} \in nseq(\alpha)$ for a path expression α, then $\mathbf{t} \models \langle\alpha\rangle$ if there are $\epsilon \preceq x \preceq y$ such that $\mathbf{t} \models \psi$, $\mathbf{t} \models \varphi$, $\mathbf{t}|_x \models \varphi$, $\mathbf{t}|_y \models \psi$ and $\mathbf{t}|_y \models \eta$. Let $\mathsf{wit}_\mathbf{t} : \mathsf{pos}(\mathbf{t}) \times \mathsf{XPath}(\downarrow_*) \to \wp(\mathsf{pos}(\mathbf{t}))$ be a *witness* function such that for any $x \in \mathsf{pos}(\mathbf{t})$ and $\alpha \in \mathsf{XPath}(\downarrow_*)$ such that $\mathbf{t}|_x \models \langle\alpha\rangle$ there is $S \in nseq(\alpha)$ where

- all elements in $\mathsf{wit}_\mathbf{t}(x, \alpha) = \{x_1, \ldots, x_n\}$ belong to the same branch, $x_1 \prec x_2 \prec \cdots \prec x_n$;
- there are $i_1 \leq \cdots \leq i_{|S|}$ such that $\{i_1, \ldots, i_{|S|}\} = \{1, \ldots, n\}$, and $\mathbf{t}|_{x_{i_j}} \models \bigwedge S(j)$ for every j.

Note that in particular $|\mathsf{wit}_\mathbf{t}(x, \alpha)| \leq |S|$ for some $S \in nseq(\alpha)$, and we hence have

$$|\mathsf{wit}_\mathbf{t}(x, \alpha)| \leq |\alpha| \ . \tag{5.8}$$

In the sequel, given $\varphi \in \mathsf{XPath}(\downarrow_*)$ we write $nesting(\varphi)$ for the maximum number of nested node tests (that is, of nested '[ ]') that are in the formula φ.

We can make use of Lemma 5.52 to make sure that we can always assume $\mathsf{wit}_\mathbf{t}$ to be in a normal form where all its elements are chained with the parent/child relation. That is, that given $\mathsf{wit}_\mathbf{t}(x, \alpha) = \{x_1 \prec \cdots \prec x_n\}$, then for all i

$$x_{i+1} = x_i{\cdot}j \text{ for some } j \ . \tag{5.9}$$

We are now in a position to explain the main argument. Let $\varphi \in \mathsf{XPath}(\downarrow_*)$ and $n = nesting(\varphi)$, and suppose we have a tree $\mathbf{t}$ that satisfies φ. Next, we describe a procedure to 'mark' the important nodes in the tree. We start by marking the root with the label 'n'. Then, for every position x marked with $t \geq 0$ and for every path expression $\beta \in \mathsf{sub}(\varphi)$ such that $nesting(\beta) \leq t$ and $\mathbf{t}|_x \models \beta$, we mark all positions $y \in \mathsf{wit}_\mathbf{t}(x, \beta)$ with '$t - 1$'.

Note that all the positions marked with $\{-1, 0, \ldots, n\}$ form *one* connected component by (5.9), and that they are all at a distance from the root of at most

nesting$(\varphi).|\varphi|$ by (5.8). Let $\mathbf{t}'$ be the tree resulting from eliminating all the positions with no marking. We then have that $\mathbf{t} \models \varphi$ iff $\mathbf{t}' \models \varphi$, and hence that $\mathsf{XPath}(\downarrow_*)$ has the poly-depth model property. We conclude by Proposition 5.48 that $\mathsf{XPath}(\downarrow_*)$ is in PSPACE. □

We then have as a corollary from Proposition 5.53 and Proposition 5.51 that $\mathsf{XPath}(\downarrow_*)$ is complete for PSPACE.

Theorem 5.54. *SAT-*$\mathsf{XPath}(\downarrow_*)$ *is* PSPACE*-complete.*

So far we have that, in the presence of data values, the presence of the descendant axis $\downarrow_*$ produces an increase[6] in the complexity from PSPACE to EXPTIME. However, we argue that it is not the ability to test for data equality of distant elements what produces this increase in complexity. It is, as a matter of fact, in the ability to test data values against that of the root in formulæ like $\langle \varepsilon = \downarrow_*[a] \rangle$. We show that if we remove this kind of data tests, the resulting logic is in PSPACE, even though the fragment contains the $\downarrow_*$ axis.

Definition 5.55. We denote by $\mathsf{XPath}^{\not\varepsilon}(\downarrow_*, =)$ the fragment of $\mathsf{XPath}(\downarrow_*, =)$ where ε path formulæ are forbidden, and in general where there is no ε-testing on a path (like in $[\varphi]\downarrow_*$), that is, such that path formulæ are defined

$$\alpha \ ::= \ \downarrow_* \ \mid \ \alpha[\varphi] \ \mid \ \alpha\beta \ \mid \ \alpha \cup \beta \ .$$

Proposition 5.56. *SAT-*$\mathsf{XPath}^{\not\varepsilon}(\downarrow_*, =)$ *is* PSPACE*-complete.*

Proof. The proof of the upper-bound goes by showing the poly-depth model property. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$ be a data tree. The key observation is that any $\mathsf{XPath}^{\not\varepsilon}(\downarrow_*, =)$ expression of the form $\langle \alpha \rangle$, $\langle \alpha = \beta \rangle$ or $\langle \alpha \neq \beta \rangle$ that is satisfied at a node x of a tree, is also satisfied in *any ancestor* of x. This is because all path expressions start with a $\downarrow_*$ axis. In other words, for any pair of positions x, x' such that $x \preceq x'$, the set of formulæ of the aforementioned type that are satisfied in x' is a *subset* of those that are satisfied in x. Given a branch $\epsilon = x_0 \prec \cdots \prec x_n$ of $\mathbf{t}$ where x_{i+1} is a child of x_i for all i, and given a formula φ, consider for every $i \in [n]$

$$C_i = \{\psi \mid \psi \in \mathsf{sub}(\varphi), \ \psi \text{ of the form } \langle \alpha \rangle, \langle \alpha = \beta \rangle \text{ or } \langle \alpha \neq \beta \rangle \text{ s.t. } \mathbf{t}|_{x_i} \models \psi\}.$$

We then have $C_0 \supseteq \cdots \supseteq C_n$, and there is only a *polynomial* number of different sets. Take any two $C_i = C_j$ such that $\mathbf{a}(x_i) = \mathbf{a}(x_j)$. Then, the tree which results from replacing the subtree at x_i by the subtree at x_j preserves the satisfaction of φ at the root. Hence, the logic has the poly-depth model property and by Proposition 5.48 its satisfiability problem is in PSPACE.

The lower-bound comes from the proof of Proposition 5.51, whose encoding is in $\mathsf{XPath}^{\not\varepsilon}(\downarrow_*, =)$. □

[6] In the case PSPACE ≠ EXPTIME.

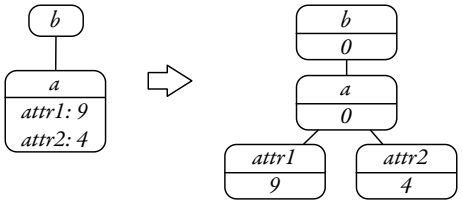

Fig. 5.12: Transformation from an XML to a data tree.

5.4.3 XML versus data trees

Here we show that all the results on satisfiability of XPath on data trees can be transferred to equivalent results on satisfiability of attrXPath on XML documents.

SAT-attrXPath$(\downarrow, \downarrow_*, =)$ *is* ExpTime*-easy.* As already mentioned, each XML document can be coded in a data tree by adding one child for each attribute with its corresponding value as in Figure 5.12. We can force this kind of model using XPath$(\downarrow_*, \downarrow, =)$ by stating that all the nodes with a symbol from $\mathbb{A}_{attr}$ are leaves,

$$\varphi_{struct} = \neg\langle \downarrow_*[\bigvee_{s \in \mathbb{A}_{attr}} s \ \wedge \langle\downarrow\rangle] \rangle \ .$$

We can interpret any attrXPath formula as an XPath formula by considering an extended alphabet $\mathbb{A} = \mathbb{A}_{elem} \cup \mathbb{A}_{attr}$ and replacing every appearance of '$@attr1$' by '$\downarrow[attr1]$'. Let us call tr to this translation.

We can then decide the satisfiability of a formula ψ of attrXPath$(\downarrow, \downarrow_*, =)$ on attributes data trees by testing the satisfiability of '$tr(\psi) \wedge \varphi_{struct}$' on data trees. Since we have that XPath$(\downarrow, \downarrow_*, =)$ is in ExpTime, we also have an ExpTime decidability procedure for the full downward fragment of attrXPath (as the translation tr is clearly performed in PTime) even with the Kleene star operator.

SAT-attrXPath$(\downarrow_*, =)$ *is* ExpTime*-hard.* On the other hand, any XPath formula on data trees can be thought of an attrXPath formula that uses at most one attribute. We can then deduce the ExpTime-hardness result of attrXPath$(\downarrow_*, =)$ from that of XPath$(\downarrow_*, =)$.

SAT-attrXPath$(\downarrow, =)$ *is* PSpace*-complete.* For the case of attrXPath$(\downarrow, =)$ we can do the same translation the only difference being that for a formula $\psi \in$ attrXPath$(\downarrow, =)$ the structure can be forced by

$$\varphi_{struct} = \bigwedge_{0 \leq n \leq d+1} \neg\langle\downarrow^n[\bigvee_{s \in \mathbb{A}_{attr}} s \wedge \langle\downarrow\rangle]\rangle$$

where d is the maximum quantity of nested occurrences of $\downarrow$ in ψ. It is easy to see that this forces the requested property for all the portion of the data tree that

```
<!ELEMENT book_list (book*)>
<!ELEMENT book ((author, birthdate?)+, chapter+)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT birthdate (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
```

Fig. 5.13: Example of a DTD under which $\mathsf{regXPath}(\downarrow,=)$ is decidable in ExpTime.

we are interested in. That is, for the whole region that $tr(\psi)$ can access. This is associated with the poly-depth model property of the logic. We then have that $\mathsf{attrXPath}(\downarrow,=)$ is PSpace-complete.

5.4.4 In the presence of regular languages

In this section we make some observations on the problem of satisfiability of downward XPath in the presence of some regular language.

In Chapter 6 we will show that satisfiability of downward XPath under a regular language is decidable. However, the problem has a very big complexity. Even for the fragment containing only the descendant axis, it can be shown that there is no algorithm that solves the satisfiability problem under a regular language in primitive recursive time or space by Corollary 4.4 of Section 4.5. However, if the language to be tested at the siblinghoods is restricted to be extensible, we can translate this problem to the emptiness problem for DD automata obtaining the following result.

Theorem 5.57. *SAT-*$\mathsf{regXPath}(\downarrow,=) + \mathscr{E}_{tree}$*, that is, the satisfiability problem for* $\mathsf{regXPath}(\downarrow,=)$ *under the class* $\mathscr{E}_{tree}$ *of extensible tree regular languages, is in* ExpTime.

Proof. Let $\varphi \in \mathsf{regXPath}(\downarrow,=)$ and $\mathcal{L} \in \mathscr{E}_{tree}$, represented as an $\mathscr{E}$-transducer $\mathcal{R}_{\mathcal{L}}$ such that $\mathcal{R}_{\mathcal{L}}(\mathbf{t} \otimes \mathbf{t})$ iff $\mathbf{t} \in \mathcal{L}$.

We first build the DD automaton $(\mathcal{R}, \mathcal{V})$ resulting from the translation of φ by Theorem 5.43. We compute the composition $\mathcal{R}' = \mathcal{R}_{\mathcal{L}} \circ \mathcal{R}$ such that $\mathcal{R}(\mathbf{t} \otimes \mathbf{t}')$ iff $\mathcal{R}_{\mathcal{L}}(\mathbf{t} \otimes \mathbf{t})$ and $\mathcal{R}(\mathbf{t} \otimes \mathbf{t}')$ in polynomial time in $\mathcal{R}$ and $\mathcal{R}_{\mathcal{L}}$. We then test the emptiness of $(\mathcal{R}', \mathcal{V})$ in exponential time in the number of states $|\tilde{Q}|$ of the automata corresponding to the (extensible) languages from the transitions of $\mathcal{R}'$. The resulting reduction is in ExpTime because the same arguments used to show 5.45 can be applied. □

The above Theorem 5.57 implies that the satisfiability of $\mathsf{regXPath}(\downarrow,=)$ under certain kind of restrictions is decidable in ExpTime. Note that these restrictions may be specified as DTD, XML Schema, Relax NG, etc. For example, it would mean that this logic is decidable under DTD whose every type is defined under some transitive operator $+$ or $*$. That is, that the definition of every type is in REG_* (as defined in Definition 5.7), like in the example of Figure 5.13.

5.5 Discussion

In Section 5.2 we introduced an automata model over data trees. Here we explore very briefly two natural extensions that could be added to this automaton. Firstly, the possibility of allowing *any* alternation free first order formula in the set Φ of properties that the verifier can test, instead of the restricted kind where no negation of a relation may occur. Secondly, what happens if we allow constants in Φ.

Negation of relations

The reader may have noticed in Definition 5.2 that the DD automata model does not allow to have negated appearances of a relation D_i under a (positive) existential quantification. This is not by chance, and in effect we can see that if we allow to have arbitrary boolean combinations of D_i relations we fall into a much harder emptiness problem. Although the decidability of the resulting model is not clear, it is possible to show that in the case it is decidable, the emptiness problem cannot be solved in primitive recursive time.

Consider the simple formula $\neg(\exists x.\neg D_1(x) \wedge D_2(x))$, such that $\mathcal{A}_1$ recognizes $\{b\, b' \mid b, b' \in \mathbb{B}\}$ and $\mathcal{A}_2$ recognizes $\{b\, b'\, b'' \mid b, b', b'' \in \mathbb{B}\}$. In other words, $\mathcal{A}_1$ simply goes to any child of the root, and $\mathcal{A}_2$ goes to any grandchild of the root. This kind of property intuitively tests that all the data values appearing at depth $l+1$ also appear at depth l. Although we will not enter into detail, it is possible to code a run of a weak version of a n-counters ICA (*cf.* Section 2.4.5) by using this kind of property. Thus, by Proposition 2.26, the emptiness problem for this extended automaton cannot be primitive recursive.

Constants

Another simple extension that DD automata may allow to have is the fact of having a set of constant data values. This does not change the complexity results. The results and definitions of Sections 5.3.1, 5.3.3, 5.3.2 remain valid. In Section 5.3.4 we modify the tree configurations by keeping explicit track of these constants, by adding the description of these constants to the data description mapping α at all configurations, and modifying the conditions of $\vdash$ accordingly. This implies that we can verify the satisfiability of $\mathsf{regXPath}(\downarrow, =)$ with constants also in ExpTime.

Final comments

We have shown the complexity of various downward fragments of XPath, as summarized in Figure 5.14. The highest complexity class we obtained is ExpTime. In the presence of data equality tests, is a well-behaved fragment considering that in the presence of all the axes XPath is undecidable. One important reason for this is the absence of any sibling axis. Indeed, as soon as any horizontal navigation is allowed in the logic, the problem becomes non-primitive recursive. However,

$\downarrow$	$\downarrow_*$	$=$	**Complexity**	**Details**
•			PSPACE-complete	Cor. 5.50
	•		PSPACE-complete	Thm. 5.54
•	•		EXPTIME-complete	(Marx, 2004)
•		•	PSPACE-complete	Prop. 5.49
	•	•	EXPTIME-complete	Thm. 5.45, Thm. 5.46
•	•	•	EXPTIME-complete	Thm. 5.45, Thm. 5.46
$\mathsf{regXPath}(\downarrow, =)$			EXPTIME-complete	Thm. 5.45, Thm. 5.46
$\mathsf{XPath}^{\not\downarrow}(\downarrow_*, =)$			PSPACE-complete	Prop. 5.56
$\mathsf{regXPath}(\downarrow, =) + \mathscr{E}_{tree}$			EXPTIME-complete	Thm. 5.57, Thm. 5.46

Fig. 5.14: Summary of results. All the bounds also hold in the absence of path unions.

we have shown that we can evaluate some restricted fragment of XML Schema or DTD that cannot limit the quantity of occurrences of nodes of a certain type, but that can verify that there is a certain structure in the siblings of the tree. For example, we can express that the children of every node with label book form a sequence of labels in the language $(\mathsf{author}\,(\mathsf{chapter})^*)^+$ (since it is an extensible language). Also, by solving the satisfiability problem we are also able to solve the containment and equivalence problems of node expressions for free, since we work with logics closed under boolean operators. We leave open the question of whether the inclusion of path expressions (as binary relations) is also decidable in EXPTIME.

We introduced the new class of Downward Data automata that capture all the expressivity of $\mathsf{regXPath}(\downarrow, =)$. This automata model is more expressive than XPath. It can test properties like, for example, that there are exactly 7 data values with label book; or that every node labeled book has between 1 and 4 children author with different data value; or that there is a data value that can be simultaneously accessed by three different branches, satisfying "$\downarrow_*[\mathsf{article}]\downarrow[\mathsf{author}]$", "$\downarrow_*[\mathsf{conference}]\downarrow[\mathsf{chair}]$", and "$\downarrow_*[\,\mathsf{scientist} \wedge \langle\, (\downarrow[\mathsf{advisor}])^*\downarrow[\mathsf{sex}]\downarrow[\mathsf{female}]\,\rangle\,]\downarrow[\mathsf{name}]$".

By the proof of decidability of the DD automata, we conclude that there is a normal form of the model for downward XPath. If a formula $\eta \in \mathsf{regXPath}(\downarrow, =)$ is satisfiable, then it is satisfiable in a model of exponential height and polynomial branching width, whose data values are such that only a polynomial number of data values can be shared between any two disjoint subtrees. This property is reflected by the fact that the emptiness of the automaton that results from the translation of a downward XPath formula only depends on a polynomial number of data values at every position. However, there is no syntactic restriction in the automaton, it can retrieve and compare any number of data values between them and the root's data value at each step of its execution.

If we extend the downward fragment with horizontal axes $\rightarrow$, $\rightarrow^*$ we obtain

the forward fragment. This fragment is decidable with non-primitive recursive complexity. In the next chapter we will show how to prove this result.

6. DOWNWARD AND RIGHTWARD NAVIGATION

6.1 Introduction

This chapter deals with logics and automata for data trees with a *forward* behavior, in the sense that we can not only move downwards in the tree, but we can also navigate (in only one sense) the sequence of siblings. In Section 6.2, we extend the model ARA(guess, spread) to the new model ATRA(guess, spread) that runs over data trees (instead of data words). The decidability of the emptiness follows easily from the decidability result shown for ARA(guess, spread). As in the case of data trees, this model allows to show the decidability of a logic.

In Section 6.4 we investigate the satisfiability of forward XPath. The satisfiability problem for this logic will follow by a reduction to the emptiness problem of ATRA(guess, spread). This reduction is not trivial, since XPath is closed under negation and our automata model is not closed under complementation. Indeed, ATRA(guess, spread) and forward XPath have incomparable expressive power.

Related work

This work is an extension of the work of Jurdziński and Lazić (2008), in the same way as the automaton of Chapter 3 is an extension of the work of Demri and Lazić (2009). In the same way as in Part I, we will show that our extensions add expressive power to the ATRA model.

6.2 Automata model

We introduce the class of Alternating Tree Register Automata by slightly adapting the definition for alternating (word) register automata. This model is essentially the same automaton presented in Part I, that works on a (unranked, ordered) *data tree* instead of a data word. The only difference is that instead of having one instruction $\triangleright$ that means 'move to the next position', we have two instructions $\triangleright$ and $\triangledown$ meaning 'move to the next sibling to the right' and 'move to the leftmost child'. This class of automata is known as ATRA(guess, spread). This model of computation will enable us to show decidability of a large fragment of XPath.

An **Alternating Tree Register Automaton** (ATRA) consists in a top-down tree walking automaton with alternating control and *one* register to store and test data. Jurdziński and Lazić (2008) showed that its finitary emptiness problem is

decidable and non primitive recursive. Here, as in the Part I, we consider an extension with the operators spread and guess. We call this model ATRA(spread, guess).

Definition 6.1. An alternating tree register automaton of ATRA(spread, guess) is a tuple $\mathcal{M} = \langle \mathbb{A}, Q, q_I, \delta \rangle$ such that $\mathbb{A}$ is a finite alphabet; Q is a finite set of states; $q_I \in Q$ is the initial state; and $\delta : Q \to \Phi$ is the transition function, where Φ is defined by the grammar

$$a \mid \bar{a} \mid \odot? \mid \mathsf{store}(q) \mid \mathsf{eq} \mid \overline{\mathsf{eq}} \mid q \wedge q' \mid q \vee q' \mid \nabla q \mid \triangleright q \mid \mathsf{guess}(q) \mid \mathsf{spread}(q, q')$$

where $a \in \mathbb{A}, q, q' \in Q$, $\odot \in \{\nabla, \bar{\nabla}, \triangleright, \bar{\triangleright}\}$.

We only focus on the differences with respect to the ARA class. ∇ and $\triangleright$ are to move to the leftmost child or to the next sibling to the right of the current position, and as before '$\odot?$' tests the current type of the position of the tree. For example, using $\bar{\nabla}?$ we test that we are in a leaf node, and by $\triangleright?$ that the node has a sibling to its right. $\mathsf{store}(q)$, eq and $\overline{\mathsf{eq}}$ work in the same way as in the ARA model. We say that a state $q \in Q$ is **moving** if $\delta(q) = \triangleright q'$ or $\delta(q) = \nabla q'$ for some $q' \in Q$.

We define two sorts of configurations: *node* configurations and *tree* configurations. In this context a **node configuration** is a tuple $\langle x, \alpha, \gamma, \Delta \rangle$ that describes the partial state of the execution at a position x of the tree. $x \in \mathsf{pos}(\mathbf{t})$ is the current position in the tree $\mathbf{t}$, $\gamma = \mathbf{t}(x) \in \mathbb{A} \times \mathbb{D}$ is the current node's symbol and datum, and $\alpha = \mathsf{type}_{\mathbf{t}}(x)$ is the tree type of x. As before, $\Delta \in \wp_{<\infty}(Q \times \mathbb{D})$ is a finite collection of execution threads. $\mathcal{N}_{\text{ATRA}}$ is the set of all node configurations. A **tree configuration** is just a finite set of node configurations, like $\{\langle \epsilon, \alpha, \gamma, \Delta \rangle, \langle 1211, \alpha', \gamma', \Delta' \rangle, \dots\}$. The run will be defined in such a way that a tree configuration never contains two node configurations in a descendant/ancestor relation. We call $\mathcal{T}_{\text{ATRA}} = \wp_{<\infty}(\mathcal{N}_{\text{ATRA}})$ the set of all tree configurations.

We define the *non-moving* relation $\to_\varepsilon$ over node configurations just as before. As a difference with the ARA mode, we have two types of moving relations. The *first-child* relation $\to_\nabla$, to move to the leftmost child, and the *next-sibling* relation $\to_\triangleright$ to move to the next sibling to the right.

The $\to_\nabla$ and $\to_\triangleright$ are defined, for any $\alpha_1 \in \{\nabla, \bar{\nabla}, \triangleright, \bar{\triangleright}\}$, $\gamma, \gamma_1 \in \mathbb{A} \times \mathbb{D}$, $h \in \{\triangleright, \bar{\triangleright}\}$, $v \in \{\nabla, \bar{\nabla}\}$, as follows

$$\langle x, (\nabla, h), \gamma, \Delta \rangle \to_\nabla \langle x{\cdot}1, \alpha_1, \gamma_1, \Delta_\nabla \rangle, \tag{6.1}$$

$$\langle x{\cdot}i, (v, \triangleright), \gamma, \Delta \rangle \to_\triangleright \langle x{\cdot}(i+1), \alpha_1, \gamma_1, \Delta_\triangleright \rangle \tag{6.2}$$

iff (i) the configuration is 'moving' (*i.e.*, all the threads (q, d) contained in Δ are of the form $\delta(q) = \nabla q'$ or $\delta(q) = \triangleright q'$); and (ii) for $\odot \in \{\nabla, \triangleright\}$, $\Delta_\odot = \{(q', d) \mid (q, d) \in \Delta, \delta(q) = \odot\, q'\}$.

Let $\to \;:=\; \to_\varepsilon \cup \to_\nabla \cup \to_\triangleright \;\subseteq \mathcal{N}_{\text{ATRA}} \times \mathcal{N}_{\text{ATRA}}$. Note that through $\to$ we obtain a run over a *branch* of the tree (if we think about the underlying binary tree according to the first-child and next-sibling relation). In order to maintain all

information about the run over all branches we need to lift this relation to tree configurations. We define the transition between tree configurations that we write $\twoheadrightarrow$. This corresponds to applying a 'non-moving' $\to_\varepsilon$ to a node configuration, or to apply a 'moving' $\to_\triangledown$, $\to_\triangleright$, or both to a node configuration according to its type. That is, we define $\mathcal{S}_1 \twoheadrightarrow \mathcal{S}_2$ iff one of the following conditions holds:

1. $\mathcal{S}_1 = \{\rho\} \cup \mathcal{S}'$, $\mathcal{S}_2 = \{\tau\} \cup \mathcal{S}'$, $\rho \to_\varepsilon \tau$;
2. $\mathcal{S}_1 = \{\rho\} \cup \mathcal{S}'$, $\mathcal{S}_2 = \{\tau\} \cup \mathcal{S}'$, $\rho = \langle x, (\triangledown, \bar{\triangleright}), \gamma, \Delta\rangle$, $\rho \to_\triangledown \tau$;
3. $\mathcal{S}_1 = \{\rho\} \cup \mathcal{S}'$, $\mathcal{S}_2 = \{\tau\} \cup \mathcal{S}'$, $\rho = \langle x, (\bar{\triangledown}, \triangleright), \gamma, \Delta\rangle$, $\rho \to_\triangleright \tau$;
4. $\mathcal{S}_1 = \{\rho\} \cup \mathcal{S}'$, $\mathcal{S}_2 = \{\tau_1, \tau_2\} \cup \mathcal{S}'$, $\rho = \langle x, (\triangledown, \triangleright), \gamma, \Delta\rangle$, $\rho \to_\triangledown \tau_1$, $\rho \to_\triangleright \tau_2$.

A *run* over a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$ is a non-empty sequence $\mathcal{S}_1 \twoheadrightarrow \cdots \twoheadrightarrow \mathcal{S}_n$ with $\mathcal{S}_1 = \{\langle \epsilon, \alpha_0, \gamma_0, \Delta_0\rangle\}$ and $\Delta_0 = \{(q_I, \mathbf{d}(\epsilon))\}$ (*i.e.*, the thread consisting in the initial state with the root's datum), such that for every $i \in [n]$ and $\langle x, \alpha, \gamma, \Delta\rangle \in \mathcal{S}_i$: (1) $x \in \mathsf{pos}(\mathbf{t})$; (2) $\gamma = \mathbf{t}(x)$; and (3) $\alpha = \mathsf{type}_{\mathbf{t}}(x)$. As before, we say that the run is *accepting* if

$$\mathcal{S}_n \subseteq \{\langle x, \alpha, \gamma, \emptyset\rangle \mid \langle x, \alpha, \gamma, \emptyset\rangle \in \mathcal{N}_{\text{ATRA}}\}.$$

The ATRA model is closed under all boolean operations (Jurdziński and Lazić, 2008). However, the extensions introduced guess and spread, while adding expressive power, are not closed under complementation as a trade-off for decidability. It is not surprising that the same properties as for the case of data words apply here.

Proposition 6.2. *ATRA(spread, guess) models have the following properties:*

(i) they are closed under union,

(ii) they are closed under intersection,

(iii) they are not closed under complement.

Example 6.3. We show an example of the expressiveness that guess adds to ATRA. Although as a corollary of Proposition 3.2 we have that the ATRA(guess, spread) class is more expressive than ATRA, we give an example that inherently uses the tree structure of the model. We force that the node at position 2 and the node at position 1·1 of a data tree to have the same data value without any further data constraints. Note that this datum does not necessarily has to appear at some common ancestor of these nodes. Consider the ATRA(guess) defined over $\mathbb{A} = \{a\}$ with

$$\delta(q_0) = \mathsf{guess}(q_1), \quad \delta(q_1) = \triangledown q_2, \quad \delta(q_2) = q_3 \wedge q_4,$$
$$\delta(q_3) = \triangledown q_5, \quad \delta(q_4) = \triangleright q_5, \quad \delta(q_5) = \mathsf{eq}.$$

For the data trees of Figure 6.1, any ATRA either accepts both, or rejects both. This is because when a thread is at position 1, and performs a moving operation

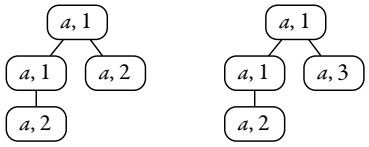

Fig. 6.1: Two indistinguishable data trees for ATRA.

splitting into two threads, one at position 1·1, the other at position 2, none of these configurations contain the data value 2. Otherwise, the automaton should have read the data value 2, which is not the case. But then, we see that the continuation of the run from the node configuration at position 2 and at position 1 are isomorphic independently of which data values we choose (either the data 2 and 3, or the data 2 and 2). Hence, both trees are accepted or rejected. However, the ATRA(guess) we just built distinguishes them, as it can introduce the data value 2 in the configuration, without the need of reading it from the tree. Equivalently, the property of "there are two leaves with the same data values" is expressible in ARA(guess) and not in ARA. □

6.3 The emptiness problem

We show that the emptiness problem for this model is decidable, reusing the results of Part I. We remind the reader that the decidability of the emptiness of ATRA was proved by Jurdziński and Lazić. Here we extend the approach used for ARA and show the decidability of the two extensions spread and guess.

Theorem 6.4. *The finitary emptiness problem of ATRA(guess, spread) is decidable.*

Proof. The proof goes as follows. We will reuse the wqo $\precsim$ used in Section 3.3.2, which here is defined over the *node configurations*. The only difference being that to use $\precsim$ over $\mathcal{N}_{\text{ATRA}}$ we work with *tree* types instead of word types. Since $\to_{\triangleright}$ and $\to_{\triangledown}$ are analogous, by the same proof as in Lemma 3.10 we obtain the following.

Lemma 6.5. $(\mathcal{N}_{\text{ATRA}}, \to)$ *is* rdc *with respect to* $(\mathcal{N}_{\text{ATRA}}, \precsim)$.

We now lift this result to *tree configurations*. We instantiate Proposition 2.16 by taking $\to_1$ as $\to$, $\leq$ as $\precsim$, and taking $\leq_\wp$ the dominance order over $(\mathcal{N}_{\text{ATRA}}, \precsim)$. We take $\to_2$ to be $\twoheadrightarrow$ as it verifies the hypothesis demanded in the Lemma. As a result we obtain the following.

Lemma 6.6. $(\mathcal{T}_{\text{ATRA}}, \twoheadrightarrow)$ *is* rdc *with respect to* $(\mathcal{T}_{\text{ATRA}}, \leq_\wp)$.

Hence, condition (1) of Proposition 2.7 is met. Let us write $\equiv$ for the equivalence relation over $\mathcal{T}_{\text{ATRA}}$ such that $\mathcal{S} \equiv \mathcal{S}'$ iff $\mathcal{S} \leq_\wp \mathcal{S}'$ and $\mathcal{S}' \leq_\wp \mathcal{S}$. Similarly as for Part I, we have that $(\mathcal{T}_{\text{ATRA}}/\equiv, \twoheadrightarrow)$ is finitely branching and effective. That is, the $\twoheadrightarrow$-image of any configuration has only a finite number of configurations up to isomorphism of the data values contained (remember that only equality between

data values matters), and representatives for every class are effectively computable. Hence, we have that $(\mathcal{T}_{\text{ATRA}}/\equiv, \twoheadrightarrow, \leq_\wp)$ verifies condition (2) of Proposition 2.7. Finally, condition (3) holds as $(\mathcal{T}_{\text{ATRA}}/\equiv, \leq_\wp)$ is a wqo (by Proposition 2.15) that is a computable relation. We conclude the proof by the following obvious statement.

Lemma 6.7. *The set of accepting tree configurations is downwards closed with respect to* $\leq_\wp$.

Hence, by Lemma 2.8, we conclude as before that the emptiness problem for ATRA(guess, spread) is decidable. □

6.4 Satisfiability of forward XPath

We consider a navigational fragment of XPath 1.0 with data equality and inequality. In particular this logic is here defined over *data trees*. However, an XML document may typically have not *one* data value per node, but a set of *attributes*, each carrying a data value. This is not a problem since every attribute of an XML element can be encoded as a child node in a data tree labeled by the attribute's name. Thus, all the decidability results hold also for XPath with attributes over XML documents.

We define $\mathsf{sub}(\varphi)$ to denote the set of all substrings of φ which are formulæ, $\mathsf{psub}(\varphi) := \{\alpha \mid \alpha \in \mathsf{sub}(\varphi), \alpha \text{ is a path expression}\}$, and $\mathsf{nsub}(\varphi) := \{\psi \mid \psi \in \mathsf{sub}(\varphi), \psi \text{ is a node expression}\}$.

Key constraints It is worth noting that $\mathsf{XPath}(\mathfrak{F}, =)$, contrary to $\mathsf{XPath}^\varepsilon(\mathfrak{F}, =)$, can express unary *key constraints*. That is, whether for some symbol a, all the a-elements in the tree have different data values.

Lemma 6.8. *For every* $a \in \mathbb{A}$ *let* $key(a)$ *be the property over a data tree* $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$*: "For every two different positions* $x, x' \in \mathsf{pos}(\mathbf{t})$ *of the tree, if* $\mathbf{a}(x) = \mathbf{a}(x') = a$*, then* $\mathbf{d}(x) \neq \mathbf{d}(x')$*". Then,* $pk(a)$ *is expressible in* $\mathsf{XPath}(\mathfrak{F}, =)$ *for any* a.

Proof. It is easy to see that the *negation* of this property can be tested by first *guessing* the closest common ancestor of two different a-elements with equal datum in the underlying 'first child'-'next sibling' binary tree coding. At this node, we verify the presence of two a-nodes with equal datum, one accessible with a "$\downarrow_*$" relation and the other with a compound "$\rightarrow^+ \downarrow_*$" relation (hence the nodes are different). The expressibility of the property then follows from the logic being closed under negation. The reader can check that the following formula expresses the property.

$$key(a) \equiv \neg\langle\, \downarrow_*[\, \langle \varepsilon[a] = \downarrow_+[a] \rangle \vee \langle \downarrow_*[a] = \rightarrow^+ \downarrow_*[a] \rangle\,]\, \rangle$$ □

6.4.1 Satisfiability problem

This section is mainly dedicated to the decidability of $\mathsf{XPath}(\mathfrak{F}, =)$, known as 'forward-XPath'. This is proved by a reduction to the emptiness problem of the automata model $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ introduced in Section 6.2.

Jurdziński and Lazić show that ATRA captures the fragment $\mathsf{XPath}^{\varepsilon}(\mathfrak{F}, =)$. It is immediate to see that ATRA can easily capture the Kleene star operator on any path formula, obtaining decidability of $\mathsf{regXPath}^{\varepsilon}(\mathfrak{F}, =)$. However, these decidability results cannot be generalized to the full unrestricted forward fragment $\mathsf{XPath}(\mathfrak{F}, =)$ as ATRA is not powerful enough to capture the full expressivity of the logic. It cannot express, for instance, that there are two different leaves with the same data value. On the other hand, $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ can express this property. But it cannot express the *negation* of the property!

Indeed, the $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ model cannot capture $\mathsf{XPath}(\mathfrak{F}, =)$. Indeed, data tests of the form $\neg\langle\alpha = \beta\rangle$ are impossible to perform for $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ as this would require—in some sense—the ability to guess two disjoint sets of data values S_1, S_2 such that all α-paths lead to a data value of S_1, and all β-paths lead to a data value of S_2. Still, in the sequel we show that there exists a reduction from the satisfiability of $\mathsf{regXPath}(\mathfrak{F}, =)$ to the emptiness of $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$, and hence that the former problem is decidable. This result settles an open question regarding the decidability of the satisfiability problem for the forward-XPath fragment: $\mathsf{XPath}(\mathfrak{F}, =)$. The main results that will be shown in Section 6.4.2 are the following.

Theorem 6.9. *Satisfiability of* $\mathsf{regXPath}(\mathfrak{F}, =)$ *in the presence of DTDs (or any regular language) and unary key constraints is decidable, non primitive recursive.*

And hence the next corollary follows from the logic being closed under boolean operations.

Corollary 6.10. *The query containment and the query equivalence problems are decidable for* $\mathsf{XPath}(\mathfrak{F}, =)$.

Moreover, these decidability results hold for $\mathsf{regXPath}(\mathfrak{F}, =)$ and even for two extensions:

- a navigational extension with *upward* axes (in Section 6.4.3), and
- a generalization of the data tests that can be performed (in Section 6.4.5).

Data trees or XML documents?

Although our main motivation for working with trees is related to static analysis of logics for XML documents, we work with *data trees*, being a simpler formalism to work with, from where results can be transferred to the class of XML documents. We discuss briefly how all the results we give on XPath over data trees, also hold for the class of XML documents.

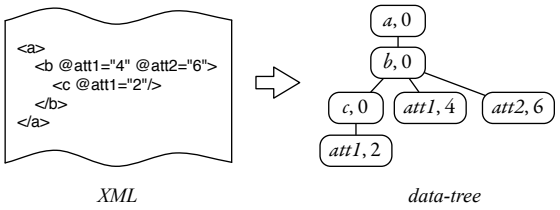

Fig. 6.2: From XML documents to data-trees.

While a data tree has *one* data value for each node, an XML document may have *several* attributes at a node, each with a data value. However, every attribute of an XML element can be encoded as a child node in a data tree labeled by the attribute's name, as in Figure 6.2. This coding can be enforced by the formalisms we present below, and we can thus transfer all the decidability results to the class of XML documents. In fact, it suffices to demand that all the attribute symbols can only occur at the leaves of the data tree and to interpret attribute expressions like '@*attrib1*' of XPath formulæ as child path expressions '$\downarrow[attrib1]$'.

6.4.2 Decidability of forward XPath

This section is devoted to the proof of the following statement.

Proposition 6.11. *For every* $\eta \in \mathsf{regXPath}(\mathfrak{F}, =)$ *there exists an effectively computable* $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ *automaton* $\mathcal{M}$ *such that* $\mathcal{M}$ *is non-empty iff* η *is satisfiable.*

Markedly, the $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ class does not capture $\mathsf{regXPath}(\mathfrak{F}, =)$. However, given formula η, it is possible to construct an automaton that tests a property that guarantees the existence of a data tree verifying η.

Disjoint values property

To show the above proposition, we need to work with runs with the *disjoint values property* as stated next.

Definition 6.12. A run $\mathcal{S}_1 \twoheadrightarrow \cdots \twoheadrightarrow \mathcal{S}_n$ on a data tree $\mathbf{t}$ has the **disjoint values property** if for every $x{\cdot}i \in \mathsf{pos}(\mathbf{t})$ and ρ a *moving* node configuration of the run with position $x{\cdot}i$, then

$$data(\mathbf{t}|_{x\cdot i}) \;\cap \bigcup_{\substack{x\cdot j \in \mathsf{pos}(\mathbf{t}) \\ j>i}} data(\mathbf{t}|_{x\cdot j}) \;\subseteq\; data(\rho)\ .$$

Figure 6.3 illustrates this property.

The proof of Proposition 6.11 can be sketched as follows:

1. We show that for every nonempty automaton $\mathcal{M} \in \mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ there is an accepting run on a data tree with the disjoint values property.

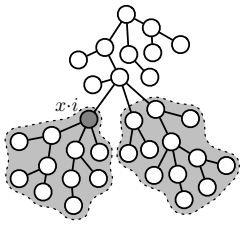

Fig. 6.3: The disjoint values property states that for every position $x{\cdot}i$, the intersection of the grey zones is present in the last configuration for $x{\cdot}i$ appearing in the run.

2. We give an effective translation from an arbitrary forward XPath formula η to an ATRA(guess, spread) automaton $\mathcal{M}$ such that
   - any tree accepted by a run of the automaton $\mathcal{M}$ with the disjoint values property verifies the XPath formula η, and
   - any tree verified by the formula η is accepted by a run of the automaton $\mathcal{M}$ with the disjoint values property.

We start by proving the disjoint values property normal form.

Proposition 6.13. *For any nonempty automaton $\mathcal{M} \in$ ATRA(guess, spread) there exists a run over a data tree with the disjoint values property.*

Proof. Given any accepting run $\mathcal{S}_1 \twoheadrightarrow \cdots \twoheadrightarrow \mathcal{S}_n$ on a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$, we show how to modify the run and the tree in order to satisfy the disjoint values property. We only need to replace some of the data values, so that the resulting tree and accepting run will be essentially the same. For any two positions $x, y \in \mathsf{pos}(\mathbf{t})$ let us write $x \preceq^{\dagger} y$ if x is an ancestor of y in the first-child and next-sibling underlying binary tree. That is, if from x we can access y by traversing the tree through the operations 'go to the leftmost child' and 'go to the next sibling to the right'.

Take any $x{\cdot}k \in \mathsf{pos}(\mathbf{t})$, and let $\rho_{x\cdot k} \in \mathcal{S}_i$ for some i be such that $\rho_{x\cdot k}$ is a moving node configuration with position $x{\cdot}k$. Consider any injective function

$$f \;:\; \mathit{data}(\mathbf{t}|_{x\cdot k}) \setminus \mathit{data}(\rho_{x\cdot k}) \;\to\; \mathbb{D} \setminus \mathit{data}(\mathbf{t})$$

and let $\hat{f} : \mathit{data}(\mathbf{t}|_{x\cdot k}) \to \mathbb{D}$ such that $\hat{f}(d) = d$ if $d \in \mathit{data}(\rho_{x\cdot k})$, or $\hat{f}(d) = f(d)$ otherwise. Note that $\hat{f}$ is injective. Let us consider then $\mathcal{S}'_1, \ldots, \mathcal{S}'_n$ where $\mathcal{S}'_j$ consists in replacing every $\rho \in \mathcal{S}_j$ with $\rho' = h(\rho)$, where[1]

$$h(\rho) = \begin{cases} \hat{f}(\rho) & \text{if } \rho \text{ has a position } y \succeq x{\cdot}k \\ \rho & \text{otherwise.} \end{cases}$$

Let us also consider the data tree $\mathbf{t}'$ that results from the replacement in $\mathbf{t}$ of every data value of a position $y \succeq x{\cdot}k$ by $\hat{f}(\mathbf{d}(y))$. We show that $\mathcal{S}'_1 \twoheadrightarrow \cdots \twoheadrightarrow \mathcal{S}'_n$ is still an

[1] By $h(\rho)$ we denote the replacement of every data value d by $h(d)$ in ρ.

accepting run of $\mathcal{M}$ over $\mathbf{t}'$. Take any leaf y which is rightmost (*i.e.*, with no siblings to its right) and consider the sequence of node configurations $\rho_1 \in \mathcal{S}_1, \ldots, \rho_n \in \mathcal{S}_n$ that are ancestors of y in the first-child next-sibling underlying tree structure. That is, every ρ_i carries a position z such that $z \preceq^\dagger y$. This is the 'sub-run' that leads to y: for every ρ_i, ρ_{i+1} either $\rho_i = \rho_{i+1}$ or $\rho_i \to \rho_{i+1}$. If $y \succeq x{\cdot}k$, take ℓ to be the first index such that ρ'_ℓ has position $x{\cdot}y$. Note that the new sequence $\rho'_\ell \in \mathcal{S}'_\ell, \ldots, \rho'_n \in \mathcal{S}'_n$ where $\rho'_j = \hat{f}(\rho_j) = h(\rho_j)$ is isomorphic, modulo renaming of data values, to $\rho_\ell, \ldots, \rho_n$ since $\hat{f}$ is injective. We have that $\rho'_1, \ldots, \rho'_\ell, \ldots \rho'_n$ is a correct run on node configurations, since

- $\rho'_1, \ldots, \rho'_{\ell-1}$ is equal to $\rho_1, \ldots, \rho_{\ell-1}$ (it is not modified by h),
- $\rho'_\ell, \ldots, \rho'_n$ is isomorphic to $\rho_\ell, \ldots, \rho_n$ (we applied an injection $\hat{f}$ to every data value), and
- $\rho'_{\ell-1}, \rho'_\ell$ are isomorphic to $\rho_{\ell-1}, \rho_\ell$, as h does not modifies any shared value.

On the other hand, if $y \not\succeq x{\cdot}k$, then nothing was modified: $\rho'_1 = \rho_1 \in \mathcal{S}_1, \ldots, \rho'_n = \rho_n \in \mathcal{S}_n$. In any case, we have that $\rho'_1, \ldots, \rho'_n$ is a correct (sub-)run on node configurations.

This means that the modified data values are innocuous for the run. As the structure of the run is not changed, this implies that $\mathcal{S}'_1 \twoheadrightarrow \cdots \twoheadrightarrow \mathcal{S}'_n$ is an accepting run that verifies the disjoint values property for $x{\cdot}k$. If we perform the same argument for every position of the tree, we end up with an accepting run and tree with the disjoint values property. □

Using the disjoint values property, we define a translation from $\mathsf{regXPath}(\mathfrak{F}, =)$ formulæ to $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$. Let η be a $\mathsf{regXPath}(\mathfrak{F}, =)$ formula and let $\mathcal{M}$ be the corresponding $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ automaton defined by the translation. We show that (i) if a data tree $\mathbf{t}$ is accepted by $\mathcal{M}$ by a run verifying the disjoint values property, then $\mathbf{t} \models \eta$, and in turn (ii) if $\mathbf{t} \models \eta$, then $\mathbf{t}$ is accepted by $\mathcal{M}$. Thus, by the disjoint values normal form (Proposition 6.13) we obtain our main result of Proposition 6.11, which by decidability of $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ (Theorem 6.4) implies that the satisfiability problem for $\mathsf{regXPath}(\mathfrak{F}, =)$ is decidable.

The translation

Let η be a $\mathsf{regXPath}(\mathfrak{F}, =)$ node expression in negation normal form (nnf for short). For succinctness and simplicity of the translation, we assume that η is in a normal form such that the $\downarrow$-axis is interpreted as the *leftmost* child. To obtain this normal form, it suffices to replace every appearance of '$\downarrow$' by '$\downarrow \to^*$'. For every path expression $\alpha \in \mathsf{psub}(\eta)$, consider a deterministic complete finite automaton $\mathcal{H}_\alpha$ over the alphabet $\mathbb{A}_\eta = \{\varphi \mid \varphi \in \mathsf{nsub}(\eta)\} \cup \{\downarrow, \to\}$ which corresponds to that regular expression. We assume the following names of its components: $\mathcal{H}_\alpha = \langle \mathbb{A}_\eta, \delta_\alpha, Q_\alpha, 0, F_\alpha \rangle$, where $Q_\alpha \subseteq \mathbb{N}$ is the finite set of states and $0 \in Q_\alpha$ is

$$\begin{aligned}
\mathsf{nnf}(\varphi \wedge \psi) &:= \mathsf{nnf}(\varphi) \wedge \mathsf{nnf}(\psi) & \mathsf{nnf}(\varphi \vee \psi) &:= \mathsf{nnf}(\varphi) \vee \mathsf{nnf}(\psi)\\
\mathsf{nnf}(\neg(\varphi \wedge \psi)) &:= \mathsf{nnf}(\neg\varphi) \vee \mathsf{nnf}(\neg\psi) & \mathsf{nnf}(\neg(\varphi \vee \psi)) &:= \mathsf{nnf}(\neg\varphi) \wedge \mathsf{nnf}(\neg\psi)\\
\mathsf{nnf}(\alpha\,\beta) &:= \mathsf{nnf}(\alpha)\,\mathsf{nnf}(\beta) & \mathsf{nnf}([\varphi]) &:= [\mathsf{nnf}(\varphi)]\\
\mathsf{nnf}(\alpha^*) &:= (\mathsf{nnf}(\alpha))^* & \mathsf{nnf}(o) &:= o \quad o \in \mathfrak{F}\\
\mathsf{nnf}(\langle \alpha \circledast \beta\rangle) &:= \langle \mathsf{nnf}(\alpha) \circledast \mathsf{nnf}(\beta)\rangle & \mathsf{nnf}(\neg\langle \alpha \circledast \beta\rangle) &:= \neg\langle \mathsf{nnf}(\alpha) \circledast \mathsf{nnf}(\beta)\rangle\\
\mathsf{nnf}(\mathsf{a}) &:= \mathsf{a} & \mathsf{nnf}(\neg\mathsf{a}) &:= \neg\mathsf{a}\\
\mathsf{nnf}(\neg\neg\varphi) &:= \mathsf{nnf}(\varphi) & \mathsf{nnf}(\langle\alpha\rangle) &:= \langle \mathsf{nnf}(\alpha)\rangle
\end{aligned}$$

Fig. 6.4: Definition of the Negation Normal Form for XPath.

the initial state. We next show how to translate η into an ATRA(guess, spread) automaton $\mathcal{M}$. For the sake of readability we define the transitions as positive boolean combinations of $\vee$ and $\wedge$ over the set of basic tests and states. Any of these —take for instance $\delta(q) = (\mathsf{store}(q_1) \wedge \nabla q_2) \vee (q_3 \wedge \bar{\mathsf{a}})$— can be rewritten into an equivalent ATRA with at most one boolean connector per transition (as in Definition 6.1) in polynomial time. The most important cases are those relative to the following data tests:

$$(1)\ \langle \alpha = \beta\rangle \qquad (2)\ \langle \alpha \neq \beta\rangle \qquad (3)\ \neg\langle \alpha = \beta\rangle \qquad (4)\ \neg\langle \alpha \neq \beta\rangle$$

We define the ATRA(guess, spread) automaton

$$\mathcal{M} := \langle \mathbb{A}, Q, (\!|\eta|\!), \delta\rangle$$

with

$$\begin{aligned}
Q := \{&(\!|\varphi|\!), (\!|\alpha|\!)^{\circledast}_{\mathcal{C},i}, (\!|\alpha|\!)^{\circledast}_{\mathsf{F}}, (\!|\alpha,\beta|\!)^{\circledast}_{\mathcal{C},i,\mathcal{E},j} \mid \varphi \in \mathsf{nsub}^{\neg}(\eta),\\
&\alpha,\beta \in \mathsf{psub}^{\neg}(\eta), \circledast \in \{=,\neq,\neg=,\neg\neq\},\\
&i \in Q_\alpha, \mathcal{C} \subseteq Q_\alpha, j \in Q_\beta, \mathcal{E} \subseteq Q_\beta\}
\end{aligned}$$

where $\mathsf{op}^{\neg}$ is the smallest superset of op closed under negation under nnf, *i.e.*, if $\varphi \in \mathsf{op}^{\neg}(\eta)$ then $\mathsf{nnf}(\neg\varphi) \in \mathsf{op}^{\neg}(\eta)$ (where nnf is defined as shown in Figure 6.4). The idea is that a state $(\!|\varphi|\!)$ verifies the formula φ. A state $(\!|\alpha|\!)^{=}_{\mathcal{C},i}$ (resp. $(\!|\alpha|\!)^{\neq}_{\mathcal{C},i}$) verifies that there is a forward path in the tree ending at a node with the same (resp. different) data value as the one in the register, such that there exists a run of $\mathcal{H}_\alpha$ over such path that starts in state i and ends in a final state. Similarly, a state $(\!|\alpha,\beta|\!)^{=}_{\mathcal{C},i,\mathcal{E},j}$ (resp. $(\!|\alpha,\beta|\!)^{\neq}_{\mathcal{C},i,\mathcal{E},j}$) verifies that there are *two* paths ending in two nodes with the same (resp. one equal, the other different) data value as that of the register; such that one path has a partial accepting run of $\mathcal{H}_\alpha$ starting in state i, and the other has a partial accepting run of $\mathcal{H}_\beta$ starting in state j. The sets $\mathcal{C}, \mathcal{E}$ are not essential to understand the general construction, and they have as

only purpose to disallow non-moving loops in the definition of δ. A state like $(\!|\alpha|\!)^{=}_{\mathsf{F}}$ is simply to mark that the run of $\mathcal{H}_\alpha$ on a path has ended, and the only remaining task is to test for equality of the data value with respect to the register. Finally, a state of the sort $(\!|\cdots|\!)^{\neg=}_{\ldots}$ (resp. $(\!|\cdots|\!)^{\neg\neq}_{\ldots}$) has a similar meaning, but it verifies that *all* the data of the nodes along the path(s) are different (resp. equal) to the datum stored in the register. We first take care of the boolean connectors and the simplest tests.

$$\delta((\!|\mathsf{a}|\!)) := \mathsf{a} \quad \delta((\!|\varphi \vee \psi|\!)) := (\!|\varphi|\!) \vee (\!|\psi|\!) \quad \delta((\!|\neg\mathsf{a}|\!)) := \bar{\mathsf{a}} \quad \delta((\!|\varphi \wedge \psi|\!)) := (\!|\varphi|\!) \wedge (\!|\psi|\!)$$

The tests $\langle\alpha\rangle$ and $\neg\langle\alpha\rangle$ are coded in a standard way (*cf.* Jurdziński and Lazić, 2008). Here we focus on the data-aware cases. Using the guess operator, we can easily define the cases corresponding to the data test cases (1) and (2) as follows. Here, $(\!|\alpha|\!)_{\mathsf{F}}$ holds at the endpoint of a path matching α.

$$\delta((\!|\alpha = \beta|\!)) := \mathsf{guess}((\!|\alpha,\beta|\!)^{=}) \quad \delta((\!|\alpha,\beta|\!)^{=}) := (\!|\alpha|\!)^{=}_{\emptyset,0} \wedge (\!|\beta|\!)^{=}_{\emptyset,0} \quad \delta((\!|\alpha|\!)^{=}_{\mathsf{F}}) := \mathsf{eq}$$
$$\delta((\!|\alpha \neq \beta|\!)) := \mathsf{guess}((\!|\alpha,\beta|\!)^{\neq}) \quad \delta((\!|\alpha,\beta|\!)^{\neq}) := (\!|\alpha|\!)^{=}_{\emptyset,0} \wedge (\!|\beta|\!)^{\neq}_{\emptyset,0} \quad \delta((\!|\alpha|\!)^{\neq}_{\mathsf{F}}) := \overline{\mathsf{eq}}$$

We define the transitions associated to each $\mathcal{H}_\alpha$, for $i \in Q_\alpha, \mathcal{C} \subseteq Q_\alpha, \circledast \in \{=,\neq\}$.

$$\delta((\!|\alpha|\!)^{\circledast}_{\mathcal{C},i}) := \bigvee_{\substack{\varphi\in\mathsf{nsub}(\alpha),\\ i':=\delta_\alpha(\varphi,i),\, i'\notin\mathcal{C}}} \big((\!|\varphi|\!) \wedge (\!|\alpha|\!)^{\circledast}_{\mathcal{C}\cup\{i'\},i'}\big)$$
$$\vee\ \nabla(\!|\alpha|\!)^{\circledast}_{\emptyset,\delta_\alpha(\downarrow,i)} \vee \triangleright(\!|\alpha|\!)^{\circledast}_{\emptyset,\delta_\alpha(\rightarrow,i)} \vee \bigvee_{i\in F_\alpha} (\!|\alpha|\!)^{\circledast}_{\mathsf{F}}$$

The test case (4) involves also an *existential* quantification over data values. In fact, $\neg(\alpha \neq \beta)$ means that either (1) there are no nodes reachable by α, or (2) there are no nodes reachable by β, or (3) there *exists* a data value d such that both (a) all elements reachable by α have datum d, and (b) all elements reachable by β have datum d.

$$\delta((\!|\neg\alpha \neq \beta|\!)) := (\!|\neg\langle\alpha\rangle|\!) \vee (\!|\neg\langle\beta\rangle|\!) \vee \mathsf{guess}((\!|\alpha,\beta|\!)^{\neg\neq})$$
$$\delta((\!|\alpha,\beta|\!)^{\neg\neq}) := (\!|\alpha|\!)^{\neg\neq}_{\emptyset,0} \wedge (\!|\beta|\!)^{\neg\neq}_{\emptyset,0} \qquad \delta((\!|\alpha|\!)^{\neg\neq}_{\mathsf{F}}) := \mathsf{eq} \qquad \delta((\!|\alpha|\!)^{\neg=}_{\mathsf{F}}) := \overline{\mathsf{eq}}$$

$$\delta((\!|\alpha|\!)^{\neg\circledast}_{\mathcal{C},i}) := \bigwedge_{\substack{\varphi\in\mathsf{nsub}(\alpha),\\ i':=\delta_\alpha(\varphi,i),\, i'\notin\mathcal{C}}} ((\!|\bar{\varphi}|\!) \vee (\!|\alpha|\!)^{\neg\circledast}_{\mathcal{C}\cup\{i'\},i'})$$
$$\wedge\ (\bar{\nabla}? \vee \nabla(\!|\alpha|\!)^{\neg\circledast}_{\emptyset,\delta_\alpha(\downarrow,i)}) \wedge (\bar{\triangleright}? \vee \triangleright(\!|\alpha|\!)^{\neg\circledast}_{\emptyset,\delta_\alpha(\rightarrow,i)})$$
$$\wedge \bigwedge_{i\in F_\alpha} (\!|\alpha|\!)^{\neg\circledast}_{\mathsf{F}} \quad \text{where } \bar{\varphi} \text{ stands for } \mathsf{nnf}(\neg\varphi).$$

The difficult part is the translation of the data test case (3). The main reason for this difficulty is the fact that $\mathsf{ATRA}(\mathsf{guess},\mathsf{spread})$ automata do not have the

expressivity to make these kinds of tests. An expression $\neg\langle\alpha = \beta\rangle$ forces the set of data values reachable by an α-path and the set of those reachable by a β-path to be disjoint. We show that nonetheless the automaton can test for a condition that is equivalent to $\neg\langle\alpha = \beta\rangle$ if we assume that the run and tree have the disjoint values property. For the coding of this property we use a weaker version of spread, and we define $\mathsf{spread}(q) := \bigwedge_{q' \in Q} \mathsf{spread}(q', q)$.

Example 6.14. As an example, suppose that $\eta = \neg\langle{\downarrow}\alpha = {\rightarrow}\beta\rangle$ is to be checked for satisfiability. One obvious answer would be to test separately α and β. If both tests succeed, one can then build a model satisfying η out of the two witnessing trees by making sure they have disjoint sets of values. Otherwise, η is clearly unsatisfiable. Suppose now that we have $\eta = \varphi \wedge \neg\langle{\downarrow}\alpha = {\rightarrow}\beta\rangle$, where φ is any formula with no data tests of type (3). One could build the automaton for φ and then ask for "$\mathsf{spread}(\langle\!|{\downarrow}\alpha|\!\rangle_0^{\neg=} \vee \langle\!|{\rightarrow}\beta|\!\rangle_0^{\neg=})$" in the automaton. This corresponds to the property "*for every data value d taken into account by the automaton (as a result of the translation of φ), either all elements reachable by α do not have datum d, or all elements reachable by β do not have datum d*". If φ contains a $\langle\alpha' = \beta'\rangle$ formula, this translates to a *guessing* of a witnessing data value d. Then, the use of spread takes care of this particular data value, and indeed of all other data values that were guessed to satisfy similar demands. In other words, it is *not* because of d that $\neg\langle{\downarrow}\alpha = {\rightarrow}\beta\rangle$ will be falsified. But then, the disjoint values property ensures that no pair of nodes accessible by α and β share the same datum. This is the main idea we encode next. □

We define $\delta(\langle\!|\neg\langle\alpha = \beta\rangle|\!\rangle) := \langle\!|\alpha, \beta|\!\rangle^{\neg=}_{\emptyset,0,\emptyset,0}$. Given $\neg\langle\alpha = \beta\rangle$, the automaton systematically looks for the closest common ancestor of every pair (x, y) of nodes accessible by α and β respectively, and tests, for every data value d present in the node configuration, that either (1) all data values accessible by the remaining path of α are different from d, or (2) all data values accessible by the remaining path of β are different from d.

$$\begin{aligned}\delta(\langle\!|\alpha, \beta|\!\rangle^{\neg=}_{\mathcal{C}_1,i,\mathcal{C}_2,j}) := \mathsf{spread}\big(\langle\!|\alpha|\!\rangle^{\neg=}_{\emptyset,i} \vee \langle\!|\beta|\!\rangle^{\neg=}_{\emptyset,j}\big) \quad &\wedge \bigwedge_{i \in F_\alpha} \langle\!|\beta|\!\rangle^{\neg=}_{\emptyset,j} \quad \wedge \bigwedge_{j \in F_\alpha} \langle\!|\alpha|\!\rangle^{\neg=}_{\emptyset,i} \\ \wedge \quad \nabla\langle\!|\alpha, \beta|\!\rangle^{\neg=}_{\emptyset,\delta_\alpha({\downarrow},i),\emptyset,\delta_\beta({\downarrow},j)} \quad &\wedge \quad \triangleright\langle\!|\alpha, \beta|\!\rangle^{\neg=}_{\emptyset,\delta_\alpha({\rightarrow},i),\emptyset,\delta_\beta({\rightarrow},j)} \\ \wedge \bigwedge_{\substack{\varphi \in \mathsf{nsub}(\alpha),\\ i' := \delta_\alpha(\varphi,i),\, i' \notin \mathcal{C}_1}} &(\langle\!|\bar{\varphi}|\!\rangle \vee \langle\!|\alpha, \beta|\!\rangle^{\neg=}_{\mathcal{C}_1 \cup \{i'\},i',\mathcal{C}_2,j}) \\ \wedge \bigwedge_{\substack{\varphi \in \mathsf{nsub}(\beta),\\ j' := \delta_\beta(\varphi,j),\, j' \notin \mathcal{C}_2}} &(\langle\!|\bar{\varphi}|\!\rangle \vee \langle\!|\alpha, \beta|\!\rangle^{\neg=}_{\mathcal{C}_1,i,\mathcal{C}_2 \cup \{j'\},j'})\end{aligned}$$

The following lemmas then follow from the discussion above.

Lemma 6.15. *For any data tree* $\mathbf{t}$*, if* $\mathbf{t} \models \eta$*, then* $\mathcal{M}$ *accepts* $\mathbf{t}$*.*

Lemma 6.16. *For any data tree* $\mathbf{t}$*, if* $\mathcal{M}$ *accepts* $\mathbf{t}$ *with a run that has the disjoint values property, then* $\mathbf{t} \models \eta$*.*

Lemmas 6.15 and 6.16 together with Proposition 6.13 conclude the proof of Proposition 6.11. We then have that Theorem 6.9 holds.

Proof of Theorem 6.9. By Proposition 6.11, satisfiability of $\mathsf{regXPath}(\mathfrak{F}, =)$ is reducible to the nonemptiness problem for $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$. On the other hand, we remark that $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ automata can encode any regular tree language —in particular a DTD, the core of XML Schema, or Relax NG— and are closed under intersection by Proposition 6.2. Also, the logic can express any unary key constraint as stated in Lemma 6.8. Hence, by Theorem 6.4 the decidability follows. □

Extensions

We consider some operators that can be added to $\mathsf{regXPath}(\mathfrak{F}, =)$ preserving the decidability of the satisfiability problem. For each of these operators, we will see that they can be coded as a $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ automaton, following the same lines of the translation in Section 6.4.2.

6.4.3 Allowing upward axes

Here we explore one possible decidable extension to the logic $\mathsf{regXPath}(\mathfrak{F}, =)$, whose decidability can be reduced to that of $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$.

Let $\mathsf{regXPath}^{\mathfrak{B}}(\mathfrak{F}, =)$ be the fragment of $\mathsf{regXPath}(\mathfrak{F} \cup \mathfrak{B}, =)$ where $\mathfrak{B} := \{\uparrow, \uparrow^*, \leftarrow, {}^*\leftarrow\}$ defined by the grammar

$$\varphi, \psi ::= \neg \mathtt{a} \mid \mathtt{a} \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle \alpha_{\mathfrak{f}} \rangle \mid \langle \alpha_{\mathfrak{b}} \rangle \mid \langle \alpha_{\mathfrak{f}} \circledast \beta_{\mathfrak{f}} \rangle \mid \neg \langle \alpha_{\mathfrak{f}} \circledast \beta_{\mathfrak{f}} \rangle \mid \neg \langle \alpha_{\mathfrak{b}} = \beta_{\mathfrak{f}} \rangle \mid \neg \langle \alpha_{\mathfrak{b}} \neq \beta_{\mathfrak{f}} \rangle$$

with $\circledast \in \{=, \neq\}$, $\mathtt{a} \in \mathbb{A}$, and

$$\begin{aligned} \alpha_{\mathfrak{f}}, \beta_{\mathfrak{f}} &::= [\varphi] \mid \alpha_{\mathfrak{f}} \beta_{\mathfrak{f}} \mid o\, \alpha_{\mathfrak{f}} \mid (\alpha_{\mathfrak{f}})^* && o \in \{\downarrow, \rightarrow, \varepsilon\}, \\ \alpha_{\mathfrak{b}}, \beta_{\mathfrak{b}} &::= [\varphi] \mid \alpha_{\mathfrak{b}} \beta_{\mathfrak{b}} \mid o\, \alpha_{\mathfrak{b}} \mid (\alpha_{\mathfrak{b}})^* && o \in \{\uparrow, \leftarrow, \varepsilon\}. \end{aligned}$$

We must note that $\mathsf{regXPath}^{\mathfrak{B}}(\mathfrak{F}, =)$ contains $\mathsf{regXPath}(\mathfrak{F}, \mathfrak{B})$, that is, the full data-unaware fragment of XPath. We also remark that it is *not* closed under negation. Indeed, we cannot express the negation of "there exists an a such that all its ancestors labeled b have different data value" which is expressed by $\downarrow_*[a \wedge \neg\langle\uparrow^*[b] = \varepsilon\rangle]$. As shown in Proposition 3.2, if the negation of this property were expressible, then its satisfiability would be undecidable. It is not hard to see that we can decide the satisfiability problem for this fragment.

Consider first the data test expressions of the types

$$\neg\langle \alpha_{\mathfrak{b}} = \beta_{\mathfrak{f}} \rangle \qquad \text{and} \qquad \neg\langle \alpha_{\mathfrak{b}} \neq \beta_{\mathfrak{f}} \rangle$$

where $\beta_{\mathfrak{f}} \in \mathsf{regXPath}(\mathfrak{F}, =)$ and $\alpha_{\mathfrak{b}} \in \mathsf{regXPath}(\mathfrak{B})$. We can decide the satisfaction of these kinds of expressions by means of $\mathsf{spread}(\ ,\ )$, using carefully its first parameter to select the desired threads from which to collect the data values we are interested in. Intuitively, along the run we throw threads that save current data value and try out all possible ways to verify $\alpha^r_{\mathfrak{b}} \in \mathsf{regXPath}(\mathfrak{F}, =)$, where $(\ )^r$ stands for the *reverse* of the regular expression. Let the automaton arrive with a thread $(\langle\!|\alpha_{\mathfrak{b}}|\!\rangle, d)$ whenever $\alpha^r_{\mathfrak{b}}$ is verified. This signals that there is a backwards path from the current node in the relation $\alpha_{\mathfrak{b}}$ that arrives at a node with data value d. Hence, at any given position, the instruction $\mathsf{spread}(\langle\!|\alpha_{\mathfrak{b}}|\!\rangle, \langle\!|\alpha_{\mathfrak{f}}|\!\rangle^{\neg^{\circledast}})$ translates correctly the expression $\neg\langle\alpha_{\mathfrak{b}} \circledast \beta_{\mathfrak{f}}\rangle$. Furthermore, $\alpha_{\mathfrak{b}}$ need not be necessarily in $\mathsf{regXPath}(\mathfrak{B})$, as its intermediate node tests can be formulæ from $\mathsf{regXPath}(\mathfrak{F}, =)$. We then obtain the following result.

Theorem 6.17. *Satisfiability for* $\mathsf{regXPath}^{\mathfrak{B}}(\mathfrak{F}, =)$ *under key constraints and DTDs is decidable.*

6.4.4 XML versus data trees

The decidability of SAT-$\mathsf{XPath}(\mathfrak{F}, =)$ on data trees entails the decidability of SAT-$\mathsf{attrXPath}(\mathfrak{F}, =)$ on XML documents. The way to transfer this results is by the same coding as shown in Section 5.4.3 for downward XPath, and can obviously be done in forward XPath since it is an extension of the downward fragment.

6.4.5 Allowing stronger data tests

Consider the property "*there are three descendant nodes labeled a, b and c with the same data value*". That is, there exists some data value d such that there are three nodes accessible by $\downarrow_*[a]$, $\downarrow_*[b]$ and $\downarrow_*[c]$ respectively, all carrying the datum d. Let us denote the fact that they have the same or different datum by introducing the symbols '$\sim$' and '$\not\sim$', and appending it at the end of the path. Then in this case we write that the elements must satisfy $\downarrow_*[\mathsf{a}]\sim$, $\downarrow_*[\mathsf{b}]\sim$, and $\downarrow_*[\mathsf{c}]\sim$. We then introduce the node expression $\{\!\{\alpha_1 s_1, \ldots, \alpha_n s_n\}\!\}$ where α_i is a path expression and $s_i \in \{\sim, \not\sim\}$ for all $i \in [1..n]$. Semantically, it is a node expression that denotes all the tree positions x from which we can access n positions $x_1, \ldots, x_n$ such that there exists $d \in \mathbb{D}$ where for all $i \in [n]$ the following holds: $(x, x_i) \in [\![\alpha_i]\!]$; if $s_i = \sim$ then $\mathbf{d}(x_i) = d$; and if $s_i = \not\sim$ then $\mathbf{d}(x_i) \neq d$. Note that now we can express $\langle \alpha = \beta \rangle$ as $\{\!\{\alpha\sim, \beta\sim\}\!\}$ and $\langle \alpha \neq \beta \rangle$ as $\{\!\{\alpha\sim, \beta\not\sim\}\!\}$. Let us call $\mathsf{regXPath}^{+}(\mathfrak{F}, =)$ to $\mathsf{regXPath}(\mathfrak{F}, =)$ extended with the construction just explained. This is a more expressive formalism since the first mentioned property —or, to give another example, $\{\!\{\downarrow_*[\mathsf{a}]\sim, \downarrow_*[\mathsf{b}]\sim, \downarrow_*[\mathsf{a}]\not\sim, \downarrow_*[\mathsf{b}]\not\sim\}\!\}$— is not expressible in $\mathsf{regXPath}(\mathfrak{F}, =)$.

We argue that satisfiability for this extension can be decided in the same way as for $\mathsf{regXPath}(\mathfrak{F}, =)$. It is straightforward to see that *positive* appearances can easily be translated with the help of the guess operator. On the other hand, for negative appearances, like $\neg\{\!\{\alpha_1 s_1, \ldots, \alpha_n s_n\}\!\}$, we proceed in the same way as we did for $\mathsf{regXPath}(\mathfrak{F}, =)$. The only difference being that in this case the automaton will

simulate the simultaneous evaluation of the n expressions and calculate all possible configurations of the closest common ancestors of the endpoints, performing a spread at each of these intermediate points.

Theorem 6.18. *Satisfiability of* $\mathsf{regXPath}^{+}(\mathfrak{F}, =)$ *under key constraints and DTDs is decidable.*

6.5 Discussion

Remark 6.19. In Section 3.5 we showed that $\mathsf{LTL}^{\downarrow}(\mathsf{U}, \mathsf{X})$ extended with quantifiers over data values $\forall^{\downarrow}_{\leq}$ and $\exists^{\downarrow}_{\geq}$ is decidable (Theorem 3.31), even in the presence of a linear order. Indeed, by an analogous reduction to $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$, these operators can be added to a CTL version of this logic over data trees, or to the μ-calculus treated by Jurdziński and Lazić (2007)[2]. However, adding the *dual* of any of these operators results in an undecidable logic.

We presented a simplified framework to work with one-way alternating register automata on data words and trees, enabling the possibility to easily show decidability of new operators by proving that they preserve the downward compatibility of a well-structured transition system. It would be interesting to hence investigate more decidable extensions, to study the expressiveness limits of decidable logics and automata for data trees.

Also, this work argues in favor of exploring computational models that although they might be not closed under all boolean operations, can serve to show decidability of logics closed under negation such as forward-XPath.

Another natural fragment of XPath is the case of vertical XPath: the fragment $\mathsf{XPath}(\downarrow, \downarrow_*, \uparrow, \uparrow^*, =)$ containing both downward and upwards navigation. This fragment will be shown to be decidable in our next chapter.

[2] This is the conference version of (Jurdziński and Lazić, 2008).

7. DOWNWARD AND UPWARD NAVIGATION

Two-way automata on data words and trees have frequently an undecidable emptiness problem. We introduce a decidable automata model that, being bottom-up, presents several features that allows to make tests by navigating the tree in both directions: upwards and downwards. This two-way flavor is witnessed by the fact that these automata can decide vertical XPath.

7.1 Introduction

In this chapter we introduce a novel decidable class of automata over unranked data tree, that we denote BUDA, for Bottom-Up alternating 1-register Data tree Automata. The BUDA are essentially alternating bottom-up tree automata with one register, without the ability of testing for "horizontal" properties on the siblings of the tree, such as for example bounding the rank of the tree. However, an automaton of this class has the ability to test rich data properties on the subtrees, which in some sense corresponds to a downward behavior.

The decidability of this automaton is proven in Section 7.3 using a WSTS, with somewhat similar techniques as in Chapters 6 and 3. However, finding the correct wqo that is compatible with the automaton is non-trivial. Since the automaton can faithfully simulate an ARA when going up to the root, the complexity of the emptiness problem is necessarily non-primitive recursive. The absence of horizontal tests is essential to obtain our decidability results. In fact, one can see that the model would become undecidable could it force a bound on the tree's rank.

As a result of the "two-wayness" flavor of the automata model, it can capture the vertical fragment of XPath. The vertical fragment $\mathsf{XPath}(\mathfrak{V}, =)$ is the one containing downward axes $\downarrow$, $\downarrow_*$ and upward axes $\uparrow$, $\uparrow^*$. Our main result on XPath is then the following.

Theorem 7.1. *The satisfiability problem for* $\mathsf{regXPath}(\mathfrak{V}, =)$ *is decidable.*

In this way we answer positively to the open question raised by Benedikt and Koch (2008, Question 5.10) and Benedikt et al. (2008), regarding the decidability of the satisfiability for vertical XPath.

All the results contained in this chapter are joint work with Luc Segoufin and correspond to the work contained in (Figueira and Segoufin, 2011).

7.1.1 Related work

The automata ATRA of Jurdziński and Lazić (2008) and ATRA(guess, spread) of Chapter 6 are the closest to the automata model we present. However, the cited automata are top-down instead of bottom-up, and they can test for horizontal properties. For example, they can express that every node has at most one child, something that cannot be tested by BUDA. On the other hand, BUDA can test properties like (T7) from Example 1.2, that cannot be expressed by any decidable formalism we have mentioned so far. Also, they can express the *inclusion dependency constraint* (cf. § 4.1.3) as (T4). However, the BUDA automata are bottom-up, and cannot test for a property on the siblings.

$\mathsf{EMSO}^2(+1, \sim)$ (Bojańczyk et al., 2009) over data trees is also a logic which is close to $\mathsf{XPath}(\mathfrak{V}, =)$ and BUDA automata since in some sense it is also *two-way*. This logic can express horizontal properties, like the property that restricts the tree to be linear which cannot be expressed by BUDA, but cannot test any property that requires a non-local test, like (T7), or that requires testing some label along the path, like (T6).

7.2 The automata model

In this section we introduce the BUDA model. It is essentially a bottom-up tree automata with one register and an alternating control. We show that these automata are at least as expressive as vertical XPath. In Section 7.3 we will show that their emptiness problem is decidable. Theorem 7.1 then follows immediately.

Recall that the ATRA model of Jurdziński and Lazić (2008), is essentially a *top-down* tree automata with one register and alternating control. An extension of this model can decide *forward* XPath as shown in Chapter 6. We aim at defining a decidable class of automata that can express data properties both in the downwards and the upwards directions. To obtain such a model of automata, the switch from top-down to bottom-up is essential. As a result, this class can capture vertical XPath, and in particular is expressively incomparable with respect to ATRA or ATRA(guess, spread). It also makes the decidability of its emptiness problem significantly more difficult.

An automaton $\mathcal{A} \in \mathsf{BUDA}$ that runs over data trees of $\mathit{Trees}(\mathbb{A} \times \mathbb{D})$ is defined as a tuple $\mathcal{A} = (\mathbb{A}, \mathbb{B}, Q, q_0, \delta_\epsilon, \delta_{up}, \mathcal{S}, h)$ where $\mathbb{A}$ is the finite alphabet of the tree, $\mathbb{B}$ is an internal finite alphabet of the automaton (whose purpose will be clear later), Q is a finite set of states, q_0 is the initial state, $\mathcal{S}$ is a finite semigroup, h is a semigroup homomorphism from $(\mathbb{A} \times \mathbb{B})^+$ to $\mathcal{S}$, δ_ϵ is the ϵ-transition function of $\mathcal{A}$, and δ_{up} is the *up*-transition function of $\mathcal{A}$.

δ_{up} is a partial function from states to formulas. For $q \in Q$, $\delta_{up}(q)$ is either undefined or a formula consisting in a disjunction of conjunctions of states. δ_ϵ is also a partial function from states to disjunctions of conjunctions of 'atoms' of one

of the following forms:

$$p \mid \mathsf{guess}(p) \mid \mathsf{univ}(p) \mid \mathsf{store}(p) \mid \mathsf{eq} \mid \overline{\mathsf{eq}} \mid$$
$$\mid \langle\mu\rangle^{=} \mid \langle\mu\rangle^{\neq} \mid \overline{\langle\mu\rangle^{=}} \mid \overline{\langle\mu\rangle^{\neq}} \mid \mathsf{root} \mid \overline{\mathsf{root}} \mid \mathsf{leaf} \mid \overline{\mathsf{leaf}} \mid a \mid \bar{a} \mid b \mid \bar{b}$$

where $\mu \in \mathcal{S}$, $p \in Q$, $a \in \mathbb{A}$, $b \in \mathbb{B}$.

We first describe the battery of tests the automata can perform. All these tests are explicitly closed under negation, denoted with the $\overline{\cdot}$ notation, and of course they are also closed under intersection and union using the alternating and nondeterministic control of the automata. The automata can test the label and internal label of the current node and also whether the current node is the root, a leaf or an internal node. The automata can test (in)equality of the current data value with the one stored in the register (eq and $\overline{\mathsf{eq}}$). Finally the automata can test the existence of some downward path, starting from the current node and leading to a node whose data value is (or is not) equal to the one currently stored in the register, such that the path satisfies some regular property on the labels. These properties are specified using the finite semigroup $\mathcal{S}$ and the morphism $h : (\mathbb{A} \times \mathbb{B})^+ \to \mathcal{S}$ over the words made of the label of the tree and the internal label. For example, $\langle\mu\rangle^{=}$ tests for the existence of a path that evaluates to μ via h, which starts at the current node and leads to a node whose data value matches the one currently stored in the register. Similarly, $\langle\mu\rangle^{\neq}$ tests that it leads to a data value different from the one currently in the register.

Based on the result of these tests, the automata can perform the following actions. They can change state, store the current data value in the register ($\mathsf{store}(p)$), or store an arbitrary data value nondeterministically chosen ($\mathsf{guess}(p)$). Finally, a transition can demand to start a new thread in state p for every data value of the subtree with the operation $\mathsf{univ}(p)$. The automata can also decide to move up in the tree according to the *up*-transition function.

Before we move on to the formal definition, we stress that the automata model is not closed under complementation because its set of actions are not closed under complementation: guess is a form of existential quantification while univ is a form of universal quantification, but they are not dual. Actually, we will show in the journal version of this paper that adding any of their dual would yield undecidability of the model.

We now turn to the formal definition. A data tree $\mathbf{a} \otimes \mathbf{d} \in \mathit{Trees}(\mathbb{A} \times \mathbb{D})$ is accepted by $\mathcal{A}$ iff there exists an internal labeling $\mathbf{b} \in \mathit{Trees}(\mathbb{B})$ with $\mathsf{pos}(\mathbf{b}) = \mathsf{pos}(\mathbf{a} \otimes \mathbf{d})$ such that there is an accepting run on $\mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$. We focus now on the definition of a run.

A **configuration** of a BUDA $\mathcal{A}$ is a set $\mathcal{C}$ of threads, viewed as a finite subset of $Q \times \mathbb{D}$. A configuration $\mathcal{C}$ is said to be **initial** iff it is of the form $\{(q_0, e)\}$ for some $e \in \mathbb{D}$. A configuration $\mathcal{C}$ is **accepting** iff it is empty.

ϵ-transitions. Let $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ and $x \in \mathsf{pos}(\mathbf{t})$. Given two configurations $\mathcal{C}$ and $\mathcal{C}'$ of $\mathcal{A}$, we say that there is an ϵ-transition of $\mathcal{A}$ at x between $\mathcal{C}$ and $\mathcal{C}'$,

denoted $(x, \mathcal{C}) \rightarrowtail_\epsilon (x, \mathcal{C}')$ (assuming $\mathcal{A}$ and $\mathbf{t}$ are understood from the context) if the following holds: there is a thread with state q and with a data value d (*i.e.*, $(q, d) \in \mathcal{C}$) where $\delta_\epsilon(q) = \bigvee_{i \in I} \gamma_i$. Each γ_i is a conjunction of atoms, and there must be one $i \in I$ with $\gamma_i = \bigwedge_{j \in J} \alpha_j$ and $\mathcal{C}' = (\mathcal{C} \setminus \{(q, d)\}) \cup \hat{\mathcal{C}}$ such that the following holds for all $j \in J$:

- If α_j is one of the tests a, b, root, leaf, eq or its negations, it must be true with the obvious semantics as described above.
- If α_j is $\langle \mu \rangle^=$ then there is a downward path in $\mathbf{t}$ starting at x and ending at some descendant position y with $\mathbf{d}(y) = d$, such that the sequence of labels in $\mathbb{A} \times \mathbb{B}$ read while going from x to y along this path (including the endpoints) evaluates to μ via h. The case of $\langle \mu \rangle^\neq$ is treated similarly replacing $\mathbf{d}(y) = d$ by $\mathbf{d}(y) \neq d$. The tests $\overline{\langle \mu \rangle^=}$ and $\overline{\langle \mu \rangle^\neq}$ correspond to the negation of these tests.
- If α_j is p for some $p \in Q$, then $(p, d) \in \hat{\mathcal{C}}$,
- if α_j is $\mathsf{store}(p)$ then $(p, \mathbf{d}(x)) \in \hat{\mathcal{C}}$,
- if α_j is $\mathsf{guess}(p)$ then $(p, d') \in \hat{\mathcal{C}}$ for some $d' \in \mathbb{D}$,
- if α_j is $\mathsf{univ}(p)$, then for all $d' \in \mathit{data}(\mathbf{t}|_x)$, $(p, d') \in \hat{\mathcal{C}}$,
- nothing else is in $\hat{\mathcal{C}}$.

The **ϵ-closure** of a pair $(x, \mathcal{C})$ is defined as the reflexive transitive closure of $\rightarrowtail_\epsilon$, i.e. the set of configurations reachable from $(x, \mathcal{C})$ by a finite sequence of ϵ-transitions.

up-transitions We say that a configuration $\mathcal{C}$ is **moving** iff for all $(q, d) \in \mathcal{C}$, $\delta_{up}(q)$ is defined. Given two configurations $\mathcal{C}$ and $\mathcal{C}'$ of $\mathcal{A}$, we say that there is an *up*-transition of $\mathcal{A}$ between $\mathcal{C}$ and $\mathcal{C}'$, denoted $\mathcal{C} \rightarrowtail_{up} \mathcal{C}'$ (assuming $\mathcal{A}$ is understood from the context) if the following conditions hold:

- $\mathcal{C}$ is moving,
- for all $(q, d) \in \mathcal{C}$, if $\delta_{up}(q) = \bigvee_{i \in I} \bigwedge_{j \in J} p_{i,j}$ then there is $i \in I$ such that for all $j \in J$, $(p_{i,j}, d) \in \mathcal{C}'$,
- nothing else is in $\mathcal{C}'$.

Remark 7.2. In the definition of the run the automaton behaves synchronously: all threads move up at the same time. This is only for convenience of presentation. Since all the threads are independent, one can also define a run in which each thread moves independently. This alternative definition would be equivalent to the current one. Observe that this is not the case for the definition of the run of the automaton model $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$ of Chapter 6. There, the synchronous beheviour is crucial to define the spread operator.

Runs We are now ready to define a **run** ρ of $\mathcal{A}$ on $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$. It is a function associating a configuration to any node x of $\mathbf{t}$ such that

1. for any leaf x of $\mathbf{t}$, $\rho(x) = \{(q_0, \mathbf{d}(x))\}$,
2. for any inner position x of $\mathbf{t}$ whose children are $x{\cdot}1, \ldots, x{\cdot}n$, then there are configurations $\mathcal{C}'_1, \cdots, \mathcal{C}'_n$ and $\mathcal{C}''_1, \cdots, \mathcal{C}''_n$ such that for all $i \in [n]$, $(x{\cdot}i, \mathcal{C}''_i)$ is in the ϵ-closure of $(x{\cdot}i, \rho(x{\cdot}i))$, $\mathcal{C}''_i \rightarrowtail_{up} \mathcal{C}'_i$, and $\rho(x) = \bigcup_{i \in [n]} \mathcal{C}'_i$.

The run ρ is **accepting** if moreover at the root (i.e., for the position ϵ), the ϵ-closure of $\rho(\epsilon)$ contains an accepting configuration.

We have the following properties.

Proposition 7.3. *The class of languages $\mathcal{L}(\mathsf{BUDA})$ definable by the BUDA class is:*

(i) closed under union,

(ii) closed under intersection, and

(iii) not closed under complementation.

Proof. Closure under union and intersection is due to the fact that ϵ-transitions have disjunction and conjunction.

The fact that it is not closed under complementation, is immediate from the fact that BUDA can simulate a ARA(guess) when going up along a branch, and that if we can test the negation of a property expressible by an ARA(guess) automaton, the emptiness problem becomes undecidable (cf. Proposition 3.2). This would imply that the emptiness problem for BUDA is undecidable, which contradicts our main result (stated in Theorem 7.5), that its emptiness problem is decidable. □

Automata normal form

We now present a normal form of BUDA, removing all the redundancy in its definition. This normal form simplifies the technical details in the proof of decidability presented in the next section.

(NF1) The semigroup $\mathcal{S}$ and morphism h have the following property.

$$\text{For all } w \in (\mathbb{A} \times \mathbb{B})^+ \text{ and } c \in \mathbb{A} \times \mathbb{B}, \quad h(w) = h(c) \text{ iff } w = c.$$

(NF2) In the definition of δ_{up} of $\mathcal{A}$, there is exactly one disjunct that contains exactly one conjunct. That is, for all $q \in Q$, $\delta_{\mathrm{up}}(q)$ is undefined or $\delta_{\mathrm{up}}(q) = p$ for some $p \in Q$.

(NF3) For all $q \in Q$, $\delta_\epsilon(q)$ is defined either as an atom, as $p \wedge p'$ or as $p \vee p'$ for some $p, p' \in Q$.

(NF4) For all $q \in Q$, $\delta_\epsilon(q)$ does not contain tests for labels (a, $\bar{a}$, b, $\bar{b}$), eq, $\overline{\mathsf{eq}}$, store, leaf or $\overline{\mathsf{leaf}}$.

An automaton $\mathcal{A} \in \mathsf{BUDA}$ is said to be in **normal form** if it satisfies (NF1), (NF2), (NF3) and (NF4). Notice that once (NF1) holds, then any test concerning a label (a, $\bar{a}$, b, or $\bar{b}$) can be simulated using tests of the form $\langle\mu\rangle$ for some appropriate μ. Using similar ideas, it is not hard to check that:

Proposition 7.4. *For any $\mathcal{A} \in$ BUDA, there is an equivalent $\mathcal{A}' \in$ BUDA in normal form that can be effectively obtained.*

Proof. First, given a finite semigroup we can easily compute another one that satisfies ((NF1)), only by adding some extra elements to the domain in order to tell apart all the one letter words for each symbol of the finite alphabet.

Second, notice that ((NF2)) is without any loss of generality, since any positive boolean combination of δ_{up} can be simulated using ϵ-transitions right after the *up*-transition is done.

On the other hand, we can simulate the fact that $\delta_\epsilon(q)$ is undefined by defining $\delta_\epsilon(q) = q$. Also, $\delta_\epsilon(q) = \bigvee_{i\in I}\bigwedge_{j\in J}\varphi_{i,j}$ —where the $\varphi_{i,j}$'s are atoms— can be decomposed into binary disjunctions and conjunctions, adding a suitable set of extra states. We can then assume without any loss of generality that the automaton verifies ((NF3)).

In the sequel we use $\langle\mu\rangle$ as a shortcut for $\langle\mu\rangle^{=} \vee \langle\mu\rangle^{\neq}$ and $\overline{\langle\mu\rangle}$ for its negation.

To show that ((NF4)) can always be assumed, note that any test for label can be simulated using $\langle\mu\rangle$. Indeed, a test a with $a \in \mathbb{A}$ can be simulated with $\bigvee_{b\in\mathbb{B}}\langle h(a,b)\rangle$, $\bar{a}$ with $\bigwedge_{b\in\mathbb{B}}\overline{\langle h(a,b)\rangle}$, and similar tests can simulate b and $\bar{b}$ for $b \in \mathbb{B}$. What is more, the ϵ-transition $\langle\mu\rangle$ of a BUDA can be simulated using $\langle\mu\rangle^{=} \vee \langle\mu\rangle^{\neq}$. The ϵ-transition eq can be simulated using $\bigvee_{a\in\mathbb{C}}\langle h(a)\rangle^{=}$. Similarly $\overline{\mathsf{eq}}$ can be simulated using $\langle\mu\rangle^{\neq}$. Also, store can be simulated using $\mathsf{guess} \wedge \mathsf{eq}$. Lastly, leaf and $\overline{\mathsf{leaf}}$ can be tested with $\langle\mu\rangle$ and $\overline{\langle\mu\rangle}$ using some $\bigvee_{\mu\notin\{h(a)|a\in\mathbb{C}\}}\langle\mu\rangle^{=}\vee\langle\mu\rangle^{\neq}$. Thus, we can suppose that the automaton $\mathcal{A}$ does not contain any transition that uses tests for labels, eq, $\overline{\mathsf{eq}}$, $\langle\mu\rangle$, store, leaf and $\overline{\mathsf{leaf}}$ without any loss of generality. □

7.3 The emptiness problem

The goal of this section is to show the following.

Theorem 7.5. *The emptiness problem for BUDA is decidable.*

In order to achieve this, we associate to each BUDA a WSTS that simulates its runs. The transition system works on sets of *abstract configurations*. Given an automaton, an abstract configuration is meant to contain all the information that is necessary to collect at the root of a given subtree in order to continue the simulation of the automaton from there. The aforesaid transition system works with *sets* of such abstract configurations in order to capture the bottom-up behavior of

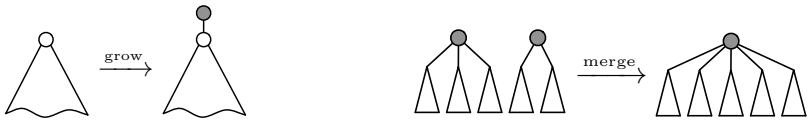

Fig. 7.1: The *grow* and *merge* operations.

the automaton on unranked trees. The transition relation of the WSTS essentially corresponds to the transitions of the automaton except for the *up*-transition. An *up*-transition of the automaton is simulated by a succession of two types of transitions of the WSTS, called *grow* and *merge*. The object of doing this is to avoid having transitions that take an unbounded number of arguments (as the *up* relation in the run of the automaton does). The *grow* transition adds a node on top of the current root, and the *merge* transition identifies the roots of two abstract configurations. Intuitively, these transitions correspond to the operations on trees of Figure 7.1. This is necessary because we do not know in advance the arity of the tree and therefore the transition system has to build one subtree at a time.

We then exhibit a wqo on abstract configurations and show that the transition system is N-downward compatible with respect to this wqo for some N that depends on the automaton. Decidability will then follow from Lemma 2.13.

In the sequel we implicitly assume that all our BUDA are in normal form.

7.3.1 Abstract configurations

Given a BUDA $\mathcal{A} = (\mathbb{A}, \mathbb{B}, Q, q_0, \delta_\epsilon, \delta_{up}, \mathcal{S}, h)$, we start the definition of its associated WSTS by defining its universe: finite sets of abstract configurations of $\mathcal{A}$.

An **abstract configuration** of $\mathcal{A}$ is a tuple (Δ, Γ, r, m) where r and m are either true or false, Δ is a finite subset of $Q \times \mathbb{D}$ and Γ is a finite subset of $\mathcal{S} \times \mathbb{D}$ such that

$$\Gamma \text{ contains exactly one pair of the form } (h(c), d) \text{ with } c \in \mathbb{A} \times \mathbb{B}. \qquad (\star)$$

This unique element of $\mathbb{A} \times \mathbb{B}$ is denoted as the **label** of the abstract configuration and the unique associated data value is denoted as the **data value** of the abstract configuration.

Intuitively r says whether the current node should be treated as the root or not, m says whether we are in a phase of *merging configurations* or not (a notion that will become clear when we introduce our transition system later on), Δ is the set of ongoing threads (corresponding to the configuration of the automaton) and a pair $(\mu, d) \in \Gamma$ simulates the existence of a downward path evaluating to μ and whose last element carry the datum d.

In the sequel we will use the following notation: $\Delta(d) = \{q \mid (q, d) \in \Delta\}$, $\Gamma(d) = \{\mu \mid (\mu, d) \in \Gamma\}$, $\Delta(q) = \{d \mid (q, d) \in \Delta\}$, $\Gamma(\mu) = \{d \mid (\mu, d) \in \Gamma\}$. We also use the notation $\Delta \otimes \Gamma : \mathbb{D} \to \wp(Q) \times \wp(\mathcal{S})$ with $(\Delta \otimes \Gamma)(d) = (\Delta(d), \Gamma(d))$. Given a data value d and an abstract configuration θ, $(\Delta \otimes \Gamma)(d)$ is also denoted as the

type of d **in** θ. We use the letter θ to denote an abstract configuration and we write AC to denote the set of all abstract configurations. Similarly, we use Θ to denote a finite set of abstract configurations and $\wp_{<\infty}(\mathrm{AC})$ for the set of finite sets of abstract configurations.

An abstract configuration $\theta = (\Delta, \Gamma, r, m)$ is said to be **initial** if it corresponds to a leaf node, i.e., is such that $\Delta = \{(q_0, d)\}$ and $\Gamma = \{(h(a), d)\}$ for some $d \in \mathbb{D}$ and $a \in \mathbb{A} \times \mathbb{B}$. It is said to be **accepting** if Δ is empty and r is *true*.

Two configurations θ_1 and θ_2 are said to be **equivalent** if there is a bijection $f : \mathbb{D} \to \mathbb{D}$ such that $f(\theta_1) = \theta_2$ (with some abuse of notation). In this case we note $\theta_1 \sim \theta_2$.

Finally, we write Θ_I to denote the set of all initial abstract configurations modulo $\sim$ (i.e., a set containing at most one element for each $\sim$ equivalence class). Note that Θ_I is *finite* and effective. A set of abstract configurations is said to be **accepting** iff it contains an accepting abstract configuration.

7.3.2 Well-quasi-orders

We now equip $\wp_{<\infty}(\mathrm{AC})$ with a well-quasi-order $(\wp_{<\infty}(\mathrm{AC}), \leq_{min})$. The order $\leq_{min}$ builds upon a wqo $(\mathrm{AC}, \precsim)$ over abstract configurations. Let us define these orderings precisely.

The **profile of an abstract configuration** $\theta = (\Delta, \Gamma, r, m)$, denoted by $\mathit{profile}(\theta)$, is $\mathit{profile}(\theta) = (\ A_0,\ A_1,\ r,\ m\ )$ with $A_i = \{(S, \chi) \in \wp(Q) \times \wp(\mathcal{S}) : |(\Delta \otimes \Gamma)^{-1}(S, \chi)| = i\}$.

We first define the quasi-order $\preceq$ over abstract configurations, and then we define the order $(\mathrm{AC}, \precsim)$ as $(\mathrm{AC}, \preceq)$ modulo $\sim$.

Definition 7.6 ($\preceq$)**.** Given two abstract configurations $\theta_1 = (\Delta_1, \Gamma_1, r_1, m_1)$ and $\theta_2 = (\Delta_2, \Gamma_2, r_2, m_2)$, we denote by $\theta_1 \preceq \theta_2$ the fact that

- $\mathit{profile}(\theta_1) = \mathit{profile}(\theta_2)$, and
- $(\Delta_1 \otimes \Gamma_1) \subseteq (\Delta_2 \otimes \Gamma_2)$

Remark 7.7. Notice that due to condition ($\star$), $\theta_1 \preceq \theta_2$ implies that θ_1 and θ_2 have the same label and same data value.

We now define $\precsim$ as follows.

Definition 7.8 ($\precsim$)**.** $\theta_1 \precsim \theta_2$ iff $\theta_1' \preceq \theta_2$ for some $\theta_1' \sim \theta_1$.

We are now ready to define our wqo over $\wp_{<\infty}(\mathrm{AC})$.

Definition 7.9 ($\leq_{min}$)**.** Given Θ_1 and Θ_2 in $\wp_{<\infty}(\mathrm{AC})$ we define $\Theta_1 \leq_{min} \Theta_2$ iff for all $\theta_2 \in \Theta_2$ there is $\theta_1 \in \Theta_1$ such that $\theta_1 \precsim \theta_2$. That is, every element from Θ_2 is *minorized* by an element of Θ_1.

We are now ready to show that the order we use in our proof is a wqo. Recall the definitions of $\precsim$ and $\leq_{min}$ in Section 7.3.2.

Lemma 7.10. $(\mathrm{AC}, \precsim)$ *is a* ω^2*-wqo.*

Proof. Given an abstract configuration $\theta = (\Delta, \Gamma, r, m)$, let us define $\bar{x}(\theta)$ as function from $\wp(Q) \times \wp(\mathcal{S})$ to $\mathbb{N}$. We define $\bar{x}(\theta)(S, \chi) = |(\Delta \otimes \Gamma)^{-1}(S, \chi)|$. Notice that $\bar{x}(\theta)$ can be seen as a vector of natural numbers. Let $\leq_{\bar{x}}$ be the componentwise order of vector of natural numbers. It then follows that for every θ_1, θ_2, $\mathit{profile}(\theta_1) = \mathit{profile}(\theta_2)$ and $\bar{x}(\theta_1) \leq_{\bar{x}} \bar{x}(\theta_2)$ iff $\theta_1 \precsim \theta_2$. By Corollary 2.24 we know that $(\mathrm{AC}, \leq_{\bar{x}})$ is a ω^2-wqo.

Let us define $\approx_{\mathit{prof}}$ as the relation between abstract configurations that corresponds to equality of profiles. Since there are only finitely many different profiles, it follows that there are finitely many equivalence classes of $\approx_{\mathit{prof}}$. Then, it cannot contain an isomorphic copy of the Rado structure (Definition 2.17), and $(\mathrm{AC}, \approx_{\mathit{prof}})$ is then a ω^2-wqo.

By Proposition 2.21, ω^2-wqo are closed under intersection, and we then obtain that $(\mathrm{AC}, (\leq_{\bar{x}} \cap \approx_{\mathit{prof}}))$ is a ω^2-wqo. In other words, $(\mathrm{AC}, \precsim)$ is a ω^2-wqo. □

And finally:

Lemma 7.11. $(\wp_{<\infty}(\mathrm{AC}), \leq_{\mathit{min}})$ *is a wqo.*

Proof. This is a direct consequence of Proposition 2.20 and Lemma 7.10. □

Finally, the following obvious lemma will be necessary to apply Lemma 2.13.

Lemma 7.12. $\{\Theta \in \wp_{<\infty}(\mathrm{AC}) \mid \Theta \textit{ is accepting}\}$ *is downward closed for* $(\wp_{<\infty}(\mathrm{AC}), \leq_{\mathit{min}})$.

7.3.3 Transition system

We now equip $\wp_{<\infty}(\mathrm{AC})$ with a transition relation $\Rightarrow$. This transition relation is built upon a transition relation $\rightarrow$ over AC.

Let us first define $\rightarrow$ over AC. It is specified so that it reflects the transitions of $\mathcal{A}$. For each possible test of the automaton there is a transition that simulates this thest by using the information contained in Γ, and removes the corresponding in Δ. And for every operation store, guess, univ there is a transition that modifies Δ. We call the ϵ-transitions of $\rightarrow$, that we note $\rightarrow_\epsilon$. On the other hand, any *up* transition of $\mathcal{A}$ is decomposed into transitions $\xrightarrow{\text{merge}}$ and $\xrightarrow{\text{grow}}$ that modify not only Δ but also Γ accordingly.

We start with ϵ-transitions. Given two abstract configurations $\theta_1 = (\Delta_1, \Gamma_1, r_1, m_1)$ and $\theta_2 = (\Delta_2, \Gamma_2, r_2, m_2)$, we say that $\theta_1 \rightarrow_\epsilon \theta_2$ if $m_1 = m_2 = \mathit{false}$ (the merge information is used for simulating an *up*-transition as will be explained later), $r_2 = r_1$ (whether the current node is the root or not should not change), θ_1 and θ_2 have the same label and data value, $\Gamma_2 = \Gamma_1$ (the tree is not affected by an ϵ-transition) and, furthermore, one of the following conditions holds:

1. $\theta_1 \xrightarrow{\text{univ}} \theta_2$. This transition can happen if there is $(q, d) \in \Delta_1$ with $\delta_\epsilon(q) = \mathsf{univ}(p)$ for some $p, q \in Q$. In this case θ_2 is such that $\Delta_2 = (\Delta_1 \setminus \{(q, d)\}) \cup \{(p, e) : \exists \mu \,.\, (\mu, e) \in \Gamma_1\}$.

2. $\theta_1 \xrightarrow{\mathsf{guess}} \theta_2$. This transition can happen if there is $(q,d) \in \Delta_1$ with $\delta_\epsilon(q) = \mathsf{guess}(p)$ for some $p, q \in Q$. In this case θ_2 is such that $\Delta_2 = (\Delta_1 \setminus \{(q,d)\}) \cup \{(p,d')\}$ for some $d' \in \mathbb{D}$.

3. $\theta_1 \xrightarrow{\langle\mu\rangle^=} \theta_2$ (resp. $\theta_1 \xrightarrow{\langle\mu\rangle^\neq} \theta_2$). This transition can happen if there is $(q,d) \in \Delta_1$ with $\delta_\epsilon(q) = \langle\mu\rangle^=$ (resp. $\delta_\epsilon(q) = \langle\mu\rangle^\neq$) for some $q \in Q$, $\mu \in \mathcal{S}$, and $\mu \in \Gamma_1(d)$ (resp. there exists $e \in \mathbb{D}$, $e \neq d$ such that $\mu \in \Gamma_1(e)$). In this case θ_2 is such that $\Delta_2 = (\Delta_1 \setminus \{(q,d)\})$. The negation of these tests $\xrightarrow{\overline{\langle\mu\rangle^=}}$ and $\xrightarrow{\overline{\langle\mu\rangle^\neq}}$ are defined in a similar way.

4. $\theta_1 \xrightarrow{\mathsf{root}} \theta_2$ (resp. $\theta_1 \xrightarrow{\overline{\mathsf{root}}} \theta_2$). This transition can happen if there is $(q,d) \in \Delta$ with $\delta_\epsilon(q) = \mathsf{root}$ and $r_1 = \mathit{true}$ (resp. $\delta_\epsilon(q) = \overline{\mathsf{root}}$ and $r_1 = \mathit{false}$). In this case θ_2 is such that $\Delta_2 = (\Delta_1 \setminus \{(q,d)\})$.

5. $\theta_1 \xrightarrow{\wedge} \theta_2$. This transition can happen if there is $(q,d) \in \Delta_1$ with $\delta_\epsilon(q) = p \wedge p'$ for some $p, p', q \in Q$. In this case θ_2 is such that $\Delta_2 = (\Delta_1 \setminus \{(q,d)\}) \cup \{(p,d),(p',d)\}$.

6. $\theta_1 \xrightarrow{\vee} \theta_2$. This transition can happen if there is $(q,d) \in \Delta_1$ with $\delta_\epsilon(q) = p \vee p'$ for some $p, p', q \in Q$. In this case θ_2 is such that $\Delta_2 = (\Delta_1 \setminus \{(q,d)\}) \cup A$, for $A = \{(p,d)\}$ or $A = \{(p',d)\}$.

Note that by ((NF3)) and ((NF4)) for every possible definition of $\delta_\epsilon(q)$ there is one transition that simulates it. To simulate δ_{up}, it turns out that we will need one extra ϵ-transition that makes our trees fatter. This transition assumes the same constraints as for $\rightarrow_\epsilon$ except that we no longer have $\Gamma_2 = \Gamma_1$. The idea is that this transition corresponds to duplicating all the immediate subtrees of the root. For example, if the root has $\mathbf{t}_1$ and $\mathbf{t}_2$ as subtrees, consider the operation of now having $\mathbf{t}_1$ $\mathbf{t}_2$ $\mathbf{t}'_1$ $\mathbf{t}'_2$ as subtrees, where $\mathbf{t}'_1$ and $\mathbf{t}'_2$ are identical to $\mathbf{t}_1$ and $\mathbf{t}_2$ except for one data value with profile (S,χ) that in $\mathbf{t}'_1$ and $\mathbf{t}'_2$ is replaced by a fresh data value. Here is the definition that follows this idea in terms of our transition system. We say that $\theta_1 \xrightarrow{\mathrm{inc}(S,\chi)} \theta_2$ for some pair $(S,\chi) \in \wp(Q) \times \wp(\mathcal{S})$ if $|(\Delta_1 \otimes \Gamma_1)^{-1}(S,\chi)| \geq 1$ and, either $\chi = \emptyset$ or $|(\Gamma_1)^{-1}(\chi)| \geq 2$. Then θ_2 is such that $\mathit{data}(\theta_2) = \mathit{data}(\theta_1) \cup \{e\}$ for some $e \notin \mathit{data}(\theta_1)$, $(\Delta_2 \otimes \Gamma_2)(e) = (S,\chi)$, and for all $d \neq e$, $(\Delta_2 \otimes \Gamma_2)(d) = (\Delta_1 \otimes \Gamma_1)(d)$. Observe that $\xrightarrow{\mathrm{inc}(S,\chi)}$ does not change the truth value of any test. Indeed, any test ($\langle\mu\rangle^=$, $\overline{\langle\mu\rangle^=}$, eq, etc.) that is true in θ, continues to be true after a $\xrightarrow{\mathrm{inc}(S,\chi)}$ transition, and vice-versa.

We define the transitions of the WSTS that correspond to *up*-transitions in the automaton. We split them into two phases: adding a new root symbol and merging the roots.

1. $\theta_1 \xrightarrow{\mathrm{grow}} \theta_2$. Given two abstract configurations θ_1 and θ_2 as above, we say $\theta_1 \xrightarrow{\mathrm{grow}} \theta_2$ if $r_1 = m_1 = \mathit{false}$, and for all $(q,d) \in \Delta_1$, $\delta_{up}(q)$ is defined and θ_2 is such that $\Delta_1 \rightarrowtail_{up} \Delta_2$, and $\Gamma_2 = \{(\mu', e) : (\mu, e) \in \Gamma_1, \mu' = h(c) \cdot \mu\} \cup \{(h(c), d)\}$, for some $c \in \mathbb{A} \times \mathbb{B}$ and $d \in \mathbb{D}$. Notice that c and d are then the label and data

value of θ_2. As a consequence of the normal form (NF1) of the semigroup, this operation preserves property $(\star)$.

2. $\theta_1, \theta_2 \xrightarrow{\text{merge}} \theta_0$. Given 3 abstract configurations $\theta_1 = (\Delta_1, \Gamma_1, r_1, m_1)$, $\theta_2 = (\Delta_2, \Gamma_2, r_2, m_2)$, $\theta_0 = (\Delta_0, \Gamma_0, r_0, m_0)$ we define $\theta_1, \theta_2 \xrightarrow{\text{merge}} \theta_0$ if they all have the same label and data value, $m_1 = m_2 = \mathit{true}$, $r_1 = r_2 = r_0$, $\Delta_0 = \Delta_1 \cup \Delta_2$, and $\Gamma_0 = \Gamma_1 \cup \Gamma_2$. Notice that this operation preserves property $(\star)$.

Remark 7.13. $\xrightarrow{\text{inc}(S,\chi)}$ can be seen as a kind of $\xrightarrow{\text{merge}}$ which preserves the truth of tests.

Definition 7.14. We define that $\Theta_1 \Rightarrow \Theta_0$ if one the following conditions holds:

1. There is $\theta_1 \in \Theta_1$ and $\theta_1' \sim \theta_1$ such that $\theta_1' \rightarrow_\epsilon \theta_0$ or $\theta_1' \xrightarrow{\text{inc}(S,\chi)} \theta_0$ or $\theta_1' \xrightarrow{\text{grow}} \theta_0$, for some θ_0, χ, and $\Theta_0 = \Theta_1 \cup \{\theta_0\}$.

2. There are $\theta_1, \theta_2 \in \Theta_1$ and $\theta_1' \sim \theta_1$, $\theta_2' \sim \theta_2$ such that $\theta_1', \theta_2' \xrightarrow{\text{merge}} \theta_0$ for some θ_0, and $\Theta_0 = \Theta_1 \cup \{\theta_0\}$.

In the definition of the transition system, the m flag is simply used to constrain the transition system to have all its $\xrightarrow{\text{merge}}$ operations right after $\xrightarrow{\text{grow}}$ and before any $\rightarrow_\epsilon$. Thus, if we take a derivation and examine the kind of $\rightarrow$ transitions that originated each $\Rightarrow$ transition, we obtain a word described by the following regular expression

$$\big((\rightarrow_\epsilon \mid \xrightarrow{\text{inc}(S,\chi)})^* \xrightarrow{\text{grow}} (\xrightarrow{\text{merge}})^*\big)^* (\rightarrow_\epsilon \mid \xrightarrow{\text{inc}(S,\chi)})^* \ . \qquad (\dagger)$$

7.3.4 Compatibility

We now show that all the previous definitions were chosen appropriately and that the transition system defined in Section 7.3.3 is compatible with the wqo defined in Section 7.3.2. The proof of this result is very technical and consists in a case analysis over each possible kind of transition. In this proof, the operation $\xrightarrow{\text{inc}(S,\chi)}$ becomes crucial to show that the downwards compatibility can always be done in a bounded amount of N steps.

We start with some technical comments.

Definition 7.15. We write $A \approx_\ell B$ iff $A = B$ or $|A \cap B| \geq \ell$.

Definition 7.16. We define the relation $\preceq_l$ as follows. $\theta_1' \preceq_l \theta_1$ iff

1. $\theta_1' \preceq \theta_1$, and

2. for every (S, χ) we have that $(\Delta_1 \otimes \Gamma_1)^{-1}(S, \chi) \approx_l (\Delta_1' \otimes \Gamma_1')^{-1}(S, \chi)$, for $l := \max(3, 2.|\mathcal{S}|.|Q|)$.

As before, we define $\precsim_l$ as $\preceq_l$ modulo $\sim$.

The following lemma illustrates the use of transitions $\xrightarrow{\mathrm{inc}(S,\chi)}$ by showing that, in a sense, $\precsim_{\boldsymbol{l}}$ can be assumed from $\precsim$ without loss of generality.

Lemma 7.17. *Let $\Theta_1, \Theta_1' \in \wp_{<\infty}(\mathrm{AC})$ such that $\Theta_1 \leq_{min} \Theta_1'$. Given $\theta_1 \in \Theta_1$ and $\theta_1' \in \Theta_1'$ such that $\theta_1 \precsim \theta_1'$, there exists $\tilde{\Theta}$ and $\tilde{\theta} \in \tilde{\Theta}$ such that*

1. *$\Theta_1 \Rightarrow^n \tilde{\Theta}$ for $n \leq (|\mathcal{S}|.|Q|).\boldsymbol{l}$,*
2. *$\theta_1 \precsim \tilde{\theta} \precsim_{\boldsymbol{l}} \theta_1'$,*
3. *$\Theta_1 \leq_{min} \tilde{\Theta} \leq_{min} \Theta_1'$.*

Proof. Let $\theta_1 = (\Delta_1, \Gamma_1, r_1, m_1)$ and $\theta_1' = (\Delta_1', \Gamma_1', r_1', m_1')$. Given $S \subseteq Q$, and $\chi \subseteq \mathcal{S}$, let

$$n = \min(\ \boldsymbol{l},\ |(\Delta_1' \otimes \Gamma_1')^{-1}(S,\chi)| - |(\Delta_1 \otimes \Gamma_1)^{-1}(S,\chi)|\ )\ .$$

Since $\theta_1 \precsim \theta_1'$, it follows that $n \geq 0$. Notice that if $|(\Delta_1 \otimes \Gamma_1)^{-1}(S,\chi)| = 1$ then by definition of $\precsim$ we also have $|(\Delta_1' \otimes \Gamma_1')^{-1}(S,\chi)| = 1$ and $n = 0$. On the other hand, if $|(\Delta_1 \otimes \Gamma_1)^{-1}(S,\chi)| > 1$ we can apply a transition $\xrightarrow{\mathrm{inc}(S,\chi)}$ to θ_1 resulting in an abstract configuration θ_2. By definition of $\xrightarrow{\mathrm{inc}(S,\chi)}$, we have $n - 1 = |(\Delta_1' \otimes \Gamma_1')^{-1}(S,\chi)| - |(\Delta_2 \otimes \Gamma_2)^{-1}(S,\chi)|$ and $\theta_1 \precsim \theta_2 \precsim \theta_1'$. Then, if we apply n transitions $\xrightarrow{\mathrm{inc}(S,\chi)}$ successively to θ_1 we end up with an abstract configuration θ_{n+1} such that $\theta_1 \,(\xrightarrow{\mathrm{inc}(S,\chi)})^n\, \theta_{n+1}$, and such that $(\Delta_1' \otimes \Gamma_1')^{-1}(S,\chi) \approxeq_{\boldsymbol{l}} (\Delta_{n+1} \otimes \Gamma_{n+1})^{-1}(S,\chi)$, $\theta_1 \preceq \theta_{n+1} \precsim \theta_1'$. Then, for $\Theta_{i+1} = \Theta_i \cup \{\theta_{i+1}\}$, we have that $\Theta_i \Rightarrow \Theta_{i+1}$ and $\Theta_i \leq_{min} \Theta_{i+1}$.

We can repeat this for every (S,χ), ending up with $\tilde{\Theta}$ containing some $\tilde{\theta}$ with the desired properties. □

We are now ready to prove the main technical lemma of this section proving that the transitions on AC are compatible with the order on AC.

Lemma 7.18. *Let $\theta_1, \theta_2, \theta_0$ be abstract configurations such that $\theta_1 \to_\epsilon \theta_0$ (or $\theta_1, \theta_2 \xrightarrow{merge} \theta_0$).*

Given $\theta_1' \precsim_{\boldsymbol{l}} \theta_1$ (and $\theta_2' \precsim_{\boldsymbol{l}} \theta_2$), then there is θ_0' such that $\theta_0' \precsim \theta_0$ and $\theta_1' \to_\epsilon \theta_0'$ (or $\theta_1', \theta_2' \xrightarrow{merge} \theta_0'$).

Proof of Lemma 7.18. This is done by a case analysis depending on where $\to$ comes from. Throughout the proof we use the following notation: $\theta_i = (\Delta_i, \Gamma_i, r_i, m_i)$, similarly for θ_i'. Moreover c_i and d_i denote the label and data value associated to θ_i (similarly for θ_i'). As we work modulo equivalence we can further assume that $\theta_1' \preceq_{\boldsymbol{l}} \theta_1$ (and $\theta_2' \preceq_{\boldsymbol{l}} \theta_2$). In particular we have $c_1 = c_1'$ and $d_1 = d_1'$.

1. $\xrightarrow{\mathrm{grow}}$. Suppose $\theta_1' \preceq_{\boldsymbol{l}} \theta_1 \xrightarrow{\mathrm{grow}} \theta_0$.

   We first show that a *grow transition* can be applied to θ_1'. Consider $e \in \mathbb{D}$ and $q \in Q$ such that $(\Delta_1' \otimes \Gamma_1')(e) = (S,\chi)$ and $q \in S$. Then from $\theta_1' \preceq_{\boldsymbol{l}} \theta_1$ we have

by definition that $(\Delta_1 \otimes \Gamma_1)(e) = (S, \chi)$. Therefore, as we already know that a $\xrightarrow{\text{grow}}$ transition can be applied from θ_1, we have $\delta_{\text{up}}(q) = p$ for some $p \in Q$ as desired.

Let θ'_0 be a configuration such that $\theta'_1 \xrightarrow{\text{grow}} \theta'_0$ with $r'_0 = r_0$. Note that by definition of $\xrightarrow{\text{grow}}$ we have the freedom to choose the label and the data value in θ'_0. We show that θ'_0 can be chosen such that $\theta'_0 \preceq \theta_0$ concluding this case.

For the label of θ'_0 we choose $c'_0 = c_0$, and for the data value of θ'_0 we choose $d'_0 = d_0$. In the sequel, we make use of the following claim.

Claim 7.18.1. There exists $\theta''_1 \sim \theta'_1$ such that $\theta''_1 \preceq_\iota \theta_1$ and $d_0 \in \mathit{data}(\theta_1)$ iff $d_0 \in \mathit{data}(\theta''_1)$.

Proof. If $d_0 \in \mathit{data}(\theta'_1)$, then by $\theta'_1 \preceq \theta_1$ we have that $d_0 \in \mathit{data}(\theta_1)$ and we take $\theta''_1 = \theta'_1$. Otherwise, suppose $d_0 \notin \mathit{data}(\theta'_1)$ and $d_0 \in \mathit{data}(\theta_1)$. Let $(S_1, \chi_1) = (\Delta_1 \otimes \Gamma_1)(d_0)$. Because $\theta'_1 \preceq_\iota \theta_1$, θ_1 and θ'_1 have the same profile, and there exists a data value e such that $(\Delta'_1 \otimes \Gamma'_1)(e) = (S_1, \chi_1)$. In this case, let f be a bijection on $\mathbb{D}$ that sends e to d_0 and is the identity on all other data values of $\mathit{data}(\theta'_1)$. Hence, since f preserves the profiles and the types of data values, we have that if $\theta'_1 \preceq_\iota \theta_1$ then $f(\theta'_1) \preceq_\iota \theta_1$ and $d_0 \in \mathit{data}(f(\theta'_1))$, and moreover $f(\theta'_1) \sim \theta'_1$ since f is a bijection. Thus, we can take $\theta''_1 = f(\theta'_1)$. □

By the previous claim we will assume, without any loss of generality, that $d_0 \in \mathit{data}(\theta_1)$ iff $d_0 \in \mathit{data}(\theta'_1)$ (otherwise we can pick θ''_1 as described in the claim). Now that θ'_0 is completely defined, we show that $\theta'_0 \preceq \theta_0$, that is, that (a) $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$, and (b) $\mathit{profile}(\theta'_0) = \mathit{profile}(\theta_0)$.

(a) $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$. Assume $e \in \mathit{data}(\theta'_0)$, and let us show that

$$(\Delta'_0 \otimes \Gamma'_0)(e) = (\Delta_0 \otimes \Gamma_0)(e).$$

If $e \in \mathit{data}(\theta'_1)$, then by $\theta'_1 \preceq \theta_1$, $e \in \mathit{data}(\theta_1)$. Let $(S, \chi) = (\Delta_1 \otimes \Gamma_1)(e) = (\Delta'_1 \otimes \Gamma'_1)(e)$. Notice that $(\Delta'_0 \otimes \Gamma'_0)(e)$ is completely determined from (S, χ), a'_0 and the fact that e is equal to d'_0 or not, according to the rules of $\xrightarrow{\text{grow}}$. Similarly, $(\Delta'_0 \otimes \Gamma'_0)(e)$ is determined by (S, χ), c_0 and the fact that e is equal to d'_0 or not, according to the rules of $\xrightarrow{\text{grow}}$. Thus, since $c_0 = c'_0$ and $d_0 = d'_0$ we have that $(\Delta_0 \otimes \Gamma_0)(e) = (\Delta'_0 \otimes \Gamma'_0)(e)$.

If $e \notin \mathit{data}(\theta'_1)$, we have two cases. If $e \neq d'_0$, this leads to a contradiction, since there is a data value in θ'_0 which is not in θ'_1 and it is not the root, which is in opposition with the rules of the $\xrightarrow{\text{grow}}$ transition. Otherwise, we have $e = d'_0 = d_0$, and by Claim 7.18.1, $e \notin \mathit{data}(\theta_1)$. But in this case we have that $(\Delta_0 \otimes \Gamma_0)(e) = (\Delta'_0 \otimes \Gamma'_0)(e) = (\emptyset, \{h(c_0)\})$ according to the rules of $\xrightarrow{\text{grow}}$, since $c_0 = c'_0$ and e is neither in θ_1 nor in θ'_1.

(b) $\mathit{profile}(\theta'_0) = \mathit{profile}(\theta_0)$. We already have by construction $r'_0 = r_0$ and $m'_0 = m_0 = \mathit{true}$. From Item (a) for all (S, χ) we have that $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S, \chi)| > 0$

implies $|(\Delta_0 \otimes \Gamma_0)^{-1}(S,\chi)| > 0$ and that $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S,\chi)| > 1$ implies $|(\Delta_0 \otimes \Gamma_0)^{-1}(S,\chi)| > 1$. It remains to show the converse of these implications. We only show the second one as the first one is similar and simpler.

Assume we have two distinct data values $e_1 \neq e_2$ such that $(\Delta_0 \otimes \Gamma_0)(e_1) = (\Delta_0 \otimes \Gamma_0)(e_2) = (S_0, \chi_0)$. Let $(S_1,\chi_1) = (\Delta_1 \otimes \Gamma_1)(e_1)$ and $(S_2,\chi_2) = (\Delta_1 \otimes \Gamma_1)(e_2)$. From $\mathit{profile}(\theta'_1) = \mathit{profile}(\theta_1)$ we know that there exist data values e'_1 and e'_2 such that $(S_1,\chi_1) = (\Delta'_1 \otimes \Gamma'_1)(e'_1)$ and $(S_2,\chi_2) = (\Delta'_1 \otimes \Gamma'_1)(e'_2)$. Even in the case where $(S_1,\chi_1) = (S_2,\chi_2)$ we get from $\mathit{profile}(\theta'_1) = \mathit{profile}(\theta_1)$ that e'_1 and e'_2 can be chosen distinct. Moreover, by Claim 7.18.1 we can also assume that $e'_1 = d_0$ iff $e_1 = d_0$ and $e'_2 = d_0$ iff $e_2 = d_0$. Hence, since $c_0 = c'_0$, by definition of $\xrightarrow{\text{grow}}$ we have $(\Delta'_0 \otimes \Gamma'_0)(e'_1) = (\Delta'_0 \otimes \Gamma'_0)(e'_2) = (S_0, \chi_0)$.

2. $\xrightarrow{\text{merge}}$. Suppose $\theta_1, \theta_2 \xrightarrow{\text{merge}} \theta_0$, with $\theta'_1 \preceq_{\boldsymbol{l}} \theta_1$ and $\theta'_2 \preceq_{\boldsymbol{l}} \theta_2$.

In this case it will be important that we work with $\preceq_{\boldsymbol{l}}$ instead of just $\preceq$. We make use of the following claim.

Claim 7.18.2. There exist $\hat{\theta}'_1 \sim \theta'_1$ and $\hat{\theta}'_2 \sim \theta'_2$ such that for every S_1, S_2 and χ_1, χ_2,

$$\begin{array}{c} (\Delta_1 \otimes \Gamma_1)^{-1}(S_1,\chi_1) \cap (\Delta_2 \otimes \Gamma_2)^{-1}(S_2,\chi_2) \\ \cong_2 \\ (\hat{\Delta}'_1 \otimes \hat{\Gamma}'_1)^{-1}(S_1,\chi_1) \cap (\hat{\Delta}'_2 \otimes \hat{\Gamma}'_2)^{-1}(S_2,\chi_2) \end{array} \qquad (\cap)$$

Proof. By the definition of $\boldsymbol{l}$ and the fact that $\theta'_1 \precsim_{\boldsymbol{l}} \theta_1$ and $\theta'_2 \precsim_{\boldsymbol{l}} \theta_2$ there are enough data values available so that we can assume without any loss of generality that $(\cap)$ holds, by using a suitable bijection on $\mathbb{D}$. We need at most 2 data values for every possible (S_1,χ_1), (S_2,χ_2), which is exactly what is given by our definition of $\boldsymbol{l}$. □

Hence, using Claim 7.18.2 we can assume without any loss of generality that θ'_1, θ'_2 verify property $(\cap)$. From $\theta'_i \preceq \theta_i$ and $\theta_1, \theta_2 \xrightarrow{\text{merge}} \theta_0$ we have $c_0 = c_1 = c_2 = c'_1 = c'_2$, $d_0 = d_1 = d_2 = d'_1 = d'_2$, $r_0 = r_1 = r_2 = r'_1 = r'_2$ and $m_1 = m_2 = m'_2 = m'_1 = \mathit{true}$. This shows that a *merge transition* can be applied to θ'_1, θ'_2. Let θ'_0 be such that $\theta'_1, \theta'_2 \xrightarrow{\text{merge}} \theta'_0$, where we choose $m'_0 = m_0$. We show that $\theta'_0 \preceq \theta_0$ concluding this case. That is, that (a) $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$, and (b) $\mathit{profile}(\theta'_0) = \mathit{profile}(\theta_0)$.

(a) First, we show that $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$. Let $d \in \mathit{data}(\theta'_0)$. By definition of $\xrightarrow{\text{merge}}$, we have that $\Delta'_0(d) = \Delta'_1(d) \cup \Delta'_2(d)$ and $\Gamma'_0(d) = \Gamma'_1(d) \cup \Gamma'_2(d)$. Since $\theta'_1 \preceq \theta_1$ and $\theta'_2 \preceq \theta_2$, we have $\Delta_i(d) = \Delta'_i(d)$ and $\Gamma_i(d) = \Gamma'_i(d)$ for $i \in \{1,2\}$, and hence by definition of $\xrightarrow{\text{merge}}$, $\Delta_0(d) = \Delta'_1(d) \cup \Delta'_2(d) = \Delta'_0(d)$, and $\Gamma_0(d) = \Gamma'_1(d) \cup \Gamma'_2(d) = \Gamma'_0(d)$ as desired.

(b) Finally, we show $\mathit{profile}(\theta'_0) = \mathit{profile}(\theta_0)$. We have already seen that $m'_0 = m_0$ and $r'_0 = r_0$. As before, from Item (a) it remains to show that for all (S, χ) we have: $|(\Delta_0 \otimes \Gamma_0)^{-1}(S, \chi)| > 0$ implies $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S, \chi)| > 0$ and $|(\Delta_0 \otimes \Gamma_0)^{-1}(S, \chi)| > 1$ implies $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S, \chi)| > 1$. We only show the second one as the first one is similar and simpler.

Assume we have two distinct data values $e \neq e'$ such that $(\Delta_0 \otimes \Gamma_0)(e) = (\Delta_0 \otimes \Gamma_0)(e') = (S_0, \chi_0)$. Let

$$(S_1, \chi_1) = (\Delta_1 \otimes \Gamma_1)(e), \qquad (S'_1, \chi'_1) = (\Delta_1 \otimes \Gamma_1)(e'),$$
$$(S_2, \chi_2) = (\Delta_2 \otimes \Gamma_2)(e), \text{ and} \qquad (S'_2, \chi'_2) = (\Delta_2 \otimes \Gamma_2)(e').$$

By definition of $\xrightarrow{\text{merge}}$ we have $S_0 = S_1 \cup S_2 = S'_1 \cup S'_2$ and $\chi_0 = \chi_1 \cup \chi_2 = \chi'_1 \cup \chi'_2$.
We then necessarily have $|(\Delta_1 \otimes \Gamma_1)^{-1}(S_1, \chi_1) \cap (\Delta_2 \otimes \Gamma_2)^{-1}(S_2, \chi_2)| \geq 1$ and $|(\Delta_1 \otimes \Gamma_1)^{-1}(S'_1, \chi'_1) \cap (\Delta_2 \otimes \Gamma_2)^{-1}(S'_2, \chi'_2)| \geq 1$. Hence by Claim 7.18.2 there are two data values d, d' such that $(S_1, \chi_1) = (\Delta'_1 \otimes \Gamma'_1)(d)$, $(S_2, \chi_2) = (\Delta'_2 \otimes \Gamma'_2)(d)$, $(S'_1, \chi'_1) = (\Delta'_1 \otimes \Gamma'_1)(d')$ and $(S'_2, \chi'_2) = (\Delta'_2 \otimes \Gamma'_2)(d')$. Moreover, even in the case where $S_1 = S'_1$, $S_2 = S'_2$, $\chi_1 = \chi'_1$ and $\chi_2 = \chi'_2$, we can assume that $d \neq d'$. By definition of $\xrightarrow{\text{merge}}$ we immediately get that $(\Delta'_0 \otimes \Gamma'_0)(d) = (\Delta'_0 \otimes \Gamma'_0)(d') = (S_0, \chi_0)$.

3. $\xrightarrow{\text{inc}(S,\chi)}$. Suppose $\theta'_1 \preceq_\iota \theta_1 \xrightarrow{\text{inc}(S,\chi)} \theta_0$.

By definition of $\xrightarrow{\text{inc}(S,\chi)}$, this implies that $|(\Delta_1 \otimes \Gamma_1)^{-1}(S, \chi)| \geq 1$ and $|\Gamma_1^{-1}(\chi)| \geq 2$ (or $\chi = \emptyset$). We treat only the case where $\chi = \emptyset$, the other one being actually simpler.

As $\mathit{profile}(\theta'_1) = \mathit{profile}(\theta_1)$ we know that if $|(\Delta'_1 \otimes \Gamma'_1)^{-1}(S, \chi)|$ is 0 (or 1) then $|(\Delta_1 \otimes \Gamma_1)^{-1}(S, \chi)|$ is 0 (or 1). Hence $|(\Delta'_1 \otimes \Gamma'_1)^{-1}(S, \chi)| \geq 1$ and $|\Gamma_1^{-1}(\chi)| \geq 2$. Therefore a transition $\xrightarrow{\text{inc}(S,\chi)}$ can be applied to θ'_1 resulting in an abstract configuration θ'_0. Let e_0 be the only data value in $\mathit{data}(\theta_0) \setminus \mathit{data}(\theta_1)$ given by the definition of $\xrightarrow{\text{inc}(S,\chi)}$. Similarly let e'_0 be the only data value in $\mathit{data}(\theta'_0) \setminus \mathit{data}(\theta'_1)$. Using a claim similar to Claim 7.18.1 we can assume without loss of generality that $e'_0 = e_0$.

We show that $\theta'_0 \preceq \theta_0$ concluding this case. That is, that (a) $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$, and (b) $\mathit{profile}(\theta'_0) = \mathit{profile}(\theta_0)$.

(a) First, we show that $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$. Let $d \in \mathit{data}(\theta'_0)$ and let $(S_0, \chi_0) = (\Delta'_0 \otimes \Gamma'_0)(d)$. If $d \neq e'_0$, by definition of $\xrightarrow{\text{inc}(S,\chi)}$, we have that $(\Delta'_1 \otimes \Gamma'_1)(d) = (S_0, \chi_0)$. As $\theta'_1 \preceq_\iota \theta_1$, we have $(\Delta_1 \otimes \Gamma_1)(d) = (S_0, \chi_0)$. And as $e_0 = e'_0$, $d \neq e_0$ and by definition of $\xrightarrow{\text{inc}(S,\chi)}$, $(\Delta_0 \otimes \Gamma_0)(d) = (S_0, \chi_0)$ as desired. If $d = e'_0$, by definition of $\xrightarrow{\text{inc}(S,\chi)}$, $(\Delta'_0 \otimes \Gamma'_0)(e'_0) = (S, \chi) = (\Delta_0 \otimes \Gamma_0)(e_0)$ and we are done as $e'_0 = e_0$.

(b) Finally, we show $\mathit{profile}(\theta'_0) = \mathit{profile}(\theta_0)$. By definition of $\xrightarrow{\mathrm{inc}(S,\chi)}$ and the fact that $\theta'_1 \preceq_\iota \theta_1$ we have $m'_0 = m'_1 = m_1 = m_0$ and $r'_0 = r'_1 = r_1 = r_0$. As before, from Item (a) it remains to show that for all (S,χ) we have that $|(\Delta_0 \otimes \Gamma_0)^{-1}(S,\chi)| > 0$ implies $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S,\chi)| > 0$ and that $|(\Delta_0 \otimes \Gamma_0)^{-1}(S,\chi)| > 1$ implies $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S,\chi)| > 1$. We only show the second one as the first one is similar and simpler.

Assume we have two distinct data values $e \neq e'$ such that $(\Delta_0 \otimes \Gamma_0)(e) = (\Delta_0 \otimes \Gamma_0)(e') = (S_0,\chi_0)$. If $(S_0,\chi_0) \neq (S,\chi)$ then we have $(\Delta_1 \otimes \Gamma_1)(e) = (\Delta_1 \otimes \Gamma_1)(e') = (S_0,\chi_0)$. By the fact that $\mathit{profile}(\theta'_1) = \mathit{profile}(\theta_1)$, we have $|(\Delta'_1 \otimes \Gamma'_1)^{-1}(S_0,\chi_0)| \geq 2$ and we get the two desired data values d and d' as by definition of $\xrightarrow{\mathrm{inc}(S,\chi)}$ we will have $(\Delta'_0 \otimes \Gamma'_0)(d) = (\Delta'_0 \otimes \Gamma'_0)(d) = (S_0,\chi_0)$. If $(S_0,\chi_0) = (S,\chi)$ then we consider two cases, either $|(\Delta_1 \otimes \Gamma_1)^{-1}(S,\chi)| \geq 2$ and we reason as above, or $|(\Delta_1 \otimes \Gamma_1)^{-1}(S,\chi)| = 1$ and in this case $|(\Delta'_1 \otimes \Gamma'_1)^{-1}(S,\chi)| = 1$ and $\xrightarrow{\mathrm{inc}(S,\chi)}$ provides the second data value we are looking for in θ'_0. Notice that $|(\Delta_1 \otimes \Gamma_1)^{-1}(S,\chi)|$ cannot be 0 as $\xrightarrow{\mathrm{inc}(S,\chi)}$ introduces only one data value.

4. $\xrightarrow{\mathsf{univ}}$. Suppose $\theta'_1 \preceq_\iota \theta_1 \xrightarrow{\mathsf{univ}} \theta_0$.

By definition of $\xrightarrow{\mathsf{univ}}$ there is $(q,d) \in \Delta_1$ with $\delta(q) = \mathsf{univ}(r)$, $c_0 = c_1$, $d_0 = d_1$, $\Gamma_0 = \Gamma_1$ and $\Delta_0 = (\Delta_1 \setminus \{(q,d)\}) \cup \{(r,e) : (\mu,e) \in \Gamma_1\}$. We first check that $\xrightarrow{\mathsf{univ}}$ can be applied to θ'_1. Let $(S,\chi) = (\Delta_1 \otimes \Gamma_1)(d)$. Since θ_1 and θ'_1 have the same profile, there must be a data value in $(\Delta'_1 \otimes \Gamma'_1)^{-1}(S,\chi)$. As in Claim 7.18.1 we can assume without loss of generality that this data value is d. Hence $(q,d) \in \Delta'_1$ and we can apply the transition $\xrightarrow{\mathsf{univ}}$ to θ'_1 resulting in θ'_0 be such that $d'_0 = d'_1$, $c'_0 = c'_1$, $\Gamma'_0 = \Gamma'_1$ and $\Delta'_0 = (\Delta'_1 \setminus \{(q,d)\}) \cup \{(r,e) : (\mu,e) \in \Gamma'_1\}$.

We now show that $\theta'_0 \preceq \theta_0$, that is, that (a) $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$, and (b) $\mathit{profile}(\theta'_0) = \mathit{profile}(\theta_0)$.

(a) First, we show that $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$. Let $e \in \mathit{data}(\theta'_0)$ and let $(S,\chi) = (\Delta'_0 \otimes \Gamma'_0)(e)$. We remark that for $e = d$, the data value that triggers the transition,

$$(\Delta'_0 \otimes \Gamma'_0)(d) = (\Delta_0 \otimes \Gamma_0)(d), \tag{7.1}$$

since $(\Delta'_1 \otimes \Gamma'_1)(d) = (\Delta_1 \otimes \Gamma_1)(d)$ by $\preceq_\iota$. Assume that $e \neq d$. By definition of Δ'_0, we have $(S',\chi) = (\Delta'_1 \otimes \Gamma'_1)(e)$ for either $S' = S$ or $S' = S \setminus \{r\}$. By $\theta'_1 \preceq_\iota \theta_1$, we have $(S',\chi) = (\Delta_1 \otimes \Gamma_1)(e)$ and by definition of $\xrightarrow{\mathsf{univ}}$, $(S,\chi) = (\Delta_0 \otimes \Gamma_0)(e)$.

(b) We now show that $\mathit{profile}(\theta_0) = \mathit{profile}(\theta'_0)$. Notice that $r_0 = r_1 = r'_1 = r'_0$ and $m_0 = m_1 = m'_1 = m'_0$. Hence by Item (a) it remains to prove that for all (S,χ): $|(\Delta_0 \otimes \Gamma_0)^{-1}(S,\chi)| > 0$ implies $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S,\chi)| > 0$ and $|(\Delta_0 \otimes \Gamma_0)^{-1}(S,\chi)| > 1$ implies $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S,\chi)| > 1$. We only show the second one as the first one is similar and simpler.

Assume we have two distinct data values $e \neq e'$ such that $(\Delta_0 \otimes \Gamma_0)(e) = (\Delta_0 \otimes \Gamma_0)(e') = (S_0, \chi_0)$. Let

$$(S_1, \chi_1) = (\Delta_1 \otimes \Gamma_1)(e), \qquad (S'_1, \chi'_1) = (\Delta_1 \otimes \Gamma_1)(e').$$

By definition of $\xrightarrow{\text{merge}}$ we have $S_1 = S_0$ or $S_1 = S_0 \setminus \{r\}$, similarly for S'_1. As $\mathit{profile}(\theta_1) = \mathit{profile}(\theta'_1)$ we get two distinct data values d and d' such that $(S_1, \chi_1) = (\Delta'_1 \otimes \Gamma'_1)(d)$, $(S'_1, \chi'_1) = (\Delta'_1 \otimes \Gamma'_1)(d')$. Notice now that by definition of $\xrightarrow{\text{merge}}$ we get $(S_0, \chi_0) = (\Delta'_0 \otimes \Gamma'_0)(d) = (\Delta'_0 \otimes \Gamma'_0)(d')$ as desired.

5. The cases $\xrightarrow{\langle\mu\rangle^=}$, $\xrightarrow{\langle\mu\rangle^\neq}$, $\xrightarrow{\mathsf{root}}$ and their negative counterparts are treated similary. We only treat in detail the case of $\xrightarrow{\langle\mu\rangle^=}$.

   By definition of $\xrightarrow{\langle\mu\rangle^=}$ there is $(q, d) \in \Delta_1$ with $\delta(q) = \langle\mu\rangle^=$. Then, θ_0 is defined as $c_0 = c_1$, $d_0 = d_1$, $\Gamma_0 = \Gamma_1$ and $\Delta_0 = (\Delta_1 \setminus \{(q, d)\})$.

   We first check that $\xrightarrow{\langle\mu\rangle^=}$ can be applied to θ'_1. Let $(S, \chi) = (\Delta_1 \otimes \Gamma_1)(d)$. Since θ_1 and θ'_1 have the same profile, there must be an element in $(\Delta'_1 \otimes \Gamma'_1)^{-1}(S, \chi)$. As in Claim 7.18.1 we can assume without any loss of generality that $(\Delta'_1 \otimes \Gamma'_1)(d) = (S, \chi)$.

   Hence $\mu \in \Gamma'_1(d)$ we can apply $\xrightarrow{\langle\mu\rangle^=}$ to θ'_1.

   Let θ'_0 be such that $d'_0 = d'_1$, $c'_0 = c'_1$, $\Gamma'_0 = \Gamma'_1$ and $\Delta'_0 = (\Delta'_1 \setminus \{(d, q)\})$. Then we have by definition $\theta'_1 \xrightarrow{\langle\mu\rangle^=} \theta'_0$.

   We show that $\theta'_0 \preceq \theta_0$, that is, that (a) $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$, and (b) $\mathit{profile}(\theta'_0) = \mathit{profile}(\theta_0)$.

   (a) We first show $(\Delta'_0 \otimes \Gamma'_0) \subseteq (\Delta_0 \otimes \Gamma_0)$. Let $e \in \mathit{data}(\theta'_0)$ and let $(S, \chi) = (\Delta'_0 \otimes \Gamma'_0)(e)$. If $e \neq d$, from the definition of $\xrightarrow{\langle\mu\rangle^=}$ it follows $(\Delta'_1 \otimes \Gamma'_1)(e) = (S, \chi)$. Because $\theta'_1 \preceq_\iota \theta_1$ we also have $(\Delta_1 \otimes \Gamma_1)(e) = (S, \chi)$. Hence, by definition of $\xrightarrow{\langle\mu\rangle^=}$, $(\Delta_0 \otimes \Gamma_0)(e) = (S, \chi)$ as desired.

   On the other hand, we remark that for the data value d that triggers the transition $(\Delta'_1 \otimes \Gamma'_1)(d) = (S \cup \{q\}, \chi)$. As $\theta'_1 \preceq_\iota \theta_1$ we have $(\Delta_1 \otimes \Gamma_1)(d) = (S \cup \{q\}, \chi)$ and therefore $(\Delta_0 \otimes \Gamma_0)(d) = (S, \chi)$ by definition of $\xrightarrow{\langle\mu\rangle^=}$.

   (b) We now show $\mathit{profile}(\theta_0) = \mathit{profile}(\theta'_0)$. Notice that $r_0 = r_1 = r'_1 = r'_0$ and $m_0 = m_1 = m'_1 = m'_0$. Hence by Item (a) it remains to prove that for all (S, χ): $|(\Delta_0 \otimes \Gamma_0)^{-1}(S, \chi)| > 0$ implies $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S, \chi)| > 0$ and $|(\Delta_0 \otimes \Gamma_0)^{-1}(S, \chi)| > 1$ implies $|(\Delta'_0 \otimes \Gamma'_0)^{-1}(S, \chi)| > 1$. We only show the second one as the first one is similar and simpler.

   Assume we have two distinct data values $e_1 \neq e_2$ such that $(\Delta_0 \otimes \Gamma_0)(e_1) = (\Delta_0 \otimes \Gamma_0)(e_2) = (S_0, \chi_0)$. Let

$$(S_1, \chi_1) = (\Delta_1 \otimes \Gamma_1)(e_1), \qquad (S'_1, \chi'_1) = (\Delta_1 \otimes \Gamma_1)(e_2.$$

As $\mathit{profile}(\theta_1) = \mathit{profile}(\theta'_1)$ we get two distinct data values e'_1 and e'_2 such that $(S_1, \chi_1) = (\Delta'_1 \otimes \Gamma'_1)(e'_1)$, $(S'_1, \chi'_1) = (\Delta'_1 \otimes \Gamma'_1)(e'_2)$. As $\theta_1 \preceq_{\boldsymbol{l}} \theta'_1$ and $\boldsymbol{l} \geq 3$ we can further choose e'_1 and e'_2 such that $e'_1 = d$ iff $e_1 = d$ and $e'_2 = d$ iff $e_2 = d$. Hence, by definition of $\xrightarrow{\langle\mu\rangle^=}$ we get $(\Delta'_0 \otimes \Gamma'_0)(e'_1) = (\Delta'_0 \otimes \Gamma'_0)(e'_2) = (S_0, \chi_0)$ as desired.

For the other cases, we do exactly the same as for $\xrightarrow{\langle\mu\rangle^=}$. We only need to verify that whenever $\theta'_1 \preceq_{\boldsymbol{l}} \theta_1$, if θ_1 satisfies the condition for any transition in $\{\xrightarrow{\langle\mu\rangle^\neq}, \xrightarrow{\mathsf{root}}\}$ or their negative counterparts, then θ'_1 also satisfies this condition. For all these cases this is a simple consequence of θ_1 and θ'_1 having the same profile. Note that by Claim 7.18.1 we can assume without loss of generality that $(\Delta_1 \otimes \Gamma_1)(d) = (\Delta'_1 \otimes \Gamma'_1)(d)$.

In the case $\overline{\langle\mu\rangle^=}$, if $\mu \notin \Gamma_1(d)$, then the same hold in θ'_1.

For the case $\langle\mu\rangle^\neq$, if $\mu \in \Gamma_1(d')$ for some $d' \neq d$, then by equality of profiles there is a data value e such that $(\Delta'_1 \otimes \Gamma'_1)(e) = (\Delta_1 \otimes \Gamma_1)(d')$. Notice that even in the case when $(\Delta_1 \otimes \Gamma_1)(e) = (\Delta_1 \otimes \Gamma_1)(d)$ the definition of *profile* and the fact that $d \neq d'$ guarantees that we can always choose $e \neq d$. Hence $\mu \in \Gamma_1(e)$ and $e \neq d$ as required.

For the case $\overline{\langle\mu\rangle^\neq}$, if $\mu \notin \Gamma_1(d')$ for all $d' \neq d$, $\Gamma_1(d)$ is the unique one that may contain μ. By equality of the profile, this must also be the case for $\Gamma'_1(d)$ as desired.

The cases $\xrightarrow{\mathsf{root}}$ and $\overline{\xrightarrow{\mathsf{root}}}$ are straightforward as the value of r is preserved by $\precsim$.

6. $\xrightarrow{\mathsf{guess}}$. Suppose we have $\theta'_1 \preceq_{\boldsymbol{l}} \theta_1 \xrightarrow{\mathsf{guess}} \theta_0$.

   By definition of $\xrightarrow{\mathsf{guess}}$ there is $(q, d) \in \Delta_1$ with $\delta(q) = \mathsf{guess}(p)$. Then, θ_0 is defined as $c_0 = c_1$, $d_0 = d_1$, $r_0 = r_1$, $m_0 = m_1$, $\Gamma_0 = \Gamma_1$ and $\Delta_0 = (\Delta_1 \setminus \{(q, d)\}) \cup \{(p, e)\}$ for some data value e. By Claim 7.18.1 we can assume without loss of generality that $e \in \mathit{data}(\theta_1)$ iff $e \in \mathit{data}(\theta'_1)$.

   Let $(S, \chi) = (\Delta_1 \otimes \Gamma_1)(d)$. From $\theta'_1 \preceq_{\boldsymbol{l}} \theta_1$ we know that the set $(\Delta'_1 \otimes \Gamma'_1)^{-1}(S, \chi)$ is not empty and, as in Claim 7.18.1 we can assume without loss of generality that d is in this set. Hence $(q, d) \in \Delta'_1$.

   Let θ'_0 be such that $\theta'_1 \xrightarrow{\mathsf{guess}} \theta'_0$ with $\Delta'_0 = (\Delta'_1 \setminus \{(q, d)\}) \cup \{(p, e)\}$. It is immediate to verify that $\theta'_0 \preceq \theta_0$.

7. The cases $\xrightarrow{\wedge}$ and $\xrightarrow{\vee}$ are treated similarly. Assume for instance that $\theta'_1 \preceq_{\boldsymbol{l}} \theta_1 \xrightarrow{\wedge} \theta_0$.

   By definition of $\xrightarrow{\wedge}$ there is $(q, d) \in \Delta_1$ with $\delta(q) = p \wedge p'$. Then, θ_0 is defined as $c_0 = c_1$, $d_0 = d_1$, $r_0 = r_1$, $m_0 = m_1$, $\Gamma_0 = \Gamma_1$ and $\Delta_0 = (\Delta_1 \setminus \{(q, d)\}) \cup \{(p, d), (p', d)\}$. Let $(S, \chi) = (\Delta_1 \otimes \Gamma_1)(d)$. From $\theta'_1 \preceq_{\boldsymbol{l}} \theta_1$ the set $(\Delta'_1 \otimes \Gamma'_1)^{-1}(S, \chi)$ is not empty and, by Claim 7.18.1 we can assume that d is in this set. Hence $(q, d) \in \Delta'_1$.

Let θ'_0 be such that $\theta'_1 \xrightarrow{\hat{\wedge}} \theta'_0$ with $\Delta'_0 = (\Delta'_1 \setminus \{(q,d)\}) \cup \{(p,d),(p',d)\}$. It is immediate to verify that $\theta'_0 \preceq \theta_0$. □

We can now easily lift the compatibility result of Lemma 7.18 to $\wp_{<\infty}(\mathrm{AC})$.

Theorem 7.19. *$(\wp_{<\infty}(\mathrm{AC}), \Rightarrow)$ is $\mathbf{N}$-downward compatible with respect to the wqo $(\wp_{<\infty}(\mathrm{AC}), \leq_{min})$, for $\mathbf{N} := (|\mathcal{S}|.|Q|).\boldsymbol{l}+1$.*

Proof. We prove that for every $\Theta'_1 \leq_{min} \Theta_1 \Rightarrow \Theta_2$ there exists Θ'_2 such that $\Theta'_1 \Rightarrow^n \Theta'_2$ for some $n \leq (|\mathcal{S}|.|Q|).\boldsymbol{l}+1$ and $\Theta'_2 \leq_{min} \Theta_2$.

Given $\Theta'_1 \leq_{min} \Theta_1 \Rightarrow \Theta_2$, suppose that the transition from Θ_1 to Θ_2 was applied to $\theta_1 \in \Theta_1$. Then, by $\Theta'_1 \leq_{min} \Theta_1$, there must exist some $\theta'_1 \in \Theta'_1$ with $\theta'_1 \precsim \theta_1$. By Lemma 7.17 there exists $\hat{\Theta}'_1$ with $\Theta'_1 \Rightarrow^m \hat{\Theta}'_1$, $m \leq (|\mathcal{S}|.|Q|).\boldsymbol{l}$ and some $\hat{\theta}'_1 \in \hat{\Theta}'_1$ such that $\theta'_1 \preceq \hat{\theta}'_1 \precsim_{\boldsymbol{l}} \theta_1$ and $\hat{\Theta}'_1 \leq_{min} \Theta_1$. By Lemma 7.18 we can then apply to $\hat{\theta}'_1$ the same transition that was applied on θ_1 resulting in an abstract configuration θ'_2 obtaining the desired Θ_2 from $\hat{\Theta}'_1$. □

Let us write $\equiv$ for the equivalence relation over $\wp_{<\infty}(\mathrm{AC})$ such that $\Theta \equiv \Theta'$ iff $\Theta \leq_{min} \Theta'$ and $\Theta' \leq_{min} \Theta$.

Given a BUDA $\mathcal{A}$, the WSTS $(\wp_{<\infty}(\mathrm{AC})/\equiv, \Rightarrow, \leq_{min})$ as constructed in the previous sections is called *the* WSTS *associated to* $\mathcal{A}$, and we note it with the letter $\mathcal{W}$.

Proposition 7.20. *Given a BUDA $\mathcal{A}$, it is decidable whether the* WSTS *associated to $\mathcal{A}$ can reach an accepting abstract configuration from its initial abstract configuration.*

Proof. By Theorem 7.19, condition (1) of Proposition 2.12 is met. Similarly as in Chapter 6, we have that $(\wp_{<\infty}(\mathrm{AC})/\equiv, \Rightarrow)$ is finitely branching and effective. That is, the $\Rightarrow$-image of any configuration has only a finite number of configurations modulo $\equiv$, and representatives for every class are effectively computable. Hence, we have that $(\wp_{<\infty}(\mathrm{AC})/\equiv, \Rightarrow, \leq_{min})$ verifies condition (2) of Proposition 2.12. Finally, condition (3) holds as $(\wp_{<\infty}(\mathrm{AC})/\equiv, \leq_{min})$ is a wqo (by Lemma 7.11) and a computable relation. Finally, by Lemma 7.12 we have the set of accepting configurations is downwards closed. Then, by Lemma 2.13 we can decide if there are accepting configurations that can be reached from the initial set of configurations by performing $\Rightarrow$ transitions. □

In the next section, we will show that this implies the decidability for the emptiness problem for BUDA automata.

7.3.5 From BUDA to its abstract configurations

As expected, the WSTS associated to a BUDA $\mathcal{A}$ reflects its behavior. That is, reachability of one corresponds exactly to accessibility of the other. One direction is easy as the transition system can easily simulate $\mathcal{A}$. The other direction requires more care. As evidenced in (†), the WSTS may perform a $\xrightarrow{\mathrm{inc}(S,\chi)}$ transition anytime.

However, BUDA can only make the tree grow in width when moving up in the tree. This issue is solved by showing that all other transitions commute with $\xrightarrow{\mathrm{inc}(S,\chi)}$. We first show the easy direction.

In the sequel, we say that $\Gamma \subseteq (\mathcal{S} \times \mathbb{D})$ is **consistent** with a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d} \in \mathit{Trees}(\mathbb{A} \times \mathbb{B} \times \mathbb{D})$ when for every possible (μ, d), Γ contains (μ, d) iff there is a downward path in $\mathbf{t}$ that starts at the root and ends at some position x such that $\mathbf{d}(x) = d$ and evaluates to μ via h.

We first show that the associated WSTS of a BUDA simulates at least its behavior.

Lemma 7.21. *Given $\mathcal{A} \in$ BUDA and its associated* WSTS *$\mathcal{W}_\mathcal{A}$. If $\mathcal{A}$ has an accepting run then $\mathcal{W}_\mathcal{A}$ can reach an accepting abstract configuration from its initial abstract configuration.*

Proof. We show that from every accepting run ρ of $\mathcal{A}$ on $\mathbf{t} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{d}$ there exists a finite sequence of transitions in $\wp_{<\infty}(\mathrm{AC})$ starting in Θ_I and ending in an accepting configuration.

As ρ is accepting, we can assign to any node $x \in \mathsf{pos}(\mathbf{t})$, a configuration $(x, \mathcal{C}_x)$ in the ϵ-closure of $(x, \rho(x))$ such that

1. $\mathcal{C}_x$ is moving,
2. if $x = \epsilon$ then $\mathcal{C}_x = \emptyset$, and
3. if $x = y{\cdot}i$, then $\rho(y) = \bigcup_{j \in [n]} \mathcal{C}'_{y \cdot j}$, with $\mathcal{C}_{x \cdot j} \rightarrowtail_{up} \mathcal{C}'_{y \cdot j}$ for every j, where $\mathcal{C}_{y \cdot 1}, \ldots, \mathcal{C}_{y \cdot n}$ are the associated configurations of the siblings of x.

Notice that such configurations $\mathcal{C}_x$ must exist by definition of an accepting run.

Given a position $x \in \mathsf{pos}(\mathbf{t})$, an abstract configuration of $\mathbf{t}|_x$, is a configuration of the form $\theta_x = (\Delta_x, \Gamma_x, r_x, m_x)$ such that $r_x = \mathit{false}$ (unless x is the root position), $m_x = \mathit{false}$, $\mathbf{a} \otimes \mathbf{b}(x)$ is the label of θ_x, $\mathbf{d}(x)$ is the data value of θ_x and Γ_x is consistent with $\mathbf{t}|_x$. An abstract configuration of x is the *first abstract configuration of* x if moreover $\Delta_x = \rho(x)$, and the *last abstract configuration of* x if $\Delta_x = \mathcal{C}_x$.

The set of abstract configurations Θ that we derive from Θ_I correspond to sets of abstract configurations of subtrees of $\mathbf{t}$, computed from bottom to top. More precisely, for each subtree $\mathbf{t}|_x$ we

1. first derive the last abstract configurations $\theta_{x \cdot 1}, \ldots, \theta_{x \cdot n}$ of the children positions $x{\cdot}1, \ldots, x{\cdot}n$ of x, obtaining a set of abstract configurations Θ containing $\{\theta_{x \cdot 1}, \ldots, \theta_{x \cdot n}\}$ (modulo $\sim$),
2. then we derive the first abstract configuration of x by n applications of $\xrightarrow{\mathrm{grow}}$ using $\mathbf{a} \otimes \mathbf{b}(x)$ for the label of the new abstract configuration, and $(n-1)$ applications of $\xrightarrow{\mathrm{merge}}$, and
3. finally, we derive the last abstract configuration of x by simulating the disjunction of conjunctions using $\xrightarrow{\wedge}$, $\xrightarrow{\vee}$, and each ϵ-operation of $\mathcal{A}$ using its

corresponding transition on $\mathcal{W}_{\mathcal{A}}$. Thus, we arrive at a set of abstract configurations Θ' containing Θ and also (modulo $\sim$) the last abstract configuration θ_x.

when x is a leaf, the process is initiated with Θ_I and we only apply Item (3).

Repeating this simulation we finally arrive at a configuration Θ_{Acc} containing the last abstract configurations θ_x (modulo $\sim$) of all the nodes of $\mathbf{t}$. Since θ_ϵ is accepting, Θ_{Acc} is accepting, and we then built an accepting derivation $\Theta_I \Rightarrow^* \Theta_{Acc}$. □

The other direction requires more care as a WSTS may perform a "merge" transition anytime while a BUDA can only simulate a "merge" in combination with a "grow" when moving up in the tree. This issue is solved by first showing that all other transitions commute with "merge". First, we need to prove the following lemma.

Lemma 7.22. *If we have* $\theta_1 \to_\epsilon \theta \xrightarrow{inc(S,\chi)} \theta_2$ *then either*

- $\theta_1 \xrightarrow{inc(S',\chi)} \theta' \to_\epsilon \theta_2$, *or*
- $\theta_1 \xrightarrow{inc(S',\chi)} \theta' \to_\epsilon \theta'' \to_\epsilon \theta_2$

for some θ', θ'' *and* S'.

Proof. Suppose that $\theta \xrightarrow{\mathrm{inc}(S,\chi)} \theta_2$. By definition we have $|(\Delta \otimes \Gamma)^{-1}(S,\chi)| \geq 1$ and $|\Gamma^{-1}(\chi)| \geq 2$ (or $\chi = \emptyset$) and a new data value e of type (S,χ) has been introduced. We then have that $e \notin data(\theta_1)$.

An important observation is that if $\theta_1 \to_\epsilon \theta$ then for any data value $d \in data(\theta)$ occurring in $data(\theta_1)$ we have $\Gamma_1(d) = \Gamma(d)$. The only case where $d \in data(\theta) \setminus data(\theta_1)$ is when d is generated as a fresh new data value by $\xrightarrow{\mathsf{guess}}$. In this latter case, $\Gamma(d) = \emptyset$. We therefore consider two cases depending on whether $\chi = \emptyset$ or not. Fix a data value e' such that $(\Delta \otimes \Gamma)(e') = (S,\chi)$.

Assume that $\theta_1 \xrightarrow{\mathrm{X}} \theta$ for some ϵ-transition $\xrightarrow{\mathrm{X}}$.

1. $\chi \neq \emptyset$. Then from the observation above $|\Gamma^{-1}(\chi)| \geq 2$ implies that $|\Gamma_1^{-1}(\chi)| \geq 2$.

   Let (q,d) be the thread of θ_1 where $\xrightarrow{\mathrm{X}}$ acts on to obtain θ and let $(S_1,\chi) = (\Delta_1 \otimes \Gamma_1)(e')$.

   If $e' \neq d$, notice that $\theta_1 \xrightarrow{\mathrm{inc}(S_1,\chi)} \theta' \xrightarrow{\mathrm{X}} \theta_2$, where $\xrightarrow{\mathrm{inc}(S_1,\chi)}$ introduces a copy e of e' and $\xrightarrow{\mathrm{X}}$ applies on the thread (q,d), just as before.

   If $e' = d$, notice that $\theta_1 \xrightarrow{\mathrm{inc}(S_1,\chi)} \theta' \xrightarrow{\mathrm{X}} \theta'' \xrightarrow{\mathrm{X}} \theta_2$, where $\xrightarrow{\mathrm{inc}(S_1,\chi)}$ introduces a copy e of e' while the first occurrence of $\xrightarrow{\mathrm{X}}$ applies on the thread (q,e') and the second on the thread (q,e). Notice that by construction we have $profile(\theta_1) = profile(\theta) = profile(\theta') = profile(\theta'')$ hence both occurrences of $\xrightarrow{\mathrm{X}}$ can be applied as the test they perform rely only on the profile.

2. $\chi = \emptyset$. If $|\Gamma_1^{-1}(\emptyset)| \geq 2$ we reason as in the previous case. Otherwise, as $|\Gamma^{-1}(\emptyset)| \geq 2$ we know that $|\Gamma_1^{-1}(\emptyset)| = 1$ and $\xrightarrow{\mathrm{X}}$ is $\xrightarrow{\mathsf{guess}}$, applying on a thread (q, d) and introducing a fresh new data value d' of type $(\{p\}, \emptyset)$ if $\delta_\epsilon(q) = \mathsf{guess}(p)$. We distinguish several cases.

   If $e' \in \mathit{data}(\theta_1)$ and $d \neq e'$ then $(S, \chi) = (\Delta_1 \otimes \Gamma_1)(e')$ and notice that $\theta_1 \xrightarrow{\mathrm{inc}(S,\chi)} \theta' \xrightarrow{\mathsf{guess}} \theta_2$ where $\xrightarrow{\mathrm{inc}(S,\chi)}$ introduces the appropriate copy e of e' while $\xrightarrow{\mathsf{guess}}$ introduces d'.

   If $e' \in \mathit{data}(\theta_1)$ and $d = e'$ then let $(S_1, \chi) = (\Delta_1 \otimes \Gamma_1)(e')$ and notice that $\theta_1 \xrightarrow{\mathrm{inc}(S,\chi)} \theta' \xrightarrow{\mathsf{guess}} \theta'' \xrightarrow{\mathsf{guess}} \theta_2$ where $\xrightarrow{\mathrm{inc}(S,\chi)}$ introduces a copy e of e' while both $\xrightarrow{\mathsf{guess}}$ introduces the same data value d' but apply successively on (q, e) and (q, e').

   The remaining case is when $e' \notin \mathit{data}(\theta_1)$. Hence $e' = d'$ and $S = \{p\}$ where $\delta(q) = \mathsf{guess}(p)$. Then notice that $\theta_1 \xrightarrow{\mathrm{inc}(S,\chi)} \theta' \xrightarrow{\mathsf{guess}} \theta_2$ where $\xrightarrow{\mathrm{inc}(S,\chi)}$ introduces the appropriate data value e with the right type. □

Finally we show:

Lemma 7.23. *Given a BUDA $\mathcal{A}$ and its associated* WSTS $\mathcal{W}_\mathcal{A}$. *If $\mathcal{W}_\mathcal{A}$ can reach an accepting abstract configuration from its initial abstract configuration then $\mathcal{A}$ has an accepting run.*

Proof. We show that for every finite sequence of transitions $\Theta_0 \Rightarrow \cdots \Rightarrow \Theta_n$ where $\Theta_0 = \Theta_I$ is the initial set of configurations and Θ_n a final configuration over $\mathcal{W}_\mathcal{A}$, for an automaton $\mathcal{A} \in \mathsf{BUDA}$, there exists a data tree $\mathbf{t} = \mathbf{a} \otimes \mathbf{d}$ accepted by $\mathcal{A}$.

The general idea is simple. We construct by induction a set T_i for each set of abstract configuration Θ_i containing a triplet $(\mathbf{t}_\theta, \rho_\theta, f_\theta)$ per abstract configuration $\theta \in \Theta_i$. This triplet consists in a data tree $\mathbf{t}_\theta \in \mathit{Trees}(\mathbb{A} \times \mathbb{B} \times \mathbb{D})$, a run over $\mathbf{t}_\theta$, and a function that associates a configuration to each node of $\mathbf{t}_\theta$, such that:

(i) ρ_θ is a valid run of $\mathcal{A}$ on $\mathbf{t}_\theta$,

(ii) $f_\theta(\epsilon) = \Delta_\theta$ is in the ϵ-closure of $\rho_\theta(x)$,

(iii) if $x{\cdot}1, \ldots, x{\cdot}k$ are the children of x then $\rho(x) = \bigcup_i \mathcal{C}_i$ where $f_\theta(x{\cdot}i) \rightarrowtail_{up} \mathcal{C}_i$, and

(iv) Γ_θ is consistent with $\mathbf{t}_\theta$.

By Lemma 7.22 and because of the flag m of each abstract configuration, we can assume without loss of generality that the derivation $\Theta_I \Rightarrow \Theta_n$ is a repetition of the following pattern: a sequence of ϵ-transitions not involving $\xrightarrow{\mathrm{inc}(S,\chi)}$, a transition $\xrightarrow{\mathrm{grow}}$, a sequence of transitions $\xrightarrow{\mathrm{merge}}$, a sequence of transition $\xrightarrow{\mathrm{inc}(S,\chi)}$.

The induction step is straightforward for all ϵ-transitions: the data tree and the run remain unchanged while $f_{\theta'}$ is constructed by modifying, for the position

x of $\mathbf{t}_\theta$, $f_\theta(x)$ using the appropriate ϵ-transition (that can be applied by Item (iv)) yielding $\Delta_{\theta'}$ as desired.

For a transition of the form $\theta \xrightarrow{\text{grow}} \theta'$, the data tree $\mathbf{t}_{\theta'}$ is constructed from $\mathbf{t}_\theta$ by adding a new root node that has the same label and data value as θ'.

The run $\rho_{\theta'}$ is the extension of ρ_θ setting $\rho_{\theta'}|_1 = \rho_{\theta'}$ and $\rho_{\theta'}(\epsilon) = \Delta_{\theta'}$. The function $f_{\theta'}$ is the extension of f_θ setting $f_{\theta'}|_1 = f_\theta$ and $f_{\theta'}(\epsilon) = \Delta_{\theta'}$. The reader can verify that we do have $f_{\theta'}(1) \rightarrowtail_{up} \Delta_{\theta'}$, as desired for our inductive hypothesis.

For a transition of the form $\theta_1\theta_2 \xrightarrow{\text{merge}} \theta'$, the data tree $\mathbf{t}_{\theta'}$ is constructed from $\mathbf{t}_{\theta_1}$ and $\mathbf{t}_{\theta_2}$ by identifying their root nodes (notice that by definition of $\xrightarrow{\text{merge}}$ the two roots have the same label and data value). That is, we set $\mathbf{t}_{\theta'}(\epsilon) = \mathbf{t}_{\theta_1}(\epsilon) = \mathbf{t}_{\theta_2}(\epsilon)$; $\mathbf{t}_{\theta'}|_i = \mathbf{t}_{\theta_1}|_i$ for all $i \in [k]$ where k is the number of children of the root of $\mathbf{t}_{\theta_1}$; and $\mathbf{t}_{\theta'}|_{k+i} = \mathbf{t}_{\theta_2}|_i$ for all children i of the root of $\mathbf{t}_{\theta_2}$. The run $\rho_{\theta'}$ and (resp. the function $f_{\theta'}$) are constructed in a similar fashion by setting $\rho_{\theta'}(\epsilon) = \rho_{\theta_1}(\epsilon) \cup \rho_{\theta_2}(\epsilon)$ (resp. $f_{\theta'}(\epsilon) = f_{\theta_1}(\epsilon) \cup f_{\theta_2}(\epsilon)$) and for all other position $i{\cdot}x$ setting $\rho_{\theta_1}(i{\cdot}x)$ or $\rho_{\theta_2}((i-k){\cdot}x)$ (resp. $f_{\theta_1}(i{\cdot}x)$ or $f_{\theta_2}((i-k){\cdot}x)$) depending whether $i \leq k$ or not (*i.e.*, whether it comes from $\mathbf{t}_{\theta_1}$ or $\mathbf{t}_{\theta_2}$). It remains to verify that these definitions have the desired properties by induction on the length of the sequence of $\xrightarrow{\text{merge}}$ that $\rho_{\theta'}(\epsilon) = f_{\theta'}(\epsilon) = \Delta_{\theta'} = \bigcup_i \mathcal{C}_i$ such that $f_{\theta'}(i) \rightarrowtail_{up} \mathcal{C}_i$ were i ranges over all the children of the root. Notice that it is important here that no ϵ-transition occur between a $\xrightarrow{\text{grow}}$ and a $\xrightarrow{\text{merge}}$.

The transitions $\xrightarrow{\text{inc}(S,\chi)}$ are treated as a special case of $\xrightarrow{\text{merge}}$ where θ_2 is a copy of θ_1 except for one of the data values of type (S, χ). As we work modulo equivalence we can always assume that such a θ_2 is in our set of abstract configurations whenever θ_1 is.

This concludes the proof. The set T_n must contain a tuple $(\mathbf{t}_\theta, \rho_\theta, f_\theta)$ for θ a final abstract configuration. For this configuration we will have by Item (i) that ρ_θ is a valid run of $\mathcal{A}$ for $\mathbf{t}_\theta$, and by Item (ii) that $f_\theta(\epsilon) = \Delta_\theta = \emptyset$ is in the ϵ-closure of $\rho_\theta(x)$. Hence, ρ_θ is an accepting run of $\mathcal{A}$ for $\mathbf{t}_\theta$. □

By combining the previous Lemmas we immediately obtain:

Proposition 7.24. *Let $\mathcal{A}$ be a BUDA. Let $\mathcal{W}$ be the* WSTS *associated to $\mathcal{A}$. Then $\mathcal{A}$ has an accepting run iff $\mathcal{W}$ can reach an accepting set of abstract configurations from the initial set of abstract configurations.*

Proof. Immediate from Lemmas 7.21 and 7.23. □

Combining Proposition 7.24 and Proposition 7.20, we prove Theorem 7.5.

7.4 Satisfiability of vertical XPath

The goal of this section is to prove that the class BUDA captures regXPath$(\mathfrak{V}, =)$. Given a formula η of regXPath$(\mathfrak{V}, =)$, we say that a BUDA $\mathcal{A}$ is *equivalent* to η if a data tree $\mathbf{t}$ is accepted by $\mathcal{A}$ iff $[\![\eta]\!]^{\mathbf{t}} \neq \emptyset$. We then obtain the following.

Proposition 7.25. *For every $\eta \in \mathsf{regXPath}(\mathfrak{V}, =)$ there exists an equivalent $\mathcal{A} \in$ BUDA computable from η.*

The idea is that it is easy to simulate any positive test $\langle \alpha = \beta \rangle$ or $\langle \alpha \neq \beta \rangle$ of vertical XPath by a BUDA using guess, $\langle \mu \rangle^{=}$ and $\langle \mu \rangle^{\neq}$. For example, consider the property $\langle \downarrow_* [a] \neq \uparrow\downarrow [b] \rangle$, which states that there is a descendant labeled a with a different data value than a sibling labeled b. A BUDA automaton can test this property as follows.

1. It guesses a data value d and stores it in the register.
2. It tests that d can be reached by $\downarrow[a]$ with a test $\langle \mu \rangle^{=}$ for a suitable μ.
3. It moves up to its parent.
4. It tests that a different value than d can be reached in one of its children labeled with b, using the test $\langle \mu \rangle^{\neq}$ for a suitable μ.

The simulation of negative tests ($\neg\langle \alpha = \beta \rangle$ or $\neg\langle \alpha \neq \beta \rangle$) is more tedious as BUDA is not closed under complementation. Nevertheless, the automaton has enough universal quantifications (in the operations univ, $\overline{\langle \mu \rangle^{=}}$ and $\overline{\langle \mu \rangle^{\neq}}$) in order to do the job. Consider for example the formula $\neg\langle \uparrow^* [b] \downarrow [a] = \downarrow_* [c] \rangle$, that states that no data value is shared between a descendant labeled c and an a-child of a b-ancestor. The automaton behaves as follows.

1. It creates one thread in state q for every data value in the subtree, using $\mathsf{univ}(q)$.
2. q tests that if the data value of the register is reachable by $\downarrow_* [c]$, using a test $\langle \mu \rangle^{=}$. If it is, it changes to state p.
3. p moves up to the root, and each time it finds a b, it tests that the currently stored data value cannot be reached by $\downarrow[a]$. This is done with a test $\overline{\langle \mu \rangle^{=}}$.

Using these ideas, we show a more rigorous construction of the BUDA automaton from a formula η.

Proof of Proposition 7.25. We denote by $\mathsf{nsub}(\eta)$ the node subformulas of η, by $\mathsf{psub}(\eta)$ the path subformulas of η and by $\mathsf{nsub}^{\neg}(\eta)$ the closure of $\mathsf{nsub}(\eta)$ under simple negations, i.e. $\mathsf{nsub}(\eta) \subseteq \mathsf{nsub}^{\neg}(\eta)$ and for $\varphi \in \mathsf{nsub}(\eta)$ then either it is of the form $\varphi = \neg\psi$ or $\neg\varphi$ is in $\mathsf{nsub}^{\neg}(\eta)$ as well.

Given a node expression $\eta \in \mathsf{regXPath}(\mathfrak{V}, =)$ we construct a BUDA $\mathcal{A}$ that tests whether *for all the leaves of a tree, η holds*. Note that this is without any loss of generality, since it is then easy to test any formula η at the *root* with $\langle \uparrow^* [\neg\langle\uparrow\rangle \wedge \eta] \rangle$.

Let us consider the alphabet

$$\mathbb{A}_\eta = \{\uparrow, \downarrow, [\varphi] \mid \varphi \in \mathsf{nsub}(\eta)\}$$

Every path expression $\alpha \in \mathsf{psub}(\eta)$ can then be interpreted as a regular expression over $\mathbb{A}_\eta$, and every word $w \in \mathbb{A}_\eta^*$ either is satisfied or not at a node x of a data tree $\mathbf{t}$ with the expected semantics. We call $w \in \mathbb{A}_\eta^*$ 'looping' iff it contains as many $\downarrow$ as $\uparrow$ letters. Therefore there exists a monoid $\mathcal{M}$ and a morphism g such that for every $\alpha \in \mathsf{psub}(\eta)$ there is a $S_\alpha \subseteq \mathcal{M}$ where for $w \in \mathbb{A}_\eta^*$ is recognized by α iff $g(w) \in S_\alpha$. Let us call ε to the identity of $\mathcal{M}$, and let us write ν, ν' to denote elements of $\mathcal{M}$.

We define $\mathcal{A} \in \mathsf{BUDA}$ as the tuple

$$\mathcal{A} := \langle \mathbb{A}, \mathbb{B}, Q, (\!|\eta|\!), \delta_\epsilon, \delta_{up}, \mathcal{S}, h \rangle.$$

The internal alphabet of $\mathcal{A}$ is $\mathbb{B} = \wp_{<\infty}(\mathcal{M})$. Intuitively, a node x is labeled with $b \in \mathbb{B}$ if it verifies the following property:

$\nu \in b$ iff there is a looping $w \in \mathbb{A}_\eta^*$ verified at the root of $\mathbf{t}|_x$ with $g(w) = \nu$.

The set of states Q contains the following symbols:

$$Q := \{(\!|\varphi|\!), (\!|\alpha|\!)^{\circledast}_{\nu}, (\!|\alpha, \beta|\!)^{\odot}_{\nu,\nu'} \mid \varphi \in \mathsf{nsub}^{\neg}(\eta), \alpha, \beta \in \mathsf{psub}(\eta) \\ \nu, \nu' \in \mathcal{M}, \circledast \in \{=, \neq, \neg=, \neg\neq\}, \odot \in \{\neg=, \neg\neq\}\} .$$

The idea is that a state $(\!|\varphi|\!)$ verifies that the formula $\varphi \in \mathsf{nsub}^{\neg}(\eta)$ holds at the current node. A state $(\!|\alpha|\!)^{=}_{\nu}$ (resp. $(\!|\alpha|\!)^{\neq}_{\nu}$) verifies that there is a traversal $w \in \mathbb{A}_\eta^+$ starting at the current node and ending at some node with the same (resp. different) data value as the one in the register. Moreover, w must verify that $\nu \cdot g(w) \in S_\alpha$. The meaning of the remaining states will become clear later.

We now describe the semigroup and morphism of the automaton $\mathcal{S}$ and $h : (\mathbb{A} \times \mathbb{B})^+ \to \mathcal{S}$. We will constantly need to be able to test whether a child or some specific descendant of the current node contains a certain internal label b for some $b \in \mathbb{B}$. This will be taken care of by $\mathcal{S}$ and h.

(1) For every $\nu \in \mathcal{M}$ and $\alpha \in \mathsf{psub}(\eta)$ the set of words $(a_1, b_1), \ldots, (a_n, b_n)$ of $(\mathbb{A} \times \mathbb{B})^+$ such that there are $\nu^1 \in b_1, \ldots, \nu^n \in b_n$ with

$$\nu \cdot (\nu^1 \cdot g(\downarrow) \cdot \nu^2 \cdot g(\downarrow) \cdot \cdots \cdot \nu^{n-1} \cdot g(\downarrow) \cdot \nu^n) \in S_\alpha.$$

is a regular language.

(2) For every $\nu \in \mathcal{M}$ the set of words $(a_1, b_1)(a_2, b_2)$ with $\nu \in b_2$ is also a regular language.

Hence there is a semigroup $\mathcal{S}$ and a morphism h such that its elements distinguishes between the properties above: For every $\nu \in \mathcal{M}$ and $\alpha \in \mathsf{psub}(\eta)$ there is a subset $\mathit{down}(\nu, \alpha) \subseteq \mathcal{S}$ recognizing (1) and for every $\nu \in \mathcal{M}$ there is a subset $\mathit{next}(\nu) \subseteq \mathcal{S}$ recognizing (2).

It now remains to set the transition functions in order to ensure all the properties stated above. In the sequel we will use the notation $\langle\mu\rangle$ as a shortcut for $\langle\mu\rangle^{=} \vee \langle\mu\rangle^{\neq}$ and $\overline{\langle\mu\rangle}$ as a shortcut for the negation of $\langle\mu\rangle$.

We first show how to ensure that the internal labels are set in the appropriate way.

First we need to ensure that everywhere in the tree, if $b \in \mathbb{B}$ holds in the node and $\nu \in b$ then there is a loop in the corresponding subtree evaluating to ν. For this the automata checks whether one of the following case holds:

- $\nu = \varepsilon$,
- $\nu = g([\varphi])\cdot\nu'$ for $\varphi \in \mathsf{nsub}(\eta)$ and $\nu' \in b$ and φ holds at the current node. The last condition is checked by starting a new thread in state $(\!|\varphi|\!)$,
- $\nu = \nu'\cdot g([\varphi])$ for $\varphi \in \mathsf{nsub}(\eta)$ and $\nu' \in b$ and φ holds at the current node. The last condition is checked by starting a new thread in state $(\!|\varphi|\!)$,
- $\nu = g(\downarrow)\cdot\nu'\cdot g(\uparrow)$ and there is a child containing ν' in its $\mathbb{B}$-label. The last condition is checked using the test $\langle\mu\rangle$ for some $\mu \in next(\nu')$.

We also need to enforce the complement: whenever there is a loop in the corresponding subtree evaluating to ν then $\nu \in b$. For this the automata checks whether one of the following case holds:

- $\nu \neq \varepsilon$,
- for every $\nu' \in \mathcal{M}$ and $\varphi \in \mathsf{nsub}(\eta)$ such that $\nu = g([\varphi])\cdot\nu'$ then either $\nu' \notin b$ or φ does not hold at the current node. The last condition is checked by starting a new thread in state $(\!|\overline{\varphi}|\!)$, where $\overline{\varphi}$ is the negation of φ,
- for every $\nu' \in \mathcal{M}$ and $\varphi \in \mathsf{nsub}(\eta)$ such that $\nu = \nu'\cdot g([\varphi])$ then either $\nu' \notin b$ or φ does not hold at the current node. The last condition is checked by starting a new thread in state $(\!|\overline{\varphi}|\!)$, where $\overline{\varphi}$ is the negation of φ,
- for every $\nu' \in \mathcal{M}$ such that $\nu = g(\downarrow)\cdot\nu'\cdot g(\uparrow)$ we test that there is no child that contains ν' in its $\mathbb{B}$-label. The last condition is checked using the tests $\overline{\langle\mu\rangle}$ for the appropriate $\mu \in \mathcal{S}$.

We now explain how the automata can enforce that the states have the intended meanings. In this description we assume that α and β are in $\mathsf{psub}(\eta)$.

Positive tests.

- Any positive boolean test is immediate to translate using $\wedge$ and $\vee$ in the transition formulæ.

- If the automaton is in a state $(\!|\langle \alpha = \beta \rangle|\!)$. This means that it has to perform a test $\langle \alpha = \beta \rangle$. In this case $\mathcal{A}$ guesses a data value, stores it in its register using guess and continues the execution with both states $(\!|\alpha|\!)^{=}_{\varepsilon}$ and $(\!|\beta|\!)^{=}_{\varepsilon}$, that will test if there exist two nodes accessible by α and β respectively such that both carry the data value of the register.

- Equivalently, if $\mathcal{A}$ is in a state $(\!|\langle \alpha \neq \beta \rangle|\!)$, it guesses a data value, stores it in its register and continues the execution with both states $(\!|\alpha|\!)^{=}_{\varepsilon}$ and $(\!|\beta|\!)^{\neq}_{\varepsilon}$, which are responsible of testing that there is a α path leading to a node with the same data value as the one in the register and a β path leading to a node with a different data value.

- If $\mathcal{A}$ is in state $(\!|\alpha|\!)^{=}_{\nu}$ (resp. $(\!|\beta|\!)^{\neq}_{\nu}$), it chooses nondeterministically to perform one of the following actions.

  - It checks that there is $\nu' \in \mathbf{b}(x)$ such that $\nu{\cdot}\nu' \in S_\alpha$ and that the current node has the appropriate data value using eq (resp. $\overline{\mathsf{eq}}$).
  - It checks that for some $\mu \in \mathit{down}(\nu, \alpha)$ the required data value is already in the making the test $\langle \mu \rangle^{=}$ (resp $\langle \mu \rangle^{\neq}$).
  - It moves up and switches state to $(\!|\alpha|\!)^{=}_{\nu \cdot g(\uparrow)}$ (resp. $(\!|\alpha|\!)^{\neq}_{\nu \cdot g(\uparrow)}$).

Negative tests. Finally, the states $(\!|\ ,\ |\!)^{\neg=}_{\mu,\mu'}$, $(\!|\ ,\ |\!)^{\neg\neq}_{\mu,\mu'}$, $(\!|\ |\!)^{\neg=}_{\mu}$ and $(\!|\ |\!)^{\neg\neq}_{\mu}$ are used to test the negation of the conditions just described. However, it is not immediate that this can be achieved from the previous tests, since the BUDA class is not closed under complementation.

First, let us see how to code a test of the form $\neg\langle \alpha = \beta \rangle$.

- If $\mathcal{A}$ is in a state $(\!|\neg\langle \alpha = \beta \rangle|\!)$, it continues the execution with state $(\!|\alpha, \beta|\!)^{\neg=}_{\varepsilon}$, where ε is the identity of $\mathcal{M}$.

- The state $(\!|\alpha, \beta|\!)^{\neg=}_{\nu}$ tests that it is not true that there are two nodes with the same data value reachable by α and β by some traversals $w, w' \in \mathbb{A}^*_\eta$ such that $\nu{\cdot}g(w) \in S_\alpha$ and $\nu{\cdot}g(w') \in S_\beta$. To attain this, it performs all the following actions using alternation:

  - If the test $\overline{\mathsf{root}}$ succeeds, then it moves up, and updates its state to $(\!|\alpha, \beta|\!)^{\neg=}_{\nu \cdot g(\uparrow)}$.
  - For every $\mu \in \mathit{down}(\nu, \alpha)$ it performs a univ operation to a state that tests $\langle \mu \rangle^{=}$. If the test succeeds it changes to state $(\!|\beta|\!)^{\neg=}_{\nu}$.
  - For every $\mu \in \mathit{down}(\nu, \beta)$ it performs a univ operation to a state that tests $\langle \mu \rangle^{=}$. If the test succeeds it changes to state $(\!|\alpha|\!)^{\neg=}_{\nu}$.

- A state $(\!|\alpha|\!)^{\neg=}_{\nu}$ tests that it is not true that there is a node having the same data value as the one currently in the register, reachable by some traversals

$w \in \mathbb{A}_\eta^*$ such that $\nu{\cdot}g(w) \in S_\alpha$. To attain this, it performs all the following actions using alternation:

- moves up (if the test $\overline{\mathsf{root}}$ succeeds), and updates its state to $(\!|\alpha|\!)^{\neg=}_{\nu\cdot g(\uparrow)}$.
- tests for every $\mu \in \mathit{down}(\nu, \alpha)$ that $\overline{\langle\mu\rangle^=}$ holds.

There is a small technical problem in the description above. In order to simulate correctly the test $\neg\langle\alpha = \beta\rangle$, it is important that the run of the automata is done using the appropriate sequencing:

Now let us turn to the case of $\neg\langle\alpha \neq \beta\rangle$.

- At state $(\!|\neg\langle\alpha \neq \beta\rangle|\!)$, $\mathcal{A}$ continues with state $(\!|\alpha, \beta|\!)^{\neg\neq}_\varepsilon$, which behaves as described next.
- The state $(\!|\alpha, \beta|\!)^{\neg\neq}_\nu$ performs all the following actions using alternation:
  - If the test $\overline{\mathsf{root}}$ succeeds, then it moves up, and updates it state to $(\!|\alpha, \beta|\!)^{\neg\neq}_{\nu\cdot g(\uparrow)}$.
  - For every $\mu \in \mathit{down}(\nu, \alpha)$ it performs a univ operation to a state that tests $\langle\mu\rangle^=$. If the test succeeds it changes to state $(\!|\beta|\!)^{\neg\neq}_\nu$.
  - For every $\mu \in \mathit{down}(\nu, \beta)$ it performs a univ operation to a state that tests $\langle\mu\rangle^=$. If the test succeeds it changes to state $(\!|\alpha|\!)^{\neg\neq}_\nu$.
- Finally, a state $(\!|\alpha|\!)^{\neg\neq}_\nu$ performs all the following actions.
  - Moves up (if the test $\overline{\mathsf{root}}$ succeeds), updating its state to $(\!|\alpha|\!)^{\neg\neq}_{g(\uparrow)\cdot\nu}$.
  - For every $\mu \in \mathit{down}(\nu, \beta)$ it checks $\overline{\langle\mu\rangle^\neq}$.

We then have that if the state $(\!|\eta|\!)$ is verified at every leaf then η holds, and that every tree satisfying η must have an accepting run of $\mathcal{A}$. We conclude that the BUDA class effectively captures all formulæ of $\mathsf{regXPath}(\mathfrak{V}, =)$. □

Hence, by Proposition 7.25 together with the fact that satisfiability of BUDA is decidable (Theorem 7.5), we obtain that Theorem 7.1 holds: satisfiability of vertical XPath is decidable.

7.4.1 XML versus data trees

As we have seen in previous chapters, the decidability of SAT-$\mathsf{XPath}(\mathfrak{V}, =)$ on data trees entails the decidability of SAT-$\mathsf{attrXPath}(\mathfrak{V}, =)$ on XML documents. The way to transfer this result is by the same coding as shown in Section 5.4.3 for downward XPath, and can obviously be done in vertical XPath since it is an extension of the downward fragment.

7.5 Discussion

Our automata model BUDA is a decidable class of automata that allows to make complex data tests that navigate the tree in both upward or downward directions. This automata model is powerful enough to code node expressions of $\mathsf{regXPath}(\mathfrak{V}, =)$. Therefore, as node expressions of $\mathsf{regXPath}(\mathfrak{V}, =)$ are closed under negation, we have shown decidability of the emptiness, inclusion and equivalence problems for node expressions of $\mathsf{regXPath}(\mathfrak{V}, =)$.

Our automata model BUDA allows to make complex data tests that navigate the tree in both upward or downward directions. However, it cannot capture vertical XPath in all generality. This is because it cannot make nested tests. However, we could extend or modify the automaton to be able to label an element each time it performs an *up* transition, and we can extend the monoid to also access this guessed letters. This would be a decidable extension that can effectively capture any formula of vertical XPath.

Finally, let us remark that this automata model relies heavily on the fact that the data tree is unranked. By the coding techniques already used in Chapter 3 we can code the run of an incrementing counter automaton along a branch, and it is hence not difficult to see that we can force the tree to have a node with many (like the Ackermann function applied to the size of the formula) children.

8. CONCLUDING REMARKS

This work had as objective the development of techniques and decidable formalisms to work with data values. We have seen several automata models that are decidable over data trees, and one over data words. We introduced formalisms that can express different kind of properties, and are on the limit of decidability. We make some comments concerning the relation between these models of automata and the logics we have treated.

Automata Notice that DD automata and ATRA(guess, spread) are incomparable in expressive power. Indeed, DD automata can capture downward XPath formulæ like $\neg\langle\downarrow_*[a] = \downarrow_*[b]\rangle$, that cannot be expressed by ATRA(guess, spread), as already argued. Moreover, DD automata can express, *e.g.*, that there are three different data values that can be accessed by $\downarrow_+[a]$—something that cannot be expressed by ATRA(guess, spread). In turn, ATRA(guess, spread) can express unary key constraints, or any regular tree language on the finite labeling, whereas DD automata cannot express key constraints and the properties that can express on the siblinghoods is very limited. In effect, DD automata cannot express that the tree is ranked: *i.e.*, given k, that all nodes have not more than k children.

The BUDA and ATRA(guess, spread) automata are also incomparable, it suffices to note that the former can express a unary inclusion dependency constraint but not a unary key constraint, while the latter can express a unary key constraint but not a unary inclusion dependency constraint.

Unfortunately, if we add the possibility to test for unary key constraints to the BUDA automata it becomes undecidable. Likewise, if we add the possibility of testing unary inclusion dependency constraints to ATRA(guess, spread) (or to ARA(guess, spread) for the matter) it becomes undecidable. There is hence no hope of having a decidable automata model containing the behavior of BUDA and ATRA(guess, spread) which is closed under intersection. It can further be shown that even DD automata with (unary) keys constraints and inclusion dependencies is undecidable.

Finally, BUDA and DD automata are also incomparable. DD automata cannot express unary inclusion dependency constraints, and BUDA automata cannot express that there are three different data values that can be accessed $\downarrow_+[a]$. However, we argue that it is possible to extend the BUDA class to allow this kind of tests—or any test that can be performed by DD automata—while preserving the decidability of the emptiness problem.

$\downarrow$	$\downarrow_+$	$\uparrow$	$\uparrow^+$	$\rightarrow$	$\rightarrow^+$	$\leftarrow$	${}^+\leftarrow$	**Complexity**	**Details**
•								PSPACE-complete	5.49
	•							EXPTIME-complete	5.45, 5.46
•	•							EXPTIME-complete	5.45, 5.46
•	•			•	•			Decidable, NPR	6.9
•	•	•	•					Decidable, NPR	7.1
			•					Decidable, NPR	4.5
	•		•					Decidable, NPR	4.5, 7.1
	•		•	•				Undecidable	4.7
					•		•	Undecidable	4.7
•		•			•			Undecidable	4.8

Fig. 8.1: Summary of main results on XPath with data values. NPR stands for a non-primitive recursive lower bound.

Logics On XPath, we showed decidability of the downward, forward, and vertical fragments, thus settling some open questions. We have now a clearer landscape of the decidability status of XPath according to the set of axes that it uses. The main results are summarized in Figure 8.1. Compared with the fragments treated by Benedikt et al. (2008) and Geerts and Fan (2005), our work addresses the satisfiability problem of XPath in the presence of recursive axes, data tests, and negation (in contrast to positive fragments), in the absence of DTDs. Also, in the presence of DTDs (or regular languages) we obtain that the forward and downward XPath fragments are decidable with a non-primitive recursive lower bound, and that vertical XPath is undecidable.

We remark that although we showed that each of the aforesaid fragments is decidable, this does not mean that we can *combine* these results. Our results do not yield the possibility to test the satisfiability of a boolean combination of formulæ where each of them belong to one of these fragments. Indeed, consider the following problem:

$$\text{Given } \eta_{\mathfrak{v}} \in \mathsf{XPath}(\mathfrak{V}, =) \text{ and } \eta_{\mathfrak{f}} \in \mathsf{XPath}(\mathfrak{F}, =), \text{ is } \eta_{\mathfrak{v}} \wedge \eta_{\mathfrak{f}} \text{ satisfiable?}$$

This is an undecidable problem, since $\eta_{\mathfrak{f}}$ can test that the tree is word-like, and $\eta_{\mathfrak{v}}$ can easily code an accepting run of a Minsky automaton over linear trees.

Future work

All our investigation on XPath focuses on the satisfiability problem. As we have seen in Section 4.2, the problem of query equivalence and inclusion reduce to this problem. But this concerns only queries of *node* expressions. Whether the techniques we developed can be adapted to show similar decidability results on

the problems of query equivalence and query containment of *path* expressions is a moot point.

Question 8.1. What is the decidability status of the inclusion and equivalence problems of path expressions of downward, forward and vertical XPath?

The fragments treated here are all navigational fragments of XPath 1.0. However XPath 2.0 has many rich features that we do not consider. We leave open the question of whether the results of this thesis can be extended to incorporate some of the distinctive features of XPath 2.0.

Another relevant issue is to try to add more domain specific relations to our models of automata. In that direction, we discussed that a linear order can be added to $\mathsf{ARA}(\mathsf{guess}, \mathsf{spread})$—or $\mathsf{LTL}^{\downarrow}_{\mathsf{nnf}}(\mathfrak{F}, \exists^{\downarrow}_{\geq}, \forall^{\downarrow}_{\leq})$—without losing decidability. Moreover, with the same kind of analysis it can be further extended to $\mathsf{ATRA}(\mathsf{guess}, \mathsf{spread})$. It would be interesting to explore other relations. For example if the data domain is the set of strings $\mathbb{D} = \mathbb{A}^*$, we may want to have the substring, prefix, and suffix relations; and if it is numerical $\mathbb{D} = \mathbb{N}$ we may want to use some arithmetic. A possible future work would be hence to extend the automata treated here with some relations or functions of this kind.

BIBLIOGRAPHY

Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 313–321, 1996. (Cited on page 7.)

Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160 (1–2):109–127, 2000. (Cited on page 7.)

Parosh Aziz Abdulla, Johann Deneux, Joël Ouaknine, and James Worrell. Decidability and complexity results for timed automata via channel machines. In *ICALP*, pages 1089–1101, 2005. doi:10.1007/11523468_88. (Cited on page 38.)

Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. doi:10.1016/0304-3975(94)90010-8. (Cited on pages 8 and 37.)

Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994. doi:10.1145/174644.174651. (Cited on page 6.)

Carlos Areces, Patrick Blackburn, and Maarten Marx. A road-map on complexity for hybrid logics. In *EACSL Annual Conference on Computer Science Logic (CSL'99)*, number 1683 in Lecture Notes in Computer Science, pages 307–321. Springer, 1999. (Cited on page 6.)

Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore D. Zuck. Parameterized verification with automatically computed inductive assertions. In *International Conference on Computer Aided Verification (CAV'01)*, pages 221–234. Springer, 2001.

Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. The emptiness problem for tree automata with global constraints. In *Annual IEEE Symposium on Logic in Computer Science (LICS'10)*. IEEE Computer Society Press, 2010. http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/BCGJV-lics10.pdf. (Cited on page 66.)

Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM*, 55(2):1–79, 2008. doi:10.1145/1346330.1346333. (Cited on pages 65, 69, 70, 71, 111, 112, 139, and 170.)

Michael Benedikt and Christoph Koch. XPath leashed. *ACM Computing Surveys*, 41(1), 2008. doi:10.1145/1456650.1456653. (Cited on page 139.)

Henrik Björklund and Mikołaj Bojańczyk. Bounded depth data trees. In *International Colloquium on Automata, Languages and Programming (ICALP'07)*, volume 4596 of *Lecture Notes in Computer Science*, pages 862–874. Springer, 2007. doi:10.1007/978-3-540-73420-8_74. (Cited on page 64.)

Henrik Björklund, Wim Martens, and Thomas Schwentick. Optimizing conjunctive queries over trees using schema information. In *International Symposium on Mathematical Foundations of Computer Science (MFCS'08)*, volume 5162 of *Lecture Notes in Computer Science*, pages 132–143. Springer, 2008. doi:10.1007/978-3-540-85238-4_10. (Cited on page 66.)

Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411(4-5):702–715, 2010. doi:10.1016/j.tcs.2009.10.009. (Cited on pages 23 and 64.)

Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal logic*. Cambridge University Press, 2001. ISBN 0-521-80200-8. (Cited on pages 7 and 70.)

Bruno Bogaert and Sophie Tison. Equality and disequality constraints on direct subterms in tree automata. In *International Symposium on Theoretical Aspects of Computer Science (STACS'92)*, pages 161–171. Springer, 1992. ISBN 3-540-55210-3. (Cited on page 66.)

Mikołaj Bojańczyk and Sławomir Lasota. An extension of data automata that captures XPath. In *Annual IEEE Symposium on Logic in Computer Science (LICS '10)*, 2010. (Cited on page 64.)

Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. *Journal of the ACM*, 56(3):1–48, 2009. doi:10.1145/1516512.1516515. (Cited on pages 60, 65, and 140.)

Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 7–16. IEEE Computer Society Press, 2006. doi:10.1109/LICS.2006.51. (Cited on pages 22, 23, and 64.)

Ahmed Bouajjani, Peter Habermehl, Yan Jurski, and Mihaela Sighireanu. Rewriting systems with data. In *International Symposium on Fundamentals of Computation Theory (FCT'07)*, pages 1–22, 2007. doi:10.1007/978-3-540-74240-1_1. (Cited on page 24.)

Patricia Bouyer, Antoine Petit, and Denis Thérien. An algebraic approach to data languages and timed languages. *Inf. Comput.*, 182(2):137–162, 2003. doi:10.1016/S0890-5401(03)00038-5. (Cited on page 22.)

Balder ten Cate. The expressivity of XPath with transitive closure. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'06)*, pages 328–337. ACM Press, 2006. doi:10.1145/1142351.1142398. (Cited on page 61.)

Balder ten Cate and Luc Segoufin. XPath, transitive closure logic, and nested tree walking automata. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'08)*, pages 251–260. ACM Press, 2008. doi:10.1145/1376916.1376952. (Cited on page 61.)

Pierre Chambart and Philippe Schnoebelen. The ordinal recursive complexity of lossy channel systems. In *Annual IEEE Symposium on Logic in Computer Science (LICS'08)*, pages 205–216. IEEE Computer Society Press, 2008. doi:10.1109/LICS.2008.47. (Cited on page 38.)

James Clark and Steve DeRose. XML path language (XPath). Website, 1999. W3C Recommendation. http://www.w3.org/TR/xpath. (Cited on page 60.)

Claire David. Complexity of data tree patterns over XML documents. In *International Symposium on Mathematical Foundations of Computer Science (MFCS'08)*, volume 5162 of *Lecture Notes in Computer Science*, pages 278–289. Springer, 2008. doi:10.1007/978-3-540-85238-4_22. (Cited on page 66.)

Claire David, Leonid Libkin, and Tony Tan. Data trees with set and linear constraints. Unpublished manuscript, 2010.

Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*, 10(3), 2009. doi:10.1145/1507244.1507246. (Cited on pages v, vii, 5, 6, 7, 15, 19, 20, 21, 22, 24, 25, 26, 28, 30, 38, 41, 43, 49, 51, 52, 54, 64, and 123.)

Stéphane Demri, Ranko Lazić, and David Nowak. On the freeze quantifier in constraint LTL: Decidability and complexity. In *International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 113–121. IEEE Computer Society Press, 2005. doi:10.1016/j.ic.2006.08.003. (Cited on page 38.)

Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *International Conference on Database Theory (ICDT '09)*, pages 252–267. ACM Press, 2009. doi:10.1145/1514894.1514924. (Cited on page 24.)

Leonard E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *The American Journal of Mathematics*, 35(4):413–422, 1913. doi:10.2307/2370405. (Cited on page 10.)

Diego Figueira. Satisfiability of downward XPath with data equality tests. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*

(PODS'09), pages 197–206. ACM Press, 2009. doi:10.1145/1559795.1559827. (Cited on page 70.)

Diego Figueira. Forward-XPath and extended register automata on data-trees. In *International Conference on Database Theory (ICDT'10)*. ACM Press, 2010. doi:10.1145/1804669.1804699. (Cited on pages 21 and 77.)

Diego Figueira, Santiago Figueira, Sylvain Schmitz, and Philippe Schnoebelen. Ackermann and primitive-recursive bounds with Dickson's lemma. *ArXiv e-prints*, 2010a. http://arxiv.org/abs/1007.2989. (Cited on page 38.)

Diego Figueira, Piotr Hofman, and Sławomir Lasota. Relating timed and register automata. In *International Workshop on Expressiveness in Concurrency (EXPRESS'10)*, 2010b. (Cited on pages 22 and 37.)

Diego Figueira and Luc Segoufin. Future-looking logics on data words and trees. In *International Symposium on Mathematical Foundations of Computer Science (MFCS'09)*, volume 5734 of *LNCS*, pages 331–343. Springer, 2009. doi:10.1007/978-3-642-03816-7_29. (Cited on pages 22, 43, 67, and 77.)

Diego Figueira and Luc Segoufin. Bottom-up automata on data trees and vertical XPath. In *International Symposium on Theoretical Aspects of Computer Science (STACS'11)*. Springer, 2011. (Cited on page 139.)

Emmanuel Filiot, Jean-Marc Talbot, and Sophie Tison. Satisfiability of a spatial logic with tree variables. In Jacques Duparc and Thomas A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2007. doi:10.1007/978-3-540-74915-8_13. (Cited on page 66.)

Emmanuel Filiot, Jean-Marc Talbot, and Sophie Tison. Tree automata with global constraints. In Masami Ito and Masafumi Toyama, editors, *Developments in Language Theory*, volume 5257 of *Lecture Notes in Computer Science*, pages 314–326. Springer, 2008. doi:10.1007/978-3-540-85780-8_25. (Cited on page 66.)

Alain Finkel. A generalization of the procedure of Karp and Miller to well structured transition system. In Thomas Ottmann, editor, *Proceedings of the 14th International Colloquium on Automata, Languages and Programming (ICALP'87)*, volume 267 of *Lecture Notes in Computer Science*, pages 499–508. Springer-Verlag, 1987. http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/F-icalp87.pdf. (Cited on page 7.)

Alain Finkel. Reduction and covering of infinite reachability trees. *Information and Computation*, 89(2):144–179, 1990. (Cited on page 7.)

Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001. doi:10.1016/S0304-3975(00)00102-X. (Cited on pages 7, 10, 11, and 12.)

Massimo Franceschet, Maarten de Rijke, and Bernd-Holger Schlingloff. Hybrid logics on linear structures: Expressivity and complexity. In *International Symposium on Temporal Representation and Reasoning (TIME'03)*, pages 166–173. IEEE Computer Society Press, 2003. doi:10.1.1.125.8491. (Cited on page 6.)

M.R. Garey and D.S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. A series of books in mathematical sciences. W. H. Freeman, 1979. ISBN 9780716710448. http://books.google.com/books?id=E2VgSQAACAAJ. (Cited on page 112.)

Floris Geerts and Wenfei Fan. Satisfiability of XPath queries with sibling axes. In *International Symposium on Database Programming Languages (DBPL'05)*, volume 3774 of *Lecture Notes in Computer Science*, pages 122–137. Springer, 2005. doi:10.1007/11601524_8. (Cited on pages 60, 65, and 170.)

Valentin Goranko. Hierarchies of modal and temporal logics with reference pointers. *Journal of Logic, Language and Information*, 5(1):1–24, 1996. doi:10.1.1.36.4479. (Cited on page 6.)

Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005. doi:10.1145/1071610.1071614. (Cited on page 60.)

Eyal Harel, Orna Lichtenstein, and Amir Pnueli. Explicit clock temporal logic. In *Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 400–413. IEEE Computer Society Press, 1990. doi:10.1109/LICS.1990.113765. (Cited on page 6.)

Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society (3)*, 2(7):326–336, 1952. doi:10.1112/plms/s3-2.1.326. (Cited on page 10.)

Petr Jančar. A note on well quasi-orderings for powersets. *Information Processing Letters*, 72(5-6):155–160, 1999. doi:10.1016/S0020-0190(99)00149-0. (Cited on pages 13 and 14.)

Marcin Jurdziński and Ranko Lazić. Alternation-free modal mu-calculus for data trees. In *Annual IEEE Symposium on Logic in Computer Science (LICS'07)*, pages 131–140. IEEE Computer Society Press, 2007. doi:10.1109/LICS.2007.11. (Cited on page 137.)

Marcin Jurdziński and Ranko Lazić. Alternating automata on data trees and XPath satisfiability. *Computing Research Repository (CoRR)*, 2008. arXiv:0805.0330. (Cited on pages v, vii, 5, 6, 7, 64, 65, 123, 125, 126, 128, 133, 137, and 140.)

Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994. doi:10.1016/0304-3975(94)90242-9. (Cited on page 22.)

Michael Kaminski and Tony Tan. Tree automata over infinite alphabets. In *Pillars of Computer Science*, volume 4800 of *Lecture Notes in Computer Science*, pages 386–423. Springer, 2008. doi:10.1007/978-3-540-78127-1_21. (Cited on page 64.)

Ahmet Kara, Thomas Schwentick, and Thomas Zeume. Temporal logics on words with multiple data values. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'10)*, 2010. (Cited on page 24.)

Olga Kouchnarenko and Philippe Schnoebelen. A model for recursive-parallel programs. In B. Steffen and Didier Caucal, editors, *Proceedings of the 1st International Workshop on Verification of Infinite State Systems (INFINITY'96)*, volume 5 of *Electronic Notes in Theoretical Computer Science*, page 30. Elsevier Science Publishers, 1997. doi:10.1016/S1571-0661(05)82512-5. (Cited on page 7.)

Joseph B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960. doi:10.2307/F1993287. (Cited on page 36.)

Richard E. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal on computing*, 6(3):467–480, 1977. (Cited on page 70.)

Sławomir Lasota and Igor Walukiewicz. Alternating timed automata. *ACM Transactions on Computational Logic*, 9(2), 2008. doi:10.1145/1342991.1342994. (Cited on page 8.)

Alexei Lisitsa and Igor Potapov. Temporal logic with predicate lambda-abstraction. In *International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 147–155, 2005. doi:10.1.1.59.6646. (Cited on page 52.)

M.H. Löb and S.S. Wainer. Hierarchies of number theoretic functions, I. *Archiv für Mathematische Logik und Grundlagenforschung*, 13:39–51, 1970. doi:10.1007/BF01967649. (Cited on page 38.)

Christof Löding and Karianto Wong. On nondeterministic unranked tree automata with sibling constraints. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'09)*, volume 4 of *LIPIcs*, pages 311–322, 2009. doi:10.4230/LIPIcs.FSTTCS.2009.2328. (Cited on page 66.)

Denis Lugiez. Counting and equality constraints for multitree automata. In *FOSSACS'03/ETAPS'03: Proceedings of the 6th International conference on Foundations of Software Science and Computation Structures and joint European conference on Theory and practice of software*, pages 328–342. Springer-Verlag, 2003. ISBN 3-540-00897-7. (Cited on page 66.)

Carsten Lutz. Description logics with concrete domains—a survey. In *Advances in Modal Logics Volume 4*. King's College Publications, 2003. (Cited on page 7.)

Amaldev Manuel. Two orders and two variables. In *International Symposium on Mathematical Foundations of Computer Science (MFCS'10)*, Lecture Notes in Computer Science. Springer, 2010. (Cited on page 23.)

Amaldev Manuel and R. Ramanujam. Counting multiplicity over infinite alphabets. In *International Workshop on Reachability Problems (RP'09)*, pages 141–153. Springer, 2009. doi:10.1007/978-3-642-04420-5_14. (Cited on page 23.)

Alberto Marcone. Foundations of bqo theory. *Transactions of the American Mathematical Society*, 345:641–660, 1994. doi:10.1090/S0002-9947-1994-1219735-8. (Cited on page 14.)

Maarten Marx. XPath with conditional axis relations. In *International Conference on Extending Database Technology (EDBT'04)*, volume 2992 of *Lecture Notes in Computer Science*, pages 477–494. Springer, 2004. doi:10.1007/b95855. (Cited on pages 70 and 120.)

Maarten Marx. First order paths in ordered trees. In *International Conference on Database Theory (ICDT'05)*, pages 114–128. Springer, 2005. doi:10.1.1.96.6923. (Cited on page 65.)

Richard Mayr. Undecidable problems in unreliable computations. *Theoretical Computer Science*, 297(1-3):337–354, 2003. doi:10.1016/S0304-3975(02)00646-1. (Cited on page 15.)

Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967. ISBN 0-13-165563-9. (Cited on page 15.)

Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5 (3):403–435, 2004. doi:10.1145/1013560.1013562. (Cited on page 22.)

Joël Ouaknine and James Worrell. On the language inclusion problem for timed automata: Closing a decidability gap. In *Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 54–63. IEEE Computer Society Press, 2004. doi:10.1.1.66.2375. (Cited on page 8.)

Klaus Reinhardt. *Counting as Method, Model and Task in Theoretical Computer Science*. Habilitation thesis, University of Tübingen, 2005. URL http:

//www2-fs.informatik.uni-tuebingen.de/~reinhard/Habil.pdf. (Cited on page 64.)

Philippe Schnoebelen. Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In *International Symposium on Mathematical Foundations of Computer Science (MFCS'10)*, volume 6281 of *Lecture Notes in Computer Science*, pages 616–628. Springer, 2010. doi:10.1007/978-3-642-15155-2_54. (Cited on pages 15 and 38.)

Thomas Schwentick and Thomas Zeume. Two-variable logic with two order relations. In *EACSL Annual Conference on Computer Science Logic (CSL'10)*, 2010. (Cited on page 23.)

INDEX