université de BORDEAUX

LaBRI

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET

INFORMATIQUE

SPÉCIALITÉ : INFORMATIQUE

Par **Sougata BOSE**

On decision problems on word transducers with origin semantics

Sous la direction de : **Anca MUSCHOLL**
Co-directeur : **Gabriele PUPPIS**

Soutenue le **12 Mars 2021** devant un jury composé de

Anca MUSCHOLL	Professeur	Université de Bordeaux	Directrice
Gabriele PUPPIS	Professeur Assistant	University of Udine	Co-Directeur
Paul GASTIN	Professeur	ENS Paris-Saclay	Rapporteur
Emmanuel FILIOT	Chercheur Qualifié	Université libre de Bruxelles	Rapporteur
Thomas COLCOMBET	Directeur de Recherche	Université Paris Diderot	Examinateur
Marc ZEITOUN	Professeur	Université de Bordeaux	Examinateur et Président

Titre : Sur les problèmes de décision concernant les transducteurs de mots avec la sémantique d'origine

Résumé : La sémantique d'origine pour les transducteurs de mots a été introduite par Bojańczyk en 2014 afin d'obtenir une caractérisation indépendante de la machine pour les fonctions mot à mot définies par les transducteurs. Notre objectif principal était d'étudier certains problèmes de décision classiques pour les transducteurs dans la sémantique d'origine, tels que le problème d'inclusion et d'équivalence. Nous avons montré que ces problèmes deviennent décidables dans la sémantique d'origine, même si la version classique est indécidable.

Motivé par l'observation que la sémantique d'origine est plus fine que la sémantique classique, nous avons défini les resynchroniseurs comme un moyen de décrire les distorsions d'origine et d'étudier les problèmes ci-dessus de manière relaxée. Nous avons étendu le modèle des resynchroniseurs rationnels, introduit par Filiot et al. pour les transducteurs unidirectionnels, aux resynchroniseurs réguliers, qui fonctionnent pour des classes de transducteurs plus grandes.

Nous avons étudié les deux variantes du problème d'inclusion relative à une resynchronisation, qui demande si un transducteur est contenu dans un autre jusqu'à une distorsion spécifiée par un resynchroniseur. Nous avons montré que le problème peut être résolu lorsque le resynchroniseur fait partie de l'entrée. Lorsque le resynchroniseur n'est pas spécifié dans l'entrée, nous avons cherché à synthétiser un tel resynchroniseur, chaque fois que cela était possible. Nous appelons cela le problème de synthèse pour les resynchroniseurs et nous montrons qu'il est indécidable en général. Nous avons identifié quelques cas restreints où le problème devient décidable. Nous avons également étudié le problème de resynchronisabilité unidirectionnelle, qui demande si un transducteur bidirectionnel donné est resynchronisable dans un transducteur unidirectionnel, et nous avons montré que ce problème est également décidable.

Mots-clés : Transducteurs de mots, La sémantique d'origine, Resynchroniseurs, Équivalence, Inclusion, Synthèse

Title: On decision problems on word transducers with origin semantics

Abstract: The origin semantics for word transducers was introduced by Bojańczyk in 2014 in order to obtain a machine-independent characterization for word-to-word functions defined by transducers. Our primary goal was to study some classical decision problems for transducers in the origin semantics, such as the containment and the equivalence problem. We showed that these problems become decidable in the origin semantics, even though the classical version is undecidable.

Motivated by the observation that the origin semantics is more fine-grained than classical semantics, we defined resynchronizers as a way to describe distortions of origins, and to study the above problems in a more relaxed way. We extended the model of rational resynchronizers, introduced by Filiot et al. for one-way transducers, to regular resynchronizers, which work for larger classes of transducers.

We studied the two variants of the containment up to resynchronizer problem, which asks if a transducer is contained in another up to a distortion specified by a resynchronizer. We showed that the problem is decidable when the resynchronizer is given as part of the input. When the resynchronizer is not specified in the input, we aimed to synthesize such a resynchronizer, whenever possible. We call this the synthesis problem for resynchronizers and show that it is undecidable in general. We identified some restricted cases when the problem becomes decidable. We also studied the one-way resynchronizability problem, which asks whether a given two-way transducer is resynchronizable in a one-way transducer, and showed that this problem is decidable as well.

Keywords: String transducers, Origin semantics, Resynchronizers, Equivalence, Containment, Synthesis

Laboratoire d'acceuil:

Laboratoire Bordelais de Recherche en Informatique (LaBRI)
Unité mixte de recherche CNRS (UMR 5800)
351, cours de la libération F-33405 Talence cedex,
France

Acknowledgements

I would like to start by thanking my advisors Anca and Gabriele for all the support they have provided over the last three years. Starting from helping me settle in France with the bureaucratic works to constantly helping me during the pandemic times, they have been a constant source of support. They have encouraged me and pushed me to work harder. I have learnt from them the importance of aesthetics and presentation in research by choosing appropriate notations, and trying to make proofs simpler and more elegant. I would also like to thank them for reading through the thesis several times to look for errors. I am extremely grateful to have them as my advisors.

I would also like to thank Vincent, who was involved in many of the works in this thesis and often acted as an informal advisor to me. I am thankful to him for the many advices he gave for preparing talks and all the discussions we had, both academic and otherwise.

A significant part of the work for the thesis happened during my visits to Krishna S. at IIT Bombay. I sincerely thank her for hosting me and for the great discussions we had. I would also like to thank Sparsa, Aneek, Vrunda, Saptarshi and others for making my time both fruitful and fun.

I also thank the reviewers Paul Gastin and Emmanuel Filiot for agreeing to review the thesis and for their nice comments and feedback. I would also like to thank the examiners Marc Zeitoun and Thomas Colcombet who along with Paul and Emmanuel posed some very interesting questions.

I would also like to thank the professors at CMI who introduced me to Automata Theory. In particular, I owe my interest in the subject to Srivathsan who has been supportive in my research both during my time at CMI and afterwards. He has encouraged me every time we have met and I am very grateful to him.

I would like to thank my friends and colleagues in LaBRI for providing a nice work environment. Special thanks go to Olivier, Diego, Nathan and Varun for the discussions on transducers, which helped me understand the subject better. I would also like to thank Marc, Pascal, and Igor and everyone related to the ANR DeLTA, UMI Relax, and IOTTA projects for funding several trips to severals conferences and research visits. I also thank the administrative staff at LaBRI and at the university.

I would also like to thank my friends in Bordeaux and elsewhere for

making my life outside work easier. I will forever have some great memories in Bordeaux thanks to Govind, Soumyajit, Raj, Tidiane, Atilla, Kanka, Mallika, Debam, Karim 1 and 2, Trang, and many others. I also thank Suman, Prantar, Ritam, Anirban, Abhishek, Debraj, Arghya, Rajarshi and others for the online board games sessions and philosophical discussions. I would also like to thank my friends Arpita and Ranadeep for supporting me.

Last but not the least, I thank my parents for all the encouragement and cooperation they provided to pursue my interests. I am forever grateful to them for always being there to listen to me. I am also thankful to my sister who has been a source of inspiration to me. I dedicate this thesis to my parents and my sister.

Résumé de la thèse

De langages aux transducteurs

Les automates constituent l'un des modèles de calcul les plus étudiés en informatique. Un automate lit une lettre en entrée, met à jour son état en utilisant une relation de transition et passe à la lecture de la lettre en entrée suivante. Lorsqu'il atteint la fin de l'entrée, l'automate accepte ou rejette le mot, selon que l'état actuel est acceptant ou non. Le langage reconnu par l'automate est l'ensemble de tous les mots acceptés. La classe des langages reconnus par les automates est appelée classe des langages rationnels et bénéficie de différentes caractérisations. Par exemple, les langages rationnels peuvent être définis par la logique monadique du second ordre, par des expressions rationnelles, par des congruences finies, etc. Même l'ajout de diverses caractéristiques aux automates, comme le non-déterminisme ou le mouvement bidirectionnel de la tête de lecture, ne change pas l'expressivité.

Les transducteurs sur les mots ont été étudiés en informatique très tôt, en même temps que les automates, en les considérant comme des automates à bandes multiples [RS59, Sch61]. Cependant, les travaux initiaux présentaient une vision *statique* des transducteurs comme lisant des mots sur plusieurs bandes d'entrée et définissant une relation k-aire. Une vision *dynamique* des transducteurs, dans laquelle ils transforment entrée en sortie a été présentée par Elgot et Mezei [EM65]. Dans cette vision dynamique, un transducteur traite un mot d'entrée en lisant une lettre à la fois et produit la sortie en écrivant sur une bande de sortie. Ainsi, les transducteurs peuvent être considérés comme une généralisation des machines de Moore et Mealy.

Un transducteur de mots définit une transformation des mots, appelée *transduction*. Les transductions *fonctionnelles*, où chaque mot d'entrée est mis en correspondance avec au plus un mot de sortie, ont été très bien étudiées. Un transducteur déterministe définit toujours une transduction fonctionnelle. Contrairement au cas des automates, l'ajout de non-déterminisme augmente strictement l'expressivité des transducteurs, car un transducteur non-déterministe peut définir des relations, au lieu des fonctions. Le fait de permettre un mouvement bidirectionnel de la tête de lecture rend également les transducteurs plus puissants. Par exemple, en utilisant le mouvement bidirectionnel, un transducteur peut copier une entrée deux fois,

ce qui n'est pas possible sans le mouvement bidirectionnel. Par conséquent, il est particulièrement important d'identifier des classes robustes de transductions, avec différentes caractérisations équivalentes.

Les transductions définies par des transducteurs unidirectionnels sont définissables par des expressions rationnelles sur un produit de monoïdes libres, et sont donc appelées *relations rationnelles*. Cela donne aussi naturellement lieu aux *fonctions rationnelles*, qui sont des fonctions définissables par des transducteurs unidirectionnels. Cette classe a reçu beaucoup d'attention dans les premiers travaux sur les transducteurs [Sch75, Cho78].

Une classe plus intéressante de transductions contient les fonctions définies par des transducteurs bidirectionnels, qui a reçu un regain d'intérêt ces dernières années. Une caractérisation *logique* équivalente de cette classe de fonctions a été donnée dans [EH01] en utilisant la logique monadique du second ordre (MSO) [Cou97]. Plus récemment, un autre modèle des transducteurs unidirectionnels équipé d'un nombre fini des registres a été introduit et on a montré qu'il caractérise la même classe de fonctions [AC11]. L'équivalence logique-machine pour les transductions [AC10, EH01] est une réminiscence des caractérisations de Büchi-Elgot-Traktenbrot pour les automates. Pour cette raison, les fonctions définies par des transducteurs de mots bidirectionnels sont appelées *fonctions régulières*. Des travaux plus récents ont fourni des formalismes bases sur les expressions régulières, appelés *expressions combinatoires régulières* [AFR14, Gas19], et sur les *fonctions de listes régulières* [BDK18]. Pour une étude sur les propriétés logiques et algébriques des transductions de mots, voir [FR16]. Le survey 'enquête récent [MP19b] est centré sur les problèmes de décision pour les fonctions de mots réguliers et le finiment valués.

La semantique d'origine

Une sémantique alternative pour les transducteurs, appelée *sémantique d'origine*, a été introduite dans [Boj14] afin d'obtenir des transducteurs bidirectionnels fonctionnels canoniques. Dans la sémantique d'origine, la sortie est étiquetée avec les positions de l'entrée, appelées origines, qui décrivent où chaque lettre de sortie a été produite. La plupart des transformations d'objets issus d'applications s'accompagnent souvent d'une notion d'origine. Par exemple, dans le cas des transductions de mots, le sous-mot ab de $w = aab$ peut apparaître soit en choisissant le premier a du sous-mot ou le deuxième a. Même si ces choix correspondent à des positions différents dans w, ils pouvent produire le même sous-mot ab. La sémantique d'origine fait la distinction entre ces deux cas.

Dans cette thèse, nous nous intéressons à l'étude du problème d'équivalence et d'inclusion pour les transducteurs dans la sémantique d'origine. L'équivalence sous la sémantique d'origine exige que les trans-

ducteurs ne produisent pas seulement les mêmes sorties pour chaque entrée, mais aussi avec les mêmes origines. Selon la sémantique de l'origine, deux transducteurs peuvent être non équivalents même s'ils calculent la même relation dans la sémantique classique. Par conséquent, la sémantique d'origine est plus "stricte" que la sémantique classique pour les transducteurs.

Récemment, des problèmes de décision dans la sémantique d'origine ont été considérés par Filiot et al. [FJLW16]. En particulier, il y a été de montré que l'équivalence avec d'origine pour les transducteurs unidirectionnels est PSpace-compléte. Ceci est en contraste avec la sémantique classique, où l'équivalence est indécidable [FR68, Gri68]. De plus, la complexité PSpace est le meilleur résultat que l'on puisse espérer, puisque l'équivalence des automates est déjà PSpace-difficile [Koz77].

Resynchronisateurs Dans la sémantique d'origine, deux transducteurs sont équivalents s'ils produisent la même sortie d'une manière synchronisée. Une notion de resynchronisation a été introduite pour les transducteurs unidirectionnels [FJLW16] comme moyen de comparer des transducteurs ayant des origines similaires, mais pas identiques. Cela généralise la notion d'équivalence d'origine et permet d'étudier des problèmes de décision dans la sémantique d'origine d'une manière plus générale.

Contributions et structure de la thèse. Dans le chapitre 1, nous fixons d'abord les notations et les définitions utilisées plus loin dans la thèse.

Dans le chapitre 2, nous étudions la décidabilité de l'équivalence d'origine pour les transducteurs bidirectionnels et à registres. Pour les transducteurs bidirectionnels, nous montrons que le problème l'équivalence avec origine est PSpace-compléte, et qu'il a donc la même complexité que l'équivalence d'automates. Pour les transducteurs à registres, nous fournissons un algorithme ExpSpace pour décider l'équivalence avec origine, qui peut être amélioré en PSpace dans le cas déterministe. Nous identifions également une sous-classe expressivement équivalente de transducteurs à registres déterministes, pour lesquels le problème d'équivalence avec origine est en PTime. Pour les transducteurs à registres avec des updates *copyful*, nous montrons la décidabilité de l'équivalence d'origine en fournissant un algorithme de propagation en arrière.

Dans le chapitre 3, nous généralisons la notion de resynchronisation en introduisant des resynchronisateurs réguliers. Ces resynchronisateurs fonctionnent aussi pour les transducteurs bidirectionnels. Ceux-ci sont définis à l'aide de formules logiques monadiques du second ordre et sont inspirés par les transductions de graphes de Courcelle [Cou97]. L'idée derrière ce modèle de resynchronisateur est d'utiliser une formule logique move avec deux variables libres correspondant aux origines source et cible, interprétées sur le mot d'entrée. Ceci définit un changement d'origine de la source à

la cible pour chaque position de sortie. Nous étudions l'équivalence par rapport d'une resynchronisation comme une généralisation du problème de l'équivalence d'origine et montrons que le problème est décidable. Nous comparons également l'expressivité des resynchronisateurs rationnels et réguliers restreints aux transducteurs unidirectionnels.

Dans le chapitre 4, nous étudions le problème de synthèse des resynchronisateurs. Ce problème demande si deux transducteurs classiquement équivalents peuvent être rendus équivalents du point de vue de l'origine en déformant les origines à l'aide d'un resynchronisateur. Si c'est le cas, alors l'objectif est de synthétiser un tel resynchronisateur. Nous étudions ce problème à la fois pour les resynchronisateurs rationnels introduits par Filiot et al. et pour les resynchronisateurs réguliers que nous avons introduits. Pour les deux classes de resynchronisateurs, le problème est indécidable. Pour les resynchronisateurs rationnels, nous donnons un moyen de synthétiser un resynchronisateur lorsque les transducteurs donnés sont fonctionnels, ou même finiment valués. Pour les resynchronisateurs réguliers, nous donnons un algorithme pour décider et, si possible, synthétiser un resynchronisateur lorsque les transducteurs d'entrée sont non ambigus.

Enfin, nous étudions une variante du problème de définissabilité unidirectionnel dans la sémantique d'origine, dans le chapitre 5 . Le problème de définissabilité unidirectionnel dans la sémantique de l'origine s'avère trivial et est caractérisé par transducteurs qui préserve l'ordre. Une variante plus intéressante de ce problème est le problème de resynchronisabilité unidirectionnelle. Il s'agit de savoir si un transducteur bidirectionnel donné peut être resynchronisé pour devenir équivalent à un transducteur unidirectionnel quelconque. Nous fournissons une propriété basée sur le graphe des paires d'entrée-sortie pour classifier quand c'est le cas. Nous prouvons également une caractérisation équivalente sur le transducteur, ce qui donne un algorithme PSpace pour décider de la resynchronisabilité unidirectionnelle.

Chaque chapitre s'achève par une section de conclusions qui résume les résultats, mentionne les problèmes ouverts connexes et les orientations pour les travaux futurs.

Introduction

From languages to transductions

Finite-state automata are one of the most well studied model of computation that processes input words over a finite alphabet using a finite set of states. An automaton reads a letter of the input, updates its state using a transition relation and moves on to read the next letter of the input to the right. Upon reaching the end of the input, the automaton either accepts or rejects the word, based on whether the current state is accepting or rejecting. The language of an automaton is the set of all words accepted by it. The class of languages recognized by finite-state automata is called the class of *regular languages* and enjoys various different characterizations. For example, regular languages can be defined by monadic second-order logic, rational expressions, finite congruences, etc. Even adding various features, such as non-determinism or two-way movement of the reading head, does not change the expressiveness of finite-state automata.

Finite-state transducers over words were studied in computer science very early, at the same time as finite-state automata, being viewed as multi-tape automata [RS59, Sch61]. However, these works presented a *static* view of transducers as reading words from multiple input tapes and defining a k-ary relation. A *dynamic* view of transducers, where they transform input into output, was presented by Elgot and Mezei [EM65]. This view is motivated by the fact that computers typically process streams of data and transform them between different formats. This view also motivates various applications, for example in natural language processing. In this dynamic view, a finite-state transducer can be seen as processing an input word by reading a letter at a time and producing the output by writing on a output tape, while accessing a finite set of states. Thus, transducers can be seen as a generalization of Moore and Mealy machines.

A word transducer defines a transformation of words, which are called *transductions*. *Functional* transductions, where every input word is mapped to at most one output word, have been very well studied. A deterministic transducer always defines a functional transduction. However, unlike the case of automata, adding non-determinism strictly increases the expressiveness of transducers as a non-deterministic transducer can define relations,

instead of functions. Allowing two-way movement of the reading head also makes transducers more powerful. For example, using two-way movement, a transducer can copy an input twice, which is not possible without the two-way movement. Therefore, it is particularly important to identify robust classes of transductions, with different equivalent characterizations.

Transductions defined by one-way transducers are definable by rational expressions over a product of free monoids, and therefore are called *rational relations*. This also naturally gives rise to *rational functions*, which are functions definable by a one-way transducer. This class has received a lot of attention in the early works on transducers [Sch75, Cho78].

A more interesting class of transductions are functions defined by two-way transducers, which has received renewed interest in recent years. An equivalent *logical* characterization of this class of functions was given in [EH01] as transformations definable in monadic second-order logic (MSO) [Cou97]. More recently, *streaming string transducers* [AC10] were introduced and shown to characterize the same class of functions. Streaming string transducers are deterministic one-way finite-state machines that process the input in a left-to-right pass and use registers to compute the output. This model was motivated by verification of streaming algorithms [AC11], where the input is provided piece-wise and processed using registers. Streaming string transducers are in contrast with two-way transducers, where the entire output needs to be stored in the memory. The logic-machine equivalence for transductions [AC10, EH01] is reminiscent of Büchi-Elgot-Traktenbrot characterizations of finite-state automata. For this reason, functions defined by two-way word transducers are called *regular functions*. More recently, there have been works providing regular expression-like formalisms, called *regular combinator expressions* [AFR14, Gas19], and *regular list functions* [BDK18]. For a survey on logical and algebraic properties of word transductions, see [FR16]. The recent survey [MP19b] is centered on decision problems for regular word functions.

Non-determinism is a very natural and desirable feature for many finite-state machines. However, for transducers, non-determinism introduces many challenges, in particular decision problems are often intractable. As mentioned, identifying a robust class of transductions is also difficult. MSO-transductions can be extended with non-determinism giving rise to NMSO-transductions. However, the equivalence between two-way transducers and MSO-transductions does not extend to their non-deterministic counterparts [EH01]. Still, NMSO-transductions and non-deterministic streaming string transducers are expressively equivalent [AD11].

Decision problems

Definability problems. Owing to the fact that the various models of transducers studied have different expressiveness, *definability* problems arise. For two classes of transducers $\mathcal{C}_1$ and $\mathcal{C}_2$, such that $\mathcal{C}_1 \subset \mathcal{C}_2$, the definability problem asks given a transducer $T \in \mathcal{C}_2$, whether there exists a transducer $T' \in \mathcal{C}_1$ defining the same transduction as T.

One instance of this is the *one-way definability* problem, which asks if a given two-way transducer is equivalent to some one-way transducer. This problem is of interest because it corresponds to a streaming setting, where the input can be processed one letter at a time, instead of having to store the entire input, and the output is produced on the fly. The one-way definability problem is undecidable in general [BGMP18], but becomes decidable when restricted to functional transducers [FGRS13, BGMP18].

Another interesting instance is the *FO-definability problem*, which asks whether a given transduction is equivalent to some FO-definable transduction. Once again, it is important to restrict our attention to functional transductions as FO-definable transductions are by definition functional. On the side of automata, first-order definability coincides with aperiodicity of the transition monoid of the automata. For transducers, a similar (but non-effective) result characterizing FO-definable transductions was obtained for functional two-way transducers [CD15] and deterministic streaming string transducers [FKT14] by defining aperiodic transition monoids. However, the above characterizations are not effective and the problem of checking whether a transduction is FO-definable or not remains open. The problem was solved for deterministic one-way transducers [Cho03] and was recently shown to be decidable for functional non-deterministic one-way transducers (using an intermediate model called bimachines) [FGL16]. In both these cases, the key idea is to build *a canonical device* defining the transduction.

Equivalence problem. The *equivalence problem* for transducers asks whether two given transducers define the same transduction. For finite-state automata, the equivalence problem, asking whether two given automata define the same language, is decidable. Given two automata A_1 and A_2, equivalence boils down to checking emptiness of $[\![A_1]\!] \cap [\![A_2^c]\!]$, where A_2^c represents the complement of A_2. Note that in case of automata, the computation of A_1 and A_2 is synchronized. However, this is not the case, even for one-way transducers as T_1 and T_2 can read the input in a synchronized manner, but produce outputs at a different rate. In fact, the equivalence problem turns out to be undecidable, even for one-way non-deterministic transducers [FR68, Gri68]. The equivalence test is one of the most widely used operation on automata, so that it becomes a natural question to look for classes of transducers for which equivalence is decidable. Again, the class of functional transducers is well behaved in this respect, with equivalence for

functional one-way transducers, two-way transducers and streaming string transducers being decidable [Sch75, CK87, AD11]. For deterministic streaming string transducers, the problem is decidable even in the unrestricted copyful case [FR17, BDSW17]. In fact, the frontier of decidability goes beyond functional transducers. The equivalence problem is in fact decidable a more general class, called the *finite-valued* transducers. The decidability for one-way and two-way transducers is due to [CK86], whereas the one for streaming string transducers is due to [MP19a]. Another class of transductions with a decidable equivalence problem is the class of deterministic rational relations (DRat). These relations are defined by transducers that have the property that a transition either reads a single input letter or outputs a single output letter. Determinism in this case implies that fixing the choice of input letter or output letter uniquely determines the next state. The equivalence problem for DRat was shown to be decidable [Bir73], even in PTime [FG82].

Origin semantics for transducers

An alternative semantics for transducers, called *origin semantics*, was introduced in [Boj14] in order to obtain canonical functional two-way transducers. In the origin semantics, the output is tagged with positions of the input, called origins, that describe where each output letter was produced. Most transformations of objects arising from applications often come with an associated notion of origin. As an example, consider the problem of *sorting* an array given as an input. The output produced will be a sorted array and each element of the output corresponds to a particular element of the input. As another example from word transductions, the subword ab of $w = aab$ can arise either by choosing the first a in the subword or the second a. Even though these choices correspond to different choices of positions w, they still yield the same subword ab. The origin semantics distinguishes between the two cases by making the choice visible using the origin information. Most models of transducers we considered have a natural notion of origins as well.

As noted by Bojańczyk while introducing origin semantics, origin information has been used to visualize program execution and even construct deubggers. In this context, a program can be seen as a syntax tree, and the origin information can be used to identify positions of errors [vDKT93]. More recently, one-counter automata with observability semantics was introduced [Bol16], which outputs the value of the counter after every transition. The observability semantics allows to recover decidability of some problems such as universality or inclusion problem and also obtain nice closure properties. This semantics is very similar to the origin semantics because the output stores information about the execution.

Equivalence under the origin semantics requires for the transducers to

not only produce the same outputs on every input, but also with the same origins. According to the origin semantics, two transducers may be non-equivalent even when they compute the same relation in the classical semantics. For example, the identity function can be implemented by a transducer by copying a letter at a time, or by copying blocks of two letters at a time. Even though they define the same transformation, these transducers will be inequivalent in the origin semantics. Therefore, the origin semantics is more "strict" than the classical semantics for transducers with respect to equivalence. Recently, decision problems in the origin semantics were considered by Filiot et al. [FJLW16]. In particular, they showed the origin equivalence for one-way transducers to be PSPACE-complete. This is in contrast with the classical semantics, where equivalence is undecidable. Moreover, the complexity PSPACE is the best one can hope for, since equivalence of NFA is already PSPACE-complete [Koz77].

Contributions. We extend the decidability of origin-equivalence from one-way transducers to more expressive two-way transducers and streaming string transducers. For two-way transducers, we show that the origin-equivalence problem is PSPACE-complete, therefore it has the same complexity as equivalence of NFA. For streaming string transducers, we provide an EXPSPACE algorithm to decide origin-equivalence, which can be improved to PSPACE in the deterministic case. Note that the classical equivalence problem is decidable in PSPACE for both deterministic and functional streaming string transducers [AD11]. Therefore, for this case, the origin semantics does not yield better complexity bounds.

We also identify an expressively equivalent subclass of deterministic streaming string transducers, that only uses *separated* updates, i.e, whenever two registers are combined into the same register, they are separated by a non-empty output. For this subclass we show that the origin-equivalence is in PTIME. For streaming string transducers with *copyful* updates, we show decidability of origin-equivalence by providing a backward propagation algorithm.

We also show that the origin-equivalence problem for transducers with unary output alphabet is as hard as the general case. This contrasts with the classical semantics, where restricting to unary output alphabet can give better complexity for the equivalence problem. For example, the classical equivalence problem for deterministic streaming string transducers is in PSPACE, but restricting to unary output alphabet gives a PTIME algorithm [ADD$^+$13].

Resynchronizers.

It can be argued that comparing two transducers in the origin semantics is rather restrictive, because it requires that the same output is generated at precisely the same place. A natural approach to allow some 'distortion' of the origin information when comparing two transducers was proposed in [FJLW16]. *Rational* resynchronizers allow to compare *one-way* transducers (hence, the name 'rational') under origin distortions that are generated with finite control. A rational resynchronizer is simply a one-way transducer that processes an interleaved input-output string, called *synchronized word*, producing another synchronized word with the same input and output projection. After introducing resynchronizers, it was shown that origin-containment up to distortion specified by a rational resynchronizer is decidable for one-way transducers. This gives a way to relax the notion of equivalence under the origin semantics.

Contributions. The synchronized word representation of input-output pairs capturing origin information does not generalize from one-way to two-way transducers (or streaming string transducers). Therefore, we introduce a new model of resynchronizers, called *regular resynchronizers*. These are defined using MSO-formulas and are inspired by MSO-definable graph transformations of Courcelle [Cou97]. The key idea behind this resynchronizer model is to use an MSO formula move with two free variables y, z, which are interpreted as input positions i and j respectively, that defines change of origin of an output position from i to j. In this way, a regular resynchronizer defines a distortion of origins based on logical formula. We show that given a two-way transducer, and a regular resynchronizer, the set of distorted synchronized pairs can be realized by another two-way transducer. This reduces the containment up to distortion specified by a regular resynchronizer to origin-containment, which is decidable.

We also study the *synthesis problem* for resynchronizers. This problem asks whether two classically equivalent transducers can be made origin equivalent by distorting origins using a resynchronizer. If so, the objective is to synthesize such a resynchronizer. We study this problem for both rational resynchronizers introduced by Filiot et al. and also for regular resynchronizers that we have introduced. For both classes of resynchronizers, the problem is undecidable. For rational resynchronizer, we give a way to synthesize a resynchronizer when the given transducers are functional, or even finite-valued. It was already shown that a special type of resynchronizers, called bounded-delay resynchronizers, suffices [FJLW16]. However, we improve on the size of resynchronizer synthesized when the transducers given are functional. For regular resynchronizers, we give an algorithm to decide and if possible, synthesize a resynchronizer when the input transducers are unambiguous.

We also study a variant of the *one-way definability problem* in the origin-semantics. The one-way definability problem in the origin semantics turns out to be trivial and is characterized by order-preserving transducers. The variant of this problem is the *one-way resynchronizability problem*. This asks whether a given two-way transducer can be resynchronized to become equivalent to some one-way transducer. We provide a graph-based property of input-output pairs to classify when this is the case. We also prove an equivalent characterization on the transducer, which gives a PSPACE algorithm for deciding one-way resynchronizability.

Structure of the Thesis

We begin by fixing the notations and definitions used subsequently in the thesis in Chapter 1.

In Chapter 2, we investigate the origin-equivalence problem for different classes of transducers. We begin by recalling the decidability result for one-way transducers given by [FJLW16]. In Section 2.1, we show that the origin-equivalence problem for unary output alphabet is as hard as the general case. In Section 2.2, we show that origin-equivalence for two-way transducers is PSPACE-complete. This section is based on parts of the paper [BMPP18]. Finally, in Section 2.3, origin-equivalence problem is studied for various classes of streaming string transducers.

In Chapter 3, we begin by defining resynchronizations as a means to relax the notion of origin-equivalence. The model of rational resynchronizers from [FJLW16] is defined in Section 3.2 and several of its important properties are defined, such as the notion of bounded delay, lag, etc. We also recall some of the important results regarding rational resynchronizers in this section. In Section 3.3, the model of regular resynchronizers is introduced and the interesting subclass of bounded regular resynchronizer is introduced. This section presents several results from [BMPP18], and [KM20]. In Section 3.4, the models of rational and resynchronizers are compared and we show that rational resynchronizers are a subclass of regular resynchronizer. However, we show that for a subclass of transducers, called real-time transducers, the two formalisms are in fact equally expressive. This section is based on parts of the paper [BKM$^+$19]. This chapter also serves as an introduction to models of resynchronizers, which are used in Chapters 4 and 5.

In Chapter 4, the synthesis problem for resynchronizers is studied. In Section 4.1, the synthesis problem is shown to be undecidable for both rational resynchronizers and bounded regular resynchronizers. The proof we present has been adapted from [KM20]. In Section 4.2, first we show that given functional (or even finite-valued) transducers, the synthesis problem for rational resynchronizers is decidable. Then we show that the synthesis problem for bounded, regular resynchronizer is decidable when the trans-

ducers given as part of the input are unambiguous. These results are from the paper [BKM$^+$19].

In Chapter 5, we study a variant of the one-way definability problem, called the one-way resynchronizability problem, which asks whether a given two-way transducer can be resynchronized to an classically equivalent one-way transducer. In Section 5.1, we recall the results for the one-way definability problem in the classical semantics [BGMP18] as well as the origin semantics [Boj14]. In the origin semantics, one-way definability was shown to be equivalent to order-preserving transductions. In Section 5.2, we give various technical definitions such as *cross-width* and *inversions*, which are used in Section 5.3, to characterize one-way resynchronizability in the simpler case of bounded-visit transducers. Section 5.4 deals with the general case to complete the proof. This chapter is based on the paper [BKMP21].

Each Chapter finishes with a conclusions section which summarizes the results, mentions related open problems, and directions for future work.

Contents

Chapter 1

Preliminaries

In this chapter, we introduce the definitions and notations used in the thesis. First we introduce automata and regular languages. We then extend automata to transducers to define transformations of words. We also define some decision problems for transducers and some known results in this direction. Finally, we define the origin semantics for transducers and the origin-equivalence problem which is studied in this thesis.

1.1 Words and Languages

Words. An *alphabet* Σ is a finite set of symbols, which are called *letters*. A *word* w over an alphabet Σ is a finite sequence of letters from Σ. We denote the set of natural numbers $\{1, 2, \ldots\}$ by $\mathbb{N}$. We denote by $[i, j]$, where $i \leq j$, an *interval* $\{i, i+1, \ldots, j\}$. Formally, a word is a map $w : [1, n] \to \Sigma$. The set $[1, n]$ is called the *domain* of w, denoted by $\mathrm{dom}(w)$. The length of such a word w is n, denoted by $|w|$. Every $i \in \mathrm{dom}(w)$ is called a *position* of w, and $w(i)$ is called the i-th letter of w. We usually represent a word w as the sequence $w(1)w(2)\ldots w(n)$. The *empty word*, denoted by ε is the unique mapping from the empty set to Σ and has length 0.

We denote by $w([i, j])$ the infix between the i-th and j-th position of w, i.e, $w(i)w(i+1)\ldots w(j)$. A word w' is called a *factor* of w, if there exists $i \leq j \in \mathrm{dom}(w)$, such that $w' = w([i, j])$. A factor w' of w is called a *prefix*, denoted by $w' \sqsubseteq w$, if $i = 1$. Similarly, w' is called a *suffix* of w if $j = |w|$. A word w' is called a *subword* of w if there exists positions $i_1 < i_2 < \cdots < i_k$ in $\mathrm{dom}(w)$ such that $w' = w(i_1)w(i_2)\ldots w(i_k)$.

For $\Omega \subseteq \Sigma$ and $w \in \Sigma^*$, we write w_Ω to denote the projection of the word w to Ω, namely the subword of w consisting of all positions of w labeled by letters from Ω.

Cuts. A *cut* in a word is a border between two consecutive positions of the word or the border at the beginning or end of the word. A word w of

length n has cuts $0, 1, \ldots, n$, where 0 is the cut at the beginning of the word, n is the cut at the end of the word, and the cut i is in between the i-th and $i+1$-th positions. As an example, over the alphabet $\Sigma = \{a, b\}$, the word $w = aba$ has 3 positions and 4 cuts. The third letter $w(3) = a$ is between the cuts 2 and 3.

Languages. The set of all finite words over Σ is denoted by Σ^*, whereas Σ^+ denotes the set of non-empty words over Σ. A language L over an alphabet Σ is a subset of the set of all words Σ^*. For example, the set of all words of length at most 7 is a language.

1.2 Automata

1.2.1 One-way automata

A non-deterministic one-way *finite-state automaton* is a tuple $A = (Q, \Sigma, I, F, \Delta)$, where Q is a finite set of *states*, Σ is an alphabet, $I \subseteq Q$ is the set of *initial* states, $F \subseteq Q$ is the set of *final* states, and $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition* relation. We write NFA as a shorthand for finite-state automaton.

Intuitively, an NFA is a finite state machine with a reading head which reads a word, referred to as the input to the NFA, in a left-to-right scan and either *accepts* or *rejects* the input word. A *configuration* of an NFA A on a word w is a pair (q, i), where q is a state and i is a cut, indicating that the current state of the NFA is q and the reading head is at cut i. The *initial configuration* is of the form $(q, 0)$, where $q \in I$. From a configuration $(q, i-1)$, where $q \in Q$, and $1 \leq i \leq |w|$, the NFA can read the letter $w(i)$ and execute a transition $(q, a, q') \in \Delta$ such that $w(i) = a$. Upon taking such a transition, the NFA moves to the new configuration (q', i).

A *run* of A over a word w of length n is a sequence of configurations $\rho = (q_0, 0) \xrightarrow{a_1} (q_1, 1) \xrightarrow{a_2} \ldots \xrightarrow{a_n} (q_n, n)$, such that there exists transitions $t_i = (q_{i-1}, a_i, q_i) \in \Delta$, where $a_i = w(i)$ and $q_0 \in I$. In other words, the configuration (q_i, i) is reachable from $(q_{i-1}, i-1)$ by the transition t_i on the letter $w(i)$. A run is called an *accepting run* if $q_n \in F$. The *language of an automaton* A, denoted by $[\![A]\!]$, is the set of all words $w \in \Sigma^*$, such that A has an accepting run on w. Two NFA A and A' are said to be equivalent if $[\![A]\!] = [\![A']\!]$. The class of languages accepted by NFA are called *regular languages*.

Example 1.2.1. *The following NFA A accepts words over the alphabet $\Sigma = \{a, b\}$ which have "a" as the last letter. The state q_1 is the only initial state, depicted by the incoming arrow and the state q_2 is the final state, depicted by the outgoing arrow. The NFA takes the transition from q_1 to q_2 on the last letter, which must be an a.*

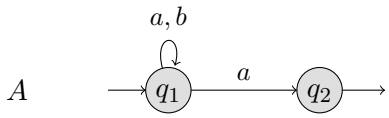

Figure 1.1 – An NFA for words with last letter a

The adjective *non-deterministic* in NFA is used to indicate that it is possible to have a non-deterministic choice among transitions on the same letter. In other words, from a configuration (q_1, i) in the above example, both the transition (q_1, a, q_1) and (q_1, a, q_2) can be executed. The adjective *one-way* signifies that the input is read in a left-to-right scan.

Deterministic finite-state automata. An NFA is called *deterministic finite-state automaton* (denoted by DFA), if there is at most one initial state q_I in I, and for all $q \in Q$ and $a \in \Sigma$, there exists at most one $q' \in Q$, such that $(q, a, q') \in \Delta$. Note that the second condition is equivalent to saying the transition relation is a partial function from $Q \times \Sigma$ to Q.

Even though DFA are defined as a restriction of NFA, DFA are in fact expressively equivalent to NFA. In fact, every NFA can be *determinized*, i.e, an equivalent DFA can be constructed, by using the subset construction, due to [RS59]. In other words, adding non-determinism does not add any expressive power to finite-state automata.

1.2.2 Two-way automata

A generalization of NFA is obtained by allowing the reading head to move in both left and right directions. *Two-way non-deterministic finite-state automata* (2NFA) are finite-state automata but enhanced with both left and right movement on the input. To define a 2NFA, we assume a word has two special delimiter symbols $\{\vdash \dashv\}$ called the left and right *endmarkers* disjoint from Σ, which occur only as the first and last letter in the word respectively. Thus, we only consider words w of length $n \geq 2$, such that the first letter $w(1)$ is $\vdash$ and the last letter $w(n)$ is $\dashv$. All the other positions $1 < i < n$ have letters $w(i)$ from Σ.

Formally, a 2NFA is a tuple $A = (Q, \Sigma, I, F, \Delta)$, where $Q = Q_{\prec} \cup Q_{\succ}$ is the set of *states* partitioned into *left-reading* states $Q_{\prec}$ and *right-reading* states $Q_{\succ}$, Σ is a finite alphabet with endmarkers $\{\vdash, \dashv\}$, $I \subseteq Q_{\succ}$ is the set of initial states, and $F \subseteq Q$ is the set of final states. and $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation. As in the case of NFA, a configuration of a 2NFA A on a word w of length n is a pair (q, i), where $q \in Q$ and $1 \leq i \leq n-1$ is a cut of w, representing the position of the reading head. Note that we assume the input head never moves to the left of $\vdash$, i.e, the cut 0, or to the right of $\dashv$, i.e, the cut n.

The partitioning of states into left and right-moving states is useful to specify which letter to read from the state. From a configuration (q, i), the 2NFA reads $w(i)$ if q is a left-reading state, i.e, $q \in Q_{\prec}$, or $w(i+1)$ if q is a right-reading state, i.e, $q \in Q_{\succ}$.

A transition $(q, a, q') \in \Delta$ connects configurations (q, i) and (q', i'), denoted by $(q, i) \xrightarrow{a} (q', i')$ if one of the following holds:

— $q \in Q_{\succ}$, $q' \in Q_{\succ}$, $a = w(i+1)$, and $i' = i+1$,
— $q \in Q_{\succ}$, $q' \in Q_{\prec}$, $a = w(i+1)$, and $i' = i$,
— $q \in Q_{\prec}$, $q' \in Q_{\succ}$, $a = w(i)$, and $i' = i$,
— $q \in Q_{\prec}$, $q' \in Q_{\prec}$, $a = w(i)$, and $i' = i-1$.

A *run* of A on w is a sequence $\rho = (q_0, i_0) \xrightarrow{a_1} (q_1, i_1) \xrightarrow{a_2} \cdots \xrightarrow{a_m} (q_m, i_m)$ of configurations connected by transitions satisfying the above conditions. Furthermore, we require (q_0, i_0) to be an initial configuration. A run is *accepting* if it ends in a final configuration, i.e, $q_m \in F$ and $i_m = n-1$.

The *language* of A, denoted by $[\![A]\!]$, is the set of words $\{w \mid w \in \Sigma^*$, and A has an accepting run on $\vdash w \dashv\}$. It is known that class of language accepted by 2NFA coincides with the class of regular languages. In particular, for every 2NFA, there exists effectively an *equivalent* NFA that accepts the same language [RS59, She59].

Deterministic two-way automata. Similar to the one-way case, we call a 2NFA A *deterministic*, if A has at most one initial state and for every state $q \in Q$ and letter $a \in \Sigma$, there is at most one transition of the form $(q, a, q') \in \Delta$. We write 2DFA as a shorthand for deterministic two-way automata.

Note that 2DFA is both an extension of DFA and a restriction of 2NFA. Since both DFA and 2NFA define the class of regular languages, 2DFA is expressively equivalent to NFA (equivalently DFA, 2NFA). Therefore, we have the expressively equivalent models of NFA, DFA, 2NFA, and 2DFA, all defining the class of regular languages.

Size of an automaton. We now define the size of an automaton (NFA or 2NFA). Given an automaton $A = \{Q, \Sigma, I, F, \Delta\}$, the *size* of A, is defined as the size of the individual components. Usually, we fix the input alphabet Σ. In this case the size of the transition relation Δ, and the set of initial and final states are polynomially bounded by $|Q|$. Therefore, we say *size of an automata* to denote $|Q|$.

Expressiveness of classes of automata. Figure 1.2 sums up the classes of automata we defined and the comparisons between the expressiveness of different classes. For example, the arrow between DFA and 2DFA denotes that 2DFA is a generalization of DFA by allowing two-way movement of the reading head.

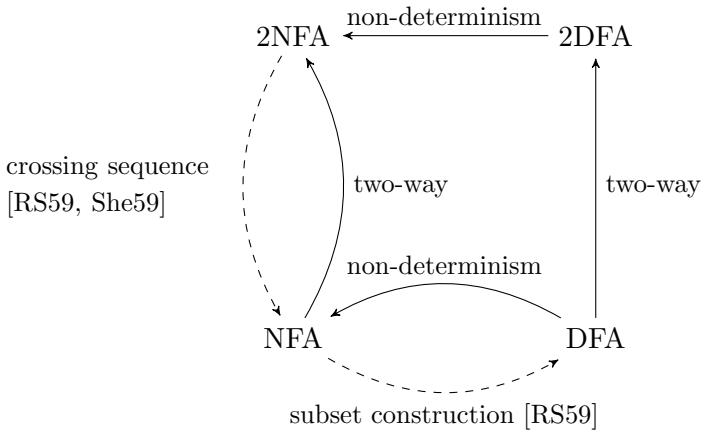

Figure 1.2 – Comparisons between classes of automata

The dashed arrows represent non-trivial inclusions. The construction from a 2NFA to an equivalent NFA is based on crossing sequences [RS59, She59], whereas the subset construction [RS59] gives an equivalent DFA for every NFA. Both these constructions result in NFA (respectively DFA) that can have size exponential in the size of the original 2NFA (respectively NFA). From a 2NFA, one can apply the two constructions to obtain an equivalent DFA, with a doubly exponential blowup. However, a construction of Vardi [Var89] (not depicted in Figure 1.2), allows a direct construction of DFA from a 2NFA with a single exponential blow-up.

Unambiguous automata. An NFA (or 2NFA) A is said to be *unambiguous*, if for every word w, A has at most one accepting run on w. For example, the NFA in Figure 1.1 is unambiguous, because every word that is accepted has exactly one accepting run. It is easy to see that every DFA is also unambiguous.

An NFA (2NFA) is called *k-ambiguous*, if for every word w, A has at most k accepting runs. Note that unambiguity corresponds to the special case of $k = 1$. An NFA (2NFA) is called *finitely ambiguous*, if it is k-ambiguous for some k. The following theorem lists some results on deciding the ambiguity of an NFA.

Theorem 1.2.2. *Given an NFA A, it is decidable in time polynomial in $|A|$,*

- *whether A is k-ambiguous, where k is fixed [SHI85].*
- *whether A is finitely ambiguous or not [WS91].*

1.3 Monadic Second-Order Logic on Words

We now present the monadic second-order (MSO) logic over words (see e.g. [Tho97]).

Signature and logical structure. A *signature* is a tuple $S = (\mathcal{P}, \mathsf{ar})$, where $\mathcal{P}$ is a set of *predicates* and $ar : \mathcal{P} \to \mathbb{N} \cup \{0\}$ is the *arity* function. A predicate $P \in \mathcal{P}$ has arity n if $ar(P) = n$. Given an alphabet Σ, we are interested in the signature $S_\Sigma = (\mathcal{P} = \{\leq, (P_a)_{a\in\Sigma}\}, ar)$, where $\leq$ has arity 2 and for every $a \in \Sigma$, P_a has arity 1.

A *structure* M over the signature S is given by a set called the $\mathrm{dom}(M)$ and an *interpretation* of every predicate. An interpretation of a predicate $P \in \mathcal{P}$ is a subset of $\mathrm{dom}(M)^{ar(P)}$. Predicates with arity 0 are interpreted as a constant in $\mathrm{dom}(M)$.

Words as logical structures. In this setting, we see a word w over the alphabet Σ as a logical structure over the signature S_Σ with domain $\mathrm{dom}(w)$ and the predicates P_a, for $a \in \Sigma$ being interpreted as the set of positions in $\mathrm{dom}(w)$ labeled by a. The predicate $\leq$ is interpreted as the order on the set of positions in $\mathrm{dom}(w)$.

Syntax of MSO. MSO formulas over the signature S_Σ is defined by the following grammar:

$$\phi \ ::= \ \phi \wedge \phi \ ; \ \neg\phi \ ; \ \exists x \phi(x) \ ; \ \exists X \phi(X) \ ; \ x \in X \ ; \ x_1 \leq x_2 \ ; \ P_a(x) \ ; \ x_1 = x_2$$

where, X ranges over a set of monadic second-order variables (often denoted by X, Y, Z, etc), and x, x_1, x_2 range over a set of first-order variables (denoted by symbols x, y, z, etc). The universal quantification over both first-order and monadic second-order variables can be defined from negation and existential quantification. The boolean connective $\phi \vee \phi$ can also be defined as usual, using $\wedge$ and $\neg$.

Free variables and satisfiability. A *free* variable in a formula ϕ is a variable that is not in the scope of a quantifier. A formula ϕ with free variables $x_1, \ldots, x_k, X_1 \ldots X_n$, is written as $\phi(x_1, \ldots, x_k, X_1, \ldots, X_n)$.

Given a set of variables $\hat{X} = \{x_1, \ldots, x_k, X_1, \ldots X_n\}$, we can extend the signature S_Σ of words over Σ to $S_{\Sigma,\hat{X}} = (\mathcal{P}', ar)$, where $\mathcal{P}' = \mathcal{P} \cup \hat{X}$. In other words, every variable is seen as a predicate. The first-order variables have arity 0 and monadic second-order variables have arity 1. Therefore, a word structure (which interprets the $\leq$ and P_a predicates only) can be expanded to an interpretation over this expanded signature, where the the first-order variables are interpreted as positions of the word and the monadic second-order variables are interpreted as sets of positions of the word. We can define

when an expanded word structure *satisfies* the formula ϕ. In particular, the base case of $x \in X$, $x_1 \leq x_2$, $P_a(x)$, $x_1 = x_2$ are defined naturally, and satisfiability for the other formulas can be defined by induction.

A word w with positions $i_1, \dots i_k \in \mathrm{dom}(w)$ and sets of positions $I_1, \dots, I_n \subseteq \mathrm{dom}(w)$, is said to *satisfy* $\phi(x_1, \dots, x_k, X_1, \dots, X_n)$ if the structure w expanded with $i_1, \dots i_k$ as interpretations of $x_1, \dots x_k$, and $I_1, \dots I_n$ as interpretations of $X_1, \dots, X_n$ *satisfies* ϕ. We usually denote this as $(w, i_1, \dots, i_k, I_1, \dots, I_n) \vDash \phi$.

Sentences and languages. An MSO formula is called a *sentence* if it has no free-variable. Given an MSO-sentence ϕ, it defines a language, namely the words that satisfies the formula ϕ. This set is denoted by $[\![\phi]\!]$. The languages defined by MSO-sentences are exactly the class of regular languages [Buc60]. Therefore, this gives an expressive equivalence between MSO logic and the various classes of automata defined earlier.

1.4 Transductions and Transducers

While automata define language of words, we now look at extensions that define transformations of words. We fix an *input* alphabet Σ and an *output* alphabet Γ. A *transduction* is a relation $\mathcal{T} \subseteq \Sigma^* \times \Gamma^*$. For a pair $(u, v) \in \mathcal{T}$, we call u the *input* word, and v the *output* word. The *domain* of a transduction $\mathcal{T}$ is the set of input words $\mathrm{dom}(\mathcal{T}) := \{u \mid \exists v, (u, v) \in \mathcal{T}\}$. Without loss of generality, we assume that Σ and Γ are disjoint. If the alphabets are not disjoint, one can tag them with Σ or Γ to make them disjoint. We will often give examples where the input and output alphabet are not disjoint. We use extensions of automata, called *transducers* as a model to study transductions of words.

1.4.1 One-way transducers

A *one-way finite-state transducer*, denoted by NFT, is an extension of NFA where the transitions read a letter from an *input* word as before, but also produce an *output*. Formally, an NFT $T = (Q, \Sigma, \Gamma, I, F, \Delta)$, where Q is a finite set of *states*, $I \subseteq Q$ is the set of *initial* states, $F \subseteq Q$ is the set of *final* states, Σ and Γ are the input and output alphabet respectively, and $\Delta \subseteq Q \times \Sigma \times 2^{\Gamma^*} \times Q$ is the *transition* relation. Moreover, for every transition $(q, a, L, q') \in \Delta$, we require L to be a *regular language* over Γ.

On an input word u, run of an NFT T is defined similarly to runs of NFA on u. A *run* of T over u is a sequence of configurations $\rho = (q_0, 0) \xrightarrow{a_1|v_1} (q_1, 1) \xrightarrow{a_2|v_2} \dots \xrightarrow{a_n|v_n} (q_n, n)$, such that there exists transitions $t_i = (q_{i-1}, a_i, L_i, q_i) \in \Delta$, where $a_i = u(i)$, $v_i \in L_i$ and $q_0 \in I$, $q_n \in F$. Therefore, for the run ρ on u, the output of the run is the word $v_1 v_2 \dots v_n$.

The transduction defined by T, denoted by $[\![T]\!]$ is the set of all pairs (u, v) such that v is the output of some accepting run ρ of T on u. We often write T instead of $[\![T]\!]$ to denote the transduction defined by a NFT T, as long as it is clear from context.

In the literature (for instance [EM65, MP19b, FR16]), $\Delta \subseteq Q \times \Sigma \times \Gamma^* \times Q$. However, the reading head is allowed to *stay* in the same position with transitions that read the empty word ε. This allows to simulate any output from a regular language L using ε-transitions. Therefore, the two formalism are equivalent. We introduce regular language outputs in order to avoid ε-transitions.

Example 1.4.1. *In Figure 1.3, the transition (q_1, a, c^*, q_2) is a transition which can output any word from c^*, i.e, c^m for any $m \geq 0$. Thus the transducer accepts any input of the form a^n and outputs c^m for every $m \geq 0$.*

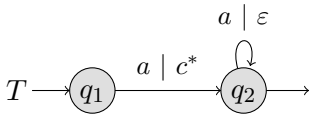

Figure 1.3 – An NFT defining pairs of the form (a^n, c^m)

Real-time transducers. An NFT T is said to be *real-time* if for every transition $(q, a, L, q') \in \Delta$, where Δ is the transition relation of T, L is finite. The name real-time is motivated by the fact that ε-transitions are not needed in the more classical representation of transition where the output is a single word v. Indeed, since L is a finite language, say $L = \{v_1, \ldots, v_n\}$, we can replace the transition (q, a, L, q') by n transitions $t_i = (q, a, v_i, q')$, for $1 \leq i \leq n$, each outputting a different word in L. When talking about real-time NFTs, we often write transitions as (q, a, v, q') instead of (q, a, L, q') with L finite.

Example 1.4.2. *An example of a real-time NFT is in Figure 1.4, which produces output $a^{|w|}$ (resp. $b^{|w|}$)on input wa (resp. wb), where $w \in \Sigma^*$, for $\Sigma = \{a, b\}$.*

Letter-to-letter transducers. A real-time NFT T is called *letter-to-letter* if for every transition (q, a, v, q') of T, we have $|v| = 1$. In other words, each transition reads a single letter of the input and produces a single letter of the output. The NFT in Figure 1.4 is letter-to-letter.

The class of letter-to-letter NFTs define *length-preserving* transductions [EM65]. A transduction $\mathcal{T}$ is called *length-preserving* if for every input-output pair $(u, v) \in \mathcal{T}$, $|u| = |v|$. While it is easy to see that letter-to-letter

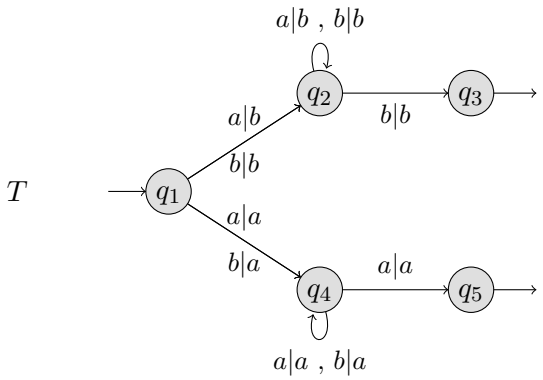

Figure 1.4 – An example of a real-time NFT

NFTs define length-preserving transductions, it was shown in [EM65] that any length-preserving transduction definable by an NFT can also be defined by a letter-to-letter NFT.

Deterministic one-way transducers. An NFT T is called deterministic, denoted by DFT, if it is real-time, it has at most one initial state, and for every state q and letter $a \in \Sigma$, there are at most one state q' and one output word $v \in \Gamma^*$, such that $(q, a, v, q') \in \Delta$. This means reading a letter a from a state q fixes the next state q' and the output produced v.

1.4.2 Two-way transducers

Similar to the case of automata, transducers can also be enhanced by two-way movement of the reading head. A *two-way finite-state transducer* T, denoted by 2NFT, is a tuple $T = (Q, \Sigma, \Gamma, I, F, \Delta)$. The set of states $Q = Q_\prec \cup Q_\succ$ is partitioned into *left-reading* states $Q_\prec$ and *right-reading* states $Q_\succ$, with set of initial state $I \subseteq Q_\succ$ and set of final states $F \subseteq Q$. As with 2NFA, there are special symbols $\vdash$ and $\dashv$, called the left and right *endmarkers*, respectively, in the alphabet Σ. Moreover, we assume the input words are of the form $\vdash u \dashv$. The transition relation $\Delta \subseteq Q \times \Sigma \times \Gamma^* \times Q$ is the set of transitions.

Configurations and *runs* of a 2NFT can be defined similar to an 2NFA, with the only difference being that a transition (q, a, v, q') now also produces an output word $v \in \Gamma^*$. We denote a transition between configurations as $(q, i) \xrightarrow{a|v} (q', i')$. Recall that we have conditions on i, i' and a depending on whether q and q' are left-reading state or right-reading state (see page 5). A *run* of T on u is a sequence $\rho = (q_0, i_0) \xrightarrow{a_1|v_1} (q_1, i_1) \xrightarrow{a_2|v_2} \cdots \xrightarrow{a_m|v_m} (q_{m+1}, i_{m+1})$ of configurations connected by transitions such that (q_0, i_0) is an initial configuration, and (q_{m+1}, i_{m+1}) is a final configuration. For an accepting run ρ, the *output* associated with ρ is any word $v_1 v_2 \cdots v_m \in \Gamma^*$.

Thus, a 2NFT T defines a relation $[\![\mathcal{T}]\!] \subseteq \Sigma^* \times \Gamma^*$ consisting of all the pairs (u, v) such that v is the output of some successful run ρ of T on u.

Note that unlike the case of NFT, here we define transitions with a single word as output instead of a regular language. For 2NFT, it does not matter whether the output in a transition is word $v \in \Gamma$ or a regular language $L \subseteq \Gamma^*$. A 2NFT can simulate ε transitions by using two-way movement of the reading head and therefore simulate any output from L by producing one letter from Γ at a time. Therefore, the real-time restriction is equivalent to the more generalized definition with regular outputs. In this thesis, we mostly consider transitions as producing a single word. However, in some cases, we use the generalization with regular outputs in a transition.

Unlike automata, 2NFT are strictly more expressive than NFT.

Example 1.4.3. *A 2NFT T can recognise the transformation that maps a word w to w^+, i.e, w^m for all $m \geq 0$ over the alphabet $\Sigma = \{a, b\}$. This is shown in Figure 1.5. The 2NFT T copies the word at state q_1, which is a right-reading state, until it reaches the right endmarker $\dashv$. After reaching the end, it moves to state q_2, which is a left-reading state and comes back to the left end of the word without outputting anything. Then it can again copy the word in a left-to-right scan at state q_1. This can be done for any number of times until the run stops at the final state q_2.*

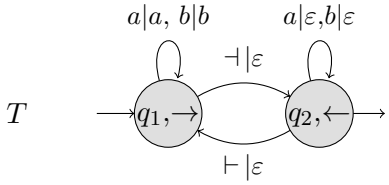

Figure 1.5 – An example of a 2NFT

Deterministic and unambiguous 2NFTs. Similar to the one-way case, we call a 2NFT T *deterministic*, denoted by 2DFT, if it has at most one initial state and, for every $q \in Q$ and $a \in \Sigma$, there is at most one transition of the form $(q, a, \{v\}, q')$ in T. Note that the 2NFT in Example 1.4.3 is not deterministic since it can either halt at the end of the word, or go back to the beginning and copy the word once again.

A 2NFT T is called *unambiguous*, if for every input u, T has at most one accepting run on u. If T has regular language outputs, then we also require the output languages in the transitions occurring in the accepting run to be singleton sets. Note that this also defines *unambiguous NFTs*. The second condition is motivated by the fact that there is an intrinsic non-determinism with regular output languages in a transition as even after choosing the

transition, there is a non-deterministic choice among the possible set of outputs.

Two-way Transducers with common guess. An extension of 2NFT is obtained by allowing *common guess*. Intuitively, this means that the input is marked with a *regular coloring*. The 2NFT then runs on the *colored input* to produce an output. Formally, a 2NFT with common guess is a tuple (T', ϕ), where $T' = (Q, \Sigma \times C, \Gamma, I, F, \Delta)$ is a 2NFT with C a finite set of colors, and ϕ is an MSO-sentence over the alphabet $\Sigma \times C$. Intuitively, ϕ defines a regular subset of $(\Sigma \times C)^*$, which are referred to as *colored words*. A word $u \in \Sigma^*$ is said to be colored as $\hat{u}$ if $\hat{u}_\Sigma = u$. Note that a word u can be colored in multiple ways by ϕ. Therefore, the common guess can introduce additional non-determinism. We denote a 2NFT with common guess by 2NFT_{CG} .

The transduction defined by a 2NFT with common guess $T = (T', \phi)$, denoted by $[\![T]\!]$, consists of pairs (u, v) such that v is the output of T' on some word $\hat{u}$ such that $\hat{u}_\Sigma = u$.

Example 1.4.4. *We give an example of an 2NFT with common guess that maps a word w to v^*, for every subword v of w. This can be done by having the set of colors $C = \{0, 1\}$, and ϕ accepts all words from $(\Gamma \times C)^*$. The 2NFT then scans the word from left-to-right, copies positions with color 1, and then again returns to the beginning of the word copying the subword with color 1 an arbitrary number of times.*

Note than 2NFT with common guess is more powerful than 2NFT, as the Example 1.4.4 cannot be realized by an 2NFT without common guess.

Size of transducers. As was the case with automata, the *size* of a transducer refers to the sum of the size of the individual component defining the transducer. However, unlike the case of automata, the size of the components need not be polynomially bounded by $|Q|$, even for a fixed alphabet, since the output component in a transition can be arbitrarily large.

For a transducer (NFT or 2NFT) T with transitions having regular output language, the *size* of T, denoted by $|T|$ is the sum of number of its states, input symbols, transitions, plus, the sizes of the NFA descriptions of the regular output languages associated with each transition rule.

Let n be a bound such that for every transition $(q, a, q', L) \in \Delta$, the NFA representation of L has size at most n. Then the *size* of the transition relation is polynomially bounded by $n|Q|$. Therefore, over a fixed alphabet, the *size* of a transducer T is also polynomially bounded by $n|Q|$.

1.4.3 Streaming string transducers

Streaming string transducers, introduced in [AC10], denoted as NSST, are an extension of real-time NFT, enhanced by registers to store and compute outputs on a run. Formally, an NSST is a tuple $T = (Q, \Sigma, \Gamma, R, I, F, out, \Delta)$, where Q is a finite set of states, Σ and Γ are finite input and output alphabets, $R = \{r_1, \ldots, r_n\}$ is a finite set of *registers*, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states.

Before defining the transition relation Δ and the *output function out*, we define register updates. A *register update up* is a function from R to $(R \cup \Gamma)^*$. We denote by $\mathcal{U}$ the set of all register updates. The *transition relation* Δ is a subset of $Q \times \Sigma \times Q \times \mathcal{U}$. The *output function out* is a function from F to R^*.

A *valuation* of the registers is a function $val : R \to \Gamma^*$. A valuation of registers can be extended to strings from $(R \cup \Gamma)^*$. For a string $\alpha = u_1 r_1 u_2 \ldots r_k u_k$ with $r_i \in R$, $u_i \in \Gamma^*$, the valuation $val(\alpha)$ of the string α is obtained by replacing every register r_j occurring in α by $val(r_j)$. Therefore, val can be extended to be a function from $(R \cup \Gamma)^* \to \Gamma^*$.

A *configuration* of NSST T on a word u of length n is a triple (q, i, val), where $q \in Q$ is the current state, i is a cut of u, $0 \leq i \leq n$, denoting the position of the reading head, and val is the current valuation of registers. Configuration $(q, i-1, val)$ is connected to configuration (q', i, val') by the transition (q, a, q', up), denoted by $(q, i-1, val) \xrightarrow{a,up} (q', i, val')$, if $u(i) = a$, and for all registers r, $val'(r) = val(up(r))$. A configuration (q, i, val) is said to be *initial* if $q \in I$, $i = 0$, and for all registers $r \in R$, $val(r) = \varepsilon$. A *run* ρ of T on a word u is a sequence $\rho = (q_0, 0, val_0) \xrightarrow{a_1,up_1} (q_1, 1, val_1) \xrightarrow{a_2,up_2} \ldots \xrightarrow{a_n,up_n} (q_n, n, val_n)$ such that $(q_0, 0)$ is an initial configuration and the transition $\xrightarrow{a_i,up_i}$ implies that $a_i = u(i)$ and $val_i(r) = val_{i-1}(up_i(r))$ for all registers $r \in R$. A run is *accepting* if it ends in a final configuration, i.e, $q_n \in F$. The output of an accepting run ρ is defined to be the word $val_n(out(q_n)) \in \Gamma^*$. The transduction defined by T, denoted by $[\![T]\!]$ is the set of all pairs (u, v) such that v is the output of some accepting run ρ of T on u.

Copyless restriction. An update *up* is called *copyless* if for every register r, $\sum_{r' \in R} |up(r')_{\{r\}}| \leq 1$. In other words, the register r occurs at most once in all the update strings $up(r')$, for $r' \in R$. A transition (q, a, q', up) is *copyless* if *up* is copyless. An output function *out* is called copyless if for every state $q_f \in F$ and every register $r \in R$, $out(q_f)$ has at most 1 occurrence of r. An NSST T is called *copyless* if the output function is copyless and every transition of T is copyless.

An NSST that is not copyless is called *copyful*. In this thesis, we write NSST to mean the *copyless* variant. The adjective copyful is used explicitly whenever we talk about the copyful variant. Copyful NSSTs are more

powerful than NSSTs. For example, a copyful NSST can define the exponentiation relation, which maps a word a^n to a^{2^n}, which cannot be done by an NSST.

Deterministic and unambiguous NSSTs. An NSST T is called *deterministic* if it has at most one initial state, and for every $q \in Q$ and $a \in \Sigma$, it has at most one transition of the form (q, a, q', up). A *deterministic* streaming string transducer is denoted by DSST. An NSST T is called *unambiguous* if for every input u, T has at most one accepting run on u.

1.5 Classes of Transducers

We have defined various classes of transducers. These classes vary in their expressive power. We look at the expressive power of the different classes defined earlier, first in the *functional* case, and then in general one.

1.5.1 Functional transductions

A transduction $\mathcal{T}$ is called *functional* if for every input $u \in \Sigma^*$, there exists at most one $v \in \Gamma^*$ such that $(u, v) \in \mathcal{T}$. Therefore, $\mathcal{T}$ is a partial function from Σ^* to Γ^*. A transducer (NFT, 2NFT, or NSST) T is called *functional* if $[\![T]\!]$ is a functional transduction. Note that every deterministic or unambiguous transducer is functional by definition. The *functionality problem* asks whether a given transducer is functional or not. For NFT, this problem was shown to decidable in [Sch75, BH77, GI83]. A PTime algorithm for checking functionality for NFT was provided in [BCPS00]. Using a logic to express structural properties of NFTs called *pattern logic*, checking functionality was shown to be decidable in NLogSpace [FMR18]. For 2NFTs, the problem was shown to be decidable in [CK87]. For NSSTs, an algorithm with PSpace complexity was given in [AD11] to check functionality. Below, we summarize the results from [MP19b], which also provides the exact complexity for 2NFT.

Theorem 1.5.1 (Theorem 3 in [MP19b])**.** *The functionality problem is*
- NLogSpace*-complete for NFT.*
- PSpace*-complete for 2NFT.*
- *in* PSpace *for NSST.*

For completeness we present here the idea behind checking functionality as given in [MP19b]. One guesses an input word and two accepting runs of the transducer on it, and then guesses either that the length of the output of the two runs are different, or there exists an output position where the output of the two runs are different. Both these conditions can be checked by an NFA equipped with a counter which can be incremented or decremented in the transitions. Therefore, the problem of checking functionality

reduces to emptiness of a *one-counter automaton* (denoted by OCA), which is decidable. For NFT, the OCA is of size polynomial in the size of the NFT. For 2NFT, the OCA uses *crossing sequences* to check the required properties, which can be of exponential size. This gives the upper bound of NLOGSPACE and PSPACE respectively. For NSSTs, the size of the OCA is exponential as we need to record the set of registers which appear to the left of the position witnessing non-equivalence in the final output.

The lower bounds are obtained by reducing emptiness of intersection of NFA (2NFA), which is NLOGSPACE-hard (PSPACE-hard [Koz77]). For NFA (2NFA) A_i, $i \in \{1, 2\}$, we can define NFT (2NFT) T_i which outputs i on input u if and only if $u \in [\![A_i]\!]$. Therefore, the intersection $[\![A_1]\!] \cap [\![A_2]\!]$ is non-empty if and only if the $T_1 \cup T_2$ is non-functional.

Expressiveness. For functional transductions, Figure 1.6 summarizes the expressive powers of the different classes of transductions. The class of functions defined by DFTs are called *sequential functions* [Eil74]. It is easy to see that sequential functions are contained in the class of functions defined by NFTs, called the *rational functions* [Sch75]. An interesting result is that functional NFTs are expressively equivalent to unambiguous NFTs (not shown in the figure) [Sch75].

It follows from definition that rational functions are definable by NSST (or 2NFT), since every NFT is also a NSST (or 2NFT). The examples shown in the Figure are examples that cannot be captured by the smaller class. The function $wa \mapsto aw$ can be realized by an NFT but not a DFT, and $w \mapsto \text{rev}(w)$, where $\text{rev}(w) = w(n) \dots w(1)$, $|w| = n$, can be realized by a 2NFT, but not an NFT.

The expressive equivalence between 2NFT and 2DFT was shown in [EH01] through MSO-definable string transductions in the spirit of graph transformation, originally defined by Courcelle [Cou94]. An MSO string-transduction, introduced in [EH01], defines a word over Γ by interpreting the $\leq$ order and $(P_a)_{a \in \Gamma}$ predicates over the set $\text{dom}(u)^k$ for a fixed k, where $u \in \Sigma^*$ is the input word. The equivalence between 2DFT and DSST (and also with MSO-transductions) was shown in [AC10]. Finally, for functional transductions, the class of functions defined by NSST and DSST were shown to be equal in [AD11]. Motivated by the various different equivalent characterizations, including the equivalence between logical MSO-transductions and machine based NSST (or 2NFT), the class of functions defined by NSST (or 2NFT) are called *regular* functions [EH01].

1.5.2 Expressiveness for non-deterministic transducers

In the non-functional case, the picture regarding the expressiveness of different classes of transducers is a bit complicated. We are interested in

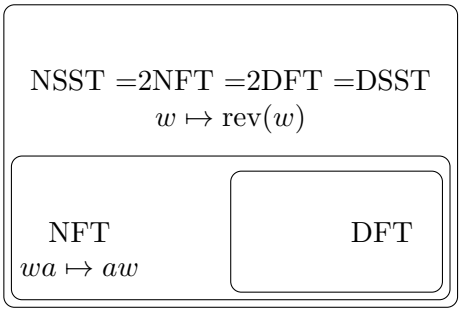

Figure 1.6 – Functional transducers

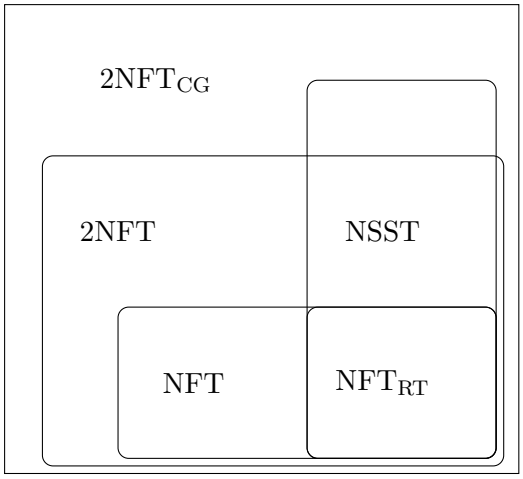

Figure 1.7 – Non-deterministic transducers

the following classes of transducers: real-time NFTs, NFTs, 2NFTs, 2NFTs with common-guess, NSSTs. These classes are depicted in Figure 1.7.

In Figure 1.7, $\mathrm{NFT}_{\mathrm{RT}}$ denotes real-time NFT having the *linear-output* property, i.e, there exists a bound k, such that for every $(u, v) \in [\![T]\!]$ on every input u, $|v| \leq k|u|$. Note that NSST have the linear-output property as well. On the other hand, the classes of transducers on the left side of the figure can produce arbitrarily long output on an input.

The containment of real-time NFT in NFT and NSST follow by definition. The main difference from the functional case is that NSST and 2NFT are now incomparable. For example, the relation $w \mapsto w^*$ can be realized by a 2NFT but not by a NSST. On the other hand the transduction $w \mapsto v^2$, where v is a subword of w, can be realized by an NSST that non-deterministically chooses a subword v and copies the letters of v into 2 different registers. A 2NFT cannot realize this transduction since it has to come back to store the guessed subword v, which is not possible.

However, by adding the feature of common guess, a 2NFT with common

guess, denoted by 2NFT_{CG} in the figure, can color a subword v of w, and the 2NFT can make 2 passes on w and copy the colored positions, realizing $w \mapsto v^2$. In fact, with common guess, the transduction $w \mapsto v^*$, where v is a subword of w can be realized by a 2NFT_{CG} . This idea can be generalized to use the common guess feature to guess a run of an NSST and then use the passes of the 2NFT over the colored input to simulate the output produced by a NSST. This shows that 2NFT with common guess are strictly more powerful than NSST and 2NFT.

Other models. The class of NSST was shown to be expressively equivalent to the *non-deterministic MSO-transductions* in [AD11]. NMSO-transductions on words, introduced in [EH01] are MSO-transduction extended with a common-guess feature, i.e, the input structure is non-deterministically colored using a MSO formula and then an MSO-transduction is applied.

In [AD11], other models, such as ε-NSSTs were introduced, which are NSSTs with transitions reading the empty input ε. This removes the restriction of output-boundedness. However, this turns out to be incomparable to both 2NFT and 2NFT with common-guess. A comparison between the different models can also be found in [BDGP17].

1.6 Decision Problems for Transducers

In this section, we describe some well-studied decision problems for transducers. We start with the containment and equivalence problems, which compare the transductions realized by two transducers.

1.6.1 Equivalence and containment problems

Given two transducers T_1, T_2, we say T_1 is *contained* in T_2, denoted by $T_1 \subseteq T_2$, if $[\![T_1]\!] \subseteq [\![T_2]\!]$. We say T_1 and T_2 are *equivalent*, denoted by $T_1 = T_2$, if $[\![T_1]\!] = [\![T_2]\!]$. This gives rise to the following decision problems for a class of transducers $\mathcal{C}$ (NFT, 2NFT, NSST, etc):

Problem 1.6.1. ***Containment Problem****: Given two transducers T_1 and T_2 from $\mathcal{C}$, is $T_1 \subseteq T_2$?*

Problem 1.6.2. ***Equivalence Problem****: Given two transducers T_1 and T_2 in $\mathcal{C}$, is $T_1 = T_2$?*

It is easy to see that the equivalence problem reduces to the containment problem. In general, the containment problem was shown to be undecidable for NFTs in [FR68]. This was strengthened in [Gri68] to show that equivalence and containment are undecidable for real-time NFTs. In [Iba78], this

was further strengthened to the case of NFTs with unary output alphabet. For completeness we show this proof below.

Theorem 1.6.3. *[FR68, Gri68, Iba78] The containment and equivalence problem for real-time NFT is undecidable, even for unary output alphabet.*

Proof. We present the proof of undecidability due to Ibarra [Iba78] adapting it to the real-time assumption.

The proof reduces the Post Correspondence Problem which asks, given two word morphisms $f, g : \Sigma^* \to \Gamma^*$, if there exist a non-empty word $w \in \Sigma^+$, such that $f(w) = g(w)$. Let k be a constant such that $|f(a)| < k$ and $|g(a) < k$ for all letters $a \in \Sigma$. We assume $\Sigma \cap \Gamma = \emptyset$.

Consider the relation R to be the set of all pairs of the form (wu, a^n), where $w \in \Sigma^*$, $u \in \Gamma^*$, and $u = f(w) = g(w)$ and $n = |u|$. Clearly, we have $n = |u| < k|w|$. Intuitively, the set R encodes the solutions of the PCP instance.

Let R' be the relation consisting of pairs (wu, a^n), where $w \in \Sigma^*$, $u \in \Gamma^*$, and $n < k|w|$. This can be realized by a real-time NFT T' that outputs at most k letters a while reading a letter of w.

Let $R^c = R' \setminus R$ consists of all pairs (wu, a^n) from $\Sigma^+\Gamma^* \times \{a\}^*$ such that, either $u \neq f(w)$, or $u \neq g(w)$, or $n \neq |u|$. We show that R^c can be recognized by a real-time NFT T. The NFT T can check that the input is from $\Sigma^+\Gamma^*$. On an input from this set, it non-deterministically guesses whether $n = |u|$, $n < |u|$ or $n > |u|$, and does the following:

- If $n > |u|$, then the T can output a^n while processing the input as follows. It non-deterministically produces $n - |u|$ many a's while reading the w part of the input and then produces a single a on reading each letter from u. This can be done in a real-time manner since $n < k|w|$.
- If $n < |u|$, then T outputs a^n on the last n positions of u in a real-time fashion.
- If $n = |u|$, then either $f(w) \neq u$ or $g(w) \neq u$. Without loss of generality, let us assume, $f(w) \neq u$. Then we can factorize the input as $w_1bw_2u_1u_2$, such that $w = w_1bw_2$, $u = u_1u_2$, and $|f(w_1)| = |u_1|$ but u_2 does not begin with $f(b)$. The transducer guesses this factorization and generates $a^{|f(w_1)|}$ while processing w_1 and $a^{|u_2|}$ while processing u_2.

The real-time NFTs T and T' are equivalent if and only if the PCP instance has no solutions. Also, we have $T \subseteq T'$ if and only if T and T' are equivalent. Therefore, the PCP reduces to the equivalence (containment) problem for real-time NFTs with unary output alphabet. □

The above proof can be modified to have T' define the universal relation $\Sigma^* \times \Gamma^*$. In this case, T has to be modified to allow to output a^n for any n

in the case $n \neq |u|$. This shows that the universality problem is undecidable for NFTs.

Corollary 1.6.3.1. *The universality problem for NFTs in undecidable.*

Decidable subclasses. While the equivalence problem is undecidable in general, there are many classes of transducers that have decidable equivalence problem. In particular, the *deterministic* subclasses enjoy decidable equivalence. One of the earliest result in this regard is due to Moore [Moo56]. The equivalence problem for DFTs was shown to be decidable by Blattner and Head [BH79]. For 2DFTs, Gurari [Gur82] proved the equivalence problem to be decidable and PSPACE-complete by a reduction to emptiness problem for *reversal-bounded counter machines*.

The equivalence problem for DSST was shown to be decidable in PSpace by Alur and Deshmukh [AD11]. In fact, they reduce the equivalence problem to checking functionality, which is in PSpace (as mentioned in Theorem 1.5.1). Given two NSSTs T_1 and T_2, $T_1 = T_2$ if and only if $T_1 \cup T_2$ is functional and $\mathrm{dom}([\![T_1]\!]) = \mathrm{dom}([\![T_2]\!])$. This also shows that the equivalence problem is decidable for functional NSSTs. For NSSTs, a matching lower bound can be obtained by reduction from equivalence of NFA, which is PSpace-hard. The exact complexity for the case of DSSTs still remains open. In [FR17, BDSW17], equivalence was shown to be decidable for copyful DSSTs as well. Over unary output alphabet, the equivalence problem for copyful DSSTs was shown to be decidable in PTime [ADD$^+$13].

Functionality of NFTs was shown to be decidable by [Sch75](see Theorem 1.5.1), which already gives decidability of the equivalence problem. Independently, the equivalence (and functionality) problem was shown to be decidable in [BH77]. Recall that checking functionality is NLogSpace-complete. However, checking whether $\mathrm{dom}([\![T_1]\!]) = \mathrm{dom}([\![T_2]\!])$ is in PSpace which dominates the complexity. However, for unambiguous NFT, the problem is in PTime, as the equivalence of domain reduces to equivalence of unambiguous NFTs, which is in PTime [SHI85].

For functional 2NFTs, equivalence was shown to be decidable in [CK87]. Though no complexity result is mentioned in this work, the complexity of equivalence problem for functional 2NFTs is PSpace-complete. The lower bound comes as usual from the emptiness of intersection of 2NFA (similar to Theorem 1.5.1). Checking functionality of $T_1 \cup T_2$ is in PSpace and the equivalence of domains can be checked in PSpace as well, due to a construction by Vardi [Var89].

The following theorem summarizes the complexity of the equivalence problem for different classes of functional transducers.

Theorem 1.6.4. *The equivalence problem for functional*

1. *DFTs is* NLogSpace*-complete.*

2. *NFTs is* PSPACE*-complete.*
3. *unambiguous NFTs is in* PTIME*.*
4. *2NFTs,2DFTs, and NSSTs is* PSPACE*-complete.*
5. *DSST is in* PSPACE*.*
6. *copyful DSSTs is decidable.*
7. *copyful DSSTs with unary output alphabet is in* PTIME*.*

Finite-valued transducers. The frontier of decidability however lies beyond the class of functional transducers. A transducer (NFT,2NFT or NSST) T is called *k-valued*, if for every input u, T has at most k different outputs on u. The special case of $k = 1$ corresponds to the functional case. A transducer is called *finite-valued* if it is k-valued for some k.

For k-valued NFTs, the equivalence problem was shown to be decidable by Culik and Karhumäki [CK86]. The following stronger result gives an alternative proof of this result.

Theorem 1.6.5. *[Web96, SdS10] A k-valued NFT can be decomposed into union of k unambiguous NFTs of exponential size.*

The decomposition according to a fixed order yields the property that NFTs T_1 and T_2 are equivalent if and only if the k unambiguous NFTs for T_1 and T_2 obtained by the above theorem are pairwise equivalent. Therefore, checking equivalence of T_1 and T_2 reduces to checking equivalence of unambiguous NFTs of exponential size. This gives an upper bound of EXPTIME.

The original proof due to Culik and Karhumäki uses *Ehrenfeucht's conjecture* to show the existence of a finite test set F_n such that two NFT with at most n states are equivalent iff they agree upon the set F_n. The Ehrenfeucht's conjecture, now a theorem, was proved independently by Albert and Lawrence and by Guba [AL85, Gub86]. We will see an application of this theorem in Chapter 2. This proof also works for finite-valued 2NFTs. However, a decomposition result as in the case of NFT is not known.

For finitely-valued NSSTs, the equivalence problem was shown to be decidable in [MP19a]. In this case, a decomposition result as in the case of NFT would prove the expressive equivalence between finite-valued NSSTs and finite-valued 2NFTs. This is because for every k-valued 2NFT, there exists an equivalent k-valued NSST. The NSST guesses a run of the 2NFT and simulates the output. Guessing the run is possible because for finitely-valued 2NFT, we can assume the crossing sequences to be of bounded size. A decomposition result for NSSTs would allow to build, given an NSST T, k NSSTs $T_1 \dots T_k$, such that $[\![T]\!] = [\![\bigcup_{i=1}^{k} T_i]\!]$. Since each of these T_i is a functional NSST, there exists equivalent 2NFTs, and therefore T is equivalent to the union of these functional 2NFTs. Therefore, the decomposition of a k-valued NSST remains an interesting open problem.

Theorem 1.6.6. *The equivalence problem for k-valued*

1. *NFT is in* ExpTime *(where k is fixed) [Web96].*
2. *2NFT and NSST is decidable [CK86, MP19a].*

1.6.2 One-way definability

A relation $R \subseteq \Sigma^* \times \Gamma^*$ is said to be *one-way definable* if $R = [\![T]\!]$ for some NFT T. The *one-way definability problem* is defined as follows.

Problem 1.6.7 (**One-way Definability**). *Given a 2NFT T, is it equivalent to some NFT T'?*

This problem is undecidable in general due to the fact that the universal relation is *one-way definable*. This means that the universality problem, which is undecidable, reduces to one-way definability problem.

Restricted to functional 2NFTs, the one-way definability problem was shown to be decidable [FGRS13], even in 2-ExpSpace [BGMP18].

1.7 Origin Semantics

In this section, we present the origin semantics for transducers. Origin semantics for transducers was introduced in [Boj14] with the motivation to study machine-independent characterization for various classes of (deterministic) transducers. In the origin semantics, the output is tagged with information about the position of the input where it was produced. The formal definition of origin information depends on the model of transducer. We start by defining origin transductions and then define the origin semantics for transducers in the classes NFT, 2NFT, and NSST.

Data words. A word over an *infinite alphabet* is defined similarly to words over finite alphabet with the difference that the letters now come from the infinite alphabet. We consider words over alphabets of the form $\Gamma \times \mathbb{N}$, where Γ is a finite alphabet, and $\mathbb{N}$ is the infinite set of natural numbers $1, 2, \ldots$. Words over such alphabets are called *data words*. For a position i in a data word $w \in (\Gamma \times \mathbb{N})^*$ labeled by (a, n), n is called the data value at position i and a is called the label at position i. For a word $v \in \Gamma^*$ and a data value $i \in \mathbb{N}$, the data word $(v(1), i)(v(2), i) \ldots v(|v|, i)$ is denoted by $v \otimes i$.

Origin transductions. A *synchronized pair* over an input alphabet Σ and output alphabet Γ is a pair (u, v), where $u \in \Sigma^*$ and $v \in (\Gamma \times \mathbb{N})^*$, such that the data values appearing in v are from the set $[1, |u|]$. An *origin transduction* over input alphabet Σ and output alphabet Γ is a set of synchronized pairs over Σ and Γ, i.e, $\mathcal{T}_o \subseteq \Sigma^* \times (\Gamma \times \mathbb{N})^*$. For a synchronized pair $(u, v) \in \mathcal{T}_o$ the word u is called the *input* and the data word v is called

the *output*. The data value at position i of v is called the *origin* of position i, written as $orig(v(i))$ and the letter from Γ at position i is denoted by $v(i)_\Gamma$.

Alternatively, the synchronized pair (u, v) can also be represented by *origin graphs*, which are graphs with vertices $V = \mathrm{dom}(u) \uplus \mathrm{dom}(v)$ for the input and output positions labeled by letters from Σ and Γ respectively. The graph consists of successor edges for the input and output positions. This defines the input and output words in the graph. The other type of edge in the graph are the *origin* edges, defined from the set $\mathrm{dom}(v)$ to the set $\mathrm{dom}(u)$. For some position $x \in \mathrm{dom}(v)$, if $orig(v(x)) = i$, then there is an origin edge (x, i). The origin graph representation was introduced in [BDGP17], and is equivalent to the synchronized word representation defined earlier. While talking about origin transductions, we mostly reason on the synchronized word representation, unless mentioned otherwise.

One-way transducers. For an NFT, the origin of a certain output position is the last input position read before producing that output. Formally, given a run $\rho = (q_0, 0) \xrightarrow{a_1|v_1} (q_1, 1) \xrightarrow{a_2|v_2} \cdots \xrightarrow{a_n|v_n} (q_n, n)$ of an NFT T on an input $u = a_1 \cdots a_n$, the *output* of ρ in the *origin semantics* will be a data word $v = v'_1 v'_2 \dots v'_n$, where $v'_i = v_i \otimes i$. Recall that $v_i \otimes i$ denotes the data word where each position has the data value i and the projection to Γ equals v_i. An NFT T defines the origin transduction $[\![T]\!]_o = \{(u, v) \mid v \text{ is the output of a run } \rho \text{ of } T \text{ on } u \text{ in the origin semantics.}\}$

A synchronized pair (u, v) is called *order-preserving* if for every positions $x < x' \in \mathrm{dom}(v)$, we have $orig(v(x)) \leq orig(v(x'))$. An origin transduction $\mathcal{T}_o$ is called *order-preserving* if every synchronized pair $(u, v) \in \mathcal{T}_o$ is order-preserving. An origin transduction defined by an NFT is order-preserving by definition.

An important feature of order-preserving synchronized pairs is that they can also be represented as an interleaving of the input and output word, called the *synchronized word*. The output v of an order-preserving synchronized pair (u, v) can be written $v = (v_1 \otimes 1)(v_2 \otimes 2) \dots (v_k \otimes k)$, where $k = |u|$ and $v_i \in \Gamma^*$ is the (possibly empty) factor of v with data value i projected to Γ. Assuming the input and output alphabets Σ and Γ are disjoint, the *synchronized word* representation of such a pair (u, v) is the word $w = u(1)v_1u(2)v_2 \dots u(k)v_k$, where v_i are the words obtained in the factorization defined earlier.

For an NFT T and synchronized pair $(u, v) \in [\![T]\!]_o$, we say T *generates* (u, v). Equivalently, if w is the synchronized word representation of (u, v), we say T *generates* w. The *synchronization language* of an NFT T is the set of all synchronized words generated by the transducer, denoted by $Sync(T)$. The synchronization language was first defined in [Niv68] and shown to be a regular language. It is clear that the synchronization language captures

the origin semantics of an NFT.

Theorem 1.7.1. *[Niv68] The synchronization language Sync(T) of an NFT T is regular.*

The above theorem immediately follows from the observation that every transition (q, a, L, q') of T can be replaced by a transition (q, aL, q') to obtain an NFA with transitions having regular languages, describing the synchronization language, which shows $Sync(T)$ is regular.

Two-way transducers. The origin semantics for 2NFTs is defined in a way similar to NFTs. The *output* associated with a successful run $\rho = (q_0, i_0) \xrightarrow{a_1|v_1} (q_1, i_1) \xrightarrow{a_2|v_2} (q_2, i_2) \cdots \xrightarrow{a_m|v_m} (q_m, i_m)$ in the *origin semantics* is the data word $v = v'_1 v'_2 \dots v'_m$, where $v'_j =$

$$v_j \otimes (i_j + 1) \text{ if } q_j \in Q_{\succ}$$
$$v_j \otimes (i_j) \text{ if } q_j \in Q_{\prec}$$

Note that the origin of v_j corresponds to the input position read by the j-th transition. As with NFTs, a 2NFT T defines the origin transduction $[\![T]\!]_o$ consisting of all synchronized pair (u, v) such that T has a run on u producing output v in the origin semantics.

Streaming string transducers. The origin semantics for NSSTs is slightly more complicated to define. Intuitively, the *origin* of an output position is the input position read by the transition that first added the output into some register. To define this formally, we change the definition of a valuation. An *valuation val* in the *origin semantics* is a function from registers to $(\Gamma \cup \mathbb{N})^*$. A *configuration* of the NSST in the origin semantics has a valuation with origins instead of the classical valuation.

Consider a run $\rho = (q_0, 0, val_0) \xrightarrow{a_1;up_1} (q_1, 1, val_1) \xrightarrow{a_2;up_2} \cdots \xrightarrow{a_n;up_n} (q_n, n, val_n)$ on an input u. As in the classical semantics, this implies that for every $1 \leq i \leq n$, $a_i = u(i)$ and $(q_{i-1}, a_i, q_i, up_i) \in \Delta$. The *valuations* are defined inductively with $val_0(r) = \varepsilon$ for every register r and $val_i(r) = val_{i-1}(up(r) \otimes i)$, where $up(r) \otimes i$ is the word over $(R \times (\Gamma \times \{i\}))^*$ obtained by replacing every letter g from Γ by (g, i). Therefore, the letters from Γ that are freshly added to registers by up_i are tagged with their origin as i. The output of ρ in the origin semantics will be $v = val_n(out(q_n))$. As with 2NFT and NFT, the origin transduction $[\![T]\!]_o$ realized by an NSST T is the set of synchronized pairs (u, v) such that v is the output of some run of T on u in the origin semantics.

Origin-containment and origin-equivalence problems. Let T_1 and T_2 be two transducers from a class $\mathcal{C}$ (NFT, 2NFT, NSST). We say T_1 is

origin-contained in T_2, denoted by $T_1 \subseteq_o T_2$, if $[\![T_1]\!]_o \subseteq [\![T_2]\!]_o$. We say T_1 and T_2 are *origin-equivalent*, denoted by $T_1 =_o T_2$, if $T_1 \subseteq_o T_2$ and $T_2 \subseteq_o T_1$, i.e, $[\![T_1]\!]_o = [\![T_2]\!]_o$.

Therefore, we define containment and equivalence problems in the origin semantics.

Problem 1.7.2. ***Origin Containment****: Given two transducers T_1 and T_2 in $\mathcal{C}$, is $T_1 \subseteq_o T_2$?*

Problem 1.7.3. ***Origin Equivalence****: Given two transducers T_1 and T_2 in $\mathcal{C}$, is $T_1 =_o T_2$?*

We will focus on these two problems throughout the thesis. When the semantics of a transducer (classical or origin) is clear from the context, we often write T instead of $[\![T]\!]$ (or $[\![T]\!]_o$).

An immediate observation is that the origin-containment (and origin-equivalence) is more *strict* than classical containment (equivalence). For example, the NSSTs in Figure 1.8 that define the reverse and identity over an unary alphabet define the function $a^n \mapsto a^n$ in the classical semantics and therefore are equivalent. However, in the origin semantics, the input a^n is mapped to $(a,1)\dots(a,n)$ by T_{id}, whereas T_{rev} maps a^n to $(a,n)\dots(a,1)$.

Figure 1.8 – NSSTs that are origin-inequivalent

All the results regarding expressiveness presented in Section 1.5 hold even in the origin semantics. Even MSO-definable transductions have a natural definition of origin and the constructions used in [EH01] or in [AD11] to show equivalence of various classes of transducers, preserve the origins in the respective models (such as NSSTs or 2NFTs) [Boj14].

Chapter 2

The Origin-Equivalence Problem

In this chapter, we study the equivalence and the containment problems for transducers under the origin semantics. Two transducers may be non-equivalent in the origin semantics even if they compute the same relation in the classical semantics. This can happen as the transducers may generate the output with different origins. Consider the reverse and the identity transductions applied to words over unary alphabet $\{a\}$. For a word $u = a^n$, the output in the classical semantics for both reverse and identity is the same word a^n. However, in the origin semantics, the outputs will be $(a,1)(a,2)\dots(a,n)$ in the identity transduction and $(a,n)(a,n-1)\dots(a,1)$ in the reverse transduction. Therefore, origin-equivalence is a refinement of classical equivalence.

The main results in this chapter show decidability of origin-equivalence for the transducer classes NFT, 2NFT, and NSST. These results contrast with the equivalence problem in classical semantics, which is undecidable for NFT [FR68, Gri68], even with unary output alphabet [Iba78]. The origin-equivalence problem was first considered for NFT [FJLW16]:

Theorem 2.0.1 ([FJLW16])**.** *The origin-equivalence problem for NFT is* PSPACE*-complete.*

The above result is obtained by reducing the problem to equivalence of NFA which accept the *synchronization languages* of the given NFT, which captures the input, and the output with the origin information (see page 22). The NFA obtained is polynomial with respect to the size of the given NFT. Since, checking equivalence of NFA is PSPACE-complete, we obtain the same complexity for origin-equivalence of NFT.

In this chapter, we first show that the origin-equivalence problem for transducers with unary output alphabet is polynomially equivalent to the general case. This is different from classical equivalence, where restricting to

unary output alphabet often helps reducing the complexity. For example, equivalence of DSSTs with unary output alphabet is solvable in PTime whereas in general it is in PSpace (mentioned in Theorem 1.6.4).

We then consider the origin-equivalence problem for 2NFTs and NSSTs. Finally, we investigate the decidability and the complexity of the origin-equivalence problem for various classes of DSSTs.

2.1 Unary output alphabet

Two transducers are classically inequivalent if there is some input on which the outputs of the two transducers are different. When restricted to unary output alphabet, different outputs means different lengths of output. The equivalence problem in general is undecidable even for unary output alphabet [Iba78] for NFT. However, for decidable subclasses, restricting to unary output alphabet sometimes yields a better complexity. For example, the classical equivalence problem for DSST is known to be in PSpace [AC11]. However, the equivalence problem for DSST with unary output alphabet can be solved in PTime [ADD+13].

In the origin semantics, two transducers can be inequivalent because of two reasons. Either, there exists an input on which the transducers produce different output; or there exists an input on which the same output has different origins. In case of an unary output alphabet, the difference can either be in the length of the outputs, or in the origins. We show below that one can encode a bigger output alphabet into a unary one using origins in such a way that the blowup in the size of the transducers is only polynomial. Using this construction, we obtain that the origin-equivalence of transducers (2NFT or NSST) is polynomially equivalent to the case of transducers restricted to unary output alphabet.

Theorem 2.1.1. *Origin-equivalence of NSST (2NFT) can be reduced in polynomial time to origin-equivalence of NSST (2NFT) with unary alphabet.*

We present the proof for NSST. A similar proof works in case of 2NFT as well.

The unary transformation. We show that an output alphabet $\Gamma = \{g_1, g_2, \ldots, g_m\}$ can be encoded using the origins and an unary output alphabet by expanding the input word.

Given an origin transduction $\mathcal{T} \subseteq \Sigma^* \times (\Gamma \times \mathbb{N})^*$, we define a function *unary* which maps *injectively* every synchronized pair $(u, v) \in \Sigma^* \times (\Gamma \times \mathbb{N})^*$ to a synchronized pair (u', v'), where $u' \in (\Sigma \uplus \Gamma)^*$ and $v' \in \mathbb{N}^*$. The new input word u' is obtained by substituting every letter a of u by ga, where $g = g_1 g_2 \ldots g_m$. For example, for input $u = aba$ over $\Sigma = \{a, b\}$ and output

alphabet $\Gamma = \{c, d\}$, the new input u' is *cdacdbcda*. Each factor ga of the input for some letter a in Σ is called a *letter-block* of u'.

The new output v' is defined as follows. The length of v' is equal to the length of the original output v. For every position $x \in \mathrm{dom}(v)$, such that $orig(v(x)) = i$ and $v(x)_\Gamma = g_j \in \Gamma$, $orig(v'(x)) = (|\Gamma| + 1)(i - 1) + j$. In other words, the origin in v' corresponds to the g_j in the i-th letter-block of u'. The following Proposition follows from definition.

Proposition 2.1.2. *The function unary* : $\Sigma^* \times (\Gamma \times \mathbb{N})^* \to (\Sigma \cup \Gamma) \times \mathbb{N})^*$ *is injective. Therefore, two origin-transductions R and R' are origin-equivalent, if and only if, unary(R) and unary(R') are origin-equivalent.*

We now show how the transformation *unary* is implemented by an NSST.

Lemma 2.1.3. *Given an origin transduction R defined by an NSST T, an NSST unary(T) can be constructed, defining the origin transduction unary(R). The size of unary(T) is polynomial in the size of T.*

Proof. Let $T = (Q, \Sigma, \Gamma, R, I, F, out, \Delta)$ be an NSST such that $\Gamma = \{g_1, g_2, \cdots, g_m\}$. For a register update up and a letter $g_i \in \Gamma$, the number of occurrences of g_i in up is defined to be $\sum_{r \in R} |up(r)_{\{g_i\}}|$, where $up(r)_{\{g_i\}}$ denotes the update string $up(r)$ projected to $\{g_i\}$. Let K be a number such that for every transition (q, a, q', up) of T and every $g_i \in \Gamma$, there are at most K occurrences of a g_i in the right-hand side of the update up. Here, the occurrences of a letter g_i can be in the right-hand side of the update for different registers

Let (u, v) be a synchronized pair of T. Recall that the input to *unary*(T) should replace every letter a of u by a letter-block ga, where $a \in \Sigma$ and $g = g_1 g_2 \dots g_m$. Let $unary(T) = (Q', \Sigma', \Gamma', R', I', F', out', \Delta')$ be an NSST uch that set of states Q' of *unary*(T) is $Q \uplus (Q \times [1, m])$. The states of the form $(q, i) \in Q'$ are said to be in the i-th *layer*. Therefore, the state space Q' is divided into m layers and one copy of the original set of states Q. The set of initial and final states $I' = I$ and $F' = F$ are the same as that of T. The new input alphabet is $\Sigma' = \Sigma \uplus \Gamma$ and the output alphabet Γ' is the unary alphabet $\{\#\}$. The set of registers R' of *unary*(T) is $R \uplus \hat{R}$, where $\hat{R} = \{r_i^j \mid 1 \le j \le m \text{ and } 1 \le i \le K\}$, where $m = |\Gamma|$ and K is the constant defined above. Therefore, there are mK new registers.

The set of transitions Δ' of *unary*(T) are grouped into three types. The first two types of transitions are used to check the validity of the letter-blocks. First, there are transitions starting from the states Q to the first layer of the states. These transitions are of the form $(q, g_1, (q, 1), up^1)$, where up^1 copies the letter # into all registers in r_i^1, for $1 \le i \le K$. The second type of transitions are the ones between the $(j-1)$-th and j-th layers. These are of the form $((q, j - 1), g_j, (q, j), up^j)$ for $1 < j \le m$, where up^j keeps the values of all register the same except for the registers r_i^j, for $1 \le i \le K$,

which are updated as $up^j(r_i^j) = \#$. Thus, these transitions check that the letter-blocks are of the correct form, and copy the output letter # into K different registers, each with origin at g_j in the letter-block.

The last type of transitions go from the last layer back to the copy of Q and simulate the original transitions of Δ. For every $(q, a, q', up) \in \Delta$, there exists a transition in Δ' of the form $((q, m), a, q', up')$, where $up'(r)$ for a register $r \in R$ is obtained by replacing all occurrences of the output letter $g_j \in \Gamma$ in the right-hand side of $up(r)$ by a register from the set r_i^j. This can be done in a copyless manner since each output letter g_j occurs at most K times on the right-hand-side. The final output update $out'(q_f)$ is the same $out(q_f)$ as T.

Note that the constants m and K are both polynomial in $|T|$. Therefore, the size of $unary(T)$ is polynomial in size of T since it consists of $m+1$ copies of states and mK additional registers. □

Theorem 2.1.1 follows from Lemma 2.1.3 and Proposition 2.1.2 for NSST. To prove Theorem 2.1.1 for 2NFT, we need to show how $unary(T)$ can be implemented by a 2NFT. We present the idea without giving the formal details.

Two-way transducers. In case of NSST, a single transition is simulated by transitions on the corresponding letter-blocks. For a 2NFT T, the 2NFT $unary(T)$ will have to move inside a letter-block to simulate a single transition of T. For example, let $\Sigma = \{a, b\}$ and $\Gamma = \{c, d\}$. To simulate a transition (q, a, cdc, q') of T, the 2NFT $unary(T)$ will have to move to the position marked by c, followed by d, followed by c in the corresponding letter-block cda. This is illustrated in Figure 2.1.

Depending on the reading direction of q and q', the sequence of transitions will begin and end at *left* or *right* border of the letter-block. In the example in Figure 2.1, assuming q, q' are both right-reading states, the transition starts from the *left* border and ends in the *right* border. Using the two-way movement of the reading head, it is possible to move to the c and d back and forth within the letter-block. In general, this may introduce intermediate state $K|Q||\Gamma|$, where K is the length of the largest word v that occurs as output in a transition (q, a, v, q'). Therefore, the size of $unary(T)$ is polynomial in the size of T.

One-way transducers. For an NFT T, although the origin transduction $unary(T)$ defined above need not be definable by a NFT, we can build a NFT which takes as input words from the *synchronization language* and outputs a symbol from an unary alphabet after reading every letter. Since, the sychronization language already encodes the origin transduction, this transformation also retains the origin information. Therefore, the case of

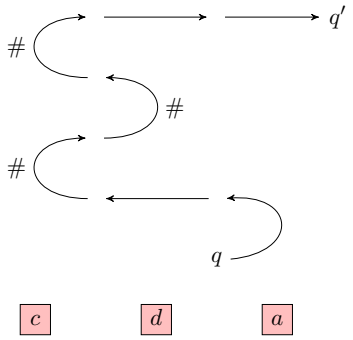

Figure 2.1 – Simulating a transition (q, a, cdc, q') using unary output alphabet

unary output alphabet is polynomially equivalent to the general case for NFT with respect to the origin-equivalence.

2.2 Origin-equivalence for Two-way Transducers

We show that the origin-equivalence problem for 2NFT can be solved in PSPACE, i.e, the same complexity as equivalence of NFA, which also provides a lower bound of PSPACE-hardness. Note that this result is shown for the *non-deterministic* transducer model, which shows that the origin semantics simplifies significantly the equivalence problem, since the equivalence problem is undecidable in the classical semantics.

Theorem 2.2.1. *Origin-containment and origin-equivalence of 2NFT is* PSPACE*-complete.*

Before getting into the formal details, we first give a brief overview of the proof. We first consider the origin-containment problem for *busy* transducers, which produce non-empty output in every transition. In this simpler case, we characterize equivalent runs using a notion of shape, that formalizes how the input head moves during the run. Two runs are origin-equivalent if and only if they have the same shape and same output at every transition, and show that equivalence can be reduced to checking emptiness of an NFA. The key observation for this reduction is that sub-runs of a transducer between an infix of the input can be abstracted by a pair of states and equivalence between these sub-runs can be checked by checking some local properties of the sub-runs. For the general case, when the transducers are not necessarily busy, we first do a normalization and then reduce the problem to the busy case.

For technical reasons, we will assume that 2NFT have regular outputs, i.e, their transitions are of the form (q, a, L, q') where the output is taken from a regular language $L \subseteq \Gamma^*$. Such a transition can output any word from

the language L. This does not add any expressive power as such a transition can be simulated by a sequence of transitions with a single output (see page 11).

The output language L of a transition is represented by an NFA. Recall that the *size* $|T|$ of the 2NFT T counts the number of its states, transitions, and the sizes of the NFA descriptions of the regular output languages associated with each transition rule. For the complexity analysis, we will need to construct NFA and 2NFTs on-the-fly. Often, the states and transitions can be enumerated using polynomial sized working space, even though the NFA or 2NFT could be exponential for some fixed parameter n. We introduce the concept of PSPACE-*constructibility* to formalize this.

PSPACE-constructibility. Given a parameter $n \in \mathbb{N}$, we say that an NFA or a 2NFT has PSPACE-*constructible transitions w.r.t.* n if its transition relation can be enumerated by an algorithm that uses working space polynomial in n. For 2NFTs, this implies that every transition has at most polynomial size in n. In particular, if a 2NFT has PSPACE-constructible transitions with respect to n, the size of the NFA representing output language is also polynomial in n.

We break the proof of Theorem 2.2.1 into two cases. We will first show how to decide origin-equivalence of 2NFT in PSPACE assuming that every transition of the 2NFT produce non-empty outputs. We call any such 2NFT *busy*. Then we show that the general problem of origin-equivalence of 2NFTs can be reduced to origin-equivalence of busy 2NFT.

Origin-equivalence of Busy 2NFT.

In a busy 2NFT, every transition produces non-empty output. Therefore, the sequence of origins of output positions determines the movement of the input head. As a consequence, runs of two busy 2NFTs can be origin-equivalent if they visit the input positions in the same sequence and have the same outputs transition-wise.

We introduce now the notions of transition shape and witness procedure that are used in the proof.

Let T_1, T_2 be two busy 2NFTs, with $T_i = (Q_i, \Sigma, \Gamma_i, \Delta_i, I_i, F_i)$ for $i = 1, 2$. Recall that the set of states Q_1 (and resp. Q_2) are assumed to be partitioned into left-reading and right-reading states $Q_{1,\prec} \cup Q_{1,\succ}$ ($Q_{2,\prec} \cup Q_{2,\succ}$ resp.). We say that two transitions $t_1 \in \Delta_1$ and $t_2 \in \Delta_2$ on the same letter, with $t_i = (q_i, a, q_i', L_i)$, have the *same shape* if $q_1 \in Q_{1,\prec} \Leftrightarrow q_2 \in Q_{2,\prec}$, and $q_1' \in Q_{1,\prec} \Leftrightarrow q_2' \in Q_{2,\prec}$.

Witness Procedures. A *witness procedure* $\mathcal{W}$ is a non-deterministic procedure that does the following. Given a transition $t_1 = (q_1, a, q_1', L_1)$ of T_1,

the procedure $\mathcal{W}$ returns a set $X \subseteq \Delta_2$ of transitions of T_2 satisfying the following property: *for some word* $v \in L_1$, we have

$$X \;=\; \big\{t_2 = (p_2, a_2, q_2, L_2) \in \Delta_2 \;:\; v \in L_2, \text{ and } t_2 \text{ has } \textit{same shape} \text{ as } t_1\big\}.$$

Intuitively, $\mathcal{W}$ fixes a choice of output v for the transition t_1 and returns all transitions of T_2 with the same shape as t_1 that output v. Note that $\mathcal{W}$ is non-deterministic: it can return several sets based on the choice of v. The witness procedure is used to deal with the intrinsic non-determinism from regular output languages. Indeed, if T_1 and T_2 had a single output word for each transition, then $\mathcal{W}$ would return only one set on $t_1 \in \Delta_1$, that is, the set of transitions of T_2 with the same shape and the same output as t_1.

Compatibility of runs. Given a run $\rho_1 = t_1 \dots t_m$ of T_1 of length m and a sequence $\xi = X_1, \dots, X_m$ of subsets of Δ_2, called *witness sequence*, we write $\xi \in \mathcal{W}(\rho_1)$ whenever $X_i \in \mathcal{W}(t_i)$ for all $1 \leq i \leq m$. We say that a run $\rho_2 = t'_1 \dots t'_m$ of T_2 is ξ-*compatible* if $t'_i \in X_i$ for all $1 \leq i \leq m$.

Intuitively, a witness sequence is a sequence of sets of transitions of T_2 returned by the witness procedure $\mathcal{W}$. A ξ-compatible run for a witness sequence will therefore be origin-equivalent to ρ_1.

Proposition 2.2.2. *Given two busy 2NFTs T_1, T_2 and a witness procedure $\mathcal{W}$, $T_1 \subseteq_o T_2$ if and only if for every successful run ρ_1 of T_1, and for every witness sequence $\xi \in \mathcal{W}(\rho_1)$, there is a successful run ρ_2 of T_2 which is ξ-compatible.*

Proof. We first assume that $T_1 \subseteq_o T_2$. Consider a successful run $\rho_1 = t_1 \dots t_m$ of T_1 on input u, with each transition t_k of the form $(q_k, i_k) \xrightarrow{b_k|L_k} (q_{k+1}, i_{k+1})$. Choose any witness sequence $\xi = X_1, \dots, X_m$ in $\mathcal{W}(\rho_1)$. Recall that each X_k corresponds to a choice of an output $v_k \in L_k$. So $v = (v_1 \otimes j_1) \cdots (v_m \otimes j_m)$ is an output (tagged with origins) produced by ρ_1, where each j_k is either i_k or $i_k - 1$ depending on whether q_k is right-reading or left-reading. (Recall that $(v_\ell \otimes j_\ell)$ denotes the output v_ℓ with all positions having origin j_ℓ, see page 23) Since $T_1 \subseteq_o T_2$, there must be a successful run $\rho_2 = t'_1 \dots t'_m$ of T_2 on the same input u that enables the same v as output. In other words, for this $\rho_2 = t'_1, \dots, t'_m$, we have $t'_k \in X_k$, since the output language of t'_k contains v_k. This shows that ρ_2 is ξ-compatible.

For the converse implication, let $\rho_1 = t_1 \dots t_m$ be a successful run of T_1 on u, with L_k output language of t_k, for all $1 \leq k \leq m$, and consider a possible output $v = (v_1, j_1) \cdots (v_m, j_m)$ produced by ρ_1. We want to show that v can also be produced by a successful run of T_2 on the same input u. According to the description of the witness procedure $\mathcal{W}$, for each k there is a set $X_k \in \mathcal{W}(t_k)$ containing precisely the transitions t' of T_2 that can output v_k and that have the same shape as t_k. Let $\xi = X_1, \dots, X_m$. By the hypothesis of the claim, there is a successful run $\rho_2 = t'_1 \dots t'_m$ of T_2

that is ξ-compatible. This means that the transitions in ρ_2 have the same shapes as those in ρ_1 and in particular, they read the same input letters and can produce the same outputs $v_1, \ldots, v_m$, tagged with the same origins $j_1, \ldots, j_m$. Thus, ρ_2 is over the same input u and enables the same output $v = (v_1, j_1) \cdots (v_m, j_m)$ as ρ_1. □

Next, we reduce the origin-equivalence problem of T_1 and T_2 to the emptiness problem of an NFA A. In this reduction, the NFA A can be exponentially larger than T_1, T_2, but will be PSPACE-constructible under suitable assumptions on T_1, T_2, and $\mathcal{W}$.

Lemma 2.2.3. *Given two busy 2NFTs T_1, T_2 with input alphabet Σ and a witness procedure $\mathcal{W}$, one can construct an NFA A that accepts precisely the words $u \in \Sigma^*$ for which there exist a successful run ρ_1 of T_1 on u and a witness sequence $\xi \in \mathcal{W}(\rho_1)$ such that no ξ-compatible run ρ_2 of T_2 is successful.*

Moreover, if T_1 and T_2 have a total number of states n and $\mathcal{W}$ uses space polynomial in n, then A is PSPACE*-constructible w.r.t. n.*

The proof of the lemma is obtained by adapting the techniques of subset construction and crossing sequences for 2NFA [Var89].

Proof. The goal is to build an NFA A which, on input u, verifies the existence of a successful run $\rho_1 = t_1, \ldots, t_m$ of T_1 on u, and of a witness sequence $\xi \in \mathcal{W}(\rho_1)$, such that there is no successful, ξ-compatible run $\rho_2 = t'_1, \ldots, t'_m$ of T_2 (the fact that ρ_2 is over the same input u as ρ_1 follows from ξ-compatibility). The goal is achieved by processing the input u from left to right, while guessing *sub-runs* of the run ρ_1 of T_1 that are *induced* by prefixes of u. At the same time, A uses the procedure $\mathcal{W}$ to 'guess' the witness sequence ξ. The sequence ξ is used to track induced pieces of runs of T_2 with the same shape as in ρ_1, such that once we choose an output word for every transition of T_1, we are guaranteed to follow all pieces of runs of T_2 that can produce the same output words.

Recall that successful runs start from the cut 1 and end at the cut $n-1$, where n is the length of the input $|u|$. A *sub-run* ρ' of a run $\rho = \rho_1\rho'\rho_2$ is a factor of the run ρ. We say that a sub-run is *induced* by the prefix $u([1, i])$ if the sub-run only reads the letters at the positions $\{1, \ldots, i\}$. Such a sub-run is called *maximal* if the last transition of ρ_1 and the first transition of ρ_2 are not on positions $\{1, \ldots, i\}$. Based on the starting position, we distinguish between *left-to-right* and *right-to-right* maximal sub-runs. A left-to-right maximal sub-run on prefix $u([1, i])$ must start from the initial state and end at the cut i. A right-to-right maximal sub-run starts and ends at from the cut i.

For a run ρ on u, there is a left-to-right induced sub-run and a number(possibly 0) of right-to-right induced sub-runs on prefix $u([1, i])$. The

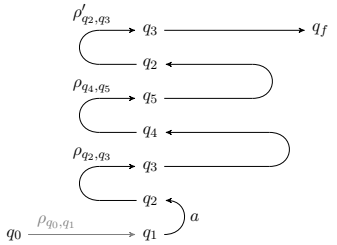

Figure 2.2 – The left-to-right and right-to-right sub-runs at position i.

number of right-to-right runs is not bounded. However, we abstract them by the start and end state in the sub-run. For example, in Figure 2.2, there are 3 right-to-right sub-runs, but they are abstracted by the state-pairs (q_2, q_3) and (q_4, q_5). In particular, the runs ρ_{q_2,q_3} and ρ'_{q_2,q_3} are both considered abstracted by the pair (q_2, q_3). Intuitively, if for the run ρ_1 in Figure 2.2 there is no compatible run of T_2, then this also holds for the run ρ'_1 obtained from ρ_1 by replacing the sub-run ρ'_{q_2,q_3} by ρ_{q_2,q_3}, since they are abstracted by the same pair of states.

The NFA A will guess the abstraction of runs of T_1 induced by longer and longer prefixes of the input. At any point, there can be exactly one left-to-right abstracted runs, and at most $|Q|^2$ right-to-right abstracted runs. At the end, A expects to see only one left-to-right induced run of T_1, that must be successful.

At the same time, A will also maintain the set of all runs of T_2 that are induced by the consumed prefix and that have been constructed using transitions provided by $\mathcal{W}$. Concerning these latter runs, A will check that none of the left-to-right runs of T_2 are successful. In fact, as we will see below, all the guessed runs are abstracted by states or pairs of states, so as to maintain at each step only a bounded, polynomial-size information. This will require some care, since, for instance, the same pair of states can represent many right-to-right runs of T_1, with different shapes.

The key observation is that it is sufficient to associate at most one left-to-right run with each state of T_1, and at most one right-to-right run with each pair of states of T_1. This is because if, say, ρ_1 has two right-to-right runs on some prefix, that start and end, respectively, in the same states, then we can replace one run by the other, and do the same for ρ_2. The property that no ξ-compatible run of T_2 is ultimately successful, will be preserved. This is formalized by the invariant maintained by the NFA A. First, we formally define the NFA A.

The states of the NFA A are tuples consisting of four different objects (note that the size of a state is polynomial in the total number n of states of T_1, T_2):

— a right-reading state $\hat{q}$ of T_1,

— a set P of pairs of left-reading and right-reading states of T_1,

— a set S of right-reading states of T_2,
— a relation R between pairs of states of T_1 and pairs of states of T_2 such that for all $\big((q_1, q_1'), (q_2, q_2')\big) \in R$, q_1 and q_2 are left-reading, and q_1' and q_2' are right-reading.

The intended meaning of the above objects is explained by the following invariant. After reading a prefix $u([1, i])$ of the input, the state $(\hat{q}, P, S, R)$ reached by A must satisfy the following properties:

1. there is a left-to-right run $\rho_{\hat{q}}$ of T_1 induced by $u([1, i])$ and ending in $\hat{q}$, and a witness sequence $\xi_{\hat{q}}$ in $\mathcal{W}(\rho_{\hat{q}})$,
2. for each pair $(q, q') \in P$, there is a right-to-right run $\rho_{q,q'}$ of T_1 that is induced by $u([1, i])$, starts in q, ends in q', and a witness sequence $\xi_{q,q'}$ in $\mathcal{W}(\rho_{q,q'})$,
3. if there is a left-to-right run ρ' of T_2 that is induced by $u([1, i])$ and is $\xi_{\hat{q}}$-compatible (cf. first item), then S contains the last state of ρ',
4. for each pair $(q, q') \in P$, if there is a right-to-right run ρ' of T_2 induced by $u([1, i])$, $\xi_{q,q'}$-compatible (cf. second item), starting in r and ending in r', then $((q, q'), (r, r')) \in R$.

The initial states of A are the tuples of the form $(\hat{q}, P, S, R)$, with $\hat{q} \in I_1$, $P = \emptyset$, and $S = I_2$. Similarly, the final states are the tuples $(\hat{q}, P, S, R)$ such that $\hat{q} \in F_1$, $P = \emptyset$, and $S \cap F_2 = \emptyset$. Assuming that Properties 1.–4. are satisfied, this will imply that A accepts some input u iff there exist a successful run ρ_1 of T_1 on u and a witness sequence $\xi \in \mathcal{W}(\rho_1)$, but no successful, ξ-compatible run of T_2 on u.

We now give the transitions of A that preserve Properties 1.–4. These are of the form

$$(\hat{q}, P, S, R) \xrightarrow{a} (\hat{q}', P', S', R')$$

and must satisfy a certain number of constraints between the various components of the source and target states. We first focus on the constraints between the first two components, i.e. $\hat{q}, P$ and $\hat{q}', P'$, which guarantee Properties 1. and 2. For this, we basically apply a variant of the classical crossing-sequence construction [She59] for simulating a 2NFA by an NFA. In the following we omit output languages in transitions, since they are determined by the source/target state, and the input letter.

— A first condition requires that $\hat{q}$ is connected to $\hat{q}'$ by a sequence of k leftward transitions on a interleaved by k right-to-right induced runs, and followed by a single rightward transition on a, as follows (see also the left hand-side of Figure 2.3):

$$(\hat{q}, a, q_1) \;\; \rho_{q_1,q_2} \;\; (q_2, a, q_3,) \;\; \rho_{q_3,q_4} \;\; \cdots \;\; \rho_{q_{2k-1},q_{2k}} \;\; (q_{2k}, a, \hat{q}')$$

where $(q_1, q_2), (q_3, q_4), \ldots, (q_{2k-1}, q_{2k}) \in P$.
For brevity, we denote this property by $\hat{q} \rightsquigarrow q_1 \rightsquigarrow q_2 \cdots \rightsquigarrow q_{2k} \rightsquigarrow \hat{q}'$.

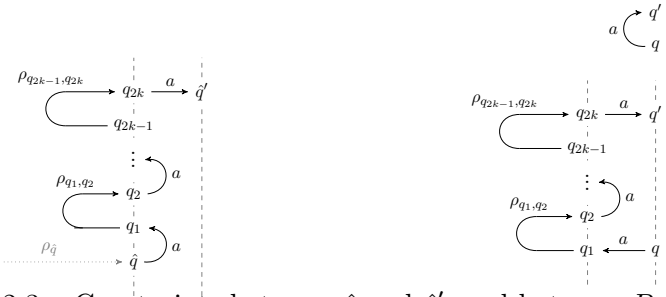

Figure 2.3 – Constraints between $\hat{q}$ and $\hat{q}'$, and between P and P'.

— In a similar way, we require that, for every pair $(q, q') \in P$, either q is connected to q' by a leftward transition on a, or q is connected to q' by a leftward transition on a, then a sequence of $k > 0$ right-to-right induced runs interleaved by $n - 1$ rightward transitions on a, followed by a right-to-left transition on a, or as follows (see the right hand-side of Figure 2.3):

$$(q, a, q_1) \;\; \rho_{q_1,q_2} \;\; (q_2, a, q_3) \;\; \rho_{q_3,q_4} \;\; \cdots \;\; \rho_{q_{2k-1},q_{2k}} \;\; (q_{2k}, a, q')$$

where $(q_1, q_2), (q_3, q_4), \ldots, (q_{2k-1}, q_{2k}) \in P$.

As before, we write for short $q \rightsquigarrow q_1 \rightsquigarrow q_2 \cdots \rightsquigarrow q_{2k} \rightsquigarrow q'$ (to distinguish the notations for the former and the latter property, it suffices to check whether the first state is right-reading or left-reading).

As concerns the constraints between S, R and S', R', we first lift the previous properties and notations to pairs of runs of T_1 and T_2, thus writing, for instance, $\binom{\hat{q}}{s} \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q_1}{s_1} \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q_2}{s_2} \ldots \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q_{2k}}{s_{2k}} \overset{\rightsquigarrow}{\rightsquigarrow} \binom{\hat{q}'}{s'}$. On top of this, we restrict the runs of T_2 to be ξ-compatible with the runs of T_1, for the corresponding witness sequence ξ. Formally, we require the following:

— Assume that $s \in S$ and that R contains the pairs $\big((q_{2i-1}, q_{2i}), (s_{2i-1}, s_{2i})\big)$, for all $1 \leq i \leq k$. Let $t = (\hat{q}, a, q_1)$, $t' = (q_{2k}, a, \hat{q}')$, and $t_i = (q_{2i-1}, a, q_{2i})$, for all $2 \leq i \leq k$, be the transitions of T_1 that are used to connect $\hat{q}$ to q_1, q_{2k} to $\hat{q}'$, etc.

Using $\mathcal{W}$, choose some witness sets $X \in \mathcal{W}(t)$, $X' \in \mathcal{W}(t')$, and $X_i \in \mathcal{W}(t_i)$, for $2 \leq i \leq k$. Then $s' \in S'$ if

$$\binom{\hat{q}}{s} \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q_1}{s_1} \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q_2}{s_2} \ldots \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q_{2k}}{s_{2k}} \overset{\rightsquigarrow}{\rightsquigarrow} \binom{\hat{q}'}{s'}$$

for some $(s, a, s_1) \in X$, $(s_{2k}, a, s') \in X'$, $(q_{2i-1}, a, q_{2i}) \in X_i$ $(2 \leq i \leq k)$.

— Assume that $(q, q') \in P'$ and R contains the pairs $\big((q_{2i-1}, q_{2i}), (s_{2i-1}, s_{2i})\big)$, for all $1 \leq i \leq k$. As before, let t, t', t_i be the transitions of T_1 that connect q to q_1, q_{2k} to q', q_{2i} to q_{2i+1}, for all $1 \leq i \leq k$.

Using $\mathcal{W}$, choose some witness sets $X \in \mathcal{W}(t))$, $X' \in \mathcal{W}(t')$, and $X_i \in \mathcal{W}(t_i)$, for $1 \leq i < k$. Then $\big((q, q'), (s, s')\big) \in R'$ if

$$\binom{q}{s} \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q_1}{s_1} \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q_2}{s_2} \ldots \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q_{2k}}{s_{2k}} \overset{\rightsquigarrow}{\rightsquigarrow} \binom{q'}{s'}$$

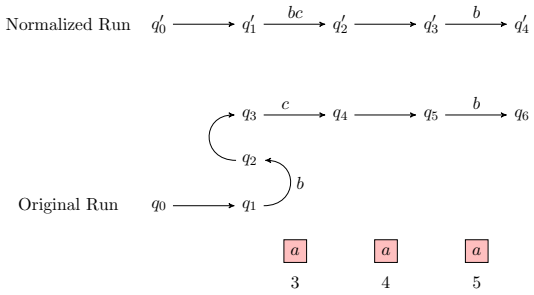

Figure 2.4 – Normalizing runs.

for some $(s, a, s_1) \in X$, $(s_{2k}, a, s') \in X'$, $(s_{2k}, a, s_{2k+1}) \in X_i$ $(1 \leq i < k)$.

It is routine to check that the above constraints guarantee all the Properties 1.–4. stated above. To conclude, we observe that the NFA $\mathcal{A}$ can be constructed by an algorithm that uses the PSPACE sub-procedures for enumerating the transitions of T_1, T_2 and the non-deterministic PSPACE procedure $\mathcal{W}$, plus additional polynomial space for storing (temporarily) the sets X, X', X_i. □

Now, we are ready to show decidability of origin-equivalence for busy 2NFT. By Proposition 2.2.2, $T_1 \subseteq_o T_2$ for busy 2NFT T_1 and T_2 amounts to checking emptiness of the NFA $\mathcal{A}$ from Lemma 2.2.3. The latter problem can be decided in PSPACE w.r.t. n, by using the PSPACE-constructibility of $\mathcal{A}$ to enumerate all transitions departing from any given state. As a consequence, we have the following result.

Corollary 2.2.3.1. *Given two busy 2NFT T_1, T_2 with a total number n of states, and given a witness procedure that uses space polynomial in n, the problem of deciding $T_1 \subseteq_o T_2$ is in* PSPACE *w.r.t. n.*

From arbitrary transducers to busy transducers

We now consider 2NFT that are not necessarily busy. We modify the 2NFT and make them busy, thus reducing the origin-equivalence problem to origin-equivalence of busy 2NFT. This is where the notion of PSPACE-constructibility will be exploited.

A naive idea would be to modify the transitions that output the empty word ε and make them output a special letter #. This however would not give a correct reduction towards origin-equivalence with busy 2NFT. Indeed, a 2NFT may produce non-empty outputs, say $v_1, v_2, \ldots$, with transitions that occur at the same position, say i, and traversing other positions of the input in between but producing only ε. This is illustrated in Figure 2.4, where there are consecutive output positions b and c produced at the same origin, but the run traverses other positions between producing b and c.

The above idea is however useful if we first *normalize* the 2NFT in such a way that maximal sub-runs generating empty outputs follow the shortest path in the input. This allows the transducer to visit positions in the input in the order they occur as origins in the output. For example, in Figure 2.4, the origins of consecutive output positions are $3, 3, 5$. The normalized transducer will move directly from input position 3 to 5 with a sequence of right moving transitions. Recall we assume in the definition of 2NFT that there are no *stay* transitions, i.e, transitions that do not move the reading head. Therefore, a contiguous block of outputs with the same origin must be produced in a single transition, such as a transition (q_1', a, bc, q_2') in the example of Figure 2.4. To do this, we need to use 2NFT where the output of a transition could be a regular language. Paired with the fact that the same input positions are visited in the order in which they occur as origins, this will give the following characterization: two arbitrary transducers are origin-equivalent if and only if their normalized versions, with empty outputs replaced by #, are also origin-equivalent.

We now describe how the normalization procedure works. Consider a 2NFT $T = (Q, \Sigma, \Gamma, \Delta, I, F)$. To normalize T we consider runs that start and end in the same cut of the input, and that produce empty output. We call such runs *lazy U-turns*, and are formally defined below.

Given an input word u, a *left* (resp. *right*) *lazy U-turn at position i of u* is any run of T on u of the form

$$(q_1, i_1) \xrightarrow{a_1|v_1} (q_2, i_2) \xrightarrow{a_2|v_2} \cdots \xrightarrow{a_m|v_m} (q_{m+1}, i_{m+1})$$

with $i_1 = i_{m+1} = i$, $i_k < i$, (resp. $i_k > i$) for all $2 \leq k \leq m$, $a_1, \ldots, a_m$ are input positions $\leq i$ (resp. $> i$) and $v_k = \varepsilon$ for all $1 \leq k \leq m$.

The pair (q_1, q_{m+1}) of states at the extremities of a left/right lazy U-turn is called a *left/right U-pair* (*at position i of u*). We denote by $U_i^{\circlearrowright}$ (resp. $U_i^{\circlearrowleft}$) the set of all left (resp. right) U-pairs at position i

Note that we have $U_i^{\circlearrowright} \subseteq Q_\prec \times Q_\succ$ and $U_i^{\circlearrowleft} \subseteq Q_\succ \times Q_\prec$. Accordingly, we define the word $u^{\circlearrowright}$ over $2^{Q_\prec \times Q_\succ}$ that has the same length as u and labels every position i with the set $U_i^{\circlearrowright}$ of left U-pairs. This $u^{\circlearrowright}$ is seen as an annotation of the original input u with the left U-pairs, and can be computed from $T = (Q, \Sigma, \Gamma, \Delta, I, F)$.

The set of lazy U-turns can be defined by a recursive rule. A pair of states $(q, q') \in u^{\circlearrowright}(i)$, if and only if,

$$q \in Q_\prec \ \wedge \ \big(q, u(i-1), \varepsilon, q', \mathit{right}\big) \in \Delta$$

$$\text{or} \quad q \in Q_\prec \ \wedge \ \exists\, (q_1, q_1'), \ldots, (q_k, q_k') \in u^{\circlearrowright}(i-1)$$

$$\begin{cases} \big(q, u(i-1), \varepsilon, q_1, \mathit{left}\big) \in \Delta \\ \big(q_j', u(i-1), \varepsilon, q_{j+1}, \mathit{left}\big) \in \Delta \quad \forall 1 \leq j \leq k \\ \big(q_k', u(i-1), \varepsilon, q_k, \mathit{right}\big) \in \Delta. \end{cases}$$

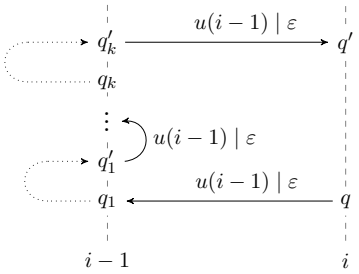

Figure 2.5 – Lazy U-turns.

This is illustrated in Figure 2.5. The annotation $u^{\circlearrowleft}$ with right U-pairs satisfies a symmetric recursive rule.

The *annotated input* $u \otimes u^{\circlearrowright} \otimes u^{\circlearrowleft}$ is the word in $(\Sigma \times 2^{Q_\prec \times Q_\succ} \times 2^{Q_\succ \times Q_\prec})^*$ where the i-th letter is $(u(i), U_i^{\circlearrowright}, U_i^{\circlearrowleft})$, i.e, the original input letter annotated with the U-pairs. The above recursive rules allow an NFA to check the annotations.

Lemma 2.2.4. *Given a transducer T, one can compute an NFA $\mathcal{U}$ such that $[\![\mathcal{U}]\!] = \{u \otimes u^{\circlearrowright} \otimes u^{\circlearrowleft} : u \in \Sigma^*\}$. Furthermore, the NFA $\mathcal{U}$ is* PSPACE*-constructible w.r.t. the number of states of T.*

Proof. The states of the NFA $\mathcal{U}$ will be $2^{Q_\prec \times Q_\succ} \times 2^{Q_\succ \times Q_\prec}$. Intuitively, the states will be used to guess the set of lazy left-to-right U-turns and the set of right-to-left U-turns. Fixing a set of lazy left-to-right U-turns at position i, the set of lazy left-to-right U-turns at position $i+1$ is determined by the recursive definition of lazy U-turns.

The states are used to guess the set of lazy left-to-right U-turns at a *cut*. Recall that a cut corresponds to the border of two consecutive positions of a word. For a word u of length n with positions 1 to n, there are $n+1$ cuts from 0 to n. The set of lazy U-turns at a cut i is therefore the pair $(u_{i-1}^{\circlearrowright}, u_i^{\circlearrowleft})$, i.e, the set of lazy left-to-right U-turns at position $i-1$ and lazy right-to-left U-turns at position i.

The transitions are then used to check validity of these guesses. The transitions also check that the annotations match with the states in the correct way, i.e, for a transition $(X, Y) \xrightarrow{(a, u^{\circlearrowright}, u^{\circlearrowleft})} (X', Y')$, it should be the case that $X = u^{\circlearrowright}$ and $Y' = u^{\circlearrowleft}$. Furthermore, the sets (X, Y) and (X', Y') must satisfy the conditions of the recursive definition of lazy U-turns (see Figure 2.5). The conditions for the sets X and X' are: a pair of states (q, q')

is in X' iff

$$q \in Q_{\prec} \ \wedge \ \big(q, a, \varepsilon, q', \mathit{right}\big) \in \Delta$$

$$\text{or} \quad q \in Q_{\prec} \ \wedge \ \exists\, (q_1, q_1'), \ldots, (q_k, q_k') \in X$$

$$\begin{cases} \big(q, a, \varepsilon, q_1, \mathit{left}\big) \in \Delta \\ \big(q_j', a, \varepsilon, q_{j+1}, \mathit{left}\big) \in \Delta \quad \forall 1 \leq j \leq k \\ \big(q_k', a, \varepsilon, q_k, \mathit{right}\big) \in \Delta. \end{cases}$$

The sets of lazy right-to-left U-turns are also checked in the similar rule.

At the first and last cut , there are no possible U-turns, since the transitions on $\vdash$ and $\dashv$ do not allow U-turns. Therefore, the set of initial and final states of $\mathcal{U}$ is $(\emptyset, \emptyset)$.

By definition of U-turns, the NFA $\mathcal{U}$ accepts correctly annotated words. Furthermore, the transitions are constructed by checking polynomially many transitions of T. Therefore, the NFA $\mathcal{U}$ is PSPACE-constructible in the number of states of T. □

We extend the 2NFT T to T_U which works on the annotated inputs. This can be obtained by replacing any transition (q, a, L, q') by $(q, (a, U^{\circlearrowright}, U^{\circlearrowleft}), L, q')$. Note that the transducer T_U does not check if the annotations is correct. However by Lemma 2.2.4, this can be done by the automaton $\mathcal{U}$. The number of transitions increases exponentially in T_U compared to T as the input alphabet grows exponentially. However, the states and transitions of T_U can be enumerated using space polynomial in the size of T since each new input letter has size polynomial in the size of T.

The normalization of T_U, which produces an origin-equivalent transducer $\mathsf{Norm}(T_U)$ with no lazy U-turns works in two steps. First, using the information provided by the annotation of the input, we shortcut all runs of T_U that consist of multiple transitions outputting at the same position and interleaved by lazy U-turns. The resulting transducer is denoted $\mathsf{Shortcut}(T_U)$. After this step, we will eliminate the lazy U-turns, thus obtaining $\mathsf{Norm}(T_U)$.

Formally, $\mathsf{Shortcut}(T_U)$ has for transitions the tuples of the form $\big(q, (a, U^{\circlearrowright}, U^{\circlearrowleft}), L, q', d\big)$, where L is the smallest language that contains every language of the form $L_1 \cdot L_2 \cdots L_k$ for which there are $q_1, q_1', \ldots, q_k, q_k'$ and $d_1, \ldots, d_k$, with $q = q_1$, $q_k' = q'$ (and hence $d_k = d$), $(q_i, a, L_i, q_i', d_i) \in \Delta$, and $(q_i', q_{i+1}) \in U^{\circlearrowright} \cup U^{\circlearrowleft}$ for all i.

Note that there is no transition $\big(q, (a, U^{\circlearrowright}, U^{\circlearrowleft}), L, q', d\big)$ when there are no languages $L_1, L_2, \ldots, L_k$ as above.

The output languages associated with the transitions of $\mathsf{Shortcut}(T_U)$ can be constructed using a classical saturation mechanism. These are regular languages, and their NFA representations are polynomial-sized w.r.t. the size of the NFA representations of the output languages of T_U. This implies

that $\mathsf{Shortcut}(T_U)$ has PSPACE-constructible transitions w.r.t. the number of its states.

Lemma 2.2.5. *Let $w = u \otimes u^{\circlearrowright} \otimes u^{\circlearrowleft}$ be an correctly annotated input. Every successful run of T_U on w is origin-equivalent to some successful run of $\mathsf{Shortcut}(T_U)$ on w without lazy U-turns. Conversely, every successful run of $\mathsf{Shortcut}(T_U)$ on w is origin-equivalent to some successful run of T_U on w.*

Proof. Consider an arbitrary run of T_U on input $w = u \otimes u^{\circlearrowright} \otimes u^{\circlearrowleft}$:

$$\rho \;=\; (q_1, i_1) \xrightarrow{b_1|v_1} (q_2, i_2) \xrightarrow{b_2|v_2} (q_3, i_3) \cdots \xrightarrow{b_m|v_m} (q_{m+1}, i_{m+1}).$$

The above run can also be seen as a run of $\mathsf{Shortcut}(T_U)$, since for every transition of T_U outputting v there is a similar transition of $\mathsf{Shortcut}(T_U)$, between the same states and outputting the same word v. We prove the claim by induction on the number of lazy U-turns in ρ. If ρ has no lazy U-turn, then the claim follows trivially. Now, for the inductive step, suppose that the factor that starts in (q_k, i_k) and ends in (q_h, i_h) is a maximal left lazy U-turn, which thus occurs at position $i = i_k = i_h$ (the case of a right lazy U-turn case is symmetric). By definition of U-turn, we know that q_k is left-reading and q_h is right-reading, and hence the transitions that immediately precede and follow the U-turn read the same input letter, i.e. $w(i) = b_{k-1} = b_h$, and output respectively the words v_{k-1} and v_h. By construction, $\mathsf{Shortcut}(T_U)$ admits a transition t that moves from configuration (q_{k-1}, i_{k-1}) to configuration (q_h, i_h), reading $w(i)$ and producing $v_{k-1} \cdot v_h$ as output. We can replace the U-turn of ρ and the surrounding transitions with the latter transition t, thus obtaining an origin-equivalent run of $\mathsf{Shortcut}(T_U)$ with a smaller number of lazy U-turns. This proves the inductive step, and thus the first claim of the lemma.

For the second claim, consider an arbitrary run of $\mathsf{Shortcut}(T_U)$ on w:

$$\rho \;=\; (q_1, i_1) \xrightarrow{b_1|v_1} (q_2, i_2) \xrightarrow{b_2|v_2} (q_3, i_3) \cdots \xrightarrow{b_m|v_m} (q_{m+1}, i_{m+1}).$$

Consider a transition along ρ, say $t : (q_k, i_k) \xrightarrow{b_k|v_k} (q_{k+1}, i_{k+1})$. By construction, there must be a sequence of states $r_1, r'_1, \ldots, r_h, r'_h$ and directions $d_1, \ldots, d_h$, for some $h \geq 2$, such that $q_k = r_1$, $q_{k+1} = r'_h$, $d_h = d$, $(r_j, a, L_j, r'_j, d_j) \in \Delta$ and $(r'_j, r_{j+1}) \in U^{\circlearrowright} \cup U^{\circlearrowleft}$ for all j, and $v_k \in L_1 \cdot L_2 \cdots L_h$. In particular, T_U admits some transitions of the form $(q_k, i_k) = (r_1, j_1) \xrightarrow{a|v'_1} (r'_1, j'_1)$, $(r_2, j_2) \xrightarrow{a|v'_2} (r'_2, j'_2)$, $\ldots$, $(r_h, j_h) \xrightarrow{a|v'_h} (r'_h, j'_h) = (q_{k+1}, i_{k+1})$, with $j'_2 = j_3$, $\ldots$, $j'_{h-1} = j_h$, and $v_k = v'_1 \cdot v'_2 \cdots v'_h$. The latter transitions, interleaved with some U-turns corresponding to the U-pairs $(r'_1, r_2), \ldots, (r'_{h-1}, r_h) \in U^{\circlearrowright} \cup U^{\circlearrowleft}$, form a valid run ρ'_t of T_U on w, which connects (q_k, i_k) to (q_{k+1}, i_{k+1}) and outputs v_k. This means that in the run ρ we can safely replace the transition t by the run ρ'_t. Doing this simultaneously for all transitions in ρ results in an origin-equivalent run of T_U. □

Lemma 2.2.5 allows to remove all the lazy U-turns without changing the semantics of the transducer. This is the final step to obtain the normalized transducer. This is done by simply forbidding the shortest possible lazy U-turns, namely, the transitions that output ε and that remain on the same input position. Since every lazy U-turn contains a transition of the previous form, forbidding this type of transitions results in forbidding arbitrary lazy U-turns. Formally, we construct from $\mathsf{Shortcut}(T_U)$ a new transducer $\mathsf{Norm}(T_U)$ by replacing every transition rule (q, a, L, q', d) with (q, a, L', q', d), where L' is either L or $L \setminus \{\varepsilon\}$, depending on whether $q \in Q_\prec \Leftrightarrow q' \in Q_\prec$ or not. The following lemma follows by construction of (T_U) and Lemma 2.2.5.

Lemma 2.2.6. *T_U and $\mathsf{Norm}(T_U)$ are origin-equivalent when restricted to correctly annotated inputs, i.e.: $[\![\mathcal{T}_U]\!]_o \cap R = [\![\mathsf{Norm}(\mathcal{T}_U)]\!]_o \cap R$, where $R = [\![\mathcal{U}]\!] \times (\Gamma \times \mathbb{N})^*$.*

The final step to reduce the origin-equivalence problem from the general case to the busy case is to make the transducer $\mathsf{Norm}(T_U)$ busy. This is done by replacing the empty output ε in a transition by a special character $\# \notin \Gamma$. We call this transducer $\mathsf{Busy}(T_U)$. Again, the states of $\mathsf{Busy}(T_U)$ are the same as those of T, and its transitions are PSPACE-constructible in the number of its states.

The proposition below follows immediately from Lemma 2.2.6, and reduces origin-containment between T_1 and T_2 to origin-containment between $\mathsf{Busy}(T_{1,U})$ and $\mathsf{Busy}(T_{2,U})$, but now relativized to correctly annotated inputs, where $\mathsf{Busy}(T_{i,U})$ denotes the transducer obtained by applying the Busy construction to T_i.

Proposition 2.2.7. *Given two transducers T_1 and T_2,*

$$T_1 \subseteq_o T_2 \quad \text{if and only if} \quad \mathsf{Busy}(T_{1,U}) \cap R \subseteq_o \mathsf{Busy}(T_{2,U}) \cap R.$$

where $R = [\![\mathcal{U}']\!] \times (\Gamma \times \mathbb{N})^*$ and $\mathcal{U}'$ is an NFA that recognizes inputs annotated with left/right U-pairs of both T_1 and T_2.

It remains to show that, given the transducers T_1, T_2, there is a PSPACE witness procedure for $\mathsf{Busy}(T_{1,U}), \mathsf{Busy}(T_{2,U})$:

Proposition 2.2.8. *Let T_1, T_2 be transducers with a total number n of states, and $\mathsf{Busy}(T_{i,U}) = (Q_i, \hat{\Sigma}, \Gamma, \Delta_i, I_i, F_i)$ for $i = 1, 2$ (the input alphabet $\hat{\Sigma}$ is the same as for $\mathcal{U}'$). There is a non-deterministic procedure $\mathcal{W}$ that works in polynomial space in n and returns on a given transition $t_1 = (q_1, a_1, q_1', L_1, d_1)$ of $\mathsf{Busy}(T_{1,U})$ any set $X \subseteq \Delta_2$ of transitions of $\mathsf{Busy}(T_{2,U})$ such that for some $v \in L_1$:*

$$X = \{t_2 = (p_2, a_2, q_2, L_2, d_2) \in \Delta_2 : v \in L_2, \text{ and } t_2 \text{ has same shape as } t_1\}.$$

Proof. Given transition $t_1 = (q_1, a, q_1', L_1, d_1)$ of $\mathsf{Busy}(T_{1,U})$, we first use the PSPACE enumeration procedure of the transitions of $\mathsf{Busy}(T_{2,U})$ to generate the set Z of all $t_2 = (q_2, a, q_2', L_2, d_2) \in \Delta_2$ with the same shape as t_1. Note that Z has polynomial size, and its representation is polynomial as well. The procedure $\mathcal{W}$ guesses first a subset Z_0 of Z. Then it guesses on-the-fly a word $v \in L_1$, and verifies that v belongs to the output language of every $t_2 \in Z_0$, and to no output language of any $t_2 \in (Z \setminus Z_0)$. Recall that $\mathsf{Busy}(T_{1,U})$ and $\mathsf{Busy}(T_{2,U})$ have PSPACE-constructible transitions w.r.t. n, and thus, by definition, the output languages are represented by NFA of size polynomial in n. Basically $\mathcal{W}$ checks a condition of the form $v \in \bigcap_i \mathcal{D}_i \setminus \bigcup_j \mathcal{D}'_j$, for polynomially many NFA $\mathcal{D}_i, \mathcal{D}'_j$ of size polynomial in n, which can be done in PSPACE. If successful, then $\mathcal{W}$ returns the set Z_0. □

Proof of Theorem 2.2.1. By Proposition 2.2.7, origin-containment for 2NFT can be reduced to origin-containment for busy 2NFT. Furthermore, the states and transitions of the busy 2NFT $\mathsf{Busy}(T_{i,U})$ can be enumerated in space polynomial in the size of T_i. The witness procedure for $\mathsf{Busy}(T_{1,U})$ and $\mathsf{Busy}(T_{2,U})$ can also be constructed in PSPACE by Proposition 2.2.8. Therefore, by Corollary 2.2.3.1, origin-containment can be checked in PSPACE.

Origin-equivalence can be checked by checking origin-containment in both directions. This concludes the proof of Theorem 2.2.1.

The lower bound of PSPACE-hard can be obtained by a reduction from the classical equivalence of NFA. □

Therefore, the origin-equivalence of 2NFT is PSPACE-complete.

2NFT with common guess. Recall that 2NFT with common guess (T, ϕ) colours the input positions non-deterministically according to the MSO-formula ϕ and runs the transducer T on the coloured word. Formally, $T = (Q, \Sigma \times \mathcal{C}, \Gamma, \Delta, I, F)$ is a transducer which takes as input words $w \in (\Sigma \times \mathcal{C})^*$ that satisfy the MSO-formula ϕ (see page 12).

Theorem 2.2.1 also holds for 2NFT with common guess. The proof works by changing Lemma 2.2.4, to have $\mathcal{U}$ check that the colouring satisfies ϕ. Assuming ϕ is given as an NFA B, the NFA $\mathcal{U}$ will be PSPACE-constructible in the size of (T, B). Therefore, the origin-equivalence problem for 2NFT with common guess is also PSPACE-complete when the *guess* is given as an NFA.

2.3 Streaming String Transducers

The decidability of origin-equivalence for NSSTs can be obtained by using the characterization of origin graphs produced by NSSTs, which was studied in [BDGP17]. It was shown that the set of origin graphs generated by an NSST is MSO-definable.

Theorem 2.3.1 ([BDGP17]). *The set of origin graphs generated by an NSST T is MSO-definable.*

Recall that the formalism of synchronized pair and origin graphs are equivalent. Over the logical signature of origin graph, which would consist of input and output successor and label predicates, and the origin relation, one can define MSO formulas describing sets of origin graphs.

Therefore, given an NSST T, one can construct an MSO formula ϕ_T describing the set of origin graphs produced by T. It is easy to see that $T_1 \not\subseteq_o T_2$ if and only if there exists an origin graph generated by T_1 which satisfies $\neg\phi_{T_2}$. Therefore, origin-containment reduces to MSO-satisfiability problem over origin graphs, which was shown to be decidable in [BDGP17]. This gives a algorithm for deciding equivalence of NSSTs. Decidability of origin-equivalence was also shown for NMSO tree-to-string transducers [FMRT15], which are a generalization of NSSTs. However, both of the above procedures does not provide any result on the complexity of the problem. We provide in this section a procedure to check containment which works on NSST directly, so without converting to a MSO formula, and show that the origin-equivalence problem for NSST is decidable in ExpSpace. We also obtain a lower bound of PSpace since checking equivalence of the underlying automaton is PSpace-hard.

Theorem 2.3.2. *The origin-equivalence problem for:*
— *NSST is in* ExpSpace *and is* PSpace*-hard;*
— *DSST is in* PSpace *and is* NlogSpace*-hard.*

Before giving the proof, we introduce some definitions and notations used therein. Recall that a run ρ of an NSST T generates a synchronized pair (u, v).

As a running example, we consider the *invert relation* which takes as input a string u and breaks it non-deterministically into a prefix p, a suffix s and the middle part m, i.e, $u = pms$ and reverses the middle part, i.e, outputs $pm^r s$. It is a relational transduction since there are multiple outputs for the same input, based on how the input is split.

Figure 2.6 shows two NSST that implement this relation. The first transducer T_1 uses three register r_p, r_m and r_s to treat the prefix, middle and the suffix respectively. The second transducer T_2, on the other hand, uses only two registers.

Flow tree of a run. We represent an update to a register r as a tree of height 1, where the root is labeled by r and the children are labeled either by a register or an output letter such that the yield of the tree is the right-hand side of the update. In this way, a transition can be represented as a forest with each tree representing the update to a register. Note that this may

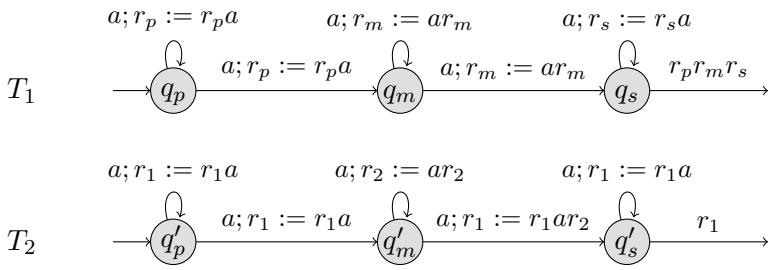

Figure 2.6 – Two NSST defining the invert relation

create multiple copies of the same output letter if it occurs more than once in the right-hand side of the update.

We can extend this representation to an accepting run ρ of an NSST T, where the sequence of register updates is $up_1,\ up_2,\ \ldots,\ up_n, out$ reaching the final state q_f and outputting $out(q_f)$. The run ρ is represented by a *flow tree* with root labeled by out, representing the final output. The children of the out node are the registers that appear in the right-hand side of the update $out(q_f)$. The *level* of the root out is $n+1$ and the children are at level n. For a node at level j labeled by register r, its children are at level $j-1$ and represent the right-hand side of the update $up_j(r)$. Intuitively, a node at level i labeled by register r corresponds to the content of r after reading the i-th letter of the input. Figure 2.7 gives an example of the flow tree for a run.

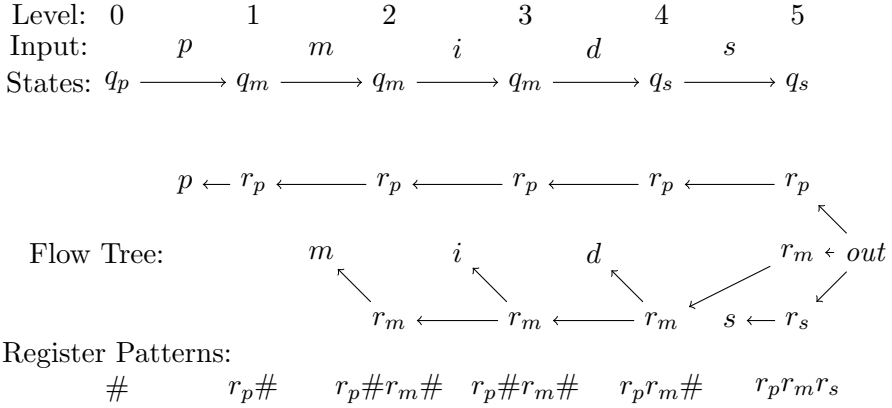

Figure 2.7 – Flow tree of the final output register of T_1

We now define *register patterns*, which are a way to represent the flow tree of a run in a level-by-level manner. A register pattern is a string over the registers and a special separator symbol #, i.e, a string over the alphabet $R \cup \{\#\}$, such that there are no consecutive #s. Additionally, no register

occurs more than once in the pattern, since we assume the NSST to be copyless.

The register pattern at level i, denoted P_i, is defined as the projection of the i-th level of the flow tree onto R, where exactly one # is inserted between two registers r, r' if there is some output with origin greater or equal to i between the current contents of r and r' in the final output. By convention, a # can also occur at the beginning (or the end) of the pattern, if there is an output position with origin greater than or equal to i before (after) the current contents of the register first (last) register in the pattern.

For example, in the pattern at level 3 in Figure 2.7, r_p and r_m are separated by # because the d in the output has origin 4 and is in between the contents of r_p and r_m.

The notion of updates can be lifted to register patterns in the natural way. Given a pattern P and a register update up, define $up(P)$ to be the string obtained by replacing each register r in P by $up(r)$. Note that $up(P)$ can contain output letters, registers and separator #. We call a register pattern P and a register update up to be *compatible* if

- $up(P)$ does not have any factor of the form rwr', where $w \in \Gamma^+$, i.e, the update does not add any output symbols between two consecutive registers not separated by #.
- $up(r)$ for any register r that occurs in P is non-empty. This ensures that in the flow tree, nodes labeled by registers have a parent.

We now, define a function $drop_\Gamma$ which maps a string s from $(R \cup \{\#\} \cup \Gamma)^*$ to the string obtained by replacing each maximal factor from $(\Gamma \cup \{\#\})^+$ in s by #. It is easy to see that for any register pattern P and update up, $drop_\Gamma(up(P))$ will be a register pattern. However, multiple register patterns P can yield the same pattern after applying the same up followed by $drop_\Gamma$.

Similarly, we define a function $drop_R$ which removes all the maximal factors from R^+ and replaces it by another special separator \$. Therefore, $drop_R(up(P))$ keeps the letters added by the update, keeps the # and replaces the registers by \$. Intuitively, \$ corresponds to outputs which are already stored in registers and # corresponds to outputs yet to be added to the registers.

Register patterns for a run. The register patterns P_i at level i are defined inductively from the end of the run. Let $up_1, up_2, \ldots, up_n, out$ be the sequence of register updates for a run, where n is the length of the input and out is the output update. We define P_n, the register pattern at level n, to be $drop_\Gamma(out(q_f))$ where q_f is the final state reached in the run. For the level i, the pattern P_i is $drop_\Gamma(up_i(P_{i+1}))$. We give an example of a run and the corresponding flow trees and register patterns in Figure 2.7. Note that given a run ρ of an NSST, the register patterns and states at each level of the run are fixed. The states and register patterns along

with the transitions encode the run and therefore by extension, the output with origin information. Furthermore, because of the copyless restriction of the updates of the NSST, a register occurs at most once in any register pattern corresponding to a level in a run. Therefore, the patterns are of size polynomial in the number of registers.

Proof of Theorem 2.3.2. The main idea of the proof is to build an automaton which uses the register patterns of T_1 and T_2 to find a witness of non-inclusion. If the automaton is unable to find a witness, i.e, the language is empty, then the inclusion holds.

Lemma 2.3.3. *Given NSST T_1 and T_2, there exists an NFA $\mathcal{A}$ such that $T_1 \subseteq_o T_2$ if and only if $[\![\mathcal{A}]\!]$ is empty. Furthermore, the number of states and transitions of $\mathcal{A}$ are exponential with respect to the size of T_2 and polynomial in the size of T_1. If T_2 is deterministic, then $\mathcal{A}$ has size polynomial in size of T_1 and T_2.*

Proof. Let $T_i = (Q_i, \Sigma, \Gamma, R_i, I_i, F_i, out_i, \Delta_i)$ for $i = 1, 2$. The NFA $\mathcal{A}$ needs to verify the existence of a successful run $\rho_1 = t_1 t_2 \dots t_n$ of T_1 on some word u, such that there is no origin-equivalent, accepting run of T_2 on u.

The NFA $\mathcal{A}$ reads the word u from left to right, guesses the register pattern at each step of ρ_1, and tracks all the runs of T_2 that have *similar* register patterns, where *similarity* of patterns is defined by having equal number of occurrences of #. The transitions will maintain the invariant that register patterns of ρ_2 substituted with the current origin-valuation of the registers are the same as that of ρ_1.

Construction of $\mathcal{A}$: The NFA $\mathcal{A}$ is the tuple $(Q, \Sigma, I, F, \Delta)$ where

- $Q = (Q_1 \times \mathcal{P}_1) \times 2^{Q_2 \times \mathcal{P}_2}$, where $\mathcal{P}_i$ is the set of register patterns over registers R_i, for $i = 1, 2$.
- $I = (I_1 \times \{\#\}) \times 2^{I_2 \times \{\#\}} \cup (I_1 \times \{\varepsilon\}) \times 2^{I_2 \times \{\varepsilon\}}$.
- $F = \{((q_1, P_1), S) \mid q_1 \in F_1,\ P_1 = out_1(q_1)$, and for every $(q_2, P_2) \in S$, either $q_2 \notin F_2$, or $P_2 \neq out_2(q_2)\}$.
  The final state checks that the state and register pattern for the first NSST corresponds to an accepting run of T_1 and all the matching runs of T_2 are either not accepting or produce different outputs.
- The set Δ contains any transition of the form $((q_1, P_1), S) \xrightarrow{a} ((q_1', P_1'), S')$ if
  1. there exists a transition (q_1, a, q_1', up_1) in Δ_1 such that up_1 and P_1' are compatible and $P_1 = drop_\Gamma(up_1(P_1'))$.
  2. for every $(q_2, P_2) \in S$, and for every transition $(q_2, a, q_2', up_2) \in \Delta_2$, there exists $(q_2', P_2') \in S'$ such that
     - up_2 and P_2' are compatible and $P_2 = drop_\Gamma(up_2(P_2'))$
     - $drop_R(up_1(P_1')) = drop_R(up_2(P_2'))$ (cf. item 1).

The first condition on the transition checks that the pair of state and register pattern of T_1 stored in the first and second component of the state of $\mathcal{A}$ extends the run of T_1 that is being tracked with a transition t_1 of T_1. The second condition checks that the set of pairs of state and register pattern of T_2, S' contains all possible extensions of pairs $(q_2, P_2) \in S$ by a transition t_2 with update up_2 that is compatible with P_2. Furthermore, the register updates of t_1 and t_2 are similar and append the same output that have with origins at the current letter a. This maintains the following invariant.

Given a pattern P and a transition $t = (q, a, q', up) \in \Delta_i$, we say a pattern P' is reachable from P on t if $P = drop_\Gamma(up(P'))$. This can be extended to define reachable patterns for a run ρ starting from the patterns $\#$, or ε.

Invariant maintained by runs of $\mathcal{A}$:

Lemma 2.3.4. *A run ρ of $\mathcal{A}$ reaches state $((q_1, P_1), S)$ after reading $u([1, i])$ if and only if*

- *there exists a run ρ_1 of T_1 which reaches the state and register pattern (q_1, P_1) after reading the prefix $u([1, i])$ with some valuation val_1, and*
- *if there exists a run ρ_2 on $u([1, i])$ of T_2 reaching state q_2 and pattern P_2 with valuation val_2, such that $val_2(P_2) = val_1(P_1)$, where val_1 is the valuation obtained from ρ_1, then $(q_2, P_2) \in S$.*

Proof. This can be proved inductively. This is clearly true at level 0 since the valuations val_1 and val_2 are empty and the patterns are either both $\#$ or both ε.

Suppose the NFA $\mathcal{A}$ reaches the states $((q_1, P_1), S)$ and $((q_1', P_1'), S')$ after reading the prefix $u([1, i])$ and $u([1, i+1])$ respectively. Let ρ_1 be a run of T_1 that reaches states, register patterns and valuations q_1, P_1, val_1 after reading $u([1, i])$. By definition, there exists a transition extending the run reaching state q_1', pattern P_1', and some valuation val_1' after reading $u([1, i + 1])$.

Let P_2' be a register pattern such that ρ_2 is a partial run of T_2 which reaches states and valuations q_2', val_2' after reading $u[1, i+1]$ and $val_1'(P_1') = val_2'(P_2')$. We need to show that $(q_2', P_2') \in S'$.

Let the state and valuation reached after reading $u([1, i])$ in ρ_2 be q_2, val_2. Let up_1 and up_2 be the updates corresponding to the $(i + 1)$-th transitions of ρ_1 and ρ_2.

Since $val_1'(P_1') = val_1(up_1(P_1'))$ and $val_2'(P_2') = val_2(up_2(P_2'))$, this implies $val_1(P_1) = val_2(P_2)$, since P_1 and P_2 are sub-strings of the arguments above obtained by applying $drop_\Gamma$ which corresponds to positions with origin $i + 1$. Similarly, $drop_R(up_1(P_1')) = drop_R(up_2(P_2'))$ as they can be obtained by replacing positions with origin less than i by \$ from $val_1(up_1(P_1'))$ and $val_2(up_2(P_2'))$ and all the output letters have origin $i + 1$. Therefore they are origin equivalent.

By definition of the transition of $\mathcal{A}$, $(q_2', P_2') \in S'$ as there exists a transition in Δ_2, the $(i+1)$-th transition of ρ_2 which satisfies the criteria for the transitions of $\mathcal{A}$, i.e, a transition that is compatible with the pattern P_1' and appends the same output. Thus, the invariant is maintained. □

Therefore, if two runs ρ_1 of T_1 and ρ_2 of T_2 on input u are origin-equivalent, the patterns which are output by ρ_2 will be in the set S and therefore the NFA $\mathcal{A}$ will not accept the word u. If the two runs are not origin-equivalent, then the set of register patterns for T_2 will either become empty at some stage and $\mathcal{A}$ accepts the input u, or have some similar pattern but do not output that pattern.

The register patterns for an NSST are of polynomial size in the number of registers of the NSST. Therefore the size of a state of $\mathcal{A}$ will be polynomial in size of T_1 and exponential in the size of T_2 since we need to store a set of register patterns, which can be exponential. Therefore, the size of the transitions of $\mathcal{A}$ will also be exponential.

However, if T_2 is a DSST, then the size of a state of $\mathcal{A}$ will be polynomial in both the size of T_1 and T_2, as the set of state and register pattern pair will be a singleton set and there will be a unique transition from any state and register pattern for T_2. □

To check origin-equivalence of two NSSTs, we can check origin-containment in both directions. Since, the states and transitions of $\mathcal{A}$ are of size exponential in T_2, we can build non-deterministically build a run of $\mathcal{A}$ on-the-fly and check for non-emptiness of $\mathcal{A}$ in space exponential in T_2. Therefore, the origin-equivalence problem is in ExpSpace.

In case of T_2 being a DSST, the size of $\mathcal{A}$ is polynomial in size of both T_1 and T_2. Therefore, the origin-containment $T_1 \subseteq_o T_2$ can be checked in PSpace. If both T_1 and T_2 are DSST, then origin-equivalence can also be checked in PSpace.

Lower bounds. A lower bound for the origin-equivalence comes from the PSpace-hardness of equivalence of NFA. If two NSST are origin-equivalent, their underlying automata must be equivalent. Since checking equivalence of NFA is PSpace-hard, checking origin-equivalence of NSST is also PSpace-hard.

For DSST, the underlying automata are DFA. Checking equivalence of DFA is NlogSpace-hard. Therefore, we obtain a lower bound of NlogSpace-hard for origin-equivalence of DSST. □

This leaves a gap in the complexity of origin-equivalence of NSSTs.

Open Problem 2.3.5. *What is the exact complexity of origin-equivalence of NSSTs (DSSTs)?*

2.3.1 Copyful DSSTs

In Theorem 2.3.2, we saw that the origin-equivalence of DSST is in PSPACE. We now consider the origin-equivalence for copyful DSSTs. Recall that a copyful DSST is a DSST without the copyless restriction in its updates, i.e, $up(r)$ for a register r can be any string over $(R \cup \Gamma)^*$. Therefore, the same register can appear multiple times in the right-hand side of an update.

We give a backward constraint propagation algorithm to solve the origin-equivalence for copyful DSSTs.

Theorem 2.3.6. *The origin-equivalence problem for copyful DSST is decidable.*

We present here the algorithm for copyful DSSTs. Let $T_1 = (Q_1, \Sigma, \Gamma, R, I_1 = \{q_1^{\text{init}}\}, F_1, out_1, \Delta_1)$ and $T_2 = (Q_2, \Sigma, \Gamma, S, I_2 = \{q_2^{\text{init}}\}, F_2, out_2, \Delta_2)$ be two copyful DSST with $R \cap S = \emptyset$. Without loss of generality, we assume they are complete, i.e, for every input word, there are runs of both T_1 and T_2 on it. If a DSST is not complete, then it can be made complete by adding a sink state.

By Theorem 2.1.1 we can assume that Γ is a singleton set, say $\Gamma = \{c\}$, so contents of registers can be viewed as belonging to $\mathbb{N}^*$ instead of $(\Gamma \times \mathbb{N})^*$, which corresponds to the origins of the c's. We define the product graph G as a labeled multi-graph with set of nodes $V = Q_1 \times Q_2$ and edges labeled by register updates. For each $(q_1, q_2) \in V$, there is an edge from (q_1, q_2) to (q_1', q_2') labeled by $up_1; up_2$ if there exists $a \in \Sigma$ such that (q_1, a, q_1', up_1) and (q_2, a, q_2', up_2) are transitions in T_1 and T_2, respectively. Note that there can be at most $|\Sigma|$ many edges between two vertices, one for each letter in Σ. This is because there is exactly one transition in T_1 from a state q_1 on an input letter a and the same for T_2, as T_1 and T_2 are deterministic.

The algorithm labels the vertices of the product graph G with constraints of the form $\alpha = \beta$ where α is a word in R^* and β is a word in S^*. We assume that the product graph G is trimmed according to accessibility from initial states, i.e, all vertices that are not reachable from $(q_1^{\text{init}}, q_2^{\text{init}})$ are removed from G.

We extend register updates to words of registers. An update for a word α in R^* is obtained by applying the update to each occurrence of a register. Given a word $\alpha = r_1 r_2 \dots r_n \in R^*$ and an update $up : R \to (R \cup \Gamma)^*$, we denote by $up(\alpha)$ the string $up(r_1)up(r_2)\dots up(r_n)$.

Let $U_1 \in (R \uplus \Gamma)^*$ and $U_2 \in (S \uplus \Gamma)^*$, i.e, potential right-hand side of updates. We call U_1, U_2 *compatible* if their projections on Γ are equal, which amounts to say that $|U_1|_c = |U_2|_c$ as $\Gamma = \{c\}$. Given two compatible update words U_1, U_2 as before, we factorize them as follows: $U_1 = \alpha_0 c \alpha_1 c \alpha_2 \cdots \alpha_{n-1} c \alpha_n$ and $U_2 = \beta_0 c \beta_1 c \beta_2 \cdots \beta_{n-1} c \beta_n$, where $\alpha_i \in R^*$ is the factor of U_1 that starts just after the i-th c and ends just before the

$(i+1)$-th c, for each $0 \leq i \leq n$, and same for $\beta_i \in S^*$ from U_2. We denote by $\mathcal{C}(U_1, U_2)$ the set of constraints $\alpha_i = \beta_i$, $0 \leq i \leq n$ (excluding the trivial constraints $\epsilon = \epsilon$). For example, the words ccr_1c and cs_1cs_2c are compatible as both of them have three occurrences of the letter c. The set of constraints $\mathcal{C}(ccr_1c, cs_1cs_2c)$ is $\{s_1 = \varepsilon, s_2 = r_1\}$.

Consider an edge e of the product graph from node v to node v', labeled by the updates $up_1; up_2$. Let $\alpha = \beta$ be a constraint labeling node v'. If $up_1(\alpha)$ and $up_2(\beta)$ are compatible, then, we denote by $\mathsf{Eq}(\alpha, \beta, e)$ the set of constraints $\mathcal{C}(up_1(\alpha), up_2(\beta)))$. Otherwise, if $up_1(\alpha)$ and $up_2(\beta)$ are not compatible, then we say that the triple (α, β, e) is an *inconsistency*.

We are now ready to describe the algorithm to check origin-equivalence of copyful DSST. The algorithm maintains two sets of constraints C_v and $\hat{C}_v$ at each node v of the product graph. The constraints in C_v are the ones that are already propagated, and $\hat{C}_v$ are the constraints that yet to be propagated.

Initial constraints. The algorithm first adds constraints to the final nodes of G, i.e, where both components in the product state are final. For a node $v = (q_1, q_2) \in F_1 \times F_2$, the algorithm adds the constraints $\mathcal{C}(out_1(q_1), out_2(q_2))$ to the set $\hat{C}_v$.

Constraint propagation. The algorithm iterates a constraint propagation step that, given any node $v' = (q_1', q_2')$, propagates the constraints $\hat{C}_{v'}$ as follows. The algorithm chooses some $\alpha = \beta$ in $\hat{C}_{v'}$. For every node v that has an edge to v', and such that this edge e is labeled by $up_1; up_2$, the algorithm first checks that $up_1(\alpha)$ and $up_2(\beta)$ are compatible. If so, it adds the constraints $\mathsf{Eq}(\alpha, \beta, e) \setminus (C_v \cup \hat{C}_v)$ to the set $\hat{C}_v$. In fact, to obtain an algorithm that terminates, only a subset of these constraints will be added. This is because there may be some newly added constraints that are already implied by other existing constraints in $\hat{C}_v$. This will be explained when we discuss the termination condition of the algorithm.

Thus, the constraints added at v by propagating $\alpha = \beta$ are added to the set of unexplored constraints, unless they are already present in the set of constraints C_v or $\hat{C}_v$. If $up_1(\alpha)$ and $up_2(\beta)$ are not compatible, the algorithm identifies (α, β, e) as inconsistency and stops.

After propagating $\alpha = \beta$ to all predecessors of v', the constraint $\alpha = \beta$ is moved from $\hat{C}_{v'}$ to $C_{v'}$. This step is iterated until the algorithm either encounters an inconsistency or reaches a fixpoint.

Example 2.3.7. *We give an example of an execution of the algorithm for origin-equivalence on the copyful DSSTs T_1 and T_2 as shown in Figure 2.8.*

The two DSST are origin-equivalent. For example, on the word aa, they produce the origin output $(c, 1)(c, 1)(c, 1)(c, 1)$

Figure 2.8 – Two origin-equivalent copyful DSSTs

The initial constraint at the node (q'_1, q'_2) of the product graph will be $(r, s_1 s_2)$. By propagating the constraint along the self edge, we obtain the constraints $rr = s_1 s_2 s_1 s_2$. By propagating the constraint n times along the edge, we will obtain $(r^n, (s_1 s_2)^n)$. However, notice that if the constraint $(r, s_1 s_2)$ is satisfied, the constraints $(r^n, (s_1 s_2)^n)$ are automatically guaranteed. In particular, the constraints $(r^n, (s_1 s_2)^n)$ are redundant. We use this observation to get a termination guarantee by only adding constraints which are not implied by the existing ones.

First, we show the correctness of the algorithm. Then, we show how we can only retain the necessary constraints and get a termination guarantee.

Proof of correctness. Since we are dealing with a unary output alphabet, we consider valuations to be functions from R and S, resp., to $\mathbb{N}^*$. This is because a register valuation in the origin semantics is a function from R (or S) to $(\Gamma \times \mathbb{N})^*$ and as Γ is unary, we can drop it to make the notations simpler.

The following simple proposition formalizes the main argument to be used in the proof.

Proposition 2.3.8. *Let $\alpha' = \beta'$ be a constraint. Consider accepting runs of ρ_1 of T_1 and ρ_2 of T_2 on a word u. This gives a unique path in the product graph G. Let $e = ((q_1, q_2), (q'_1, q'_2))$ be the j-th edge in this path, labeled by updates up_1; up_2 that are compatible with respect to $\alpha' = \beta'$. Let val_i and val'_i be the valuations reached in the run ρ_i at state q_i and q'_i respectively, for $i \in \{1, 2\}$. For any constraint $\alpha = \beta \in \mathsf{Eq}(\alpha', \beta', e)$, $val_1(\alpha) = val_2(\beta)$, if and only if, $val'_1(\alpha') = val'_2(\beta')$.*

Proof. Let $up_1(\alpha') = \alpha_1 c \alpha_2 c \dots c \alpha_k$ and $up_2(\beta') = \beta_1 c \beta_2 \dots c \beta_k$. By definition of the valuation, $val'_1(\alpha') = val_1(\alpha_1) j val_1(\alpha_2) \dots j val_1(\alpha_k)$ and similarly for $val'_2(\beta')$.

For any constraint $\alpha_i = \beta_i$ in $\mathsf{Eq}(\alpha', \beta', e)$, $i \in \{1, \dots, k\}$, $val_1(\alpha_i)$ and $val_2(\beta_i)$ only use data values $< j$. If $val_1(\alpha_i) \neq val_2(\beta_i)$, for some i, then clearly $val'_1(\alpha')$ and $val'_2(\beta')$. Otherwise, if $val_1(\alpha_i) = val_2(\beta_i)$, for all i, then obviously, we have $val'_1(\alpha') = val'_2(\beta')$. □

Using this proposition, we show the correctness of the algorithm. First, we show that if an inconsistency is detected by the algorithm, then the two DSST are not origin-equivalent.

Lemma 2.3.9. *If the algorithm detects an inconsistency (α, β, e), where e is an edge from (q_1, q_2) to (q_1', q_2') labeled by updates $up_1; up_2$, then T_1 and T_2 are not origin-equivalent.*

Proof. Let a be the letter such that there exists transitions (q_1, a, q_1', up_1) in T_1 and (q_2, a, q_2', up_2) in T_2. Such a letter exists as otherwise, there would not be an edge e in the graph.

Since we assumed T_1 and T_2 are trim, the states q_1 and q_2 are accessible from the initial state on some word, say u. This gives a run of T_1 and T_2 on the word ua. Let val_1' and val_2' be valuations in the runs of T_1 and T_2 after reading ua. As (α, β, e) is an inconsistency, this means $up_1(\alpha)$ and $up_2(\beta)$ have different number of cs. This in turn implies that $val_1'(\alpha)$ and $val_2'(\beta)$ will have different number of occurrences of $|u'| + 1$.

As $\alpha = \beta$ is a constraint at (q_1', q_2'), there exists a sequence of edges $e_1, e_2, \ldots, e_k$, where $e_1 = ((q_1', q_2'), v_1)$ and $e_i = (v_{i-1}, v_i)$ for $1 < i \leq k$ and $v_k = (f_1, f_2)$ is a pair of final state; and a constraint $\alpha_i = \beta_i$ for $1 < i \leq k$, at node v_i such that

— $\alpha_k = out_1(f_1)$ and $\beta_k = out_2(f_2)$,

— $\alpha_{i-1} = \beta_{i-1}$ is a constraint in $\mathsf{Eq}(\alpha_i, \beta_i, e_i)$,

— $\alpha = \beta$ is a constraint in $\mathsf{Eq}(\alpha_1, \beta_1, e_1)$.

The existence of such a path is guaranteed as the constraint $\alpha = \beta$ is propagated to the vertex (q_q', q_2') after a finitely many steps in the algorithm. This path also gives a word u' where $u'(i)$ is the letter corresponding to the edge e_i.

Since $val_1'; val_2'$ do not satisfy the constraint $\alpha = \beta$, by applying Lemma 2.3.8 iteratively, we get that the final valuations val_1'' and val_2'' do not satisfy $out_1(f_1) = out_2(f_2)$. Therefore, the runs of T_1 and T_2 on uau' are origin-inequivalent. □

We now prove that if the algorithm does not detect any inconsistency, then the DSST are indeed origin-equivalent. This will complete the proof of correctness of the algorithm.

Lemma 2.3.10. *If the algorithm does not detect an inconsistency, then T_1 and T_2 are origin-equivalent.*

Proof. Consider the runs of ρ_1 of T_1 and ρ_2 of T_2 on a word u. This gives a unique path $\pi = e_1 e_2 \ldots e_n$ in the product graph G such that $e_i = ((q_1^{i-1}, q_2^{i-1}), (q_1^i, q_2^i))$ are labeled by updates $up_1^i; up_2^i$.

The states (q_1^0, q_2^0) are initial states and (q_1^n, q_2^n) are final states. At every vertex $v_i = (q_1^i, q_2^i)$ in the path π, we define the set of constraints K_i inductively from the end. The set of constraints K_n is the set defined by $\mathcal{C}(out_1(q_1^n), out_2(q_2^n))$. For every node $v_{i-1} = (q_1^{i-1}, q_2^{i-1})$, the set of constraints K_{i-1} for $1 \leq i \leq n$ is defined using the set of constraints K_i and the edge e_i. Essentially, K_{i-1} is the union of the sets $\mathsf{Eq}(\alpha, \beta, e_i)$ such that

$\alpha = \beta$ is a constraint in K_i. In other words, the constraints in K_{i-1} are exactly the ones obtained by propagating the constraints from K_i along the edge e_i.

Let val_1^i (val_2^i) be the valuations at state q_1^i (q_2^i) in ρ_1 (ρ_2 respectively). The initial valuation pair (val_1^0, val_2^0) satisfy the constraints K_0 as the valuations are empty. Suppose, the constraints K_i are satisfied by the valuation pair (val_1^i, val_2^i). Then by Proposition 2.3.8, the valuation pair $(val_1^{i+1}, val_2^{i+1})$ satisfies the constraints K_{i+1}. This is because $val_1^{i+1}(r)$ equals the word obtained by replacing every c by $i+1$ in $val_1^i(up_1^{i+1}(r))$, where up_1^{i+1} is the update in the transition of T_1 corresponding to e_{i+1}. This is implied by Proposition 2.3.8.

Therefore, the final valuations satisfy $out_1(q_1^n) = out_2(q_2^n)$, which implies the outputs are origin-equivalent. This proves the lemma. □

This concludes the proof that the algorithm is correct if it terminates. We now prove that the algorithm terminates after a finite number of iteration steps.

Termination. To ensure termination of the propagation algorithm, it would be nice if we could derive a uniform bound on the number of constraints that could appear at any node in the product graph G. In general, there is no such bound, since the words α and β can be of any length. Therefore, there is no immediate termination guarantee. To ensure termination, we only keep the constraints at a node if they are not implied by the existing ones. For example, the constraints $r_1 = s_1$ and $r_2 = s_2$ implies the constraint $r_1 r_2 = s_1 s_2$. To achieve this, we define word equation systems.

For a set of variables Ω and an alphabet Σ, a *word equation* is a pair of the form (x, y), where $x, y \in \Omega^*$. A solution to the equation is a morphism $\sigma : \Omega^* \to \Sigma^*$ such that $\sigma(x) = \sigma(y)$. A system of word equations is a (possibly infinite) collection of word equations. We denote a system of equations as the tuple $(\Omega, \Sigma, \mathcal{C})$, where $\mathcal{C}$ is a set of the equations (x, y) with $x, y \in \Omega^*$.

Note that the constraints at a node obtained by the propagation algorithm can be seen as a system of word equations with $\Sigma = \mathbb{N}$ and $\Omega = R \cup S$.

Theorem 2.3.11 (Ehrenfeucht's Conjecture)**.** *Given a system of word equations $\mathcal{S} = (\Omega, \Sigma, \mathcal{C})$ over a finite alphabet Σ, there exists a subsystem of word equations $\mathcal{S}' = (\Omega, \Sigma, \mathcal{C}')$, where $\mathcal{C}'$ is a finite subset of $\mathcal{C}$ such that $\sigma : \Omega^* \to \Sigma^*$ is a solution of $\mathcal{S}$ if and only if it is a solution of $\mathcal{S}'$.*

This result, known as the Ehrenfeucht's conjecture, was proved independently by Albert and Lawrence and by Guba [AL85, Gub86]. This theorem works for finite alphabets. In the constraints obtained in our algorithm however, the valuations map registers to natural numbers $\mathbb{N}$. Therefore, we prove an equivalence between solutions of a system of word equations over

the binary alphabet $\Sigma = \{0, 1\}$ and similar systems of word equations over the infinite alphabet $\mathbb{N}$.

Lemma 2.3.12. *Suppose $\mathcal{S} = (\Omega, \Sigma = \{0, 1\}, \mathcal{C})$ and $\mathcal{S}' = (\Omega, \mathbb{N}, \mathcal{C})$ are two systems of word equations with the same set $\mathcal{C}$ of symbolic equations. There is a bijection between the solutions of $\mathcal{S}$ and the solutions of $\mathcal{S}'$.*

Proof. Consider the bijection h between $\mathbb{N}^*$ and $\{0, 1\}^*$, such that $h(n_1 n_2 \dots n_k)$ to $1^{n_1}01^{n_2}0 \dots 01^{n_k}$. This is clearly a bijection as it is injective and surjective, since $\{0, 1\}^* = (01^*)^*$. Moreover, the encoding is compatible with concatenation, i.e, $h(u.v) = h(u)h(v)$. Therefore, there is a bijection between solutions of $\mathcal{S}$ and solutions of $\mathcal{S}'$. □

Using this lemma, we now explain how to check for equivalence of systems of word equations over the binary alphabet $\Sigma = \{0, 1\}$. This would then suffice to check whether a given constraint over $\mathbb{N}^*$ is redundant or not with respect to a given system of constraints. To ensure termination, we modify the propagation step slightly. We add a constraint (α, β) to the set $\hat{C}_v$, only if $C_v \cup \hat{C}_v$ and $C_v \cup \hat{C}_v \cup \{(\alpha, \beta)\}$ are not equivalent. For example, if the constraints $r_1 = s_1$ and $r_2 = s_2$ are in C_v, then we do not add the constraint $r_1 r_2 = s_1 s_2$ to $\hat{C}_v$ even if it is obtained by propagating some constraint to the node v. Equivalence of systems of word equations over Σ^* can be checked using Makanin's algorithm [Lot02].

Note that in the case of (copyless) DSSTs, the strings in the constraints can only contain a register at most once. Therefore, the size of the constraints is bounded by the total number of registers. In particular, the number of distinct constraints is exponential in the sizes of R and S. This yields a PSPACE algorithm for copyless DSSTs, that is alternative to the one given in the proof of Theorem 2.3.2. Note that the backward propagation algorithm cannot be immediately generalized to the non-deterministic case. For a fixed input u, the constraints are propagated backwards by following u in reverse, which gives different constraints at the initial node, based on the different paths followed. To check for equivalence only one of these constraints need to be satisfied. The problem does not occur in the deterministic case, since there is at most one path from the final node to initial node that follows the word u in reverse. Therefore, the equivalence problem remains open for copyful NSSTs.

2.3.2 An expressively equivalent subclass of DSST

We saw that the origin-equivalence problem for DSST is in PSPACE. In fact, this upper bound is the same as for classical equivalence of DSST [AD11].

Restricted to unary output alphabet however, the classical-equivalence problem for DSST can be solved in PTIME, even for copyful DSST

[ADD+13]. The algorithm in the latter paper uses Karr's algorithm [MOS04] for computing invariants of affine programs. For origin-equivalence, Theorem 2.1.1 shows that the restriction to unary output alphabet does not make the equivalence problem easier, at least with our provided bounds. In particular, the best upper bound for origin-equivalence of DSST with unary output alphabet is still PSPACE, which is worse than that of classical-equivalence.

Although the exact complexity of origin-equivalence remains open for both unrestricted DSST and DSST with unary output alphabet, these results suggest that the origin semantics does not necessarily yield a better complexity for the equivalence problem.

We consider in this subsection a restricted class of DSSTs, called separated DSSTs, for which origin-equivalence can be solved in PTIME.

Separated DSST. We call an update *separated* if the right-hand side of the update never contains a subword of the form rr' for two registers r, r'. This means that occurrences of two registers are always separated by some non-empty word in the right-hand side of an update. For example the update $r := r_1abr_2$ is separated, whereas the update $r := ar_1r_2b$ is not.

We call an DSST *separated* if all its updates are separated. We show that restricting to separated updates does not decrease the expressiveness of DSST as long as we allow *bounded-copy DSST*, originally defined in [AFT12]..

To define bounded-copy DSST, we need to define the flow graph of a run of a copyful DSST. The *flow graph of a run* is a directed acyclic multi-graph, defined in a similar way to flow trees. For a run ρ of length n, the vertices of the flow graph of ρ are $(R \times \{0, 1, \ldots, n\}) \cup \{out\}$. The $(i-1)$-th and i-th level simulate the updates, i.e, there is an edge from $(r', i-1)$ to (r, i) if the i-th update to register r, $up_i(r)$ uses the register r'. As a register can occur in several updates, there can be multiple outgoing edges from any vertex, and the flow graph can have multiple edges between two vertices. There is a unique final output register out, which corresponds to the final output. All the nodes in the flow graph must have a path to the final register. This assumption ensures that all registers in the flow graph contribute to the final output.

A copyful DSST is called *bounded-copy* DSST if there exists a bound k such that $\forall$ run ρ and $\forall$ vertex (r, i) in the flow graph of ρ, there are at most k paths from of (r, i) to out. Alur et. al.[AFT12] showed that bounded-copy DSST are expressively equivalent to (copyless) DSST. The construction also preserves the origins. Therefore, the class of (copyless) DSST and bounded-copy, separated DSST are expressively equivalent in the origin semantics.

Theorem 2.3.13. *Every DSST can be transformed to an origin-equivalent bounded-copy, separated DSST with exponentially many additional registers.*

Thus, DSSTs and bounded-copy separated DSSTs are expressively equivalent, both in the classical semantics and origin semantics.

Proof. Let $T = (Q, \Sigma, \Gamma, R, I, F, out, \Delta)$ be a DSST. We want to build an origin-equivalent bounded-copy, separated DSST $T' = (Q, \Sigma, \Gamma, R', I, F, out', \Delta')$. Note that the states remain the same. The only changes are in the set of registers and updates to the registers.

Let R be the set of registers of T and Π denote the set of words $\alpha \in R^+$ such that every register occurs at most once in α. For every word $\alpha \in \Pi$, we have a register x_α in R'. The updates to a register x_α are determined by the updates to the original registers. For a transition (q, a, q', up) in Δ, we have a transition (q, a, q', up') in Δ' such that, if $\alpha = r_1 \dots r_k$, then $up'(x_\alpha)$ evaluates to $up(r_1) \dots up(r_k)$.

However, the word $up(r_1) \dots up(r_k)$ need not be separated. We use the registers in X to make the right-hand side separated. The update to register x_α, $up'(x_\alpha)$ will be the string obtained by replacing every maximal factor α' of $up(r_1)up(r_2) \dots up(r_k)$, by $x_{\alpha'}$.

Consider for example $up(r) = br'$, and $up(r') = r$ where r, r' are registers of T. The update to the string rr' is $br'r$. Therefore, the update to the register $x_{rr'}$ in T' will use the register $x_{r'r}$ instead of using the concatenation of registers $r'r$. Therefore, we have $up'(x_{rr'}) = bx_{r'r}$.

The transducers T and T' are origin-equivalent since after reading a prefix of the input, the register x_α of T' computes the value of the string α obtained after T reads the same prefix, and the origins in output of T and T' are the same.

To show that the copyful DSST T' is indeed bounded-copy, consider a run of T' of length n. We show that there is exactly one path from any node (x_α, i) to out' in the flow multi-graph using the fact that the DSST T was copyless. Let $\alpha = r_1 r_2 \dots r_k$ and suppose there exists a node (x_α, i) with more than one path to out'. This implies the contents of registers $r_1, r_2, \dots, r_k$ will have more than one path to the final output out in the flow graph of corresponding to a run of T. This contradicts the fact that T is copyless. Therefore, the DSST T' is bounded-copy with a bound of 1.

The number of register in T' is exponential in the number of registers of T. Therefore, T' is a bounded-copy, separated DSST with exponentially many additional registers that is origin-equivalent to T. □

We give an example showing that the exponential number of registers in indeed required.

Example 2.3.14. *We give an example of a DSST with n registers for which exponentially many registers are needed for an equivalent separated DSST.*

The DSST $T_n = (Q, \Sigma, \Gamma, R, I, F, out, \Delta)$ has two states $Q = \{q_0, q_1\}$, with q_0 being the initial state and q_1 being a final state. The set of registers

$R = \{r_1, \dots r_n\}$. The input alphabet is $\Sigma = \{a\} \cup \{a_{i,j} \mid \forall i, j \in \{1, \dots, n\}\}$ and the output alphabet is $\Delta = \{b_1, \dots, b_n\}$.

On the letter a, there is a transition in Δ (q_0, a, q_1, up), where $up(r_i) := b_i$. On the letters $a_{i,j}$, for all $i, j \in \{1, \dots, n\}$, there is a transition $(q_1, a_{i,j}, q_1, up_{i,j})$. The update $up_{i,j}$ swaps the contents of registers r_i and r_j and keeps all the other registers unchanged. The DSST accepts words of the form aw where all letters in w are a_{ij}, for some $i, j \in \{1, \dots, n\}$. The final output function is $out(q_1) = r_1 r_2 \dots r_n$.

Since any permutation of n elements can be generated by transpositions, the output can be any permutation of $b_1, b_2, \dots b_n$, depending on the word aw, as every transition on a_{ij} is a transposition of the registers.

In any equivalent separated DSST, there needs to be a register storing every possible permutation after reading the first a of the input. Otherwise, the registers would be storing only part of the output and as concatenation of registers is not allowed in the updates, it would not be possible to output the correct permutation.

PTime algorithm for origin-equivalence. We now give a PTIME algorithm to check origin-equivalence of copyful, separated DSST. The algorithm is a backward constraint propagation algorithm similar to the one in proof of Theorem 2.3.6 for copyful DSSTs.

Theorem 2.3.15. *The origin-equivalence problem for copyful, separated DSST is in* PTIME.

Proof. The algorithm labels the product graph of copyful, separated DSSTs T_1 and T_2 with sets of registers R and S respectively, by constraints of the form $r = s$, or $r = \varepsilon$, or $s = \varepsilon$, where $r \in R$ and $s \in S$ are registers of T_1 and T_2 respectively. We denote constraints by a pair (r, s) where r and s are either registers or the empty word ε.

The algorithm is the same as for general copyful case. However, here we have to deal with the output alphabet not being unary. This is because the transformation of DSST with an arbitrary output alphabet to one with unary output alphabet in proof of Theorem 2.1.1 does not preserve separated updates.

For this reason, we need to define *compatible* updates differently. Let r and s be registers of T_1 and T_2, or the empty word, and up_1 and up_2 be updates of T_1 and T_2 respectively such that $up_1(r) = c_1 r_1 c_2 r_2 \dots c_k r_k c_{k+1}$ and $up_2(s) = c'_1 s_1 c'_2 s_2 \dots c'_\ell s_\ell c'_{\ell+1}$, where every c_i and c'_i are letters from the output alphabet Γ and every r_i (s_i) is either a register in R (S) or the empty word. We say up_1 and up_2 are compatible with respect to registers r and s if $k = \ell$ and $c_i = c'_i$ for every $1 \leq i \leq k + 1$. In other words, the projection on to Γ are equal. The constraints implied by the two words $up_1(r)$ and $up_2(s)$ are the pairs (r_i, s_i) for all $1 \leq i \leq k$. Given an edge

e labeled with updates $up_1; up_2$ and a constraint (r, s) such that up_1, up_2 are compatible with respect to r and s, we denote by $\mathsf{Eq}(r, s, e)$ the set of constraints implied by the words $up_1(r)$ and $up_2(s)$.

With this definition of compatible updates, the algorithm in the proof of Theorem 2.3.6 works for copyful, separated DSSTs as well. The algorithm labels the product graph of T_1 and T_2 with constraints of the form $r = s$. Recall that the product graph has vertices $V = Q_1 \times Q_2$ with edges from (q_1, q_2) to (q_1', q_2') labeled with updates $up_1; up_2$ if there is a letter a such that (q_1, a, q_1', up_1) and (q_2, a, q_2', up_2) are transitions of T_1 and T_2 respectively. For a node (f_1, f_2) in the product graph, the initial constraints are $out(f_1) = out(f_2)$.

In the propagation step, if a node $v' = (q_1', q_2')$ has a constraint $r = s$, and $e = (v, v')$ is an edge labeled by $up_1; , up_2$, we check if the updates up_1 and up_2 are compatible with respect to r and s. If so, we add the constraints $\mathsf{Eq}(r, s, e)$ to the node v.

An inconsistency in the algorithm is a constraint (r, s) at a vertex v' and an edge $e = (v, v')$, such that $up_1(r)$ and $up_2(s)$ are not compatible, where $up_1; up_2$ is the label of edge e. The algorithm stops if it encounters an inconsistency or reaches a fixed point. no more new constraints can be added by propagating existing constraints. The total number of possible constraints at a vertex is polynomial in $|R| + |S|$. Therefore, the number of propagation steps needed to reach a fixed point is polynomial in $|R| + |S|$. Each propagation step needs to check compatibility of the updates, which can be done in polynomial time. Therefore, the running time of the algorithm will be polynomial in $|T_1| + |T_2|$. The correctness of the algorithm follows a proof similar to the proof of Theorem 2.3.6.

Therefore, the origin-equivalence problem of copyful, separated DSSTs is in PTime. □

2.4 Conclusions

We studied the origin-equivalence (and origin-containment) problems for 2NFTs, NSSTs and various subclasses of NSSTs. We first showed that the output alphabet can be encoded in the input using origins and therefore, the origin-equivalence problem for 2NFTs (or NSSTs) with unary output alphabet is as hard as the general case(Theorem 2.1.1). This is in contrast with the classical semantics, where restricting to unary output alphabet often makes the problem easier. For instance, the equivalence for DSSTs with unary alphabet is in PTime [ADD+13], compared to PSpace in the general case [AD11].

For 2NFTs, we showed that the origin-equivalence problem is PSpace-complete (Theorem 2.2.1), which is the lowest complexity possible as equivalence of NFA is already PSpace-hard. The result holds even for the more

general model of 2NFT with common guess. In the process, we introduce a normalization process of eliminating non-productive U-turns, which reduces the problem to a simpler case of busy 2NFTs. For the case of busy 2NFTs, we reduce the origin-equivalence problem to emptiness of NFA using variants of classical techniques involving crossing sequences.

For NSSTs, the origin-equivalence problem can be reduced to MSO-satisfiability on origin graphs, which was shown to be decidable in [BDGP17]. However, this algorithm does not give any interesting complexity bounds. We give a direct algorithm which reduces the problem to emptiness of NFA, by considering the register patterns encountered during a run. This algorithm gives an upper bound of ExpSpace for NSSTs and PSpace for DSSTs for the origin-equivalence problem (Theorem 2.3.2). The best known lower bounds are PSpace-hardness for NSSTs and NlogSpace-hardness for DSSTs obtained from the equivalence of NFA and DFA respectively. Therefore, the complexity of the origin-equivalence problem remains open. An interesting observation is that the complexity of classical equivalence for DSSTs is also between PSpace and NlogSpace. For DSSTs with unary output alphabet, the classical equivalence problem is in PTime. Therefore, the best known upper bound is better for the classical equivalence problem in this case, compared to the origin-equivalence problem.

We also show that the origin-equivalence problem for copyful DSSTs is decidable (Theorem 2.3.6). The proof uses a constraint propagation algorithm and uses Makanin's algorithm to ensure termination. The backward constraint propagation algorithm also gives an alternate PSpace algorithm for (copyless) DSSTs, as Makanin's algorithm is no longer needed to ensure termination. Thus, we have both a forward and a backward algorithm for origin-equivalence of DSSTs. In the copyful case, we do not obtain any good complexity bounds, since the algorithm has to use the Makanin's algorithm as a subroutine. Note that the classical equivalence problem for copyful DSSTs is also decidable [FR17].

We also introduce a subclass of DSSTs, called *separated*-DSSTs, in which registers can be appear in the right-hand side of the same update, only if a letter separates them in the update. We show that any DSST is equivalent to a bounded-copy separated DSST (Theorem 2.3.13). However, the obtained separated DSST can be exponential in size with respect to the original DSST. We also show that the origin-equivalence problem is in PTime, even for copyful separated DSSTs (Theorem 2.3.15).

Another class of word transductions with a decidable equivalence problem is the logic $\mathcal{L}_T$ [DFL18]. However, this class is incomparable to the classes of transducers we have considered.

To conclude, the complexity for origin-equivalence (or even classical equivalence) problem for DSSTs remains a major open problem. Another possible direction of future work would be to study origin-equivalence problem for transductions on other objects, such as trees, data words, etc, by

introducing appropriate notions of origins. For transductions on trees, the origin-equivalence problem for deterministic top-down tree-to-string transducers was shown to be decidable in [FMRT15], which is open in the classical semantics. For data words, transducers in the origin semantics were considered in [DGH16, Pra20]. However, to the best of our knowledge, the origin-equivalence problem has not been considered.

Chapter 3

Resynchronizers

We saw in Chapter 2 that unlike the classical equivalence problem, the origin-equivalence problem is decidable for several classes of nondeterministic transducers such as NFTs, 2NFTs, NSSTs. However, origin-equivalence is more fine-grained than classical-equivalence, i.e, origin-equivalence might distinguish transducers that are classically equivalent. For example, the reverse and identity transductions over an unary alphabet are classically equivalent, but not origin-equivalent (see Figure 1.8). In this chapter, we discuss resynchronizers as a means to study a relaxation of the origin-equivalence relationship. Resynchronizers were originally introduced by Filiot et al. [FJLW16] in the context of NFTs. We first present the original model of *rational resynchronizers*, followed by a larger class of logical resynchronizers, called *regular resynchronizers*, which work for 2NFTs and NSSTs as well. The model of regular resynchronizers was introduced in [BMPP18] and has been further studied in [BKM$^+$19, KM20]. Finally, we study the expressiveness of rational and regular resynchronizers when restricted to the class of NFTs [BKM$^+$19].

3.1 Resychronization and Containment problem

We start by presenting some examples of transducers which are classically equivalent but not origin-equivalent.

Example 3.1.1. *Consider the NFTs T_1 and T_2 on the left of Figure 3.1 that processes inputs from a^+. On the word a^n, T_1 produces $(c,2)\dots(c,n)$ in the origin semantics, whereas T_2 produces the output $(c,1)\dots(c,n-1)$. Therefore, both are origin-inequivalent, but classically equivalent as both correspond to classical output c^{n-1}.*

Example 3.1.2. *For the NFTs T_3 and T_4 on the right of Figure 3.1, the inputs are of the form a^n, for $n \in \mathbb{N}$. The induced output of T_3 (resp. T_4) is $(c,1)^m$ (resp. $(c,n)^m$), for any m. Therefore, they are origin-inequivalent, but classically equivalent.*

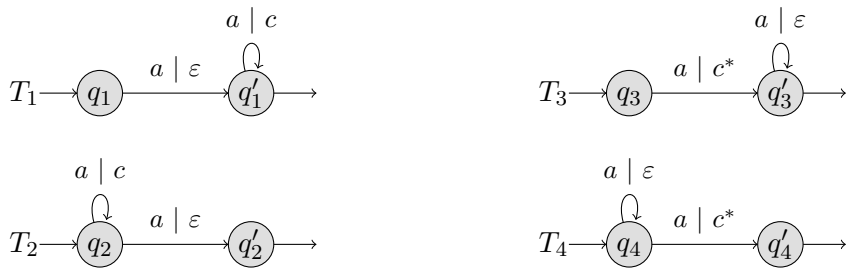

Figure 3.1 – Classically equivalent transducers that are not origin-equivalent

Note that in Example 3.1.1, for every output position, the difference between the origins in T_1 and T_2 is 1, whereas in Example 3.1.2, the difference in origins of an output position in T_3 and T_4 can be arbitrarily large.

Example 3.1.3. *Another example is that of the* identity *and* reverse *function over the* unary *alphabet* $\Sigma = \Gamma = \{a\}$*. The identity function maps* a^n *to* $(a,1)(a,2)\dots(a,n)$*, whereas the reverse function maps* a^n *to* $(a,n)(a,n-1)\dots(a,1)$*. These functions can be implemented by two NSSTs (given on 24) or 2NFTs, which will be equivalent in the classical semantics but inequivalent in the origin semantics.*

Resynchronization. A *resynchronization* relation R is a binary relation over synchronized pairs, so $R \subseteq (\Sigma^* \times (\Gamma \times \mathbb{N})^*)^2$. Any pair $((u,v),(u',v'))$ in the relation R satisfies the following properties:

— the inputs are the same, i.e, $u = u'$,

— the outputs are the same except for the origins, i.e, $v_\Gamma = v'_\Gamma$, where v_Γ (v'_Γ, resp.) represents v (v', resp.) projected onto Γ.

We call the pair (u,v) the *source* synchronized pair and (u',v') the *target* synchronized pair.

We use a resynchronization relation as a way to *change the origins* in a transduction while preserving the input and output of the source and target pairs. Given a resynchronization relation R, and a transducer T, we write $R(T)$ to denote the set of synchronized pairs $\{(u',v') \mid \exists (u,v) \in [\![T]\!]_o \text{ s.t. } ((u,v),(u',v')) \in R\}$. Note that we use the notation $R(T)$ instead of $R([\![T]\!]_o)$ for the sake of clarity, even though the resynchronizer is applied to the transduction $[\![T]\!]_o$ and not directly to the transducer T.

We define below the problem of *containment up to resynchronization*, which consists of deciding whether a transducer is contained in the other up to a change in origins defined by a given resynchronization relation.

Problem 3.1.4 (**Containment Up to Resynchronization**)**.** *Given two transducers* T_1 *and* T_2 *and a resynchronization relation* R*, is* $T_1 \subseteq_o R(T_2)$*, i.e,* $[\![T_1]\!]_o \subseteq R(T_2)$*?*

If this is the case, we say T_1 *is contained in* T_2 *up to the resynchronization* R. Intuitively, $T_1 \subseteq_o R(T_2)$ means that any synchronized pair of T_1 can be obtained by applying the resynchronization relation R to some synchronized pair of T_2. We often write $T_1 \subseteq_R T_2$ to denote $T_1 \subseteq_o R(T_2)$.

One can also consider the symmetric variant asking whether $R(T_1) \subseteq_o T_2$. Intuitively, this would mean the resynchronizer R changes the origins of the synchronized pairs of T_1 to obtain a synchronized pair of T_2. However, in this case, we would always have a resynchronization relation, namely the *empty* resynchronization $R = \emptyset$, such that $R(T_1) \subseteq_o T_2$, for any transducers T_1 and T_2. To avoid this, we consider the variant $T_1 \subseteq_o R(T_2)$.

We now present examples of resynchronization relations.

Example 3.1.5. *The universal resynchronization* R_{univ} *allows the origins of an output position to move in any possible way. Therefore,* R_{univ} *contains all pairs of synchronized pairs* $((u, v), (u', v'))$*, such that* $u = u'$ *and* $v_\Gamma = v'_\Gamma$*. A transducer* T_1 *is contained in* T_2 *up to* R_{univ} *if and only if* T_1 *is classically contained in* T_2*.*

Example 3.1.6. *The resynchronization* $R_{\textit{first-to-last}}$ *moves the origins of all output positions from the first input position to the last position of the input. So this relation contains all pairs* $((u, v), (u, v'))$ *such that* $orig(v(x)) = 1$ *and* $orig(v'(x)) = |u|$ *for all output positions* $x \in dom(v)$*.*

Example 3.1.7. *The resynchronization* R_{shift} *shifts the origins by one position to the right. It contains those pairs* $((u, v), (u, v'))$ *such that for every output position* x*,* $orig(v'(x)) = orig(v(x)) + 1$ *if* $orig(v(x)) < |u|$*. The domain of* R *excludes pairs* $((u, v), (u, v'))$ *such that* v *has a position* x *with* $orig(v(x)) = |u|$*.*

Example 3.1.5 shows that containment up to a given resynchronization, in general, is as hard as classical containment of transducers, and therefore undecidable. We will consider in the next sections two classes of resynchronization relations for which the problem of containment up to resynchronization is decidable, relative to specific classes of transducers.

3.2 Rational Resynchronizers

In this section, we present the model of *rational resynchronizers* for NFTs, which was introduced by Filiot et al. [FJLW16]. Recall from Section 1.7 (Page 22), that a synchronized pair (u, v) produced by an NFT is order-preserving, i.e, for any positions x and x' of v, $orig(v(x)) \leq orig(v(x'))$ if $x < x'$. Given such an order-preserving synchronized pair $(u, v) \in \Sigma^* \times \Gamma^*$, the *synchronized word* representation of (u, v) is the word $w = u(1)v_1u(2)v_2 \dots u(n)v_n \in (\Sigma \uplus \Gamma)^*$, where $u(i)$ is the i-th letter of u, and v_i is the projection on Γ of the maximal factor of v with origin i. Recall that

in order to define the synchronization language, we assume that Σ and Γ are disjoint. In case they are not, we can always tag the letter with information about whether it is an input or output letter. The set of *synchronized words* defined by an NFT T is called the *synchronization language* of T, denoted by $Sync(T)$, and it is a regular language (see Theorem 1.7.1).

Since the synchronization language captures the origin information of an NFT, transformations of synchronization languages can be used to define a resynchronization relation. In fact, the name *resynchronization*, introduced by Filiot et al. [FJLW16], is motivated by the observation that resynchronizations are transformations of synchronization languages. In this section, we present some results from [FJLW16] about (rational) resynchronizers.

We fix Σ and Γ to be disjoint input and output alphabet. A *rational resynchronizer* is an NFT R satisfying the following properties. We refer to the input and output of R as the *source* and the *target* respectively. The resynchronizer R has the same source and target alphabet $\Sigma \uplus \Gamma$. Moreover, for all source-target pair $(w, w') \in [\![R]\!]$, the projections of w and w' over Σ (resp. Γ) coincide, so $w_\Sigma = w'_\Sigma$ and $w_\Gamma = w'_\Gamma$. By slight abuse of notation, we write R instead of $[\![R]\!]$ when no confusion arises.

Because of the restriction that projections of w and w' to both Σ and Γ are equal, a rational resynchronizer is a length-preserving transducer. In the thesis, we assume that a rational resynchronizer is a letter-to-letter transducer, i.e, it processes a single letter in the source and produces a single target letter in a single transition, even though the original definition does not require this assumption [FJLW16]. Note that this is not a restriction, as all length-preserving transducers can be equivalently expressed by letter-to-letter NFTs, see [EM65].

Example 3.2.1. *The following resynchronizer R in Figure 3.2 shifts the origin of the output positions by* 1 *to the left and makes the NFTs in Example 3.1.1 equivalent modulo R.*

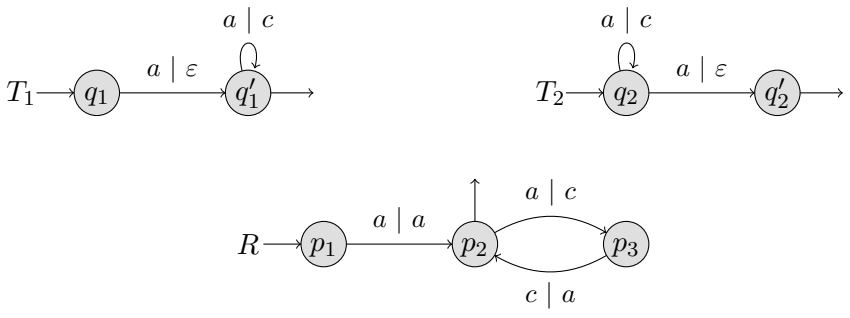

Figure 3.2 – Resynchronizer R such that T_1 and T_2 are origin-equivalent modulo R

The image of the synchronization language $Sync(T)$ of an NFT T through a rational resynchronizer R is the set of words $R(Sync(T)) = \{w' \mid \exists w \in Sync(T) \text{ such that} (w, w') \in R\}$. It is a regular synchronization language and a corresponding NFA can be obtained by taking the product of the NFA representation of $Sync(T)$ and the resynchronizer R. The NFA representation of $R(Sync(T))$ thus obtained is polynomial in the size of R and T. We often write $R(T)$ instead of $R(Sync(T))$ as long as it is clear from the context that T is an NFT.

Therefore, given two NFTs T_1 and T_2, the containment up to a rational resynchronizer R, $T_1 \subseteq_o R(T_2)$ reduces to the containment problem between the regular languages $Sync(T_1)$ and $R(Sync(T_2))$. Since both languages are described by NFA of size polynomial in T_1, T_2 and R, we have that containment up to rational resynchronizer is in PSPACE (and also PSPACE-hard, because of containment of NFA).

Theorem 3.2.2 (Theorem 7 in [FJLW16])**.** *The containment up to a given rational resynchronizer for NFT is* PSPACE*-complete.*

We now introduce two technical notions specific to rational resynchronizers, the *lag* and *delay*.

Lag of a rational resynchronizer. A rational resynchronizer processes a synchronized word as the source and produces another synchronized word as the target. Recall that synchronized words are words over the disjoint union of input and output alphabet, i.e, $\Sigma \uplus \Gamma$. The notion of *lag* counts the difference between the number of input letters from Σ consumed in the source and produced in the target. The *lag* depends on a run of the resynchronizer. Given a partial run $\rho = q_0 \xrightarrow{a_1 \mid c_1} q_1 \xrightarrow{a_2 \mid c_2} \dots \xrightarrow{a_n \mid c_n} q_n$ of the (letter-to-letter) resynchronizer R, its lag is $\mathsf{lag}(\rho) = |a_1 \dots a_n|_\Sigma - |c_1 \dots c_n|_\Sigma$, where $|u|_\Sigma$ denotes the length of the word u projected to Σ.

A transition $q \xrightarrow{a \mid c} q'$ in ρ increases $\mathsf{lag}(\rho)$ by 1 if $a \in \Sigma$ and $c \in \Gamma$, and decreases $\mathsf{lag}(\rho)$ by 1 if $a \in \Gamma$ and $c \in \Sigma$. Otherwise if both $a, c \in \Sigma$ or $a, c \in \Gamma$, then $\mathsf{lag}(\rho)$ remains unchanged. Note that since R is letter-to-letter, we could have equivalently defined $\mathsf{lag}(\rho)$ by counting the difference between the number of output letters instead of the input letters produced in the source and consumed in the target. Further, note that the lag of a successful run is always 0, since R preserves the input projection, and thus the number of input letters in source and target must be same.

Even though the lag depends on the run, it can be easily seen that it is in fact a property of the states:

Lemma 3.2.3 (Lemma 15 in [BKM$^+$19])**.** *Assuming that R is trimmed and has exactly one initial state, for every two runs ρ_1 and ρ_2 of R that begin with the same state and end with the same state, $\mathsf{lag}(\rho_1) = \mathsf{lag}(\rho_2)$.*

Proof. Since R is trimmed, both runs ρ_1 and ρ_2 can be completed to some successful runs of the form $\rho'\rho_1\rho''$ and $\rho'\rho_2\rho''$. From $\mathsf{lag}(\rho'\rho_1\rho'') = 0 = \mathsf{lag}(\rho'\rho_2\rho'')$, it immediately follows that $\mathsf{lag}(\rho_1) = -(\mathsf{lag}(\rho') + \mathsf{lag}(\rho'')) = \mathsf{lag}(\rho_2)$. □

Note that we can assume, without loss of generality that R is trimmed and has exactly one initial state. Using the above lemma, we can uniquely associate a lag with each state q of R, denoted by $\mathsf{lag}(q)$, by choosing an arbitrary run ρ that starts with the initial state of R and ends with q, and letting $\mathsf{lag}(q) = \mathsf{lag}(\rho)$. Note also that all lags range over the finite set $\{-|Q|, \ldots, |Q|\}$, where Q is the state space of R, since each transition of R can only increase or decrease the lag by 1.

Bounded-delay resynchronizers. A different notion that measures how much the source and the target have diverged in a given resynchronization, called *delay*, was introduced in Filiot et al. [FJLW16]. Unlike the lag, the delay is defined as a property of a source and target pair, instead of being a property of the resynchronizer. Given two words w and w' over $\Sigma \uplus \Gamma$ such that $w_\Sigma = w'_\Sigma$, let $v = w_\Gamma$ and $v' = w'_\Gamma$. Define $\mathit{diff}(v, v')$ as the word $v^{-1}v'$ over $\Gamma \cup \Gamma^{-1}$, modulo the cancellation rule $a^{-1}a = \epsilon$. Intuitively, $\mathit{diff}(v, v')$ removes the (longest) common prefix of v and v', keeping the remainders. The word u^{-1} is, as usual, $u(n)^{-1} \ldots u(1)^{-1}$. For example, $\mathit{diff}(ab, ac) = b^{-1}c$. If w is a prefix of w', then $\mathit{diff}(w, w') = w^{-1}w' \in \Gamma^*$. Similarly, if w' is a prefix of w, then $\mathit{diff}(w, w') = ((w')^{-1}w)^{-1} \in (\Gamma^{-1})^*$.

Let $w = u(1)v_1u(2)v_2 \ldots u(n)v_n$, $w' = u(1)v'_1u(2)v'_2 \ldots u(n)v'_n$ be two synchronized words with the same input and output, so $v_1 \ldots v_n = v'_1 \ldots v'_n$. The $\mathit{delay}(w, w')$ is defined as $\max_i (|\mathit{diff}(v_1 \ldots v_i, v'_1 \ldots v'_i)|)$. Intuitively, a delay of k implies that after consuming equal parts of the input, the word w is ahead or behind of w' by at most k positions in the output.

Using the notion of delay, we can define the k-delay resynchronization relation $\mathrm{Del}_k = \{(w, w') \mid w_\Sigma = w'_\Sigma\ ,\ w_\Gamma = w'_\Gamma \text{ and } \mathit{delay}(w, w') \leq k\}$. Note that this relation depends only on the definition of delay. As it turns out, the k-delay resynchronization relation Del_k is rational [FJLW16].

Proposition 3.2.4 (Proposition 8 in [FJLW16])**.** *For every k, there exists a rational resynchronizer D_k defining Del_k.*

The proof given by Filiot et al [FJLW16] gives a rational resynchronizer whose states correspond to words $w \in \Gamma^* \cup (\Gamma^{-1})^*$, where $|w| \leq k$. The transitions consume a single input letter in both source and target at a time and the state corresponds to $\mathit{diff}(v_1 \ldots v_k, v'_1 \ldots v'_k)$ after reading k input letters. When consuming output letters, the *diff* between the output read in source and target can change, and this corresponds to an update in the state. However, the obtained rational resynchronizer is not a letter-to-letter NFT. Converting it into a letter-to-letter NFT is of course possible, but the

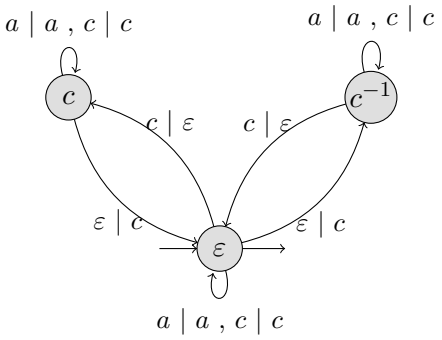

Figure 3.3 – 1-delay resynchronizer D_1

states lose their intuitive meaning. We present an example of the 1-delay rational resynchronizer with $\Sigma = \{a\}$ and $\Gamma = \{c\}$ without the letter-to-letter restriction in Figure 3.3. An important observation here is that since the states corresponds to words of length at most k, there are exponentially many states. Therefore, the rational resynchronizer D_k has state space exponential in k. As a consequence of Proposition 3.2.4 and Theorem 3.2.2, containment up to a bounded-delay resynchronizer for a fixed bound k is decidable in ExpSpace (Theorem 9 in [FJLW16]).

To illustrate the difference between *lag* and *delay*, we give an example.

Example 3.2.5. *The NFTs T_1 and T_2 in Figure 3.4 produce, on the input aa, output $(c,1)^n$ and $(c,2)^n$ respectively, for any $n \geq 1$. The delay between the pair of synchronized words $(ac^n a, aac^n)$ is n, since after the first a, the difference in number of cs is n. Therefore, the maximum delay among the pairs of synchronized words accepted by R is unbounded.*

However, the lag at state p_1, p_2 and p_4 of the resynchronizer R is 0, and at state p_3 the lag is -1.

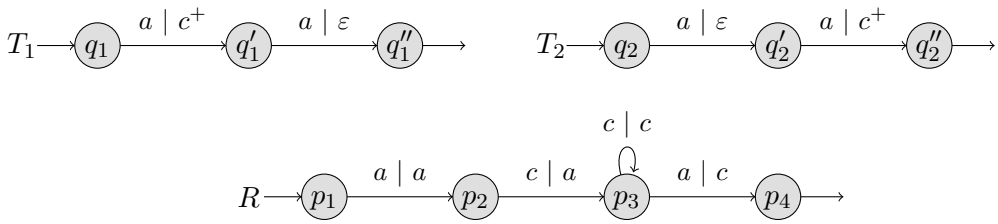

Figure 3.4 – Resynchronizer with unbounded delay

The main difference between delay and lag is that while lag compares the difference in the input projection (or, equivalently, in the output projection) between prefixes of the synchronized words of equal length, the delay

compares the difference in total length between prefixes of the synchronized words which have equal projection to the input alphabet.

Note that in Example 3.2.5, the delay is unbounded because the length of the output between consecutive input positions was unbounded. For example, for every $n \geq 1$, T_1 produces the synchronized word $ac^n a$, whereas T_2 produces the synchronized word aac^n.

A resynchronization relation $[\![R]\!]$ defined by a rational resynchronizer R, such that $[\![R]\!] \subseteq \mathrm{Del}_k$ for some $k \geq 0$, is called a *bounded-delay resynchronization*.

We now restrict our attention to the class of real-time NFTs, whose synchronized words have a bounded number of output letters between consecutive input positions (see Page 8). Recall that real-time NFTs have transitions with finite language output, i.e, in a transition (q, a, L, q'), the language L is a finite language.

The following theorem from [FJLW16] shows that for real-time NFTs, bounded-delay resynchronizations are equally expressive as rational resynchronizers.

Theorem 3.2.6 (Theorem 11 in [FJLW16])**.** *Given two real-time NFTs T_1 and T_2 and a rational resynchronizer R such that $T_1 \subseteq_o R(T_2)$, an integer k can be computed such that $T_1 \subseteq_o Del_k(T_2)$ holds. Furthermore, the value of k is polynomially bounded in the sizes of T_1, T_2, R.*

We show an alternative proof of Theorem 3.2.6 using the definition of lag. For the real-time NFTs T_1 and T_2, consider the length of the longest word appearing in a language L in any of the transitions of T_1 or T_2. Call this number ℓ. Note that ℓ must be polynomial in the size of T_1 and T_2. We prove the following lemma.

Lemma 3.2.7. *Let R be a letter-to-letter rational resynchronizer with maximum lag k'. Let (w, w') be a pair of synchronized words such that $(w, w') \in R$, $w \in Sync(T_2)$ and $w' \in Sync(T_1)$. Then $delay(w, w') \leq k'\ell$.*

Proof. Let $w = u(1)v_1u(2)v_2 \dots u(n)v_n$ and $w' = u(1)v'_1u(2)v'_2 \dots u(n)v'_n$. We need to show that $|\mathit{diff}(v_1 \dots v_m, v'_1 \dots v'_m)| \leq k'\ell$ for all $1 \leq m \leq n$. Suppose for some m, $|u(1)v_1 \dots u(m)v_m| = |u(1)v'_1 \dots u(m)v'_m| + C$, where $C > 0$ (the case of $C \leq 0$ can be handled similarly). Let w'' be the prefix of w' that has the same length as $|u(1)v_1 \dots u(m)v_m|$. Let the run of R corresponding to the pair (w, w') reach some state q after reading the prefix $u(1)v_1 \dots u(m)v_m$ in source and producing w'' in the target. The lag at the state q is at most k', therefore there are at most $m + k'$ input positions in w'', since there are m input positions in $u(1)v_1 \dots u(m)v_m$. Between two consecutive input positions, there are at most ℓ output positions. Therefore $|w''| - |u(1)v'_1 \dots u(m)v'_m|$ is at most $k'\ell$. Since this holds for every m, $delay(w, w')$ is at most $k'\ell$. □

Let us now turn to the proof of Theorem 3.2.6. Since the delay between pairs (w, w') in R, such that $w \in \mathit{Sync}(T_2)$ and $w' \in \mathit{Sync}(T_1)$ is at most $k'\ell$, $R(T_2) \subseteq \mathrm{Del}_{k'\ell}(T_2)$. Therefore, $T_2 \subseteq_o R(T_1) \subseteq_o \mathrm{Del}_{k'\ell}(T_1)$. Since k' is polynomial in the size of R and ℓ is polynomial in the size of T_1 and T_2, $k'\ell$ is polynomial in the size of R, T_1 and T_2. This concludes the proof of Theorem 3.2.6.

Rational resynchronizability. Given NFTs T_1 and T_2, we say T_1 is resynchronizable in to T_2 by a rational resynchronizer, if there exists a rational resynchronizer R such that $T_1 \subseteq_o R(T_2)$. We denote this by $T_1 \preceq_{\mathrm{rat}} T_2$. We call this relation the *rational resynchronizability* relation on NFTs and present some of its properties.

Properties of rational resynchronizability. First, it is easy to see that rational resynchronizability is in between origin-equivalence and classical equivalence, i.e, (1) $T_1 \subseteq_o T_2$ implies $T_1 \preceq_{\mathrm{rat}} T_2$; and (2) $T_1 \preceq_{\mathrm{rat}} T_2$ implies $T_1 \subseteq T_2$, where $\subseteq$ is the classical containment relation. The first condition follows simply by considering the identity resynchronizer. The second condition follows from the definition of resynchronizability, since $T_1 \subseteq_o R(T_2)$ for some R and $R(T_2) \subseteq T_2$, and therefore, $T_1 \subseteq T_2$.

By the above arguments, it is not clear whether rational resynchronizability coincides with classical equivalence or not, i.e, for every $T_1 \subseteq T_2$, whether $T_1 \preceq_{\mathrm{rat}} T_2$ or not. As it turns out, they do not coincide.

Example 3.2.8. *Consider the NFT T_1 and T_2 given in Figure 3.5. Both T_1 and T_2 read words of the form a^k, for any k, as input and produces a word from $(c, 1)^*$ and $(c, k)^*$ respectively. We prove that there is no rational resynchronizer R, such that $T_1 \subseteq_o R(T_2)$.*

Figure 3.5 – T_1 not rational resynchronizable to T_2

Proposition 3.2.9. *For any rational resynchronizer R, $T_1 \not\subseteq_o R(T_2)$ for T_1, T_2 in Figure 3.5.*

Proof. Suppose there exists a rational resynchronizer R such that $T_1 \subseteq_o R(T_2)$. Let k be an upper bound on the $|\mathsf{lag}(q)|$ at any state q of R. Recall that $\mathsf{lag}(q) = k$ means that for any run of R' that reaches q after reading a prefix of the input u and producing v, we have $|u_\Sigma| - |v_\Sigma| = k$. Consider the pair of synchronized words $w_1 = ac^{k+2}a^{k+1}, w_2 = a^{k+2}c^{k+2}$. The

synchronized word w_1 is produced by T_1 and the word w_2 is produced by T_2. Note that R must contain (w_1, w_2) as $T_1 \subseteq_o R(T_2)$ and w_2 is the only synchronized word in T_2 that can be mapped to w_1 by R.

Consider the prefixes ac^{k+1} and a^{k+2} of w_1 and w_2 respectively. Suppose that after reading the prefix ac^{k+1}, R reaches state q. Clearly, $|\mathsf{lag}(q)| = k+1 > k$. This is a contradiction. Therefore, such a rational resynchronizer cannot exist. □

Remark 1. *While, rational resynchronizability and classical containment do not coincide in general, we will see in Chapter 4, that the two properties are equivalent for the class of finitely-valued NFTs. In other words, given two k-valued NFTs T_1 and T_2, $T_1 \subseteq T_2$* if and only if*, there exists a (bounded-delay) rational resynchronizer R such that $T_1 \subseteq_o R(T_2)$.*

The identity resynchronization, $R = \{(w, w) \mid w \in (\Sigma \uplus \Gamma)^*\}$ can be expressed by a rational resynchronizer and witnesses the reflexivity of the $\preceq_{\text{rat}}$ relation. The transitivity of the $\preceq_{\text{rat}}$ relation follows from composition of rational resynchronizers. Since rational resynchronizers are NFTs, they are closed under composition. If R_1, R_2 are rational resynchronizers, such that $T_1 \subseteq_o R_1(T_2)$ and $T_2 \subseteq_o R_2(T_3)$, then $T_1 \subseteq_o R_1 \cdot R_2(T_3)$.

However, the relation $\preceq_{\text{rat}}$ is not symmetric. For example, let $T_\emptyset$ be an NFT defining the empty transduction. Clearly $T_\emptyset \preceq_{\text{rat}} T$ for any NFT T. But the opposite is not true as long as T contains at least one synchronized word w.

The above example is not a corner case. We give a general technique to find NFTs T_1, T_2 such that $T_1 \preceq_{\text{rat}} T_2$, but $T_2 \not\preceq_{\text{rat}} T_1$. Let T_1', T_2' be NFTs such that $T_1' \not\preceq_{\text{rat}} T_2'$. In particular, we can take the NFTs from Example 3.2.8. Let $T_1 = T_1' \cup T_2$, $T_2 = T_2'$. Clearly $T_2 \subseteq_o R_{id}(T_1)$. However, we can show by contradiction that $T_1 \not\preceq_{\text{rat}} T_2$. Suppose $T_1 \subseteq_o R(T_2)$ for some resynchronizer R. Then $T_1' \subseteq_o T_1 \subseteq_o R(T_2) =_o R(T_2')$, which contradicts the assumption that $T_1' \not\preceq_{\text{rat}} T_2'$.

It is possible to define *an equivalence of NFT up to rational resynchronizers* by requiring $T_1 \preceq_{\text{rat}} T_2$ and $T_2 \preceq_{\text{rat}} T_1$. This will be in between classical and origin equivalence. However, in this thesis, we are mainly interested in the rational resynchronizability relation.

3.3 Regular Resynchronizers

For transductions defined by 2NFT or NSST, we can no longer use rational resynchronizers, because we cannot work with the synchronization language. Therefore, we introduce a logic-based model of resynchronization, called *regular resynchronizers*, which is motivated by logical transformation of graphs [Cou94]. Regular resynchronizers were originally introduced in [BMPP18] with the name of MSO-resynchronizers and were applied to origin

graphs. This model has been further studied [BKM$^+$19, KM20, BKMP21]. Here, we apply regular resynchronizers to synchronized pairs instead of origin graphs (the two formalisms are in fact equivalent, see page 21).

The key idea is to define an MSO-formula move with two-free variables interpreted over positions of the input word. Intuitively, if $(u, i, j) \vDash \mathsf{move}$, then output positions with origin i can be moved to have origin j. We illustrate this with an example.

Example 3.3.1. *Consider the NFTs in Figure 3.6, which accept words of the form a^{2n} and output c^{2n}, but with different origins. For example, on input aa, the output of T_1 is $(c,1)(c,2)$ and the output of T_2 is $(c,1)(c,1)$.*

To make T_1 contained in T_2 modulo a resynchronization relation, the move formula must change the origins that are even numbers in T_2 by shifting to the right by 1, and keep the origins that are odd unchanged.

Therefore, the formula $\mathsf{move}(y,z) : (y = z) \vee (y = z+1)$, will allow to shift the origins of some output positions (e.g. the even ones) by 1 to the right and keep the origins of other positions unchanged. However, there are many other possible shifts of origins.

For example, on the synchronized pair $(aa, (c,1)(c,2))$, the origin of the first output position can be kept unchanged by choosing $y = z$ and the origin of the second position can be changed to 1 by choosing $y = z+1$. This choice is shown in figure at the bottom of Figure 3.6. The source origin is denoted by blue arrow, whereas the target origin is denoted by a brown arrow.

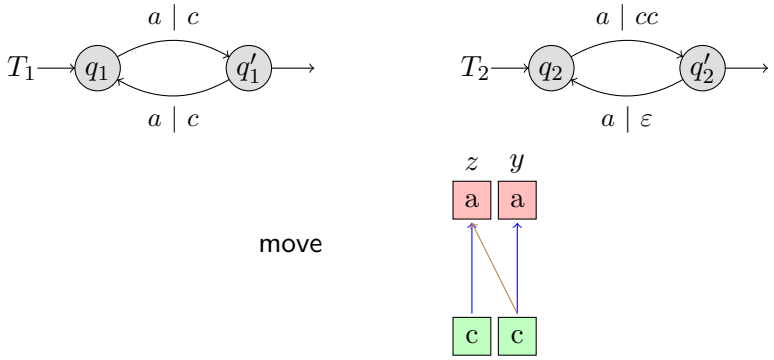

Figure 3.6 – $\mathsf{move}(y,z) : (y = z) \vee (y = z+1)$

To further restrict changes in origins as defined by the move formula, we add other components to the resynchronizer. The first one is *parameters* (or colors). Output parameters label the output positions using a finite set of parameters $\overline{O} = (O_1, O_2, \ldots, O_k)$. This defines different types of output positions based on which of the parameters are true in a given position as well as based on the letter at that position. Formally, the type τ of an output position is (a, S) where, $a \in \Gamma$ is the letter at the position and $S \subseteq \overline{O}$ is

the set of output parameters hold at that position. Therefore, the output-type can be thought of as the letter of the output position in the extended alphabet $\Gamma' = \Gamma \times 2^{\overline{O}}$.

Output-types are then used to relativize move formulas. For example, if there are output parameters $\overline{O} = \{\text{odd}, \text{even}\}$, that identify even and odd output positions, then we can have $\mathsf{move}_{\text{odd}}(y, z) := (y = z)$ and $\mathsf{move}_{\text{even}}(y, z) := (y = z + 1)$. Note that here since there is only one letter $c \in \Gamma$, we do not explicitly write c in the output types. The regular resynchronizer R defined by $\mathsf{move}_{\text{odd}}, \mathsf{move}_{\text{even}}$, assuming the parameters odd, even correctly identify odd and even positions, will satisfy $T_1 =_o R(T_2)$ as in Example 3.3.1.

We can also use input parameters $\overline{I} = \{I_1, I_2, \ldots, I_\ell\}$, which label the input positions. Then, an input position will be labeled by a letter from $\Sigma' = \Sigma \times 2^{\overline{I}}$. The move relation can inspect both the letter and the associated input parameters.

Both input and output parameters must satisfy some constraints defined by MSO-formulas ipar and opar, interpreted over the annotated input and output words, respectively. In the above example, the opar formula can be used to check that the even and odd positions in the output are correctly annotated with the parameters even and odd. We denote by extended alphabets $\Sigma' = \Sigma \times 2^{\overline{I}}$ and $\Gamma' = \Gamma \times 2^{\overline{O}}$, and call words over this alphabet as annotated inputs and annotated outputs respectively.

Finally, we add to a resynchronizer some formulas next, which are, as move, MSO-formulas with two free variables interpreted over the annotated input word. The idea behind the next formulas is to constraint the target origins of consecutive output positions. The intended meaning of next is that if two consecutive output positions x, $x + 1$ get target origins z and z', respectively, by the move formula, then $(\hat{u}, z, z') \vDash \mathsf{next}$. Essentially, next allows to discard some of the synchronized pairs obtained by applying move. Like the move formula, the next formula, can also take into account the output-types of the output positions x and $x + 1$, i.e, for every pair of output-type τ, τ', we have a formula $\mathsf{next}_{\tau,\tau'}$.

For example, to define order-preserving synchronized pairs, the next formula can constrain the target origins z, z' by enforcing $z \leq z'$. This means that for any output positions x with target origin z, the output position $x + 1$ must have target origin $z' \geq z$.

Regular resynchronizers Formally, a regular resynchronizer R is defined as a tuple $R = (\overline{I}, \overline{O}, \mathsf{ipar}, \mathsf{opar}, (\mathsf{move})_\tau, (\mathsf{next})_{\tau,\tau'})$, where

— $\overline{I}$ and $\overline{O}$ are finite sets of input and output parameters,

— ipar and opar are MSO-sentences on words over the alphabets $\Sigma' = \Sigma \times 2^{\overline{I}}$ and $\Gamma' = \Gamma \times 2^{\overline{O}}$ respectively,

— for every $\tau \in \Gamma'$, move_τ is an MSO-formula over Σ' with two first-

order free variables (usually denoted by y, z),
— for every $\tau, \tau' \in \Gamma'$, $\mathsf{next}_{\tau,\tau'}$ is an MSO-formula over Σ' with two first-order free variables (usually denoted by z, z').

Semantics of regular resynchronizers. Consider a regular resynchronizer $R = (\overline{I}, \overline{O}, \mathsf{ipar}, \mathsf{opar}, (\mathsf{move})_\tau, (\mathsf{next})_{\tau,\tau'})$. Let (u, v) and (u, v') be two synchronized pairs with $v_\Gamma = v'_\Gamma$. The pair $((u, v), (u, v'))$ belongs to $[\![R]\!]$ if and only if there exist words $\hat{u} \in (\Sigma \times 2^{\overline{I}})^*$ and $\hat{v} \in (\Gamma \times 2^{\overline{O}})^*$ satisfying the following conditions:

— $\hat{u} \vDash \mathsf{ipar}$ and $\hat{v} \vDash \mathsf{opar}$,
— $\hat{u}_\Sigma = u$, $\hat{v}_\Gamma = v_\Gamma$,
— for every output position x in $\mathrm{dom}(\hat{v})$ labeled by $\tau = (a, S)$, $v(x) = (a, i)$ and $v'(x) = (a, j)$ imply $(\hat{u}, i, j) \vDash \mathsf{move}_\tau$;
— for every pair $(x, x+1)$ of consecutive output positions in $\mathrm{dom}(\hat{v})$ labeled by $\tau = (a, S)$ and $\tau' = (b, S')$, $v'(x) = (a, j)$ and $v'(x+1) = (b, j')$ imply $(\hat{u}, j, j') \vDash \mathsf{next}_{\tau,\tau'}$.

With a slight abuse of notation, we write R instead of $[\![R]\!]$ to denote the semantics of a regular resynchronizer when no confusion arises.

Note that the universal resynchronization, defined in Example 3.1.5, is regular. This can be done without any input/output parameters, just by letting move_a be the formula that is vacuously true. This implies that the containment up to any given (unrestricted) regular resynchronizer is undecidable in general. Therefore, we have the following result.

Proposition 3.3.2. *Containment (equivalence) up to a regular resynchronizer is undecidable.*

To obtain a more interesting class of resynchronizers, we introduce a restriction on the move_τ formulas.

Bounded regular resynchronizers. The restriction we consider is as follows. There exists a natural number k such that for every output type τ, every word $\hat{u} \in \Sigma'^*$ and every $j \in \mathrm{dom}(u)$, there exist at most k distinct positions $i \in \mathrm{dom}(u)$ such that $(\hat{u}, i, j) \vDash \mathsf{move}_\tau$. We call such a resynchronizer *k-bounded*. A resynchronizer is *bounded* if it is k-bounded for some k.

In other words, for a fixed input annotated with parameters, the move_τ defines a finite union of functions from target origins to source origins. Note that this restriction is decidable, i.e, we can check whether a given regular resynchronizer is bounded or not:

Proposition 3.3.3 (Proposition 15 in [BMPP18])**.** *Given a regular resynchronizer R, it is decidable to check if it is bounded.*

Proof. It suffices to check that move_τ is bounded for every fixed output-type τ. We show how this can be checked. By Büchi's theorem the move_τ formula

can be expressed by a DFA A, accepting words with two marked positions, over the alphabet $\Sigma' \times \{0,1\}^2$, where the last two components are used to mark the positions i, j corresponding to the two free variables. Therefore, A accepts words over the alphabet $\Sigma' \times \{0,1\}^2$ such that there is exactly one position with second component marked 1 and exactly one position with last component marked 1.

Consider the language obtained by projecting out the second component, i.e, the positions corresponding to i. This is a regular language and an NFA A' accepting this language can be obtained by projecting out the second component of the alphabet in the transitions.

Consider a word $(\hat{u}, j)$ accepted by A'. Note that j here refers to the position marked by the last component as 1. This word is accepted by A' if and only if, there exists $i \in \mathrm{dom}(u)$ such that $(\hat{u}, i, j)$ is accepted by A. For every such $i \in \mathrm{dom}(u)$, there exists exactly one run in A since A is a DFA. Furthermore, for positions $i \neq i' \in \mathrm{dom}(u)$ such that there is a run ρ on $(\hat{u}, i, j)$ and a run ρ' on $(\hat{u}, i', j)$, then the runs of A' corresponding to ρ and ρ' are different. If they correspond to the same run of A', then the states visited in both ρ and ρ' are the same. Therefore, the word with both i and i' marked by the third component will also be accepted by A, which is a contradiction.

Therefore, the NFA A' is at most k-ambiguous, if and only if, for every $\hat{u}$ and j, there are at most k positions $i \in \mathrm{dom}(u)$ such that $(\hat{u}, i, j)$ is accepted by A. In other words, the problem of checking whether the resynchronizer R is bounded or not reduces to checking whether A' is finitely ambiguous or not. To conclude, we recall that one can decide whether a given NFA is finitely ambiguous [WS91]. □

We now prove that for every bounded, regular resynchronizer, we can build an equivalent 1-bounded, regular resynchronizer by introducing additional output parameters.

Lemma 3.3.4. *Every k-bounded, regular resynchronizer is effectively equivalent to a 1-bounded, regular resynchronizer.*

Proof. Let $R = (\overline{I}, \overline{O}, \mathsf{ipar}, \mathsf{opar}, (\mathsf{move}_\tau)_\tau, (\mathsf{next}_{\tau,\tau'})_{\tau,\tau'})$ be a k-bounded, regular resynchronizer. Let $\hat{u}$ and $\hat{v}$ be a pair of annotated input and output satisfying ipar and opar respectively. To construct an equivalent 1-bounded regular resynchronizer R' we introduce additional output parameters. Specifically, each output position will be annotated with an output type τ from R and an additional index in $\{1, \ldots, k\}$. The intended meaning of the index is as follows: if (y, z) is the source/target origin pair associated with an output position labeled by (τ, i), $i \in \{1, \ldots, k\}$, then then there are exactly $(i-1)$ positions $y' < y$ such that $(\hat{u}, y', z) \vDash \mathsf{move}_\tau$.

Note that this indexing depends on the choice of the target origin z. Therefore, different indexing are possible for different choice of the target

origin z.

Based on the resynchronizer R, we define the new resynchronizer as $R' = (\overline{I}, \overline{O}', \mathsf{ipar}, \mathsf{opar}', (\mathsf{move}'_{(\tau,i)})_{\tau,i}, (\mathsf{next}_{(\tau,i),(\tau',i')})_{\tau,i,\tau',i'})$, where

— $\overline{O}' = \overline{O} \uplus \{O'_1, \ldots, O'_k\}$ consists of the old output parameters $\overline{O}$ of R plus some new parameters $O'_1, \ldots, O'_k$ for representing indices in $\{1, \ldots, k\}$;

— opar' defines language of all output annotations whose projections over Γ' (the output alphabet extended with the parameters of R) satisfy opar and each position is marked by exactly one index;

— given a type τ' that encodes a type τ of R and an index $i \in \{1, \ldots, k\}$, $\mathsf{move}'_{\tau'}(y, z)$ states that y is the i-th position y' satisfying $\mathsf{move}_\tau(y', z)$; This property can be expressed by the MSO-formula

$$\exists\, y_1 < \cdots < y_i = y \;\bigwedge\nolimits_j \mathsf{move}_\tau(y_j, z)$$
$$\wedge\, \forall y' \leq y \;\big(\mathsf{move}_\tau(y', z) \rightarrow \bigvee\nolimits_j y' = y_j\big);$$

— $\mathsf{next}'_{(\tau,i),(\tau',i')}(z, z')$ enforces the same property as $\mathsf{next}_{\tau,\tau'}(z, z')$.

The resynchronizer R' is 1-bounded by definition of $\mathsf{move}'_{(\tau,i)}$. If for positions $y < y'$, $(\hat{u}, y, z) \vDash \mathsf{move}'_{(\tau,i)}$ and $(\hat{u}, y', z) \vDash \mathsf{move}'_{(\tau,i)}$, then y and y' are both the i-th source position in $\hat{u}$ satisfying move_τ with target z, which is a contradiction.

We now prove that R and R' define the same relation between synchronized pairs. First we show $R' \subseteq R$. Consider $((u, v), (u, v')) \in R'$. Therefore, there exists $\hat{u} \vDash \mathsf{ipar}$ and $\hat{v} \vDash \mathsf{opar}'$ such that move' applied to positions of $\hat{v}$ give the v' witnessing $((u, v), (u, v')) \in R'$. By definition of opar', $\hat{v}_{\Gamma'} \vDash \mathsf{opar}$. Suppose, a position x of output type (τ, i) is moved from origin y in v to z in v'. This means $(\hat{u}, y, z) \vDash \mathsf{move}'_{(\tau,i)}$. Then, by definition of $\mathsf{move}'_{(\tau,i)}$, $(\hat{u}, y, z) \vDash \mathsf{move}_\tau$. This shows $R' \subseteq R$.

For the containment $R \subseteq R'$, consider $((u, v), (u, v')) \in R$. Therefore, there exists $\hat{u} \vDash \mathsf{ipar}$ and $\hat{v} \vDash \mathsf{opar}$ such that move applied to each position in $\hat{v}$ witnesses $((u, v), (u, v')) \in R$. This means for every position $x \in \mathrm{dom}(\hat{v})$ with output-type τ, there exist y, z, such that $(\hat{u}, y, z) \vDash \mathsf{move}_\tau$, $y = \mathit{orig}(v(x))$ and $z = \mathit{orig}(v'(x))$. For such a position $x \in \mathrm{dom}(\hat{v})$ of output type τ, let $i \in \{1, \ldots, k\}$ be such that there are exactly $i - 1$ positions $y_1 < y_2 < \ldots y_{i-1} < y$ such that $(\hat{u}, y_j, z) \vDash \mathsf{move}_\tau$. Let $\hat{v}'$ be the annotation of $\hat{v}$ where every position x is annotated with the index i as above. Clearly $\hat{v}' \vDash \mathsf{opar}'$ and therefore, $((u, v), (u, v')) \in R'$. We conclude $R = R'$. □

The boundedness restriction excludes resynchronizations such as the universal resynchronization R_{univ} (cf. Example 3.1.5) for which the problem of containment up to resynchronizer is undecidable. As we will see in Theorem 3.3.6 later, the boundedness restriction ensures that for every bounded, regular resynchronizer R and every 2NFT T, the relation $R(T)$ is realized

by a 2NFT with common guess (see page 12 for definition of 2NFT with common guess).

Another important property of bounded, regular resynchronizers is closure under composition.

Lemma 3.3.5. *The class of bounded regular resynchronizers is effectively closed under composition.*

Proof. Let $R = (\overline{I}, \overline{O}, \mathsf{ipar}, \mathsf{opar}, (\mathsf{move}_\tau)_\tau, (\mathsf{next}_{\tau,\tau'})_{\tau,\tau'})$ and $R' = (\overline{I}', \overline{O}', \mathsf{ipar}', \mathsf{opar}', (\mathsf{move}'_\lambda)_\lambda, (\mathsf{next}'_{\lambda,\lambda'})_{\lambda,\lambda'})$ be two bounded, regular resynchronizers. In view of Lemma 3.3.4, we can assume that both resynchronizers are 1-bounded. The composition $R \circ R'$ can be defined by combining the effects of R and R' almost component-wise. Some care should be taken, however, in combining the formulas next and next'. Formally, we define the composed resynchronizer $R'' = (\overline{I}'', \overline{O}'', \mathsf{ipar}'', \mathsf{opar}'', (\mathsf{move}''_{(\tau,\lambda)})_{\tau,\lambda}, (\mathsf{next}''_{(\tau,\lambda),(\tau',\lambda')})_{\tau,\lambda,\tau',\lambda'})$, where

- $\overline{I}''$ is the union of the parameters $\overline{I}$ and $\overline{I}'$,
- $\overline{O}''$ is the union of the parameters $\overline{O}$ and $\overline{O}'$,
- ipar'' is the conjunction of the formulas ipar and ipar';
- opar'' is the conjunction of the formulas opar and opar';
- $\mathsf{move}''_{(\tau,\lambda)}(y, z)$ states the existence of some position t satisfying both formulas $\mathsf{move}_\tau(t, z)$ and $\mathsf{move}'_\lambda(y, t)$;
- $\mathsf{next}_{(\tau,\lambda),(\tau',\lambda')}(z, z')$ requires that $\mathsf{next}_{\tau,\tau'}(z, z')$ holds and, moreover, that there exist some positions t, t' satisfying $\mathsf{move}_\tau(t, z)$, $\mathsf{move}_{\tau'}(t', z')$, and $\mathsf{next}_{\lambda,\lambda'}(t, t')$; note that these positions t, t' are uniquely determined from z, z' since R is 1-bounded, and they act, at the same time, as source origins for R and as target origins for R'.

By definition, $\mathsf{move}''_{(\tau,\lambda)}$ is 1-bounded, thus z and τ determine a unique t, which together with λ determines a unique y. It is also easy to see that R'' is equivalent to $R \circ R'$ as the positions corresponding to t in formulas $\mathsf{move}''_{(\tau,\lambda)}$ and $\mathsf{next}''_{(\tau,\lambda),(\tau',\lambda')}$ correspond to the source origin of R and target origin of R'. □

We now show why the boundedness restriction is important. For the purpose of Theorem 3.3.6 below, we assume the regular languages ipar, opar, move_τ and $\mathsf{next}_{\tau,\tau'}$ are all given as NFA. Recall that the regular languages corresponding to move_τ and $\mathsf{next}_{\tau,\tau'}$ will be over words with two marked positions, i.e, over the alphabet $\Sigma' \times \{0, 1\}^2$, with the last two components used to mark two positions in the input. The sentences ipar and opar correspond to languages over Σ' and Γ' respectively.

For complexity arguments we use the notion of PSPACE-constructibility. Recall that a 2NFT has PSPACE-constructible states and transitions with respect to n means that the states and transition relation can be enumerated using working space polynomial in n. In particular, this also means that

the NFA representation of the output languages in transitions have size polynomial in n.

Theorem 3.3.6 (Theorem 16 in [BMPP18])**.** *Given a bounded, regular resynchronizer R and a 2NFT (possibly with common guess) T one can construct a 2NFT with common guess T' such that*

$$T' =_o R(T)$$

Moreover, assuming the 2NFT T has n states and PSPACE*-constructible transitions w.r.t. n, and the resynchronizer relations* move*,* next*,* ipar *and* opar *are given as NFA of size at most k, the size of T' is exponential in n and k.*

Proof. To prove the theorem, it is convenient to first assume that R has no input/output parameters and T has no common guess. We will see later how to adapt the proof when input and output parameters and common guess are used.

Let $R = (\emptyset, \emptyset, \mathsf{ipar}, \mathsf{opar}, (\mathsf{move})_\tau, (\mathsf{next})_{\tau,\tau'})$ be a bounded regular resynchronizer with no input/output parameters. Note that the output-types τ are simply letters from Γ.

Applying the resynchronization defined by R to a 2NFT T can be seen as a repeated application of resynchronizations consisting of a single move_τ formula for every output-type τ, which changes the origins according to output-type τ, followed by resynchronizations corresponding to the $\mathsf{next}_{\tau,\tau'}$ formula, for every pair of output-types τ, τ'. The resynchronization corresponding to move_τ changes origins of output positions of type τ, whereas the resynchronization corresponding to $\mathsf{next}_{\tau,\tau'}$ does not change any origins, but discards those outputs that violate $\mathsf{next}_{\tau,\tau'}$. Given a 2NFT T and a resynchronization R corresponding to a single move_τ relation, we first show how to construct a 2NFT T' with $T' =_o R(T)$.

For sake of clarity in the proof, we assume that T outputs at most a single letter at each transition. This assumption can be made without loss of generality, since we can reproduce any word $v \in L$ that is outputted by some transition $(q, i) \xrightarrow{a|L} (q', i')$, letter by letter, with several transitions that move back and forth around position i. Note that this transformation adds polynomially many new states to simulate an original transition. This is because the transitions are PSPACE-constructible and therefore, we can use the states of the polynomial sized NFA-representation of the output language as intermediate states while simulating a transition.

Dealing with move. The idea here is that T' has to simulate an arbitrary run of T on an input u, by displacing the origins of any output letter with type τ from a source i to a target j, as indicated by $(u, i, j) \in \mathsf{move}_\tau$. The idea of the construction is as follows. Whenever T outputs a letter b

with origin i, T' non-deterministically moves to some position j such that, $(u, i, j) \in \mathsf{move}_b$ (recall that b is also the output-type of the produced letter since there are no output parameters). Then T' produces the same output b as T, but at position j. Finally, it moves back to the original position i. For the latter step, we will exploit the fact that move_b is bounded.

Since move_b is a regular property, there is a corresponding finite monoid $(M, \cdot, F)$, where $F \subseteq M$, and a monoid morphism $h : (\Sigma \times \{0,1\}^2)^* \to M$, such that $(u, i, j) \in \mathsf{move}_b$ if and only if $h(u, i, j) \in F$.

Recall that we denote by $u(k, k')$, $k \leq k'$, to be the factor between the k-th and k'-th positions of u (both included). Let $u' \in (\Sigma \times \{0,1\}^2)^*$ be a word with two marked positions. For any such word u' and for all $1 \leq k \leq k' \leq |u'|$, we then define $\ell_k = h(u'(1, k-1))$, $r_{k'} = h(u'(k'+1, |u'|))$, and $m_{k,k'} = h(u'(k, k'))$. We observe that

$$u' = (u, i, j) \in \mathsf{move}_b \qquad \text{iff} \qquad \begin{cases} \ell_i \cdot m_{i,j} \cdot r_j \in F & \text{if } i \leq j \\ \ell_j \cdot m_{j,i} \cdot r_i \in F & \text{if } i > j. \end{cases}$$

The elements ℓ_i and r_i associated with each position i of the word u' are functionally determined by u'. In particular the word $\ell_1 \dots \ell_{|u|}$ (resp. $r_1 \dots r_{|u|}$) can be seen as the run of a deterministic (resp. co-deterministic) automaton on u.

We now describe the required 2NFT with common guess T'. The transducer T' uses the common guess to annotate every input position k with ℓ_k and r_k. This can be done by an NFA for the common guess by simply checking that $\ell_i \cdot h(u(i)) = \ell_{i+1}$ and $r_i = h(u(i+1)) \cdot r_{i+1}$. The common guess NFA will need to store in its state the current ℓ_i and r_i.

Over the annotated input, the computation by T' is done in three phases, which we call the original phase, the simulation phase and the backtracking phase. The *original phase* simulates an arbitrary run of T as long as the transitions do not produce any output with letter b. In this phase, the state of T' is updated in the same way as in transitions in T.

To simulate a transition of T with output b, say $q \xrightarrow{a|b} q'$, that originates at a position i and produces the letter b, T' stores in its control state the transition rule to be simulated and the monoid element ℓ_i associated with the current position i (the source). It then guesses whether the displaced origin j (i.e. the target) is to the left or to the right of i. Consider the case where $j \geq i$ (the case $j < i$ is symmetric). In this case T' starts moving to the right, until it reaches some position $j \geq i$ such that $(u, i, j) \in \mathsf{move}_b$. This is equivalent to checking that $\ell_i \cdot m_{i,j} \cdot r_j \in F$. Once a target j is reached, T' produces the same output b as the original transition. We call this the *simulation phase*. In this phase, the states of T' need to store ℓ_i and compute $m_{i,j}$ during the run.

After the simulation phase, T' begins the *backtracking phase* for backtracking to the source i. During this phase, T' will maintain the previous

monoid elements ℓ_i, $m_{i,j}$, and, while moving backwards, compute $m_{i',j}$ for all $i' \leq j$. We claim that there is a unique i' such that $\ell_{i'} = \ell_i$ and $m_{i',j} = m_{i,j}$, and hence such i' must coincide with the source i. Indeed, if this were not the case, we could pump the factor of the input between the correct source i and some $i' \neq i$, showing that the relation move_b is not bounded.

Based on this, the transducer T' can move back to the correct source i, from which it can then simulate the change of control state from q to q' and move to the appropriate next position in the original phase. Any run of T' that simulates a run of T on input u, as described above, results in producing the same output v as T, but with the origins modified according to move_b.

In the original phase, the information needed in the states is simply the state in the original run of T. However, in the simulation phase, for moving from position i to j, T needs to store the letter b to be output, whether j is to the left or right, the monoid element ℓ_i, and compute the monoid element $m_{i,j}$ (which will be used in the backtracking phase). In the backtracking phase, this information is used to return to the correct position i. Therefore, the states of T' are in the set $Q \times \Gamma \times \{\ell, r\} \times M^2$, where Q is the set of states of T. The only dependency on the size of T is in the component Q and Γ. Therefore, the size is polynomial in n. The dependency on the resynchronizer comes from the M^2 component. Since the monoid M is exponential in the size of the NFA representation of move_b, the size of T' is exponential in k (which is a bound on the size of the NFA-representation of the resynchronizer relations).

Dealing with next. We now consider the case of $\mathsf{next}_{b,b'}$, which restricts the origins in such a way that pairs of origins j, j' associated with consecutive outputs positions x and $x+1$ labeled by letters b and b' respectively, such that (u, j, j') satisfies $\mathsf{next}_{b,b'}$. In particular, it will disallow runs that produce outputs that do not satisfy $\mathsf{next}_{b,b'}$.

To deal with this case, we assume that T' uses the common guess to annotate every input position with the set of lazy U-turns. Recall that a lazy left (right) U-turn at position i is a sequence of transitions starting at the $(i-1)$-th cut (i-th cut) and coming back to the same cut without producing any output and only reading positions to the left (right) of the cut. The set of lazy U-turns at position i can be checked using a common guess NFA as described in the proof of Theorem 2.2.1. We can normalize T with respect to lazy U-turns, i.e., construct $\mathsf{Norm}(T)$ that is origin-equivalent to T, such that $\mathsf{Norm}(T)$ does not have any lazy U-turns (see page 39). This means that if we have two consecutive output positions x and $x+1$ with origins j and j' respectively, then the run of $\mathsf{Norm}(T)$ moves from j to j' by a sequence of LR transitions (respectively RL transitions) if $j < j'$ (respectively $j > j'$). If $j = j'$, then the outputs x and $x+1$ are produced in a single

transition. Note that $\mathsf{Norm}(T)$ requires transitions with language outputs, in order to eliminate all U-turns. Note also that the states of $\mathsf{Norm}(T)$ need to store the set of lazy U-turns for the input positions. Therefore, $\mathsf{Norm}(T)$ has size exponential in $|T|$.

For the construction of T', we begin by following the same approach as in case of move. Consider the finite monoid $(M, \cdot, F)$, $F \subseteq M$, and a monoid morphism $h : (\Sigma \times \{0,1\}^2)^* \to M$, and some subset F of M, such that $(u, j, j') \in \mathsf{next}_{b,b'}$ if and only if $h((u, j, j')) \in F$. The element $h(u, j, j')$ is equal to either $\ell_j \cdot m_{j,j'} \cdot r_{j'}$ or $\ell_{j'} \cdot m_{j',j} \cdot r_j$ depending on whether $j \leq j'$ or $j \geq j'$. We assume $j \leq j'$, and the case of $j \geq j'$ can be handled similarly. As in the case of move_b, the elements ℓ_j and $r_{j'}$ are assumed to be available as explicit annotations of the input which can be checked by the common guess. The element $m_{j,j'}$, on the other hand, can be computed by T' while moving from j to j'. Note that, to compute the monoid element $m_{j,j'}$, it is important that $\mathsf{Norm}(T)$ does not have any lazy U-turns.

We now specify how T' simulates a run of $\mathsf{Norm}(T)$, while restricting the set of possible outputs. As explained above, T' maintains in its control state the monoid elements ℓ_j, $m_{j,j'}$, and r_j, where j is the origin of the last non-empty output and j' is the current input position (for simplicity, here we assume that $j \leq j'$, otherwise we swap the roles of j and j'). In addition, it also maintains whether the most recently produced output letter is b or not. Now, suppose $\mathsf{Norm}(T)$ takes a transition from the current position j' to a position j'', say with $j'' \geq j$, and outputs words from the language $L \subseteq \Gamma^*$. Let $L_\varepsilon = L \cap \{\varepsilon\}$, $L_{-b} = L \cap \Gamma^* b$, $L_{b'-} = L \cap b'\Gamma^*$ and $L_{-bb'-} = L \cap \Gamma^* bb'\Gamma^*$ and L_{rest} is the set of words in L that are in neither of the above languages. Then T' non-deterministically chooses to output a word. We describe below the updates in the states needed based on which of the above languages the word belongs to.

1. if it chooses to output a word in L_ε, then, assuming $L_\varepsilon \neq \emptyset$, it simulates the change of control state of the transition of $\mathsf{Norm}(T)$, moves from j' to j'', and updates the stored monoid element from $m_{j,j'}$ to $m_{j,j''} = m_{j,j'} \cdot h(a)$, where a is the input letter read while moving from j' to j''. This indicates the last output letter has not changed and the last position with non-empty output is still j;
2. if it chooses to output a word in $L_{b'-}$, then, assuming b was the last output symbol (given by the control state), it checks that $\ell_j \cdot m_{j,j'} \cdot r_{j'} \in F$. This corresponds to checking (u, j, j') satisfies $\mathsf{next}_{b,b'}$ and then T' simulates the change of control state of the transition of $\mathsf{Norm}(T)$, moves from position j' to j'', updates the stored monoid element from $m_{j,j'}$ to $m_{j',j''} = h(\varepsilon)$. This corresponds to the case where the last output letter was b and the first output letter in v is
3. if it chooses to output a word in $L_{-bb'-}$, then $\ell_{j'} \cdot r_{j'} \in F$. This corresponds to the case that bb' is a factor of v and checks (u, j', j')

satisfies $\mathsf{next}_{b,b'}$;

4. if it chooses to output a word in L_{-b}, then it updates in its control state the last produced output symbol to be b and updates the stored monoid element from $m_{j,j'}$ to $m_{j',j''} = h(\varepsilon)$.
5. if it chooses to output a word in L_{rest}, then it updates the stored monoid element from $m_{j,j'}$ to $m_{j',j''} = h(\varepsilon)$. This corresponds to producing a non-empty output which do not contribute to any bb' factor in the output. The control state is updated to reflect that the last produced output letter is not a b.

Note that for a choice of output in L, it might be possible for it to be included in many of the languages above, for example $v = b'ab$ will be in $L_{b'-}$ and in L_{-b}. Therefore, in general, we will need to check a combination of the above conditions.

In case conditions 2. or 3. are not satisfied, i.e, $\ell_j \cdot m_{j,j'} \cdot r_{j'}$ does not belong to the set F, then the control of T' moves to a rejecting sink state. The above behaviour guarantees the following property: for consecutive output positions x and $x+1$ labeled by b and b' and with origins j, j' in a run of T such that $(u, j, j') \notin \mathsf{next}_{b,b'}$, the corresponding run in T' is rejecting.

The transducer T' needs to store two monoid elements, the original states of $\mathsf{Norm}(T)$ and the last output letter produced. Therefore T' has states polynomial in the size of $\mathsf{Norm}(T)$ and exponential in the size of R, since the monoid M can be exponential in the size of the NFA-representation of next. Since $\mathsf{Norm}(T)$ itself is exponential in the size of T, the size of T' is exponential in both $|T|$ and $|R|$.

We now need to show that these constructions for move_b and $\mathsf{next}_{b,b'}$ are independent and can be done one after the other. Note that in each of these constructions, we build a 2NFT with common guess. The common guess can be done in one step, by having the guesses on marked by the transducer they correspond to. For example, suppose T_1 and T_2 are two 2NFTs with common guess such that the common guess of T_1 (respectively T_2) is over the alphabet C_1 (respectively C_2) such that C_1 and C_2 are disjoint. Then we can obtain a 2NFT with common guess over $C_1 \times C_2$. Therefore, the 2NFTs can be applied one-after another, i.e, we can obtain a 2NFT with common guess describing $R(T)$. Since each of the intermediate steps incur a polynomial blowup in n and exponential blowup in k, the final 2NFT T' will also be polynomial sized in n and exponential sized in k.

To complete the proof, we discuss how to generalise to the case with input and output parameters. In this case, we can modify T and add common guess such that T reads inputs over $\Sigma \times 2^{\overline{I}}$, guessing a valuation of the parameters $\overline{I}$, and produces outputs over $\Gamma \times 2^{\overline{O}}$, guessing a valuation of $\overline{O}$. We could then apply the constructions that follow, and finally modify the resulting transducer T' by projecting away the input and output anno-

tations. We observe that the projection operation on the input and output is easy and can be implemented directly at the level of the transitions of T' by taking a product with the NFA representation of the regular languages ipar and opar. Since ipar and opar are already given as part of the input, we observe that complexity bounds are preserved. □

Since $R(T)$ can be defined by a 2NFT with common guess, the problem of checking containment up to a given bounded, regular resynchronizer reduces to the origin-containment problem. As a corollary, we obtain the following result.

Corollary 3.3.6.1 (Corollary 17 in [BMPP18])**.** *Given two 2NFTs T_1 and T_2, and a bounded regular resynchronizer R, whose relations are given by NFA, one can decide whether $T_1 \subseteq_o R(T_2)$ in* ExpSpace*. Furthermore, if the bounded, regular resynchronizer R is fixed (i.e, not part of the input), the complexity decreases to* PSpace*.*

When the regular resynchronizer is given as part of the input, the 2NFT describing $R(T_2)$ can be of size exponential in size of R and T_2. Since the origin-containment problem is in PSpace, we obtain an upper bound of ExpSpace. When the regular resynchronizer considered is fixed, then the complexity decreases to PSpace, since the size of 2NFT describing $R(T_2)$ is polynomial in size of T_2.

Regular Resynchronizability. We now discuss some properties of regular resynchronizers. Note that in a bounded, regular resynchronizer, the different components are used for different purposes. The input and output parameters are used to identify special positions, the move relation is used to change origins, and the next relation is used to restrict the origins of consecutive output positions. However, the parameters and the next relations only restrict the set of origin graphs obtained. In other words, given a bounded regular resynchronizer $R = (\overline{I}, \overline{O}, \mathsf{ipar}, \mathsf{opar}, (\mathsf{move})_\tau, (\mathsf{next})_{\tau,\tau'})$, we can define $R' = (\emptyset, \emptyset, \Sigma^*, \Gamma^*, \mathsf{move}_R, \text{true})$, where there are no input and output parameters, and no restrictions defined by ipar, opar and next relations. The move_R relation in R' is the union of move_τ for all output-types $\tau \in \Gamma \times 2^{\overline{O}}$ of R. In other words, $(u, i, j) \in \mathsf{move}_R$ if and only if, there exists $\hat{u} \in \mathsf{ipar}$, and a output-type τ, such that $(\hat{u}, i, j) \in \mathsf{move}_\tau$. With slight abuse of notation, we can represent the resynchronizer R' by move_R, since the other components of the resynchronizer are trivial. The following Proposition follows by definition of move_R.

Proposition 3.3.7. *For any 2NFT T, $R(T) \subseteq_o \mathsf{move}_R(T)$.*

This result can be rephrased as follows: the parameters and the next relation only restrict the set of synchronized pairs obtained after applying

the resynchronizer. In other words, they are used to disallow some runs of the resynchronized transducer $\mathsf{move}_R(T)$, but do not add any new runs.

As in the case of rational resynchronizers, given 2NFT T_1 and T_2, we say T_1 is *resynchronizable* to T_2 by a bounded, regular resynchronizer, if there exists a bounded, regular resynchronizer R such that $T_1 \subseteq_o R(T_2)$. We denote this by $T_1 \preceq_{\text{reg}} T_2$. With respect to the regular resynchronizability relation ($T_1 \preceq_{\text{reg}} T_2$), Proposition 3.3.7 implies that we can assume regular resynchronizers do not have parameters nor next relations. It was shown by Kuperberg and Martens [KM20], that $\preceq_{\text{reg}}$ is a pre-order on 2NFTs, i.e, it is reflexive and transitive.

Proposition 3.3.8 (Lemma 11 in [KM20])**.** *The relation $\preceq_{reg}$ is a pre-order on 2NFTs.*

Proof. The identity resynchronizer can be defined by having $(u, i, i) \vDash \mathsf{move}$, for all $u \in \Sigma^*$, and for all $i \in \text{dom}(u)$. Thus $\preceq_{\text{reg}}$ is reflexive.

Transitivity of regular resynchronizability follows from closure under composition of bounded, regular resynchronizers proved in Lemma 3.3.5. □

Similar to what happens with $\preceq_{\text{rat}}$, the following example by Kuperberg and Martens [KM20], shows that $\preceq_{\text{reg}}$ is not symmetric, and therefore not an equivalence relation.

Example 3.3.9. *Let T_1 and T_2 be the NFTs given in Figure 3.7. Both T_1 and T_2 take inputs of the form a^n and outputs a word of the form c^m. The origin-output of T_1 is $(c,1)(c,2)\dots(c,n)^{(m-n+1)}$, if $m \geq n$, and $(c,1)\dots(c,m)$ otherwise. On the other hand, the origin-output of T_2 is of the form $(c,1)^m$. In other words, T_1 produces a single c for every a read, and produces the remaining c's, if any, at the last letter of the input with origin n, whereas T_2 produces the c's all at once at the first input position.*

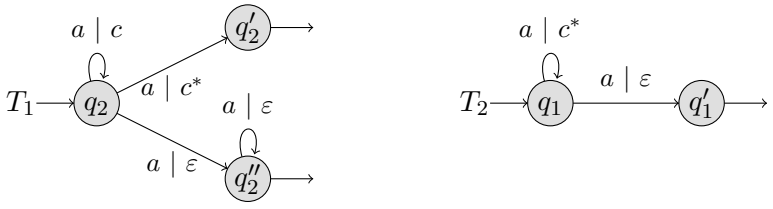

Figure 3.7 – Example to show $\preceq_{\text{reg}}$ is not symmetric

By taking R with move_c containing words of the form $(a^n, 1, j)$ for all $1 \leq j \leq n$, and $\mathsf{next}_{c,c}$ containing words of the form $(a^n, j, j+1)$ for all $1 \leq j \leq n$ and (a^n, n, n), we get $T_1 \subseteq_o R(T_2)$. Therefore, $T_1 \preceq_{\text{reg}} T_2$. However, there is no bounded regular resynchronizer R' such that $T_2 \subseteq_o R'(T_1)$. This is because the resynchronizer would need to move the origins

to a single position. Therefore, for an input a^n, the move' relation of R' needs to contain all pairs of the form $(a^n, i, 1)$ for all $1 \leq i \leq n$. This is clearly not bounded, and therefore $T_2 \not\preceq_{\text{reg}} T_1$.

We now present a machine independent characterization of the synchronized pairs defined by transducers T_1 and T_2 that captures the (non) existence of a bounded, regular resynchronizer. This result, due to Kuperberg and Martens [KM20], states that $T_1 \preceq_{\text{reg}} T_2$ if and only if pairs of origin graphs from T_1 and T_2 are *bounded-traversal*. Here, we present the notion of bounded-traversal by adapting it to the formalism of synchronized pairs instead of origin-graphs.

Bounded Traversal Resynchronizers (Definition 13 in [KM20]). Let (u, v) and (u, v'), where $v_\Gamma = v'_\Gamma$, be two synchronized pairs with the same input word and output word (without origins). Given two input positions $i, j \in \text{dom}(u)$, we say i *traverses* j, if there exists an output position x labeled by (a, i) in v, such that

— $i \leq j$ and x is labeled by (a, j') in v' for some $j' > j$ (left-to-right traversal).

— $i \geq j$ and x is labeled by (a, j') in v' for some $j' < j$ (right-to-left traversal).

Intuitively, i is said to traverse j when a resynchronizer maps (u, v) to (u, v') and moves some outputs from origin i to j' such that j is in between i and j'.

A pair $((u, v), (u, v'))$ is *k-traversal* if for every input position j, there exist at most k positions $i_1, \ldots i_k \in \text{dom}(u)$ such that i_ℓ traverses j for all $1 \leq \ell \leq k$.

An important observation in the above definition is that we require each of the i_ℓ to be the origin of some output position. Therefore, it is still possible to resynchronize the origin to an input position j' far from the source origin i, while being k-traversal.

A bounded, regular resynchronizer is said to be *bounded-traversal* if there exists a natural number k if for every pair $((u, v), (u, v'))$ in the resynchronization relation defined by R, $((u, v), (u, v'))$ is k-traversal.

One of the main results of Kuperberg and Martens [KM20] is to construct for every natural number k, a bounded, regular resynchronizer Trav_k, such that $((u, v), (u, v')) \in \text{Trav}_k$ *if and only if*, $((u, v), (u, v'))$ have k-traversal. The intuitive idea in constructing the resynchronizer Trav_k is as follows. There are $2k$ input parameters, Right_ℓ and Left_ℓ for $\ell = 1, \ldots, k$. The set of input positions defined by these parameters should satisfy the criterion that no position i in Right_ℓ (respectively Left_ℓ) traverses another position j marked by the same parameter from left to right, i.e, $i \leq j$ (respectively right to left, i.e, $i \geq j$).

The move formula of the resynchronizer Trav_k acts as follows. Let i, j

be positions in Right_ℓ such that no position between i and j is in the set Right_ℓ. The move formula can then move the origin of an output position from i to j', for any j' such that$i \leq j' \leq j$. Similarly, we can define a move formula for Left_ℓ sets. Note that the same position can be in multiple such sets Right_ℓ for different ℓ.

We illustrate this on an example using the origin graph representation. Consider the origin graphs given in Figure 3.8, where the solid lines represent a origin graph G_1 and dashed lines represent origin graph G_2. The position 1 only traverses both 1 and 2. The position 2 traverses 2 and 3. The position 3 only traverses itself, and all the traversals are left-to-right traversals. Therefore, the pair of origin graph is 2-traversal. Consider the resynchronizer with input parameters Right_1 marking the positions 1 and 3 and Right_2 marking the position 2. According to the bounded-traversal resynchronizer Trav_2, the origins can be moved from position 1 to anywhere between 1 and 3, and from position 3 to anywhere greater than or equal to 3.

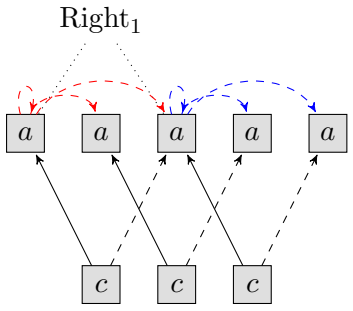

Figure 3.8 – An example to illustrate bounded-traversal resynchronizer

Note that the resynchronizer Trav_k is bounded due to the definition of the sets Right_ℓ and the corresponding move. For any target origin j, there can be exactly one source from a set Right_ℓ (or Left_ℓ). This is because if there are two positions i and i' such that $(u, i, j) \vDash \text{move}$ and $(u, i', j) \vDash \text{move}$, then both $i < j$ and $i' < j$. W.l.o.g, assume $i < i'$. Then i traverses i', as some output position can be redirected from i to j and i' is a position between i and j and this contradicts the definition of move. This also implies that for any fixed target origin j, there can be at most $2k$ distinct source positions, each corresponding to different set Right_ℓ or Left_ℓ.

The class of resynchronizers Trav_k for all $k \in \mathbb{N}$ is called the class of bounded-traversal resynchronizers. The main result about this class of resynchronizers is that they characterize the class of bounded, regular resynchronizers.

Theorem 3.3.10 (Lemma 16 in [KM20])**.** *A regular resynchronizer is bounded* if and only if *it is bounded-traversal.*

An application of this theorem is to show non-existence of a bounded, regular resynchronizer.

Example 3.3.11. *Consider the 2NFTs T_1 and T_2 that do the identity and reverse transformation on unary alphabet. Therefore, on input a^n, T_1 produces $(a,1)(a,2)\dots(a,n)$, whereas T_2 produces $(a,n)\dots(a,1)$. Suppose there exists a bounded, regular resynchronizer R such that $T_1 \subseteq_o R(T_2)$. The origin of the $(n+1)$-th position is the same in both T_1 and T_2. This means the last n positions traverses the $n+1$-th position, since they have to be redirected from some position $> n+1$ to some position $< n+1$. Therefore, R is not bounded-traversal, which is a contradiction.*

Therefore, as a consequence, for any 2NFTs, $T_1 \preceq_{\text{reg}} T_2$ if and only if there exists a k such that $T_1 \subseteq_o \text{Trav}_k(T_2)$. Therefore, bounded-traversal resynchronizers are universal in the sense that if two transducers are resynchronizable by a bounded, regular resynchronizer, they are resynchronizable by some Trav_k. This class can be seen as analogous to the class of bounded-delay resynchronizers when considering rational resynchronizers. Indeed, recall that for rational resynchronizers, Theorem 3.2.6 states that two real-time NFTs are resynchronizable if and only if they are resynchronizable by a bounded-delay resynchronizer.

Other possible formalisms for resynchronizers. We defined regular resynchronizers using the input and output parameters, the move relation to define a change of origin, and the next relation to restrict the changed origins. The most important criterion for this choice is to define change of origin without changing the underlying input-output pair. One motivation for introducing the next relation is to restrict the transformed synchronized pairs to order-preserving ones, namely those synchronized pairs that are produced by NFTs instead of 2NFTs. Definitions were also chosen so that the set of target synchronized pairs could be generated by means of a transducer. We discuss some potential alternative formalisms and the problems with those approaches.

To define resynchronizers for 2NFTs, one natural idea would be to use logical transformations on origin graphs. Recall that an origin graph G is an equivalent representation of a synchronized pair (u,v), which has nodes corresponding to input and output positions labeled with letters from Σ and Γ, and edges from output positions to the corresponding origins. Specifically, for an output position (a,x), there is an edge from that output position to the x-th input position. Therefore, classical MSO-logic based transformations of graphs [Cou97, Eng97] can be used to define resynchronization relations. We refer to [Cou97] for a introduction to MSO-definable graph transductions. A natural question is to ask whether the regular resynchronizers (with or without the boundedness restriction) is a special case of

MSO-definable transformation of origin graphs. Note that we require the input and output words to be fixed. Therefore, the graph transduction should not change the domain or the input and output orders, but only change the origin relation. Given a MSO-formula $\mathsf{move}(y, z)$, the modified origin relation $orig'(x, z)$ to define a graph transduction, denoting that the modified origin of output position x is the input position y, can be defined as the conjunction of the conditions $orig(x, y)$ and $\mathsf{move}(y, z)$. However, this runs into the problem that the same position x is assigned different origins $z_1, \dots z_k$, whenever $\mathsf{move}(y, z_i)$ is satisfied. This can be resolved by considering NMSO-transductions which allow to choose a single z_i from all the possible choices. Therefore, it is worthwhile to consider the more general NMSO-definable transductions of origin graphs as a generalisation of regular resynchronizers.

The MSO-formula defining the modified origin relation can talk about the entire origin graph. This makes it difficult to find a restriction that allows to define $R(T)$ by means of a 2NFT. However, MSO-logic over origin graphs generated by 2NFT is undecidable, as one can encode grids in those origin graphs:

Example 3.3.12. *Consider the 2NFT T that copies an input u an arbitrary number of times. Therefore on an input u of length k, the output produced can be $((u(1), 1)(u(2), 2) \dots (u(k), k))^m$ for any $m > 0$. The origin graph representation for $k = 3$ and $m = 2$ is given in Figure 3.9.*

A 2×3 grid can be interpreted in the output part with the 1st and 2nd copy of w in the output as the 1st and 2rd rows respectively. The horizontal edges correspond to the successor inside w and the vertical edges corresponds to same positions of w in consecutive copies. In this way, we can interpret an $m \times k$ grid inside an origin graph. In Figure 3.9, the output vertices are labeled by numbers from 1 to 6 to identify the corresponding vertices in the grid.

Figure 3.9 – A 2×3 grid in the output of a 2NFT

For all the above reasons, we restrict our attention to the simpler model of bounded, regular resynchronizer.

There are other decidable logics over origin graphs, such as the logic introduced by Dartois et al. in [DFL18]. However, this logic is not suitable for defining resynchronizers for various reasons. Firstly, its expressive power is incomparable to 2NFTs. Moreover, the logic defines a word transduction with origins, but a resynchronizer requires defining a transduction of origin graphs.

Another possible approach is to define resynchronizers as transformations of data words. A data word is a word over the alphabet $\Omega \times \mathbb{N}$, where Ω is a finite alphabet and $\mathbb{N}$ is an infinite set of data values. For example, the output v in a synchronized pair (u, v) is a data word. The synchronized pair can indeed be represented as the data word $(u(1), 1)(u(2), 2) \dots (u(n), n)v$. Transformations of data words have been already studied, such as the model of SDST (Streaming Data String Transducers), which was the original motivation behind studying NSSTs [AC11]. However, they fail to have various desirable properties, such as composition, and they do not enjoy equivalent characterizations. Another model of data transductions through origins was studied by Habermehl and Durand-Gasselin [DGH16]. In both these models, a finite number of data values can be stored in *data variables*. The transducer can later use these data variables to assign the data value at an output position. For the purpose of resynchronizer, the assigned data value would be the origin of the position. This means that all the assigned target origins must be at some point stored in the data variable, and thus must be source origin of some position. This restriction makes it an unsatisfactory model to define resynchronizers.

3.4 Rational vs Regular Resynchronizers

In this section, we study the expressiveness of the two models of resynchronizers previously defined, namely the rational resynchronizers and the bounded regular resynchronizers, when restricted to NFTs. Recall that the rational resynchronizer changes origins by using a transformation of the synchronization language. On the other hand, a bounded regular resynchronizer uses different formulas such as move, next, etc to define changes of origin. Given a rational resynchronizer R and a bounded regular resynchronizer R', we call them *equivalent* if $[\![R]\!] = [\![R']\!]$. Our main result in this section is to show that rational resynchronizers are a strict subclass of bounded regular resynchronizer. In other words, for every rational resynchronizer, there exists an equivalent bounded, regular resynchronizer. However, there exist bounded regular resynchronizers which cannot be equivalently expressed by any rational resynchronizer, even when the resynchronization is restricted to order-preserving synchronized pairs.

Theorem 3.4.1. *Rational resynchronizers are a strict subclass of bounded regular resynchronizers.*

We first prove that for every rational resynchronizer, we can build an equivalent bounded, regular resynchronizer.

Theorem 3.4.2. *For every rational resynchronizer R, there exists an equivalent 1-bounded, regular resynchronizer R'. In particular, for all NFTs T, T_1 and T_2, we have $R(T) =_o R'(T)$, and $T_1 \preceq_{rat} T_2$ implies $T_1 \preceq_{reg} T_2$.*

We fix an NFT R over $\Sigma \uplus \Gamma$ that defines a rational resynchronizer. We assume without loss of generality that R is letter-to-letter and *trimmed*, namely, every state in R occurs in some successful run. Note that R maps synchronized words to synchronized words. Recall that, we use the terms 'source' (resp. 'target') to refer to a synchronized word that is an input (resp. an output) of R. On the other hand, we shall use the terms 'input' and 'output' to refer to the projections of a synchronized word over Σ and Γ, respectively (note that, in this case, it does not matter whether the synchronized word is the source or the target, since these have the same projections over Σ and Γ). The goal is to construct a 1-bounded, regular resynchronizer R' (with parameters) that defines the same resynchronization as R.

The main idea is to encode the run of the rational resynchronizer R on the input and use this encoding to define the transformation of origins. Consider a successful run of R, say $\rho = q_0 \xrightarrow{c_1 \mid d_1} q_1 \xrightarrow{c_2 \mid d_2} \dots \xrightarrow{c_n \mid d_n} q_n$, and define relations omatch_ρ and imatch_ρ between transitions of ρ. These relations are used later to define a bijection between source and target origins and ultimately define the move formula.

The relation omatch_ρ consists of all pairs (i, j) of positions of ρ such that c_i and d_j are output letters and $(c_1 c_2 \dots c_i)_\Gamma = (d_1 d_2 \dots d_j)_\Gamma$ Therefore, omatch_ρ matches transitions of ρ that correspond to the same occurrence of an output letter in the source and the target. Note that omatch_ρ is in fact a partial bijection. In a similar way, we define imatch_ρ as the partial bijection that contains all pairs (i, j) of positions of ρ such that c_i and d_j are input letters and $(c_1 c_2 \dots c_i)_\Sigma = (d_1 d_2 \dots d_j)_\Sigma$. Thus, imatch_ρ matches the same input positions in source and target.

Here we give an example of the omatch_ρ relation. Consider the pair of source and target synchronized words in Figure 3.10 with $\Sigma = \{a\}$ and $\Gamma = \{b\}$ which could be realized by a successful run ρ of R. Because R is letter-to-letter, any position in any of the two words corresponds precisely to a position in the run ρ, so we can represent the relations omatch_ρ and imatch_ρ by means of edges between source and target positions. In Figure 3.10, the solid vertical edges between the source and target positions represent pairs of omatch_ρ, while the dashed edges represent some pairs of imatch_ρ. The significance of the backward horizontal edges will be explained later and can be temporarily overlooked.

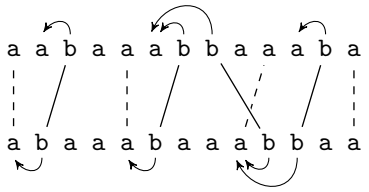

Figure 3.10 – imatch_ρ and omatch_ρ

Mapping the source to target origins. We now explain how the relations imatch_ρ and omatch_ρ can be used to define the required move formula. We do so by first using Figure 3.10. Consider any output letter at position i in the source synchronized word w, for e.g. the first occurrence of b with $i = 3$. Let i' be the last Σ-labelled position before i in w, as indicated by the backward arrow on top ($i' = 2$ in the example). This position i' determines the source origin $j = |w(1, i')_\Sigma|$ of the output position.

To find the corresponding target origin, we observe that the position i is mapped via the relation omatch_ρ (solid vertical arrow) to some position k in the target synchronized word w'. Let k' be the last Σ-labelled position before k in w'(backward arrow at the bottom), and map k' back to a position ℓ in the source via the relation imatch_ρ (dashed edges). In our example, $k = 2$ and $k' = 1 = \ell$. The position ℓ in w determines precisely the target origin $\ell' = |w(1, \ell)_\Sigma|$ of the considered output letter. The above steps describe a correspondence between two transitions i' and ℓ in ρ, with labels over Σ, that is precisely defined by

$$\exists i, k, k', j, \ell' \quad \begin{cases} \rho[i', i] \text{ consumes a word in } \Sigma\Gamma^+ \\ (i, k) \in \mathsf{omatch}_\rho \\ \rho[k', k] \text{ produces a word in } \Sigma\Gamma^+ \\ (\ell, k') \in \mathsf{imatch}_\rho \\ j = |w(1, i')|_\Sigma \\ \ell' = |w(1, l)|_\Sigma \end{cases} \qquad (\star)$$

The first (resp. third) conditions ensure that i' (resp. k') is the last input position before i (resp. k). In the above equation, $\rho[i', i]$ represents the part of ρ between positions i', i (both i', i included).

Note that the quantified positions j, ℓ' are uniquely determined by i' and ℓ thanks to the last two conditions. We denote by match_ρ the relation of all pairs (j, ℓ') that satisfy Equation $(\star)$. This determines a correspondence between source and target origins on the input projection.

We intend to use match to define the move relation for the regular resynchronizer. However, there are several issues with this approach: the relation match is not yet a partial bijection (since different output positions may have

the same source origin but different target origins), Moreover, the match relation needs to be implemented by means of a MSO-formula move_τ that only considers positions of the input, but it needs to be able to reason about a fixed run of R. Therefore, the transitions in the run on output letters need to be encoded into the input annotation. Below, we explain how to overcome these issues.

Hereafter, we call *output block* any maximal factor of a synchronized word that is labeled over Γ. Intuitively, this corresponds to a maximal factor of the output that originates at the same input position. We first consider, as a simpler case, a rational resynchronizer R that reads *source* synchronized words where the lengths of the output blocks are uniformly bounded by some constant, say B (a similar property holds for the blocks of the target synchronized words, using lag-based arguments).

Case A. Bounded output blocks: Encoding the runs. In this case we can encode any successful run ρ of R entirely on the input, by annotating every Σ-labelled position i with a factor ρ_i of ρ that reads the input symbol at position i, followed by the sequence of output symbols up to the next input symbol. Note that every factor ρ_i has length at most $B+1$ since there can be at most B many output positions in the block. This annotation can be done using the input parameters since the length of ρ_i is bounded by B. The correctness of the input annotation can be checked by an MSO-sentence ipar.

Let $\hat{u}$ denote the input word u where every position i is annotated with ρ_i. Given a factor of the run $\rho_i \in \Sigma\Gamma^+$, $\rho_i[1] \in \Sigma$ is the first position of the factor ρ_i.

In addition, we also annotate the output word with indices from $\{1, \ldots, B\}$, which we call *offsets*, in such a way that an output position x is annotated with an offset o if and only if it is the o-th output position with the same source origin. These will form the output parameters $\overline{O}$. Note that the correctness of the annotation cannot be checked by an MSO-sentence opar that refers only to the output. The check will be done instead by a combined use of the formulas move_τ and $\mathsf{next}_{\tau,\tau'}$.

Checking validity of the encoding. We first check using next that, for every pair of consecutive output positions x and $x+1$ annotated with the offsets o and o' (recall that offset is a number in $\{1, \ldots, B\}$), respectively, it holds that $o' = o+1$ or $o' = 1$, depending on whether the *source* origins of x and $x+1$ coincide or not. For this we let $(\hat{u}, k, k') \vDash \mathsf{next}_{\tau,\tau'}$, if

1. either $o' = o+1$ and there is $j = j'$ such that $(\hat{u}, j, k) \vDash \mathsf{move}_\tau$ and $(\hat{u}, j', k') \vDash \mathsf{move}_{\tau'}$,
2. or $o' = 1$ and there are $j < j'$ such that $(\hat{u}, j, k) \vDash \mathsf{move}_\tau$ and $(\hat{u}, j', k') \vDash \mathsf{move}_{\tau'}$.

Note that the above definition of the formula $\mathsf{next}_{\tau,\tau'}$ refers to the formulas move_τ and $\mathsf{move}_{\tau'}$ and needs to guess the correct source origins j and j'. However, $\mathsf{next}_{\tau,\tau'}$ must be defined in terms of the *target* origins of output positions x and $x+1$. Assuming that for every output type τ the formula move_τ, which will be defined later, determines a *partial bijection* between input positions, the above definition of $\mathsf{next}_{\tau,\tau'}$ can be used to determine the positions k and k' uniquely and hence define $\mathsf{next}_{\tau,\tau'}$ as required. We will define move_τ such that indeed it defines a partial bijection.

It remains to check that maximal offset occurring in an output block with origin j coincides with the number of output symbols produced by the corresponding factor ρ_j of the run. Thus, we modify slightly the definition of $\mathsf{next}_{\tau,\tau'}$ in case 2., as follows:

2'. or $o' = 1$ and there are $j < j'$ such that $(\hat{u}, j, k) \vDash \mathsf{move}_\tau$ and $(\hat{u}, j', k') \vDash \mathsf{move}_{\tau'}$, *and* $o = |\rho_j| - 1$.

Note that the factor ρ_j can be derived by inspecting the annotation of the input position j. The modification guarantees that the output annotation is correct for all output blocks but the last one. The annotation for the last output block can be checked by marking the last output position with a distinguished symbol (using again some parameters) and by requiring that if τ witnesses the marked symbol and the offset o, then move_τ can be satisfied by a triple of the form $(\hat{u}, j, k)$, with $o = |\rho_j| - 1$. We omit the tedious definitions in this case. This concludes the definitions of next and how it is used to check correctness of the encoding of a run.

Having the input correctly annotated with the factors ρ_j of ρ and the output correctly annotated with the offsets, we can encode the i-th transition of ρ by a pair (j, o) that consists of a position j of the input and an offset $o \in \{0, 1, \ldots, B\}$. The encoding is defined in such a way that $i = \sum_{j'<j} |\rho_{j'}| + o + 1$ (in particular, $o = 0$ when the i-th transition consumes an input symbol, otherwise $o \geq 1$). We use this encoding to translate the relations omatch_ρ, imatch_ρ, and match_ρ, to equivalent finite unions of partial bijections between input positions parameterized by output-types. We begin by explaining the translation of omatch_ρ.

Translation of omatch_ρ. Consider any pair $(i, i') \in \mathsf{omatch}_\rho$. Since the i-th transition of ρ consumes an output symbol in the source, it is encoded by a pair of the form (j, o), with $o \geq 1$. On the other hand, the i'-th transition may consume either an input letter or an output letter in the source, but always produces an output letter in the target. In the former case, i' is encoded by a pair $(j', 0)$; in the latter case, it is encoded by a pair (j', o'), with $o' \geq 1$.

As an example, in the Figure 3.11, $(7, 4) \in \mathsf{omatch}_\rho$. Position 7 of the run is encoded as $(5, 1)$ on the input. The transition at position 4 consumes an input symbol a, and produces the output symbol b, and is encoded as

$(3, 0)$.

In general, we observe that the lag induced just after the o-th transition of ρ_j must be equal to the number of output symbols produced between the $(o' + 1)$-th transition of $\rho_{j'}$ and the o-th transition of ρ_j, both included (when the lag is negative one follows the transitions in reverse order, i.e, from j' to j counting negatively). As an illustration in the figure, the lag after the first transition of ρ_5 is 2, which is the number of output symbols in the dotted box. The dotted box consists of the symbols produced between the first transition of $\rho_{3'}$ and the first transition of ρ_5, and has two output symbols.

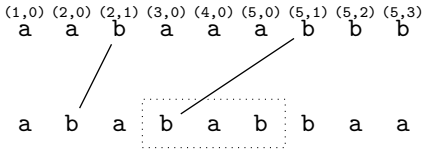

Figure 3.11 – An example illustrating omatch_ρ

Translation of imatch_ρ. The translation of the relation imatch_ρ is similar. The only difference is that now the pairs $(i, i') \in \mathsf{imatch}_\rho$ are encoded by tuples of the form $\big((j, o), (j', o')\big)$, with $o = 0$ since the i-th transition consumes an input symbol in the source. The i'-th transition, as before, can consume an input symbol or an output symbol in the source.

Consider the Figure 3.12, where $(2, 3) \in \mathsf{imatch}_\rho$. Position $i = 2$ is encoded as $(2, 0)$. The transition at position 3 consumes an output letter b (and produces the input letter a). Position 3 is encoded as $(2, 1)$.

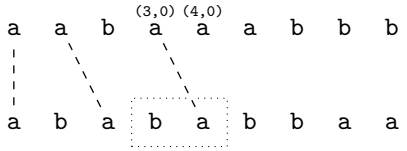

Figure 3.12 – An example illustrating imatch_ρ

The only difference here is that one has to relate the lag with the number of *input* letters produced between (both positions included) the first transition of ρ_j and the o'-th transition of $\rho_{j'}$. Again, in the figure, the lag after the first transition of ρ_3 is 1, which is the number of input symbols in the dotted box. The dotted box contains the symbols produced between the first transition of ρ_3 and the first transition of $\rho_{4'}$, and has one input symbol.

Relations encoding omatch_ρ and imatch_ρ. We can represent omatch_ρ as a finite union of relations $O_{o,o'} \subseteq (\Sigma \times 2^{\overline{I}})^* \times \mathbb{N} \times \mathbb{N}$, each describing a regular property of annotated inputs with two distinguished positions, (j, o) and (j', o') in it, in such a way that the positions are bijectively related one to another.

Likewise, we can represent imatch_ρ as a finite union of relations $I_{0,o'}$, each describing a regular property of annotated inputs with two distinguished positions encoded as $(j, 0)$ and (j', o') in it, which are bijectively related one to another.

Translation of match_ρ. We finally turn to the translation of the relation match_ρ, which will eventually determine the relations move_τ of the desired regular resynchronizer R'. This is done by mimicking Equation ($\star$) via the encoding of positions in the run ρ using pairs of input positions and offsets, and more precisely, by replacing the variables i', i, k, k', ℓ of Equation (*) with the pairs $(j, 0)$, (j, o), (m', o'), (m'', o''), $(j', 0)$.

Formally, for every offset $o \in \{1, \ldots, B\}$, we define the set M_o of all triples $(\hat{u}, y, z)$, where $\hat{u}$ is an annotated input and j, j' are positions in it that satisfy the following property:

$$\exists m', m'' \bigvee_{0 \le o', o'' \le B} \begin{cases} \rho_j[1, o+1] \text{ consumes a word in } \Sigma\Gamma^+ \\ (\hat{u}, j, m') \in O_{o,o'} \\ \rho_{m''}[o''+1, |\rho_{m''}|]\ \rho_{m''+1} \cdots \rho_{m'-1}\ \rho_{m'}[1, o'+1] \\ \qquad\qquad \text{produces a word in } \Sigma\Gamma^+ \\ (\hat{u}, j', m'') \in I_{0,o''}. \end{cases} \tag{$\star\star$}$$

Note that the first condition holds trivially by definition of ρ_j, while the third condition is easily implemented by accessing the factors $\rho_{m''}, \ldots, \rho_{m'}$ of ρ that are encoded by the input parameters. For simplicity, here we assumed that (m'', o'') is lexicographically before (m', o'); to treat the symmetric case, one has to interpret the definition by considering the sequence of transitions in reverse, i.e, from (m', o') to (m'', o''). The intended meaning of $(\hat{u}, j, j') \in M_o$ is as follows. Suppose that the input is correctly annotated with the factors ρ_j of a successful run ρ of R, and that the output position x of ρ is correctly annotated with an offset o. Assuming that x is the o-th output position with source origin j, then j' is its target origin in ρ.

Continuing with our running example, we determine the target origin for the point b annotated (5,1), whose source origin is (5,0) (see Figure 3.13). We will find the target origin of this b annotated (5,1). As seen in the computation of omatch_ρ, we know that $(u, 5, 3) \in O_{1,0}$. The factor $\rho_5 = abbb$, and $\rho_5[1, 2] = ab \in \Sigma\Gamma^+$, and as we have seen, $(u, 5, 3) \in O_{1,0}$. Now,

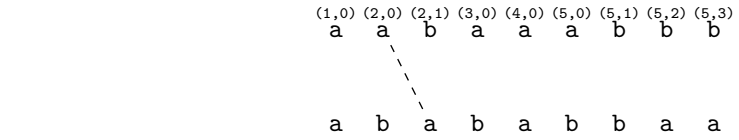

Figure 3.13 – Figure depicting the M_o relation

consider the part of the source u annotated with $(2,1)(3,0)$. This produces the output $ab \in \Sigma\Gamma^+$. That is, for $m'' = 2, o'' = 1$, and $m' = 3, o' = 0$, we have $\rho_{m''}[o''+1, 2]\ \rho_{m'}[1, o'+1] = \rho_2[2,2]\rho_3[1,1] = ba$ produces the output $ab \in \Sigma\Gamma^+$.

Consider $(j', 0) = (2, 0)$. The lag after the a at $i = 2$ annotated $(2, 0)$ is 1. Also, $(2, 3) \in \mathsf{imatch}_\rho$. The position 3 consumes an output and produces an input a. Indeed, the lag after the first transition of ρ_2 is 1, which is the number of input symbols between the first transition of ρ_2 and the second transition ($(o'+1)$-th transition) of ρ_2. That is, $(u, 2, 2) \in I_{0,1}$. Thus, starting with the b annotated $(j, o) = (5, 1)$ such that $\rho_5[1, 2] \in \Sigma\Gamma^+$, we first obtain $(m', o') = (3, 0)$ with $(u, 5, 3) \in O_{1,0}$. Further, $\rho_2[2, 2]\rho_3[1, 1]$ produces a word in $\Sigma\Gamma^+$. Finally, we have $(u, 2, 2) \in I_{0,1}$, obtaining $(u, 5, 2) \in M_1$.

Definition of move_τ. We could define move_τ just as M_o, for every $\tau = (a, o) \in \Gamma \times \{1, \ldots, B\}$. However, recall that the correctness of the output annotation is guaranteed only once if ensure that move_τ defines a partial bijection between input positions j and j' (hereafter we say for short that the relation is bijective), which is not known a priori. Bijectiveness can be enforced syntactically, by defining move_τ as $\{(\hat{u}, j, j') \in M_o \mid \forall(\hat{u}, m, m') \in M_o\ (j = m) \leftrightarrow (j' = m')\}$. Observe that either M_o is bijective, and hence $\mathsf{move}_\tau = M_o$, or it is not, and in this case move_τ defines a subrelation of M_o that is bijective. Note that, in the case where move_τ defines a subrelation of M_o, there will be no induced pair of synchronized words, since the origins of some output elements could not be redirected. This is fine, and actually needed, in order to avoid generating with R' spurious pairs of synchronized words, that are not also generated by R. On the other hand, observe that the formula move_τ does generate, for appropriate choices of the output annotations, all the pairs of synchronized words that are generated by R. We finally observe that move_τ and $\mathsf{next}_{\tau,\tau'}$ can be defined in MSO. We obtain in this way, a 1-bounded, regular resynchronizer R' equivalent to R.

Case B. The general case. We generalize the previous ideas to capture a rational resynchronizer R with source output blocks of possibly unbounded length. One additional difficulty is that we cannot anymore encode a successful run ρ of R entirely on the input, as ρ may have arbitrarily long factors

on outputs blocks. Another difficulty is that we cannot uniquely identify the positions in an output block using offsets ranging over a fixed finite set. We will see that a solution to both problems comes from covering most of the output by factors in which the positions behave similarly in terms of the source-to-target origin transformation. Intuitively, each of these factors can be thought of as a 'pseudo-position', and accordingly the output blocks can be thought of as having boundedly many pseudo-positions. This will make it possible to apply the same ideas as before. We now state the key lemma that identifies the aforesaid factors. By a slight abuse of terminology, we call output blocks also the maximal Γ-labeled factors of a synchronized word.

Lemma 3.4.3. *Let R be a rational resynchronizer with set of states Q. Let ρ be a successful run of R, and let w and w' be the source and target synchronized words induced by ρ.*

- *Every output block v of w can be factorized into $\mathcal{O}(|Q|^2)$ sub-blocks $v_1, \ldots, v_n$ such that if $|v_i| > 1$ and ρ_i is the factor of ρ that corresponds to v_i, then all states in ρ_i have the same lag, say ℓ_i, and the factor of ρ obtained by extending ρ_i to the left and to the right by exactly $|\ell_i|$ transitions forms a loop of R, i.e, begins and ends with the same state.*
- *Moreover, for every factorization $v = v_1 \ldots v_n$ as above, each sub-block v_i is also a factor of w', and hence all positions in v_i have the same target origin.*

Before proving the Lemma, we give an example to illustrate how the Lemma is used in presence of unbounded output blocks. In Figure 3.14, we show a rational resynchronizer with $\Sigma = \{a\}$, $\Gamma = \{c\}$.

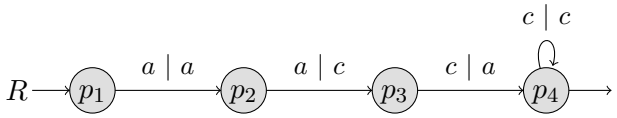

Figure 3.14 – Resynchronizer with unbounded output block

Consider the source and target synchronized words aac^{k+1} and $acac^k$ respectively, where $k \geq 0$. The source has exactly one output block c^{k+1}. The state p_3 has lag 1 whereas all other states have lag 0. The sequence of states in the run on the output block is $p_3 p_4 \ldots p_4$. Therefore, the decomposition of the output block c^{k+1} into $v_1 v_2$ with $v_1 = c$ and $v_2 = c^k$, satisfies the conditions of the Lemma. For the block $v_1 = c$, the corresponding state p_3 has lag 1. By shifting 1 to the right, we obtain a loop on state p_4. The same holds for positions in v_2 since the lag of p_4 is 0.

Furthermore, the blocks v_1 and v_2 have different origins in the target synchronized word as well, illustrating the second condition of the Lemma. The intended use of the Lemma is to decompose the unbounded block into

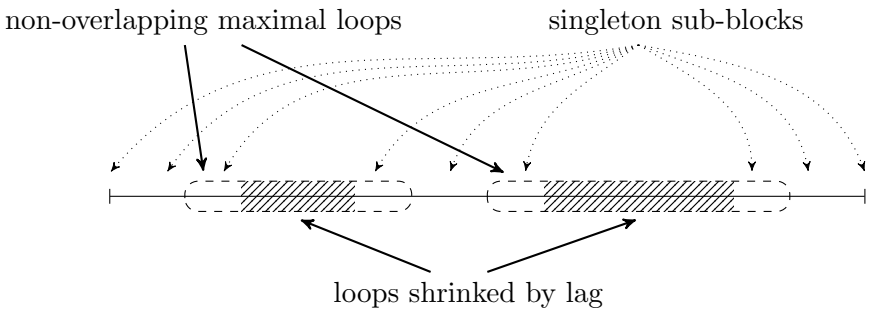

Figure 3.15 – Factorization of an output block.

a bounded number of factors, which are then moved together in the target. Now, we prove this Lemma.

Proof of Lemma 3.4.3. We prove the first claim of the lemma (Figure 3.15 provides an intuitive account of the constructions). Let v be an output block of the source synchronized word w and let ρ' be the factor of the run ρ aligned with v. As a preliminary step, we fix a maximal set of pairwise non-overlapping maximal loops inside ρ', say $\rho'_1, \ldots, \rho'_m$. A simple counting argument shows that $m \leq |Q|$ and that there are at most $|Q|$ positions in ρ' that are not covered by the loops $\rho'_1, \ldots, \rho'_m$. The latter positions determine some sub-blocks of v of length 1 in the required decomposition.

The remaining sub-blocks of v will be obtained by factorizing the loops $\rho'_1, \ldots, \rho'_m$, as follows. Consider any loop ρ'_j. By construction, all letters consumed by ρ'_j occur in v, so they are output letters. Similarly, all letters produced by ρ'_j are also output letters, since otherwise, by considering repetitions of the loop ρ'_j, one could get different lags, violating Lemma 3.2.3. This means that the lag associated with all the states along ρ'_j is constant, say ℓ_j ($\leq |Q|$). If ρ'_j has length at most $2|\ell_j|$, then we simply decompose it into $2|\ell_j|$ factors of length 1. Otherwise, we cover a prefix of ρ'_j with $|\ell_j|$ factors of length 1, and a suffix of ρ'_j with $|\ell_j|$ other factors of length 1. The remaining part of ρ'_j is covered by a last factor of length $|\rho'_j| - 2|\ell_j|$.

Overall, this induces a factorization of v into at most $|Q|$ (the sub-blocks not covered by a loop) + $|Q| \cdot (2|Q| + 1)$ is decomposed into $(2\ell_j + 1) \leq (2|Q| + 1)$ sub-blocks).

This gives $\mathcal{O}(|Q|^2)$ sub-blocks $v_1, \ldots, v_n$. Moreover, by construction, if $|v_i| > 1$, then in the corresponding factor ρ_i of ρ, all states have the same lag, say ℓ_i, and if we extend ρ_i to the left and to the right by exactly $|\ell_i|$ transitions, we get back one of the loops ρ'_j (recall that each loop ρ'_j of length $> 2|\ell_j|$ is decomposed into $|\ell_j|$ blocks of length 1, then a block of length $|\rho'_j| - |\ell_j|$, and finally, $|\ell_j|$ blocks of length 1. Clearly, if we extend the middle block on either side by blocks of length $|\ell_j|$, then we get back ρ'_j. This proves the first claim of the lemma.

As for the second claim, suppose that $v_1, \ldots, v_n$ is a factorization of an output block v of w satisfying the first claim. Clearly, every sub-block v_i of length 1 is also a factor of the target synchronized word w'. The interesting case is when a sub-block v_i has length larger than 1. In this case, by the previous claim, we know that in the corresponding factor ρ_i of ρ, all states have the same lag ℓ_i, and the factor ρ'_i of ρ that is obtained by expanding ρ_i to the left and to the right by $|\ell_i|$ transition is a loop. In fact, since ρ'_i is a loop, we also know that all states in it have lag ℓ_i. Now, to prove that v_i is a factor of the target synchronized word w', it suffices to show that every two consecutive positions of ρ_i are mapped to consecutive positions via the relation omatch_ρ. This follows almost by construction, since for every pair $(j,k) \in \mathsf{omatch}_\rho$, if j occurs inside the factor ρ_i, then k occurs inside the loop ρ'_i (recall that ρ'_i consumes and produces only output symbols), and hence $k = j - \ell_i$. In addition, if $j+1$ also occurs inside ρ_i, then clearly $(j+1, k+1) \in \mathsf{omatch}_\rho$. This proves that v_i is a factor of the target synchronized word w', and hence all positions in it have the same target origin. □

We now continue with the proof of Theorem 3.4.2. In view of the above lemma we can guess a suitable factorization of the output into sub-blocks that refine the output blocks, and treat each sub-block as if it were a single pseudo-position. In particular, we can annotate every sub-block with a unique offset from a finite set of quadratic size w.r.t. $|Q|$. The role of the offsets will be the same as in the case of bounded output blocks, namely, determine some partial bijections $O_{o,o'}$, $I_{0,o'}$, and M_o between positions of the input. In addition, we annotate every sub-block with the pair consisting of the first and last states of the factor of the successful run that consumes that sub-block. We call such a pair of states a *pseudo-transition*, as it plays the same role of a transition associated with a single output position. Finally, we annotate every input position j with a sequence of bounded length that represents a single transition on j followed by the pseudo-transitions on the subblocks with source origin j. The resulting input annotation provides an abstraction of a successful run of R.

The correctness of the above annotations can be enforced by defining suitable formulas ipar, opar, $\mathsf{next}_{\tau,\tau'}$ for the regular resynchronizer R'. We omit the tedious details concerning these formulas, and only observe that, as before, the definition $\mathsf{next}_{\tau,\tau'}$ relies on the fact that move_τ and $\mathsf{move}_{\tau'}$ define partial bijections between input positions.

Finally, we turn to describing the formulas move_τ that maps source to target origins for τ-labeled output positions. The definition is basically the same as before, based on some auxiliary relations $O_{o,o'}$ and $I_{0,o''}$ that implement omatch_ρ and imatch_ρ at the level of input positions. As before, we guarantee, by means of a syntactical trick, that move_τ determines a partial bijection between input positions. In conclusion, we get a regular

resynchronizer R', with input and output parameters, that is equivalent to the rational resynchronizer R. This concludes the proof of Theorem 3.4.2. □

Not all regular resynchronizers are rational. We now give an example of a bounded, regular resynchronizer R and NFTs T_1 and T_2, such that $T_1 \subseteq_o R(T_2)$, but for which there is no rational resynchronizer R' such that $T_1 \subseteq_o R'(T_2)$. This will show that bounded, regular resynchronizers are *strictly* more expressive than rational ones, even when applied to NFTs. We have already seen in Example 3.2.8, some T_1 and T_2 such that no rational resynchronizer exists. Recall that in the example, we had $Sync(T_1) = ac^*a^*$ and $Sync(T_2) = a^*ac^*$. In other words, the origin of every output position in T_1 is the first position and the origin of every output position in T_2 is the last position. Therefore, it suffices to define a regular resynchronizer R with $\mathsf{move}_c(y, z) : \mathit{first}(y) \wedge \mathit{last}(z)$, where $\mathit{first}(y)$ checks that y is the first input position and $\mathit{last}(z)$ checks that z is the last input position. It is easy to see that R is 1-bounded. Clearly $T_1 \subseteq_o R(T_2)$, but no rational rational resynchronizer R' exists. This concludes the proof of Theorem 3.4.1. □

Expressiveness restricted to Real-time NFTs. Observe that the NFTs T_1 and T_2 in the previous example were not real-time. For real-time NFTs, rational resynchronizers and bounded, regular resynchronizers are in fact equally expressive.

Theorem 3.4.4. *Given real-time NFTs T_1 and T_2, the following conditions are equivalent*

1. *$T_1 \subseteq_o Del_k(T_2)$ for some integer k.*
2. *$T_1 \subseteq_o R(T_2)$ for some rational resynchronizer R.*
3. *$T_1 \subseteq_o R'(T_2)$ for some bounded, regular resynchronizer R'.*
4. *$T_1 \subseteq_o Trav_k(T_2)$ for some bounded-traversal resynchronizer $Trav_k$.*

Proof. The equivalence of statements 1. and 2. follows from Theorem 3.2.6, as shown by Filiot et al. [FJLW16]. The equivalence of statements 3 and 4 follow from Theorem 3.3.10, as shown by Kuperberg and Martens [KM20]. The implication 2. to 3. follows from Theorem 3.4.2. We now prove the implication 4. to 1. which completes the proof of the Theorem.

Suppose there exists a bounded-traversal resynchronizer Trav_k for some k such that $T_1 \subseteq_o \mathrm{Trav}_k(T_2)$. We also assume that the length of the longest output in any transition of the real-time NFTs T_1 and T_2 is at most ℓ. Let $w_1 \in Sync(T_1)$ and $w_2 \in Sync(T_2)$ be synchronized words and (u, v_1) and (u, v_2) be the corresponding synchronized pairs such that $((u, v_1), (u, v_2)) \in \mathrm{Trav}_k$.

Consider an input position $i \in \mathrm{dom}(u)$. Let u' be the prefix of u of length i. Let v_1' and v_2' be the maximal prefixes of outputs v_1 and v_2 that

have origins in u'. Since $((u, v_1), (u, v_2))$ is k-traversal, there exists at most k positions $i_1, \ldots, i_k$ such that i_j traverses i. We want to prove $|\mathit{diff}(v_1', v_2')| < k \cdot \ell$.

Suppose $|v_1'| \leq |v_2'|$. Note that for every $j \in \{1, \ldots, k\}$, i_j must be the origin of some position x in v_1. If $\mathit{orig}(v_1(x)) \leq i$, then x is a position in v_1' and therefore $\mathit{orig}(v_2(x)) \leq i$, which contradicts that i_j traverses i. Therefore, $\mathit{orig}(v_1(x)) > i$ and for i_j to traverse i, it must be that $\mathit{orig}(v_2(x)) \leq i$. Therefore, x is a position in the difference between v_1' and v_2'. Since there are at most k positions that traverses i and each of these position is the origin of at most ℓ output positions, there are $k \cdot \ell$ such position in the difference of v_1' and v_2'. The case of $|v_1' > |v_2'|$ is symmetric.

Therefore, $\mathit{delay}(w_1, w_2) < k \cdot \ell$, and thus, $T_1 \subseteq_o \mathrm{Del}_{k \cdot \ell}(T_2)$. □

Summing up, our last theorem shows that relativized to real-time NFTs, the expressive power of rational resynchronizers and bounded, regular resynchronizers is the same. This establishes an equivalence between logic-based resynchronizers and machine-based resynchronizers.

3.5 Conclusions

We recalled the model of rational resynchronizers and introduced bounded, regular resynchronizers as a means to study problems for transducers between the classical and origin semantics. We showed that the resynchronized transformations can be defined by means of a transducer (NFT or 2NFT with common guess). This allows us to reduce the problem of containment up to a given resynchronizer to the origin-containment problem. A general direction for future work is to study classical problems through the lens of resynchronizers. For example, in Chapter 5, we will study the one-way definability problem up to resynchronizers, which asks whether a given 2NFT is origin-equivalent to a NFT up to a resynchronizer.

We also saw examples of transducers for which no rational/bounded, regular resynchronizers exist. The non-existence of resynchronizer can be proved by the notion of lag for rational resynchronizers and bounded-traversal for the regular counterpart. An interesting problem, which we will study in Chapter 4, is to decide whether such a resynchronizer exists or not.

In Section 3.4, we saw the comparison between the two models of resynchronizers. In particular, for real-time NFTs, the models are equally expressive, thus yielding a logic-machine equivalence result. An interesting problem is to find a logic-based model of resynchronizers for rational resynchronizers, or a machine-based model equivalent to bounded, regular resynchronizers. Another direction for future work would be to lift the notion of resynchronizers from words to more general objects such as trees, graphs, etc.

Chapter 4

Synthesis Problem for Resynchronizers

In Chapter 3, we introduced rational resynchronizers and bounded, regular resynchronizers and showed that the containment up to a given resynchronizer (of any of the two forms) is decidable. In that case, the resynchronizers and the transducers were both given as input to the problem.

In this Chapter, we consider the **resynchronizer synthesis problem**, where the input consists of two transducers $T_1 \subseteq T_2$, and the problem is to decide if there is a resynchronizer R such that $T_1 \subseteq_o R(T_2)$. If such a resynchronizer exists, we also want to synthesize it.

Problem 4.0.1. *(Resynchronizer Synthesis Problem) Given two transducers $T_1 \subseteq T_2$, compute, whenever possible, a resynchronizer R such that $T_1 \subseteq_o R(T_2)$?*

Note that checking $T_1 \subseteq T_2$ is already undecidable [FR68, Gri68], even for NFTs. Therefore, it is important to assume that the NFTs given are such that $T_1 \subseteq T_2$. Thus, this is a promise problem.

The problem of course depends on the class of transducers (such as NFTs, 2NFTs, etc) and the class of resynchronizers (rational or bounded, regular) that we consider. Recall that we have already given examples of transducers T_1 and T_2 for which no rational/bounded, regular resynchronizer R exists, such that $T_1 \subseteq_o R(T_2)$ (see Examples 3.2.8 and 3.3.9). The problem of checking whether such a resynchronizer exists (but not synthesizing) can therefore be seen as containment up to an unknown resynchronizer.

We show that checking whether a rational/bounded, regular resynchronizer exists such that $T_1 \subseteq_o R(T_2)$ is undecidable, even when the input transducers are NFTs. We also identify cases where the problem is decidable. For rational resynchronizers, we show that a rational resynchronizer can always be synthesized if the given transducers are functional or even finite-valued. For synthesis of regular resynchronizers, we look at the case where the given

transducers are unambiguous, i.e, they admit at most one accepting run on every input. For this simpler case, we show that the problem is decidable and give an algorithm to synthesize the required resynchronizer.

4.1 Synthesis Problem for NFTs

We first prove that the synthesis problem for rational resynchronizers is undecidable. The proof we present is a modification of the proof of undecidability of synthesis of bounded, regular resynchronizer for NFTs by Kuperberg and Martens [KM20].

Theorem 4.1.1. *The rational resynchronizer synthesis problem is undecidable for NFTs.*

Proof. We consider a variant of the Post Correspondence Problem (PCP), called the *PCP-boundedness problem*, and reduce it to the synthesis problem for rational resynchronizers.

An instance of the problem consists of a finite set of pairs of words $\mathcal{P} = \{(u_i, v_i) \in (\Gamma^*)^2 \mid i \in I\}$ as input, where I is a finite set of indices. Given a sequence of indices $x = i_1 i_2 \dots i_k \in I^*$, let $u_x = u_{i_1} u_{i_2} \dots u_{i_k}$, and $v_x = v_{i_1} v_{i_2} \dots v_{i_k}$. We assume that $\mathcal{P}$ satisfies the following properties:

- $\nexists x \in I^*$, such that $v_x \sqsubseteq u_x$, where $\sqsubseteq$ is the prefix relation.
- There is an infinite sequence $\hat{x} = i_1 i_2 \dots$ of indices such that for all $x \in I^*$, $u_x \sqsubseteq v_x$ if and only if $x \sqsubset \hat{x}$. In this case, x is called a partial solution to the PCP-instance.

The *PCP-boundedness* problem asks whether there exists a bound k, such that, for all partial solutions x to the PCP instance it holds that $|v_x| - |u_x| < k$. If this is the case, we say $\mathcal{P}$ is k-bounded.

The undecidability of the *PCP-boundedness* problem is obtained from the following boundedness problem: given a deterministic Turing machine (DTM) that does not halt on empty tape, does it use a bounded amount of tape? The classical reduction of DTM to PCP produces a set of pairs $\mathcal{P}$ in such a way that partial solutions x correspond to partial computations of the DTM, with v_x ahead of u_x by one configuration. Therefore, tape-boundedness of the DTM reduces to the PCP-boundedness problem.

We now construct two NFTs T_1 and T_2 which read a sequence of indices x to simulate the PCP-boundedness problem. These NFTs are shown in Figure 4.1. The NFT T_1 has only one state p_0 with a transition that reads any index i and outputs v_i. Note that T_1 is deterministic.

The NFT T_2 has initial state q_0 with a loop reading any index i and outputting the corresponding word u_i, and therefore can output u_x by staying in the state q_0. T_2 can also guess that the input sequence does not correspond to a partial solution; in that case, it takes the transition from q_0 to q_1, where it reads index i and outputs any word u_i' such that $u_i' \not\sqsubseteq u_i$ and

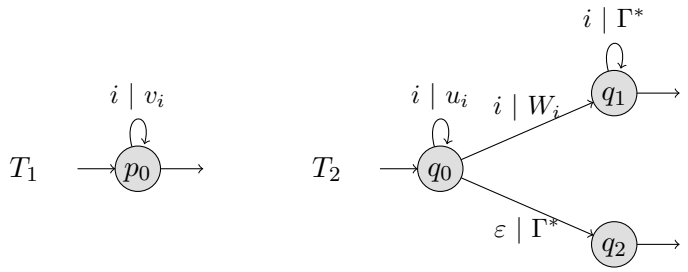

Figure 4.1 – NFTs T_1 and T_2

$|u'_i| \leq |u_i|$. The set of such words u'_i is denoted by W_i. This ensures that the output produced in this case is different from u_x. In state q_1, T_2 can read any index i and output any word in Γ^*. From q_0, T_2 can also move to state q_2 and produce any output in Γ^*. Intuitively, T_2 produces u_x, which lags behind the v_x produced by T_1. The transition to state q_2 allows T_2 to catch up with T_1.

If the input x corresponds to a partial solution of the PCP instance, then T_2 has a run staying at state q_0 which produces u_x. Since x is a solution to the PCP instance, $u_x \sqsubseteq v_x$ and therefore, T_2 can move to state q_2 and produce the rest of the output to match with v_x. If the input x does not correspond to a partial solution of the PCP instance, it can be decomposed into $x = x_1 i x_2$ such that x_1 is still a partial solution of the PCP instance, but $x_1 i$ is not. Therefore, $u_{x_1} \sqsubseteq v_{x_1}$ but $u_{x_1} u_i \not\sqsubseteq v_{x_1} v_i$. In this case, consider the run of T_2 that reads the input x_1 at state q_0, moves to state q_1 on reading i. This partial run produces the output $u_{x_1} u'_i$ such that $u'_i \not\sqsubseteq u_i$ and $|u'_i| \leq |u_i|$. Since $u_{x_1} u_i \not\sqsubseteq v_{x_1} v_i$, u'_i can be chosen such that $u_{x_1} u'_i \sqsubseteq v_{x_1} v_i$ and $|u'_i| \leq |u_i|$. At state q_1, T_2 can immediately produce the rest of the suffix of $v_{x_1} v_i$ and produce v_i on reading the index i. This run produces exactly the output v_x. Therefore, in the classical semantics, we would have $T_1 \subseteq T_2$.

Suppose $\mathcal{P}$ is k-bounded. We prove that the delay between T_1 and T_2 is bounded by k. After reading a prefix x of the input, there are two cases. If x is a partial solution to the PCP instance, T_2 has a run producing u_x. Since, $\mathcal{P}$ is B-bounded, $|\mathit{diff}(u_x, v_x)| \leq k$. Recall that $\mathit{diff}(u_x, v_x)$ here will be a word u, such that $u_x u = v_x$ (see page 66 for definition). If x does not correspond to a partial solution of the PCP instance, then let $x = x_1 i x_2$ such that $u_{x_1} \sqsubseteq v_{x_1}$, i.e, x' is the maximal prefix of x that is a partial solution of the PCP instance. Since $\mathcal{P}$ is k-bounded, $|\mathit{diff}(u_{x_1}, v_{x_1})| \leq k$. Therefore, there exists a run of T_2 that reads the letter i after x_1, produces $\mathit{diff}(u_{x_1}, v_{x_1}.v_i)$, i.e, the suffix of $v_{x_1} v_i$ after u_{x_1}. This can be done since at state q_1, T_2 can produce any arbitrary output. After this transition, T_2 can produce v_i on reading index i and therefore the delay is bounded by 0. We conclude that when $\mathcal{P}$ is k-bounded, $T_1 \subseteq_o \mathrm{Del}_k(T_2)$. Recall that Del_k is the k-delay resynchronizer which means after reading any sequence of input

indices, the difference in the lengths of the produced outputs can differ by at most k.

If $\mathcal{P}$ is not k-bounded, then for every bound k, there exists a prefix $x \sqsubseteq \hat{x}$ such that $|\mathit{diff}(u_x, v_x)| > k$. Note that since x is a prefix of $\hat{x}$, the only run of T_2 that matches with the run of T_1 is the one that stays in state q_0.

Suppose there exists a rational resynchronizer R such that $T_1 \subseteq R(T_2)$. Let ℓ be a bound on the lag in any state of R and B be a bound on the length of u_i's and v_i's. Recall that a state q of a rational resynchronizer has lag ℓ if every run that reaches q, reads ℓ more input letters in the source than in the target (see page 65 for definition).

Consider the sequence $x = i_1 \dots i_m$ such that $|v_x| - |u_x| > (\ell+1) \cdot B$. Let w_2^x denote the synchronized word $i_1 v_{i_1} \dots i_m v_{i_m}$ and $w_1^x = i_1 u_{i_1} \dots i_m u_{i_m}$‘. Since $u_x \sqsubseteq v_x$, $|w_2^x| > |w_1^x| + (\ell+1) \cdot B$. Suppose R reaches state q after reading w_2^x in the source. In the target, R would have consumed $(\ell+1) \cdot B$ more letters, which means it must have consumed at least $\ell+1$ input symbols, since each output block has size at most B. Therefore, $\mathsf{lag}(q) = \ell + 1$, which contradicts the bound on the lag at any state of R. We conclude that when $\mathcal{P}$ is not bounded, such a rational resynchronizer cannot exist. □

Based on Theorem 4.1.1, the synthesis problem for rational resynchronizers is undecidable. We observe that in this proof, one can also replace rational resynchronizers by bounded, regular resynchronizers. If $\mathcal{P}$ is bounded, existence of rational resynchronizer implies existence of an equivalent bounded, regular resynchronizer as well (see Theorem 3.4.2). For the other case, if the PCP instance is not bounded, it is easy to show that for a partial solution $x = i_1 \dots i_m$, such that $|v_x| - |u_x| > k \cdot B$, at least k positions traverse i_m. Therefore, by Theorem 3.3.10, there is no bounded, regular resynchronizer R such that $T_1 \subseteq_o R(T_2)$.

Theorem 4.1.2 (Theorem 20 in [KM20])**.** *Given two NFTs $T_1 \subseteq T_2$, it is undecidable to check if there exists a bounded, regular resynchronizer R, such that $T_1 \subseteq_o R(T_2)$.*

Therefore, the synthesis problem for both rational and bounded, regular resynchronizers are undecidable. Note that in these proofs, the left-hand side NFT T_1 is even deterministic. We now look at some restrictions on the class of transducers considered, which make the problem of synthesizing resynchronizers decidable.

4.2 Decidable restrictions

4.2.1 Synthesizing Rational Resynchronizers

We first look at the problem of synthesizing rational resynchronizers. Since the problem is undecidable for general NFTs, a natural restriction

would be to consider functional and finite valued NFTs. This is motivated by the fact that classical equivalence problem is also undecidable for NFTs, but becomes decidable when restricted to finite-valued NFTs [CK86].

Finite valued NFTs. Recall that an NFTs is finite-valued, i.e, there exists a bound k such that for every input u, there are at most k different output in the classical semantics. The following result shows that, for finite-valued NFTs, classical containment and rational resynchronizer synthesis are equivalent.

Theorem 4.2.1. *Let T_1, T_2 be k-valued NFTs. The following conditions are equivalent, and decidable:*

1. $T_1 \subseteq T_2$,
2. $T_1 \subseteq_o R(T_2)$ *for some resynchronization* R,
3. $T_1 \subseteq_o R(T_2)$ *for some rational resynchronizer* R.

Proof. The decidability comes from the fact that containment is decidable for finite-valued 2NFTs [CK86] (condition 1). The implications from 3. to 2. and 2. to 1. are trivial. We prove the only interesting implication from 1. to 3. Suppose that T_1, T_2 are k-valued NFTs such that $T_1 \subseteq T_2$. We use a result that k-valued NFTs can be decomposed into union of k unambiguous NFTs [Web96, SdS10]. Therefore, we can transform the k-valued NFT T_2 into an equivalent union of k unambiguous NFTs $T_2' = \bigcup_{i=1}^{k} T_2^i$. Note that while T_2 and T_2' are equivalent in the classical semantics, in the origin semantics, we only have $T_2' \subseteq_o T_2$. This is because in the decomposition result, the runs of T_2' correspond to some, but not all runs of T_2. Therefore some runs may not simulated in T_2' and the corresponding output in the origin semantics are not generated by T_2'.

Since $T_1 \subseteq T_2'$, we can compute a d-delay (in particular, a rational) resynchronizer Del_d such that $T_1 \subseteq_o \mathrm{Del}_d(T_2')$. This uses the following theorem from Filiot et al [FJLW16].

Theorem 4.2.2 ([FJLW16])**.** *Let T_1, T_2 be NFTs, where T_2 is k-ambiguous. One can compute a d-delay resynchronizer Del_d, for some $d \in \mathbb{N}$, such that $T_1 \subseteq T_2$ implies $T_1 \subseteq_o Del_d(T_2)$.*

Finally, since $T_2' \subseteq_o T_2$, $T_1 \subseteq_o \mathrm{Del}_d(T_2')$, and $\mathrm{Del}_d(T_2') \subseteq_o \mathrm{Del}_d(T_2)$, we get $T_1 \subseteq_o \mathrm{Del}_d(T_2)$. This concludes the proof. □

Therefore, the synthesis problem for rational resynchronizers restricted to finite-valued NFTs is decidable. Furthermore, the synthesized resynchronizer is also a bounded-delay resynchronizer.

Functional NFTs. Functional NFTs are a special case of k-valued NFTs where $k = 1$. Recall that it can be decided in PSPACE whether an NFT is functional [BCPS03], and that the classical containment problem for functional NFT is also in PSPACE [BH77]. We show a stronger result for the synthesis problem for rational resynchronizers restricted to functional NFTs than in the finite-valued setting. The result is stronger because we show that for functional NFTs, if there exists a rational resynchronizer R such that $T_1 \subseteq_o R(T_2)$, we can also assume $T_1 =_o R(T_2)$.

Theorem 4.2.3. *Let T_1, T_2 be two functional NFTs. The following conditions are equivalent, and decidable:*

1. $T_1 \subseteq T_2$,
2. $T_1 \subseteq_o R(T_2)$ *for some resynchronization* R,
3. $T_1 =_o R(T_2)$ *for some rational resynchronizer* R.

Proof. The implication from 3. to 2. and 2. to 1. are trivial. For the remaining implication, from 1. to 3., suppose that T_1, T_2 are functional NFTs such that $T_1 \subseteq T_2$. We construct a rational resynchronizer R over the disjoint union $\Sigma \uplus \Gamma$ of the input and output alphabets of T_1, T_2, using a variant of the direct product of T_1 and T_2. More precisely, let $T_1 = (Q_1, q_1, \Delta_1, F_1)$, $T_2 = (Q_2, q_2, \Delta_2, F_2)$, and $R = (Q, q, \Delta, F)$, where $Q = Q_1 \times Q_2$, $q = (q_1, q_2)$, $F = F_1 \times F_2$, Δ contains all transitions of the form $(s_1, s_2) \xrightarrow{aw_2 \,|\, aw_1} (t_1, t_2)$, with $s_i \xrightarrow{a \,|\, w_i} t_i$ in Δ_i for both $i = 1$ and $i = 2$. Intuitively, the transducer R simulates a run of T_1 and a run of T_2 in parallel, by repeatedly consuming an input symbol a and the corresponding output w_2 produced by T_2, and producing the same input symbol a followed by the corresponding output w_1 of T_1. Since T_1 and T_2 are functional and classically contained one in the other, we have that R maps strings over $\Sigma \uplus \Gamma$ to strings over $\Sigma \uplus \Gamma$ while preserving the projections on the input and on the output alphabets. This means that R is indeed a resynchronizer. Finally, T_1 is clearly origin equivalent to $R(T_2)$. Note that here R is not a letter-to-letter transducer. However, since it is a length-preserving NFT, we can obtain an equivalent letter-to-letter transducer as the required rational resynchronizer. □

In conclusion, the results we have obtained for synthesis of rational resynchronizer are very similar to the known results for classical equivalence problem. While the problem is undecidable in the unrestricted case, for the functional and the finite-valued case, a rational resynchronizer always exists and can be synthesized.

4.2.2 Regular Resynchronizers

We now consider the synthesis problem for bounded, regular resynchronizers. Note that in this case, we can have different variants of the problem,

by considering the given transducers coming from different classes (NFTs, 2NFTs, NSSTs). We look at the case when the given transducers are unambiguous 2NFTs.

Case of Unambiguous 2NFTs. The synthesis problem for rational resynchronizer, for two given functional (or even finite-valued) NFTs, is decidable. In fact, a rational resynchronizer always exists and is also bounded-delay resynchronizer. Therefore, an interesting question to ask is the bounded, regular resynchronizer synthesis problem for two given functional 2NFTs. We now present a positive result in the simpler case of unambiguous 2NFTs. Note that every functional 2NFT can be converted to a classically equivalent unambiguous 2NFT. However, such transformations do not necessarily preserve the origins. This is because a functional 2NFT, on a fixed input, can produce the same output in the classical semantics, but different outputs in the origin semantics. An unambiguous 2NFT however has a unique successful run and therefore a output, even in the origin semantics.

Our results about the regular resynchronizer synthesis problem for unambiguous 2NFTs are in contrast with result about rational resynchronizers, in the sense that a resynchronizer need not always exists. As an example, consider the 2NFT T_1 that consumes an input of the form a^* from left to right, while copying the letters to the output, and a 2NFT T_2 that realizes the same function but while consuming the input in reverse. We have seen in Example 3.3.11 that $T_1 \subseteq T_2$, but $T_1 \not\preceq_{\text{reg}} T_2$. On the other hand, for functional NFTs, we are always able to synthesize a bounded-delay resynchronizer.

As we will see, it is possible to associate a resynchronizer for pairs of classically equivalent unambiguous 2NFTs if we move beyond the class of bounded, regular resynchronizers and consider resynchronizers defined by Parikh automata. The existence of bounded, regular resynchronizers between two given unambiguous 2NFTs can thus be seen as a strengthening of the classical containment relation.

Parikh Resynchronizers. First we introduce resynchronizers definable by Parikh automata. Formally, a *Parikh automaton* is a finite automaton $A = (Q, \Sigma, I, \Delta, F, Z, S)$ equipped with a function $Z : \Delta \to \mathbb{Z}^k$ that associates vectors of integers to every transition of the automaton and a semi-linear set $S \subseteq \mathbb{Z}^k$ ($\mathbb{Z}^k$ denotes the set of $\mathbb{Z}$-vectors of dimension k). A successful run of A is a run starting in a state in I, ending in a state in F, such that the vector obtained by summing together the vectors associated with the transitions of the run belongs to the set S. The semi-linear set S can therefore be seen as an acceptance condition. A Parikh automaton can be seen as a recognizer of languages. We define the language recognized

by A as the set of all words that admit a run ending in final states and computing a vector in S.

We say that A is *unambiguous* if the underlying finite automaton is unambiguous. In this case, the runs need not end with a valuation in S, but end in a final state from F. In this case, we can associate with each input u, the vector $A(u) \in \mathbb{Z}^k$ associated with the unique run of the underlying automaton of A on u, if such a run exists; otherwise $A(u)$ is undefined.

An important property of unambiguous Parikh automata is that they are closed under the pointwise sum and difference operations. More precisely, given unambiguous Parikh automata A_1 and A_2, we can construct A_+ and A_- such that $A_+(u) = A_1(u) + A_2(u)$ and $A_-(u) = A_1(u) - A_2(u)$, for all possible inputs u.

In the rest of the thesis, we assume the set S is $\{0^k\}$. This is without loss of generality, since, for any given acceptance condition as a semi-linear set, we can always add some transitions to subtract suitable vectors from S at the end of the run.

By a slight abuse of terminology, we call *Parikh resynchronizer* any resynchronizer with parameters whose relations move_τ and $\mathsf{next}_{\tau,\tau'}$, seen as languages of annotated words with two marked positions, are recognizable by unambiguous Parikh automata, and ipar and opar are regular languages. We naturally inherit from regular resynchronizers the notion of boundedness concerning the relation move.

We introduce another technical notion, that will be helpful later. Given a resynchronizer R, we define its *target set* as the set of all pairs (u, j) where u is an input, $j \in \mathrm{dom}(u)$, and $(w, i, j) \in \mathsf{move}_\tau$ for some annotation w of u with input parameters, some input position i, and some output type τ. Essentially, this is the set of pairs (u, i) where u is an input and i is a potential origin in a target synchronized word obtained by applying the resynchronizer R. Similarly, we define the *origin set* of a 2NFT T as the set of all pairs (u, i) such that i is the origin of some position output by T on u. Formally, u is an input of T and there exists a output position $x \in \mathrm{dom}(v)$, such that $(u, v) \in [\![T]\!]_o$ and x is labeled by (a, i) for $a \in \Gamma$.

Theorem 4.2.4 (Theorem 12 in [BKM+19]). *Let T_1, T_2 be two unambiguous 2NFTs. The following conditions are equivalent:*

1. *$T_1 \subseteq T_2$,*
2. *$T_1 \subseteq_o R(T_2)$ for some resynchronization R,*
3. *$T_1 =_o R(T_2)$ for some 1-bounded, Parikh resynchronizer R whose target set coincides with the origin set of T_1.*

The third condition requires that on input u, R can redirect the origin of any output position x to some j if and only if j is the origin of some position in the output produced by T_1 on u. Note that since T_1 and T_2 are

unambiguous, the outputs are determined by the input u. We now present the proof of the theorem.

Proof. The implications from 2. to 1. and from 3. to 2. follow from the definitions. The only interesting implication is from 1. to 3, where we suppose that $T_1 \subseteq T_2$ and we aim at constructing a 1-bounded Parikh resynchronizer R such that $T_1 =_o R(T_2)$, and whose target set is the same as the origin set of T_1. The proof exploits variants of the classical constructions based on crossing sequences [She59], as well as the reduction of containment of functional 2NFT to emptiness of languages recognized by Parikh automata [MP19b]. We adapt the definition of crossing sequences as a sequence of transitions taken in a position, instead of the more classical definition as a sequence of states visited in a position. A *crossing sequence* of a unambiguous 2NFT is thus a tuple $\bar{t} = (t_1, \ldots, t_n)$ of transitions such that the source states of $t_1, t_3, \ldots$ are right-reading and the source states of $t_2, t_4, \ldots$ are left-reading. The tuple is meant to describe the transitions along a successful run that depart from configurations at a certain position i. Formally, given a run ρ, the *crossing sequence of ρ at input position i*, denoted $\rho[i]$, consists of the quadruples (q, a, v, q') such that $(q, i) \xrightarrow{a \mid v} (q', i')$ is a transition of ρ, where the occurrence order of the said transitions coincides with the order on the quadruples of the crossing sequence. Note that the crossing sequences for an unambiguous 2NFT is bounded since the same position cannot be visited with the same state more than once (otherwise there would exists multiple run on the same input). In particular, we can assume that the length of a crossing sequence is bounded by the total number of states of the 2NFT. Moreover, for unambiguous 2NFTs, the input along with a position, determines the crossing sequences of the successful run at the position. More precisely, there are regular languages $L_{\bar{t}}$, one for each possible crossing sequence $\bar{t}$, that contains precisely those inputs u with a specific position i marked on it (for short, we denote such words by $\langle u, i \rangle$), such that the crossing sequence at i of the unique successful run on u is exactly $\bar{t}$.

We can generalize the notation for words with marked positions to write $\langle u, i_1, i_2, \ldots, i_k \rangle$ to denote the word u with marked positions $i_1, \ldots, i_k$.

We now turn to the main proof, which is divided into several steps.

Encoding output positions. We begin by describing an annotation over the input that encodes an arbitrary output positions. We consider an unambiguous 2NFT T (which can be either T_1 or T_2). We also consider an arbitrary input for T, denoted by u and the unique induced successful run, denoted by ρ. To simplify the notations, we assume that T produces at most one letter at each transition.

Let n be the number of states of T. Since T is unambiguous, v contains at most n output positions with the same origin (otherwise, the same configuration would be visited at least twice along the successful run ρ, which

could then be used to contradict the assumption of unambiguity). This means that every output position x in $\mathrm{dom}(v)$ can be encoded by its origin $i_x = \mathit{orig}(v(x))$ together with a suitable index $\ell_x \in \{1, \ldots, n\}$, describing the number of output positions $x' \leq x$ with the same origin i_x as x. We recall that the input u together with the position i can be represented as an annotated input of the form $\langle u, i \rangle$.

Decoding by Parikh automata. We now show that there are Parikh automata that on input u compute the inverse of the encoding $x \mapsto (i_x, \ell_x)$ described above. More precisely, there are unambiguous Parikh automata $A_1, \ldots, A_n$ such that each A_ℓ receives as input a word $\langle u, i \rangle$ having a special position marked on it, and outputs the unique output *position* x such that $(i, \ell) = (i_x, \ell_x)$, if this exists, otherwise the output is undefined. By output x of a Parikh automaton, we mean the vector computed by A_ℓ on the input $\langle u, i \rangle$, which in our case is the number (one-dimensional vector) x.

Each automaton A_ℓ can be constructed from T and ℓ by unambiguously guessing the crossing sequences of the unique run of T on u, and by counting the number of output symbols produced in the run until a *productive* transition at the marked position i is executed for the ℓ-th time (a productive transition is a transition that produces non-empty output). This requires A_ℓ to guess which productive transitions in the run occur before the ℓ-th productive transitions at i, and this can be easily be done by such an automaton. The value x is computed by counting the number of output symbols in the transition before the (i_x, ℓ_x)-th transition.

Redirecting origins. We now apply the constructions outlined above in order to obtain the desired Parikh resynchronizer R from T_1 and T_2. Let u be some input and $v_1, v_2 \in (\Gamma \times \mathbb{N})^*$ be the output in origin semantics produced by the unique successful runs of T_1, T_2 on u. Since $T_1 \subseteq T_2$, we can further assume $v = (v_1)_\Gamma = (v_2)_\Gamma$. Consider any output position $x \in \mathrm{dom}(v)$. According to v_2, x is encoded by an input position i_x and an index $\ell_x \in \{1, \ldots, n_2\}$, where n_2 is the number of states of T_2. In a similar way, according to v_1, the same position x is encoded by some input position j_x and an index $k_x \in \{1, \ldots, n_1\}$, where n_1 is the number of states of T_1.

Furthermore, based on the previous constructions, there are unambiguous Parikh automata $A_{2,\ell}$ and $A_{1,k}$ such that

— $A_{2,\ell}(\langle u, i \rangle) = x$ if and only $(i, \ell) = (i_x, \ell_x)$,

— $A_{1,k}(\langle u, j \rangle) = x$ if and only $(j, k) = (j_x, k_x)$.

Intuitively, the position i will be the source origin (origin in T_2) and the position j will be the target origin (origin in T_1), of the output position x.

Since unambiguous Parikh automata are closed under pointwise difference, there is a unambiguous Parikh automaton $A_{\ell,k}$ that recognizes pre-

cisely the language of annotated words $\langle u, i, j\rangle$ such that

$$A_{2,\ell}(\langle u, i\rangle) - A_{1,k}(\langle u, j\rangle) = 0 \qquad (\star)$$

Note that the above language defines a partial bijection between pairs of positions i, j in the input u in such a way that i and j are the origins of the same output position x according to the unique outputs v_2, v_1 of T_2, T_1 on input u. This property can be used to define the component move_τ of the desired resynchronizer R, by simply letting

$$\mathsf{move}_\tau = \{(u, i, j) \mid A_{\ell,k}(\langle u, i, j\rangle) = 0\}$$

where $\tau = (a, \ell, k) \in \Gamma \times \{1, \ldots, n_2\} \times \{1, \ldots, n_1\}$.

To check correctness of the above definition, we need to guess the pairs of indices (ℓ, k) correctly as annotations of output positions. More precisely, we have that:

— for every output position x with origin i in v_2, and with output type $\tau = (a, \ell_x, k)$, there is at most one input position j such that $(u, i, j) \in \mathsf{move}_\tau$; in addition, if we also have $k = k_x$, then j is the origin of the position x in v_1; symmetrically,

— for every output position x with origin j in v_1, and with output type $\tau = (a, \ell, k_x)$, there is at most one input position i such that $(u, i, j) \in \mathsf{move}_\tau$; in addition, if we also have $\ell = \ell_x$, then i is the origin of x in v_2.

Based on the above properties, we need to guess suitable output parameters that associate with each position x, a correct pair (ℓ_x, k_x). We explain below how this is done using the components opar and $\mathsf{next}_{\tau,\tau'}$ of the resynchronizer.

Constraining output parameters. We first focus on the indices k_x related to T_1; we will later explain how to adapt the constructions to check the indices ℓ_x related to T_2. Recall that we want R such that $T_1 =_o R(T_2)$. Therefore, the origins in v_1 are the target origins and origins in v_2 are the source origins. As usual, we fix an input u and the unique successful run ρ_1 of T_1 on u.

The idea is that each index k_x corresponds to a certain element of the crossing sequence of ρ_1 at the target origin j_x, and knowing the correct index for x determines the correct index for the next output position $x+1$. Based on this, correctness can be verified inductively using the guessed crossing sequences and the relation $\mathsf{next}_{\tau,\tau'}$ of the resynchronizer, as explained in details below.

For the base case, we check that the first output position is correctly annotated. The encoding of the first position will be $(j, 1)$ for some input position j. Note that the index will always be 1 since this is the first output position. This can easily be checked by opar. Additionally, we also need to

check that there is no transition before the $(j,1)$-th transition in the run. For this latter constraint, we need access to the full run, which unfortunately is not available in the output annotations. We can however use move_τ to check this. Since it is interpreted over the input, which is annotated with the crossing sequences, it is easy to check that all the transitions before the $(j,1)$-th transition produce non-empty outputs.

For the inductive step, we consider an output position x and assume that it is correctly annotated with $k = k_x$. Let k' be the annotation of the next position $x+1$. To check that k' is also correct, we consider the k-th productive transition at the target origin of x and the k'-th productive transition at the target origins of $x+1$, and verify that they are connected by a non-productive run. More precisely, let z and z' be the target origins of x and $x+1$, respectively, and let $\bar{t}_z$ and $\bar{t}_{z'}$ be the crossing sequences of ρ_1 at those positions. We have that $k' = k'_{x+1}$ if and only if the k-th productive transition of $\bar{t}_z$ and the k'-th productive transition of $\bar{t}_{z'}$ are connected by a factor of the run that consists only of non-productive transitions. The latter property can be translated to a regular property $\mathsf{next}_{\tau,\tau'}$ concerning the input annotated with two specific positions, z and z', assuming that $\tau = (a,\ell,k)$ and $\tau = (a',\ell',k')$ are the letters of the output positions x and $x+1$.

It now remains to check the correctness of the output annotations w.r.t. the indices ℓ for the second 2NFT T_2. We follow a principle similar to the one described above for T_1. The base case, checking the correctness of the annotation of the first position can still be done using move_τ. The only difference is that now, in the inductive step, we have to work with the source origins y and y' of consecutive output positions x and $x+1$. The additional difficulty is that, by definition, the relation $\mathsf{next}_{\tau,\tau'}$ can only refer to target origins. We overcome this problem by exploiting the fact that move_τ defines a partial bijection between target and source origins.

Formally, we first define a relation $\mathsf{next}^{\mathrm{source}}_{\tau,\tau'}$ as before, that constrain the indices ℓ and ℓ' associated with two consecutive output positions x and $x+1$ labeled by τ and τ', respectively. We do this as if $\mathsf{next}^{\mathrm{source}}_{\tau,\tau'}$ were able to speak about source origins. We combine the relations move_τ, $\mathsf{move}_{\tau'}$ and $\mathsf{next}^{\mathrm{source}}_{\tau,\tau'}$ to obtain the relation $\mathsf{next}_{\tau,\tau'}$:

$$\big\{(u,z,z') \mid \exists y,y'\ (u,y,y') \in \mathsf{next}^{\mathrm{source}}_{\tau,\tau'},\ (u,y,z) \in \mathsf{move}_\tau,\ (u,y',z') \in \mathsf{move}_{\tau'}\big\}.$$

Since in the inductive step we assume that x is correctly annotated with the pair (ℓ,k) and $x+1$ is annotated with (ℓ',k'), where $k' = k_x$ is correct by the previous arguments, there are unique y,y' that satisfy $(u,y,z) \in \mathsf{move}_\tau$ and $(u,y',z') \in \mathsf{move}_\tau$ in the above definition, and these must be the source origins of x and $x+1$. This means that the above relation, which is definable by a unambiguous Parikh automaton, correctly verifies the correctness of the index ℓ' associated with $x+1$.

We conclude by observing a few properties of the defined Parikh resynchronizer R. As already explained, the relation move_τ defines a bijection between pairs of input positions, so R is a bijective Parikh resynchronizer. In particular, this is a functional, 1-bounded resynchronizer. As concerns its target set, that is the set of pairs (u, z) such that $(u, y, z) \in \mathsf{move}_\tau$ for some $z \in \mathrm{dom}(u)$ and some $\tau \in \Gamma \times \{1, \ldots, n_2\} \times \{1, \ldots, n_1\}$, it coincides by construction with the target set of T_1. □

Note that in the above proof, the relation $\mathsf{next}_{\tau,\tau'}$ is defined by conjoining a regular property with the properties defined by the relations move_τ and $\mathsf{move}_{\tau'}$. Therefore, we have the following property, which will be used later to prove completeness for the synthesis problem.

Lemma 4.2.5. *Let R be the Parikh Resynchronizer obtained from Theorem 4.2.4. The relation $\mathsf{next}_{\tau,\tau'}$ is regular if move_τ and $\mathsf{move}_{\tau'}$ are regular.*

From Theorem 4.2.4, we obtain a Parikh resynchronizer R. To solve the regular resynchronizer synthesis problem for unambiguous 2NFTs, we can check if the move_τ, for every type τ, defined by the Parikh automata in R, defines a regular language. If this is indeed the case, then by Lemma 4.2.5, $\mathsf{next}_{\tau,\tau'}$ is also regular. Therefore, we are able to synthesize a regular resynchronizer $\hat{R}$. The following theorem shows that this check is sufficient, i.e, if there is a suitable regular resynchronizer, then $\hat{R}$ is regular.

Theorem 4.2.6 (Theorem 13 in [BKM$^+$19])**.** *Let T_1, T_2 be two 2NFTs such that $T_1 \subseteq T_2$, and let $\hat{R}$ be the bounded Parikh resynchronizer obtained from Theorem 4.2.4. The following conditions are equivalent:*

1. *$\hat{R}$ is a regular resynchronizer,*
2. *$T_1 \subseteq_o R(T_2)$ for some bounded regular resynchronizer R,*
3. *$T_1 =_o R(T_2)$ for some 1-bounded regular resynchronizer R with the same target set as T_1.*

Proof. We prove the following implications in the order: 1. $\to$ 2. $\to$ 3. $\to$ 1.

From 1. to 2. This is trivial since $\hat{R}$ is bounded, and satisfies $T_1 =_o \hat{R}(T_2)$, and hence $T_1 \subseteq_o \hat{R}(T_2)$.

From 2. to 3. Suppose that R is a bounded, regular resynchronizer with input alphabet Σ and output alphabet Γ, such that $T_1 \subseteq_o R(T_2)$. Recall that by Lemma 3.3.4, we can assume R is 1-bounded. The goal is to construct a 1-bounded, regular resynchronizer R', with the same target set as T_1 and such that $T_1 =_o R'(T_2)$. For the sake of simplicity, we assume that R has no input parameters, and similarly T_1 has no common guess (the more general cases can be dealt with by annotating the considered inputs with the possible parameters and the common guess).

The idea for defining the desired resynchronizer R' is as follows. We first restrict each formula move_τ so as to make it a partial bijection, that is, for every input u, and every source origin $y \in \mathrm{dom}(u)$, there is an annotation $\hat{u}$ of the input and at most one target origin z such that $(\hat{u}, y, z) \in \mathsf{move}_\tau$ (and conversely, since R is 1-bounded, for every target origin z, there is a unique source origin y). This step requires the use of appropriate input parameters that determine a unique target origin z from any given source origin y. We explain later how we achieve the goal of defining a move_τ that is a partial bijection.

Then, we restrict further the formula move_τ so that every target origin z is witnessed by T_1. Formally, we introduce input parameters ranging over $\mathbb{B}^\Gamma$ and work with annotated inputs of the form $u \otimes w$, with $u \in \Sigma^*$ and $w \in (\mathbb{B}^\Gamma)^*$. Recall that $\mathbb{B}^\Gamma$ are the output types. Given $u \in \Sigma^*$, we define O_u as the set of all positions $z \in \mathrm{dom}(u)$ that are the origin of some position $x \in \mathrm{dom}(v)$.

The new formula move'_τ that redirects a source origins to target origins is defined as the following restriction of move_τ:

$$\mathsf{move}'_\tau = \big\{(u \otimes w, y, z) \;\mid\; (u, y, z) \in \mathsf{move}_\tau,\ w(z)(\tau) = 1,\ z \in O_u \big\}.$$

Intuitively, this means that there is a output position x with output-type τ (hence, $w(z)(\tau) = 1$), and target origin z (hence $z \in O_u$). Clearly, the above relation is MSO-definable and contained in move_τ. However, it is still possible that move'_τ associates multiple target origins with the same source origin.

To get a partial bijection from move'_τ we need to constrain the possible annotated input $u \otimes w$. We do so by requiring that, for every output letter $\tau \in \Gamma$ and every position y in $u \otimes w$, if there is z with $(u, y, z) \vDash \mathsf{move}_\tau$, then there is *exactly one* z' such that $(u, y, z') \vDash \mathsf{move}_\tau$ and $w(\tau)(z) = 1$. This again gives a new relation move''_τ that is contained in the previous relation move_τ. Note that the latter property is again regular, and thus could be conjoined with the original relation ipar to form the new relation ipar'. Accordingly, the formula $\mathsf{next}'_{\tau,\tau'}$ of the desired resynchronizer R' checks the same properties as $\mathsf{next}_{\tau,\tau'}$, but expanded with arbitrary input annotations over $\mathbb{B}^\Gamma$.

It is now easy to see that the the resulting resynchronizer R' is 1-bounded, and in fact, on each input, defines a partial bijection between source and target origins in such a way that the target set coincides with that of T_1. By pairing this with the containments $R'(T_2) \subseteq_o R(T_2)$ and $T_1 \subseteq_o R(T_2)$, we obtain $T_1 =_o R'(T_2)$.

From 3. to 1. Knowing that $\hat{R}(T_2) =_o T_1 =_o R(T_2)$ for some 1-bounded resynchronizers R and $\hat{R}$ obtained from Theorem 4.2.2, with the same target sets as T_1 implies that the formulas move^R_τ and $\mathsf{move}^{\hat{R}}_\tau$, from R and $\hat{R}$

respectively, coincide. Moreover, since the relation move_τ^R of R is regular, this means that $\mathsf{move}_\tau^{\hat{R}}$ is regular too. Finally, we recall from Lemma 4.2.5 that $\mathsf{next}_{\tau,\tau'}^{\hat{R}}$ is regular whenever $\mathsf{move}_\tau^{\hat{R}}$ and $\mathsf{move}_{\tau'}^{\hat{R}}$ are. We can then conclude that the relations $\mathsf{next}_{\tau,\tau'}^{\hat{R}}$ are also regular, and hence $\hat{R}$ is a bounded, regular resynchronizer. □

Finally, in order to conclude that the synthesis problem for bounded, regular resynchronizers is decidable, we use the result on decidability of regularity of languages recognized by unambiguous Parikh automata [CFM13].

Lemma 4.2.7 ([CFM13])**.** *Given an unambiguous Parikh automaton A, it is decidable to check whether $[\![A]\!]$ is a regular language.*

Proof sketch. This result requires unambiguity and uses Presburger arithmetics to determine, for each (simple) loop a threshold such that iterating the loop more than the threshold always satisfies the Parikh constraint. A run using such a simple loop adds keeps repeating the same computation. If the computation is repeated several times above the threshold, always satisfies the constraint, i.e, generates a vector that satisfies the acceptance condition. The language of the Parikh automaton is regular if and only if every (simple) loop has such a threshold. □

We thus conclude:

Corollary 4.2.7.1. *Given two unambiguous 2NFTs T_1, T_2, one can decide whether there is a regular resynchronizer R such that $T_1 \subseteq_o R(T_2)$.*

Since the above result requires unambiguity, we cannot directly extend it to functional 2NFTs. Another interesting problem would be to extend the above result to k-ambiguous 2NFTs, which have at most k different runs on an input. Recall that for NFTs, we are able to decompose k-ambiguous ones into k unambiguous NFTs. For 2NFTs, such a decomposition result remains open. Obtaining such a decomposition theorem could ease a solution to the synthesis problem as well, as we could then apply the algorithm for synthesizing a regular resynchronizer for k pairs of unambiguous 2NFTs, given by the decomposition.

4.3 Conclusions

We saw that the synthesis problem for both rational and bounded, regular resynchronizers are undecidable (Theorem 4.1.1), even for NFTs. However, the NFT T_2 used in the proof is not real-time. The synthesis problem remains thus open for real-time NFTs. Note that by Theorem 3.4.4, rational resynchronizers and bounded, regular resynchronizers are equivalent

for real-time NFTs. Therefore, the synthesis problem for bounded, regular resynchronizers given two real-time NFTs also remains open.

For finite-valued NFTs, we saw that it is always possible to synthesize a bounded-delay resynchronizer (Theorem 4.2.1). We were able to strengthen the result for functional NFTs (Theorem 4.2.3), by constructing a resynchronizer, that is of size polynomial in T_1 and T_2. In the k-valued case, the synthesized resynchronizer is a k-delay resynchronizer, which can have size exponential in k (see Proposition 3.2.4).

For 2NFTs and bounded, regular resynchronizers, we solved the synthesis problem under the assumption that the 2NFTs are unambiguous (Theorem 4.2.4). The synthesis problem for bounded, regular resynchronizer thus remains open for functional, finitely ambiguous and finite-valued 2NFTs.

Chapter 5

One-way Resynchronizability

We have seen various classes of transducers, such as NFTs, 2NFTs, NSSTs, etc, with different expressive power. The class of relations defined by NFTs is contained in the class of relations defined by 2NFTs. Therefore, a natural question to ask is the class membership problem, i.e, given an 2NFT T, is it equivalent to some NFT? This is called the *one-way definability* problem and has been well studied in the classical semantics. This problem is motivated by the fact that a NFT can produce the output in a *streaming manner*, by processing one letter at a time, whereas a 2NFT needs to store the entire input to produce the output.

In the origin semantics, the one-way definability problem becomes trivial due to a machine independent characterization of relations defined by NFTs as order-preserving transductions. In this chapter we study a variant of this problem, which we call the *one-way resynchronizability* problem. A 2NFT T is one-way resynchronizable if there exists a resynchronizer R such that $R(T)$ is origin-equivalent to some NFT. Intuitively, this means that T uses the two-way movement in a limited manner and the resynchronizer R can "undo" the effects of the two-way movement. We begin by recalling the known results about one-way definability in both classical and origin semantics and then present our results on one-way resynchronizability.

5.1 One-way Definability and Resynchronizability

Classical semantics. The one-way definability problem in the classical setting is the following.

Problem 5.1.1 (One-way Definability)**.** *Given a 2NFT T, is it classically equivalent to some NFT T'?*

If such a T' exists, T is called one-way definable. This one-way definability problem is undecidable in the unrestricted (relational) case, see [BGMP18]. However, restricted to functional transducers, the problem becomes decidable [FGRS13] in 2-ExpSpace [BGMP18].

Example 5.1.2. *Let T_{shift} be a 2DFT that shifts the last letter of the input to the beginning of the word. Therefore, on an input $u = wa$, T_{shift} produces aw by making a first pass to the end of the word, outputting the last letter, and then moving back to the beginning and copying the remaining w in a second pass. This transformation can also be implemented by an NFT which guesses and outputs the last letter, copies the suffix w, and finally checks the guess, all of this in one pass. Therefore, T_{shift} is one-way definable.*

Example 5.1.3. *Let T_{rev} be a 2NFT with $\Sigma = \Gamma = \{a\}$ doing reverse transformation. In other words, on an input $u = a^n$, T_{rev} moves to the end of the word, copies the word from the end to the beginning and moves to the end to finish the computation. In the classical semantics, it is equivalent to the identity transformation, which is definable by an NFT. Therefore, T_{rev} is one-way definable. However, if we take $\Sigma = \Gamma = \{a, b\}$, then T_{rev} is no longer one-way definable.*

Origin semantics. In the origin semantics, the one-way definability problem becomes easier. The class of one-way definable functions was characterized in [Boj14] as *order-preserving transductions*, namely, as those transductions whose origins respect the input order. Formally, a transduction T is order-preserving if for every $(u, v) \in T$, and output positions $x < x' \in \mathrm{dom}(v)$, $orig(v(x)) \leq orig(v(x'))$.

Although order-preserving transductions were originally defined for functions [Boj14], the definition can be extended to relational transductions without any modifications. Note that none of the examples presented above are one-way definable in the origin semantics. In fact, one-way definability in the origin semantics is trivial as it boils down to checking one-wayness after removing non-productive U-turns.

Proposition 5.1.4. *A 2NFT T is one-way definable in the origin semantics if and only if $[\![T]\!]_o$ is order-preserving. Moreover, the latter property is* PSPACE*-complete.*

Proof. The left-to-right direction is trivial as T is one-way definable, and therefore produces the output in such a way that the origins preserve the input order. The right-to-left direction follows from the construction of $\mathsf{Shortcut}(T)$ (see page 39), which removes all non-productive U-turns. Recall that the 2NFT $\mathsf{Shortcut}(T)$ visits input position based on the order of origins (see page 39). Therefore, $\mathsf{Shortcut}(T)$ is an NFT that is origin-equivalent to T. Thus, T is one-way definable.

For the last claim note that we can check that $[\![T]\!]_o$ is not order-preserving by guessing on-the-fly an input $uavbw$ and a run producing some output on the letter b, then on the letter a, with empty output in-between. We need to store information about non-productive U-turns to check this,

hence the PSPACE upper bound. The lower bound follows from the language emptiness of 2DFA. Given a 2DFA accepting a language L, one can construct a 2DFT that produces reverse of w, for every $w \in L$. This 2DFT is one-way definable in origin semantics if and only if L is empty.

For sake of completeness, recall that the emptiness problem for 2DFA is PSPACE-hard. Given a PSPACE Turing machine M with input w using space n in the tape that is polynomial in $|w|$, we can construct a 2DFA that reads sequences of configurations of the M and checks whether it encodes a successful run of M on w or not. We can assume that all configurations have size n, and the 2DFA can check that consecutive configurations are connected by a transition by moving back and forth using the two-way movement of the reading head. The language of the 2DFA is non-empty if and only there is an accepting run of M on w that uses space polynomial in $|w|$. □

One-way resynchronizability. Since the one-way definability problem in the origin semantics is too simple, and rather restrictive, we consider a variant of the problem, that asks for one-way definability up to *some resynchronizer*, i.e, whether there exists a regular resynchronizer R such that $R(T)$ is one-way definable in the origin-semantics. For example, T_{shift} in Example 5.1.2 can be made one-way by applying a resynchronizer that moves the origin of the last output position from the end to the beginning of the input. So there exists a regular resynchronizer R such that $R(T_{\text{shift}})$ is order-preserving.

There is however a catch in the previous attempted definition of *one-way resynchronizability*. The empty resynchronization relation applied to any transduction gives the empty transduction, which is one-way definable. This would make every 2NFT trivially resynchronizable to an NFT. To avoid this, we also require R to be *T-preserving*, i.e, $[\![T]\!]_o$ is contained in the domain of R. This ensures that every synchronized pair generated by T is mapped to some synchronized pair by R and therefore, $R(T)$ is classically equivalent to T.

We thus define the *one-way resynchronizability problem* as follows.

Problem 5.1.5 (One-way Resynchronizability)**.** *Given a 2NFT T, does there exist a T-preserving, bounded, regular resynchronizer R such that $R(T)$ is order-preserving?*

Our main result is to show that the one-way resynchronizability problem is decidable.

Theorem 5.1.6. *It is decidable in* EXPSPACE *whether a given 2NFT T is one-way resynchronizable.*

We break the proof into two parts. First, we consider the case of *bounded-visit 2NFTs*. A 2NFT T is called k-visit if every synchronized pair $(u, v) \in T$

is accepted by a run ρ which has at most k occurrences of a configuration from $Q \times \{i\}$, for every input position i. In other words, the number of visits to an input position i in the run ρ is at most k. A 2NFT T is called bounded-visit if there exists a k such that T is k-visit. This restriction allows for runs of T to be summarized by flows (a modification of the classical notion of crossing sequence of [She59]) of bounded size.

We characterize one-way resynchronizability of a bounded-visit 2NFT T by the absence of *inversions* (defined in Section 5.2), which can be detected by inspecting the flows of T. We also provide another equivalent characterization, called *bounded cross-width*, based on the set of synchronized pairs defined by the 2NFT T (also defined in Section 5.2).

For the general case, the main difficulty is that the flows are no longer bounded. Therefore, the approach of checking the flows for absence of inversions no longer works. We will show that if a 2NFT T has a *vertical loop* (sub-run starting and ending in the same configuration) which produces outputs with unbounded number of distinct origins, then T cannot be one-way resynchronizable. In absence of such loops, we will show how to build a bounded-visit 2NFT $\mathit{low}(T)$ classically equivalent to T, and reduce the problem to the bounded-visit case.

5.2 Technical ingredients

In this section, we introduce the technical definitions used to characterize one-way resynchronizability for bounded-visit 2NFTs.

Cross-width. Let $\sigma = (u, v)$ be a synchronized pair and let $X_1, X_2 \subseteq \mathrm{dom}(v)$ be sets of output positions such that, for all $x_1 \in X_1$ and $x_2 \in X_2$, $x_1 < x_2$ and $\mathit{orig}(v(x_1)) > \mathit{orig}(v(x_2))$. We call such a pair (X_1, X_2) a *cross* and define its *width* as $\min(|\mathit{orig}(X_1)|, |\mathit{orig}(X_2)|)$, where $\mathit{orig}(X) = \{\mathit{orig}(v(x)) \mid x \in X\}$ is the set of origins corresponding to a set X of output positions. Intuitively, to make the pair (u, v) order-preserving, a resynchronizer needs to either redirect the origins of X_1 or the origins of X_2.

The *cross-width* of a synchronized pair (u, v) is the maximal width of the crosses in (u, v). We say a 2NFT T has *bounded cross-width* if there exists a bound k, such that for all synchronized pairs $(u, v) \in T$, the cross-width of (u, v) is at most k.

Examples of cross-width. The synchronized pairs produced by T_{shift} from Example 5.1.2 have cross-width 1. For example, the synchronized pair $(aba, (b, 2)(a, 3)(a, 1))$, has crosses (X_1, X_2), where $X_1 = \{1\}$ and X_2 is a non-empty subset of $\{2, 3\}$. In fact, for any synchronized pair generated

by T_{shift}, a cross will always have $X_1 = \{1\}$. Therefore, the cross-width is bounded by 1.

For the 2NFT T_{rev} in Example 5.1.3, for every integer k, there is a synchronized pair with a cross of width k. For instance, the synchronized pair $(a^{2k}, (a, 2k) \dots (a, 1))$ has a cross (X_1, X_2), where $X_1 = \{1, \dots, k\}$ and $X_2 = \{k+1, \dots, 2k\}$, which has width k. Since, k can be chosen arbitrarily large, the cross-width is unbounded.

It is easy to see that order-preserving synchronized pairs have cross-width 0, since they have no crosses. One of the characterizations in Theorem 5.3.1 is that a 2NFT T is one-way resynchronizable if and only if T has bounded cross-width.

We now define the other key notion of *inversion* which will be used in Theorem 5.3.1 To define inversions, we first need to define flows (a notion inspired from crossing sequences [She59, BGMP18]).

Intervals. An *interval* of a word is a set of consecutive positions in it. Recall that an interval $[i, j]$ denotes the set $\{i, i+1, \dots, j\}$, where $i \leq j$ (see page 2). We write $I = [i, i')$ to denote the interval $[i, i' + 1]$. Given two intervals $I = [i, i']$ and $J = [j, j']$, we write $I < J$ if $i' < j$, and we say that I, J are adjacent if $i' = j - 1$. The union of two adjacent intervals $I = [i, i']$, $J = [j, j']$, denoted $I \cup J$, is the interval $[i, j']$ (if I, J are not adjacent, then $I \cup J$ is undefined).

Sub-runs. Given a run ρ of a 2NFT, a *sub-run* is a factor of ρ. Note that a sub-run of a 2NFT may visit a position of the input several times. For an input interval $I = [i, j]$ and a run ρ, we say that a sub-run ρ' of ρ *spans over* I if i (resp. j) is the smallest (resp. greatest) input position visited in the sub-run ρ'.

A subrun spanning over an interval I is said to be *maximal* if it cannot be extended within the interval. In other words, the transitions before and after the sub-run are not on a position of I. Such maximal sub-runs can be left-to-right, left-to-left, right-to-left, or right-to-right depending on where the starting and ending positions are w.r.t. the endpoints of the interval. For example, in the run shown on the left-hand side of Figure 5.1, the sub-run $(q_0, 1) \to (q_1, 2) \to (q_2, 2) \to (q_3, 1)$ is a maximal left-to-left sub-run on the interval $[2, 3]$.

Flows. Flows are used to summarize maximal sub-runs of a 2NFT that span over a given interval. Given a 2NFT T, a *flow* of T is a graph with vertices divided into L-vertices and R-vertices, labeled by states of T. The directed edges of the flow are divided into two productive and non-productive edges. We call a directed edge (v, v') an LR edge if v is a L-vertex and v' is

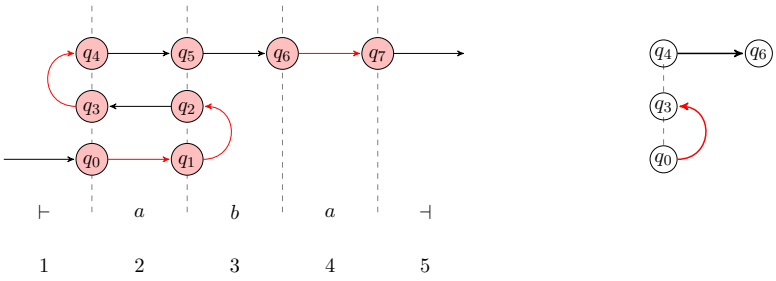

Figure 5.1 – A run ρ and the flow of interval $[2, 3]$

a R-vertex. LL, RL and RR edges are defined similarly. Moreover, an edge (v, v') must satisfy the following conditions:

- The edge source v is either an L-vertex labeled by a right-reading state, or an R-vertex labeled by a left-reading state.
- The edge destination v' is either an L-vertex labeled by a left-reading state, or an R-vertex labeled by a right-reading state.
- Each vertex is in exactly one edge, either as a source or destination.
- The set of L-vertices (R-vertices, resp.) is totally ordered, in such a way that for every LL (RR, resp.) edge (v, v'), we have $v < v'$.

Flows of a run. When considering flows of a K-visit 2NFT, we assume there are at most K L-vertices and K R-vertices. Given a run ρ of T and an interval $I = [i, i']$ on the input, *the flow of* ρ *on* I, denoted $\mathit{flow}_\rho(I)$, is obtained by identifying every configuration at cut $i - 1$ (resp. i') with a *left* (resp. *right*) vertex, labeled by the state of the configuration, and every maximal sub-run of ρ spanning over I with an edge connecting the appropriate vertices (this sub-run is called the *witnessing sub-run* of the edge of the flow). An edge is said to be *productive* if its witnessing sub-run produces non-empty output.

Figure 5.1 shows a run ρ of a 2NFT, where the red edges indicate productive transitions. The $\mathit{flow}_\rho(i)$ at any position i can be seen as the subgraph restricted to vertices at the cuts surrounding position i (shown by dotted lines). For example, the $\mathit{flow}_\rho(4)$ only has one edge $q_6 \to q_7$, which is productive. The right-hand side of the figure shows $\mathit{flow}_\rho([2, 3])$.

Juxtaposition of flows. We now define the juxtaposition of flows. Intuitively, the juxtaposition of flows is obtained by glueing together two flows F and G such that the set of R-vertices of F and L-vertices of G induce the same sequence of states. These vertices are merged to obtain a graph with three groups of vertices.

Formally, we define the *juxtaposition* FG as a graph with vertices divided into three groups, L-vertices of F, R-vertices of F and R-vertices of G. The edges consists the edges of F and G. Note that this is well-defined only

if the L-vertices of G coincides with the R-vertices of F. The result of a juxtaposition of two flows is not a flow, since it has three distinguished groups of vertices.

Hereafter, we talk about juxtaposition assuming it is well-defined. It is easy to extend juxtaposition to any number of flows, instead of just two flows.

A run of a 2NFT be seen as the juxtaposition of the flows at each input position. For example, in Figure 5.1, the flows at position i can always be juxtaposed with the flow at position $i + 1$.

Composition and flow monoid. The composition of two flows F and G is defined when they can be juxtaposed. Given such F and G, the composition results in the flow $F \cdot G$ obtained by keeping the L-vertices of F and the R-vertices of G. The edges of the concatenated flow $F \cdot G$ are obtained by shortcutting every path in the juxtaposition FG into an edge. An edge in $F \cdot G$ is *productive* if the corresponding path in FG had at least one productive edge (recall that a productive edge represents a sub-run that produces non-empty output). We call this operation of obtaining the concatenation from juxtaposition *flattening*. When the composition is undefined, we simply write $F \cdot G = \bot$. The above definitions naturally give rise to a *flow monoid* associated with the 2NFT T, where elements are the flows of T, extended with a dummy element $\bot$, and the product operation is given by the composition of flows, with the convention that $\bot$ is absorbing.

We can define the flow of a run on an interval I by flattening the juxtaposition of the flows at positions from that interval. For example, the $\mathit{flow}_\rho([2, 3])$ in Figure 5.1, is obtained by flattening the juxtaposition of flows at position 2 and 3. It is easy to verify that for any two adjacent intervals $I < J$ and a run ρ, $\mathit{flow}_\rho(I) \cdot \mathit{flow}_\rho(J) = \mathit{flow}_\rho(I \cup J)$.

Size of the flow monoid. We denote by M_T the *flow monoid* of a K-visit 2NFT T. We now give a bound on the size of M_T. If Q is the set of states of T, there are at most $|Q|^{2K}$ possible sequences of L and R-vertices; and the number of edges (marked as productive or not) is bounded by $\binom{2K}{K} \cdot (2K)^K \cdot 2^K \leq (2K + 1)^{2K}$. Including the dummy element $\bot$ in the flow monoid, we get $|M_T| \leq (|Q| \cdot (2K + 1))^{2K} + 1 =: \mathbf{M}$.

Accessibility order. Let F, G be two flows with edges f and g respectively. We say $f \preceq g$ in FG if there is a directed path from f to g in FG.

This order is well defined for flows corresponding to a run since they can always be juxtaposed. Given a run ρ on input u, a partition of the input into intervals $I_1 \dots I_k$, and an edge e_i of $\mathit{flow}_\rho(I_i)$ and e_j of $\mathit{flow}_\rho(I_j)$, we say $e_i \preceq e_j$ if there is a directed path from e_i to e_j in $\mathit{flow}_\rho(I_1) \dots \mathit{flow}_\rho(I_k)$.

Note that this depends on both the run ρ and the partition $I_1 \dots I_k$. In general, given two disjoint intervals $I = [i, i']$ and $J = [j, j']$, we can define the accessibility order between the edges of $\mathit{flow}_\rho(I)$ and $\mathit{flow}_\rho(J)$, by choosing the partition of the input into intervals defined by the positions i, i', j, j'.

An important remark is that the accessibility order defined by runs is a total order. Note that an edge in $\mathit{flow}_\rho(I)$ corresponds to a sub-run of ρ in the interval I. The accessibility relation with respect to a run is essentially the order of occurrence of these sub-runs.

Idempotent flows and loops. An *idempotent flow* is a flow F such that $F \cdot F = F$. This is an important notion that is used to identify intervals of the input on which the run can be pumped. For an idempotent flow, we denote by $F \dots F$ the n-fold juxtaposition and by $F \cdot \dots \cdot F$, the n-fold concatenation of the flow F with itself.

A loop of a run ρ over input w is an interval $I = [i, j]$ that induces an idempotent flow $F = \mathit{flow}(\rho, I)$, i.e, $F \cdot F = F$. In the presence of a loop $I = [i, j]$, the run ρ can be pumped. Given $n > 0$, we denote by $\mathsf{pump}_I^n(\rho)$ the run obtained from ρ by glueing together the sub-runs that span over the intervals $[1, i)$ and $(j, |w|]$ with n copies of the sub-runs spanning over I. The run $\mathsf{pump}_I^n(\rho)$ can be represented by $\mathit{flow}_\rho([1, i))F \dots F\mathit{flow}_\rho((j, |w|])$, where $F \dots F$ represents the n-fold juxtaposition of F with itself. Note that any edge f of the flow F occurs n times in the new run $\mathsf{pump}_I^n(\rho)$.

For an example, consider the run shown in Figure 5.2. The flow $F = \mathit{flow}_\rho(I)$, is an idempotent. The other edges of the runs are shown with dotted edges. On the right-hand side, we have the run $\mathsf{pump}_I^2(\rho)$, obtained by pumping F twice.

We call a LR or a RL edge of a flow a *straight edge*. A straight edge of a loop is a straight edge of the idempotent flow corresponding to the loop. For example, in Figure 5.2, the edges α_2, β and γ are straight edges. The Lemma below shows that such an edge acts as an invariant with respect to the accessibility order in the pumped run, i.e, an edge, that is not part of a loop, that occurs before (resp. after) a straight edge of a loop in the original run occurs before (resp. after) all copies of the straight edge in the pumped run as well. As an example, consider the edge ρ_3 in Figure 5.2. In the accessibility order, $\alpha \preceq \rho_3$. In the pumped run as well, both copies of the edge α precede the edge ρ_4.

Lemma 5.2.1. *Let ρ be a run of T on u, and $J < I < K$ be a partition of the domain of u into intervals, with I being a loop of ρ. Let $F = \mathit{flow}_\rho(J)$, $E = \mathit{flow}_\rho(I)$, and $G = \mathit{flow}_\rho(K)$ be the corresponding flows. Consider an arbitrary edge f of either F or G, and a straight edge e of the idempotent flow E such that $f \preceq e$ (resp. $e \preceq f$) in FEG. Let $e_1, e_2, \dots e_n$ be the n copies of the edge e in $\mathsf{pump}_I^n(\rho)$. Then, for all $i \in \{1, \dots, n\}$, $f \preceq e_i$*

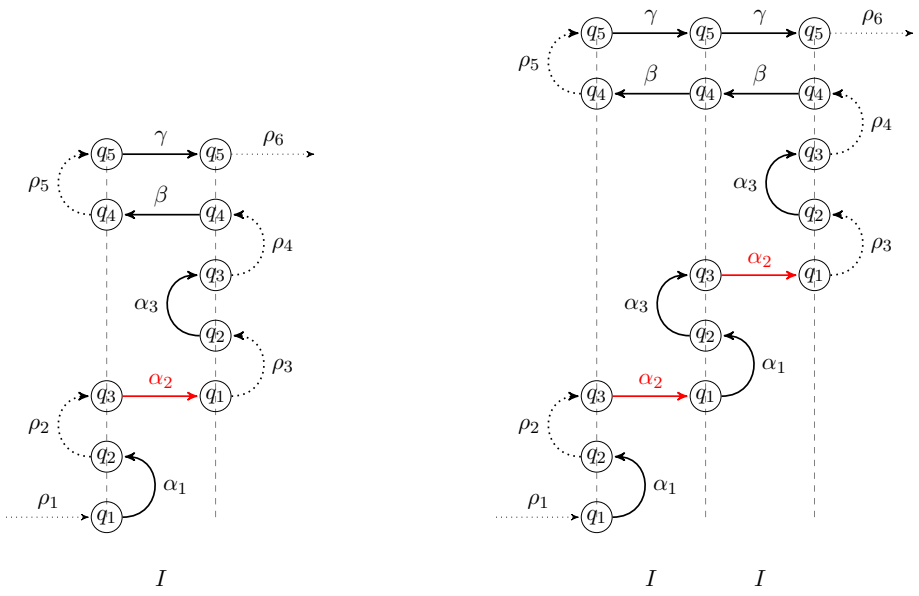

Figure 5.2 – A loop and a pumped run

(resp. $e_i \preceq f$*) in* $FE \dots EG$*.*

Proof. Consider the maximal path π inside $E \dots E$ that contains the edge e_i. Note that this path starts and ends at some extremal vertices of $E \dots E$ (otherwise the path could be extended while remaining inside $E \dots E$). Also recall that concatenation can be defined from juxtaposition by flattening. In particular, since E is idempotent, we have that $E = E \cdot \ldots \cdot E$ can be obtained from the flattening of $E \dots E$, and this operation transforms the path π into an edge e'. By construction, we have that $f \preceq e_i$ in $FE \dots EG$ if and only if $f \preceq e'$ in FEG. So it remains to prove that

$$f \preceq e \text{ in } FEG \qquad \text{iff} \qquad f \preceq e' \text{ in } FEG.$$

Clearly, this latter claim holds if the edges e and e' coincide. This is indeed the case when e is a straight edge and E is idempotent. The formal proof that this holds is rather tedious, but follows quite easily from a series of results already proven in [BGMP18]. Here, we only present a sketch with the key arguments used in the proof:

— the edges of an idempotent flow E can be grouped into *components* (cf. Definition 6.4 from [BGMP18]), so that each component contains exactly one straight edge,

— every path inside the juxtaposition EE, with E idempotent, consists of edges from the same component, say C; moreover, after the flattening from EE to E, this path becomes an edge of E that belongs again to the component C (cf. Claims 7.3 and 7.4 in the proof of Theorem 7.2 from [BGMP18]);

— every maximal path in $E \dots E$ that contains a straight edge starts and end at opposite sides of $E \dots E$ (simple observation based on the definition of concatenation and Lemma 6.6 from [BGMP18]).

In our example in Figure 5.2, the edges α_1, α_2, α_3 are in a single component, and β and γ are two components with a single edge.

To conclude, recall that π is a path inside $E \dots E$ that contains a copy e_i of the straight edge e, and that becomes the edge e' after the flattening into E. The previous properties immediately imply that $e' = e$. In our example, the path $\alpha_2\alpha_1\alpha_3\alpha_2$ is flattened to form the edge α_2. □

Inversions. An *inversion* of a run ρ is a tuple (I, e, I', e') such that

— I, I' are loops of ρ and $I < I'$,

— e, e' are productive straight edges in $\mathit{flow}(\rho, I)$ and $\mathit{flow}(\rho, I')$ respectively,

— $e' \preceq e$ in the accessibility relation defined by ρ and the intervals I and I'.

Intuitively, the run ρ on intervals I and I' are loops which have productive straight edges e and e' which are produced in the reverse order, i.e, $I < I'$ and $e' \preceq e$ in the run order. For example, in Figure 5.3, the intervals $I < I'$ are loops with productive straight edges $e' \preceq e$. Therefore, (I, e, I', e') is an inversion. A 2NFT T is said to have inversion if there is a successful run ρ of T has an inversion.

The run obtained by pumping these loops yield a run with multiple occurrences of the edges e and e', which, by Lemma 5.2.1, are such that all occurrences of e' precede all occurrences of e. The key argument is to show that these pumped runs create a cross of unbounded width.

Factorization Tree. Given a finite monoid and a sequence α of monoid elements, a *factorization tree* for α is an ordered, unranked tree where the nodes are labeled by monoid elements in such a way that

— the yield of the tree is the sequence α;

— all internal nodes have at least two children and are labeled by the product of the monoid elements at the child nodes;

— if a node p has more than two children, then all its children must have the same label as p, which must be an idempotent.

By Simon's factorization theorem [Sim90], every sequence of monoid elements has some factorization tree of height at most linear in the size of the monoid (more precisely, at most $3|M|$ for a monoid M, see e.g. [Col07]).

In our case, we will use the flow monoid M_T and look for some *factorization tree of a run* ρ of the 2NFT T. Specifically, we are interested in some factorization tree for the sequence of flows $\mathit{flow}_\rho(1), \dots \mathit{flow}_\rho(n)$ representing the run ρ. Therefore, the idempotent nodes of the factorization tree will correspond to loops of the run ρ.

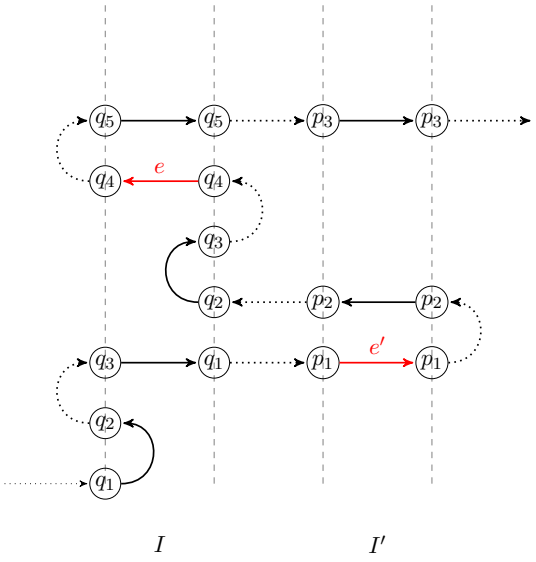

Figure 5.3 – An inversion

Consider the run to the left of Figure 5.2. Suppose that the input is of the form $\vdash a \dashv$, with $I = \{1\}$, and the sub-runs ρ_1, ρ_2, ρ_5 being transitions on $\vdash$ and ρ_2, ρ_3, ρ_6 being sub-runs on $\dashv$. Consider the pumped run $\rho' = \mathsf{pump}_I^3(\rho)$ on the input $\vdash aaa \dashv$. The factorization tree corresponding to ρ' is depicted in Figure 5.4, where F_1 is the flow on $\vdash$, F_2 is the flow on $\dashv$ and F is the idempotent flow on interval I. Note that the red colored node, which has 3 children, is labeled by an idempotent F.

The height of the factorization tree is 3. It is easy to see that on any word $\vdash a^n \dashv$, we have a factorization tree of height 3 simply by having the idempotent node labeled by F have n children labeled by F.

Given a node p of the factorization tree of a sequence α, the yield of the subtree at the node p gives a factor $\alpha([i, j])$ of α. We denote by $I(p)$ the interval $[i, j]$. For the factorization tree of a run, $I(p)$ also gives an interval of the input. For example, the red node in Figure 5.4 gives the interval $[2, 4]$ of the input.

Output blocks. Given an input interval I, we denote by $out(I)$ the set of output positions whose origins belong to I (note that this might not be an output interval). An *output block* of I is a maximal interval contained in $out(I)$. In Figure 5.5, the output blocks of interval $[2, 3]$ are marked as red.

For a node p of a factorization tree of a run ρ, the set of output positions with origins in $I(p)$ is denoted by $out(p)$ instead of $out(I(p))$. We now introduce the notion of dominant output blocks of a node p of a factorization tree, denoted by $bigout_p(I)$, and prove some of its properties.

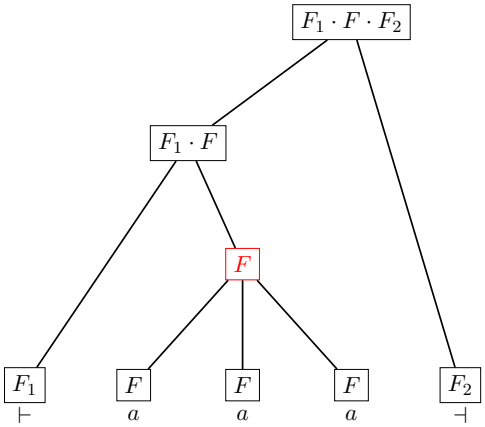

Figure 5.4 – Factorization tree of a run

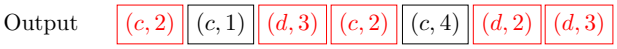

Figure 5.5 – Output blocks of interval $[2, 3]$

Dominant output blocks. For the rest of this section, we fix a synchronized pair (u, v) generated by a run ρ of a K-visit 2NFT T.

Given $n \in \mathbb{N}$, we say that a set B of output positions is *n-large* if $|orig(B)| > n$; otherwise, we say that B is *n-small*. For example, in Figure 5.5, the input interval $[2, 3]$ has three output blocks, of which the first one is 1-small and the other two are 1-large.

The constant $\mathbf{M} = (|Q| \cdot (2K+1))^{2K} + 1$ is an upper bound to the size of the flow monoid M_T. We use the derived constant $\mathbf{C} = \mathbf{M}^{2K}$ to distinguish between large and small sets of output positions.

The intuition behind this constant is that for any run ρ, a set of output positions that is $\mathbf{C}$-large must traverse a loop of ρ. This is formalized in the lemma below.

Lemma 5.2.2. *Let I be an input interval and B a set of output positions with origins inside I. If B is $\mathbf{C}$-large, then there is a loop $J \subseteq I$ of ρ such that $flow_\rho(J)$ contains a productive straight edge witnessed by a sub-run that intersects B (in particular, $out(J) \cap B \neq \emptyset$).*

Proof. The proof uses algebraic properties of the flow monoid M_T [Jec21] (see also Theorem 7.2 in [BGMP18], which proves a similar result, but with a larger constant derived from Simon's factorization theorem). In particular, [Jec21] shows that in any sequence α of monoid elements such that $|\alpha| \geq \mathbf{C}$ there is some interval with idempotent product.

Since, B is $\mathbf{C}$-large, the interval I can be partitioned into intervals $I_1, \ldots, I_{\mathbf{C}}$, where I_j is the interval between the $(j-1)$-th and j-th distinct position in $orig(B)$. The intervals I_1 and $I_{\mathbf{C}}$ start and end with the

beginning and end of I respectively. Therefore, by considering the flows of these intervals, we obtain a sub-interval J which is a loop. Since we have $J = I_j \dots I_k$, for some $i \leq k$, clearly $out(J) \cap B$ is non-empty.

□

The *dominant output interval* of I, denoted $bigout_\rho(I)$, is the smallest output interval that contains all **C**-large output blocks of I. For example, if $\mathbf{C} = 1$, then in Figure 5.5, $bigout_\rho([2,3])$ will be $(d,3)(c,2)(c,4)(d,2)(d,3)$. We sometimes write $bigout(I)$, when the run ρ is clear from the context. Note that $bigout(I)$ either is empty or begins with the first **C**-large output block of I and ends with the last **C**-large output block of I.

Properties of dominant output blocks. We present some technical lemmas that will be used in the the proof of Theorem 5.3.1. *In all lemmas till the end of this section, we assume that all successful runs of T (in particular, ρ) avoid inversions.*

The following lemma intuitively shows that the dominant output blocks respect the input order of the intervals.

Lemma 5.2.3. *Let $I_1 < I_2$ be two input intervals and B_1, B_2 output blocks of I_1, I_2, respectively. If both B_1, B_2 are **C**-large, then $B_1 < B_2$.*

Proof. B_1 and B_2 are clearly disjoint. Assume that B_1 and B_2 are **C**-large, but $B_1 > B_2$. By Lemma 5.2.2, we can find for both $i = 1$ and $i = 2$ a loop $J_i \subseteq I_i$ and a productive straight edge $e_i \in flow_\rho(J_i)$ that is witnessed by a sub-run intersecting B_i. Clearly, we have $J_1 < J_2$, and since $B_1 > B_2$, the sub-run witnessing e_1 follows the sub-run witnessing e_2. Thus, (J_1, e_1, J_2, e_2) is an inversion of ρ, which contradicts the assumption that T avoids inversions. Therefore, the lemma holds. □

The next lemma says that the number of output positions between two **C**-large output blocks of an interval, with origins outside the interval, is small.

Lemma 5.2.4. *Let I be an input interval, $B_1 < B_2$ two output blocks of I, and S the set of output positions strictly between B_1 and B_2 and with origins outside I. If B_1, B_2 are **C**-large, then S is $2\mathbf{C}$-small.*

Proof. By way of contradiction, suppose that S is $2\mathbf{C}$-large. This means that $|orig(S) \cap I'| > \mathbf{C}$ for some interval I' disjoint from I, say $I' < I$ (the case of $I' > I$ is treated similarly). By Lemma 5.2.2, we can find two loops $J \subseteq I$ and $J' \subseteq I'$ and some productive straight edges $e \in flow_\rho(J)$ and $e' \in flow_\rho(J')$ that are witnessed by sub-runs intersecting B_1 and S, respectively. Since $S > B_1$, we know that the sub-run witnessing e follows the sub-run witnessing e'. As in the previous proof, this shows the inversion (J, e, J', e'), which contradicts the assumption that T avoids inversions. □

Another lemma below shows that the number of positions added to dominant blocks by merging the dominant blocks of two consecutive intervals is small.

Lemma 5.2.5. *Let $I = I_1 \cdot I_2$, $B = bigout(I)$, and $B_i = bigout(I_i)$ for $i = 1, 2$. Then $B \setminus (B_1 \cup B_2)$ is $4K\mathbf{C}$-small.*

Proof. By Lemma 5.2.3, we have $B_1 < B_2$. Moreover, all $\mathbf{C}$-large output blocks of I_1 or I_2 are also $\mathbf{C}$-large output blocks of I, so B contains both B_1 and B_2. Let I_0 be the maximal interval to the left of I_1, and thus adjacent to it, and, similarly, let I_3 be the maximal interval to the right of I_2, and thus adjacent to it.

Suppose, by way of contradiction, that $B \setminus (B_1 \cup B_2)$ is $4K\mathbf{C}$-large. This means that there is a $2K\mathbf{C}$-large set $S \subseteq B \setminus (B_1 \cup B_2)$ with origins entirely inside $I_0 \cdot I_1$ or entirely inside $I_2 \cdot I_3$. Suppose, w.l.o.g., that the former case holds, and decompose S as a union of maximal output blocks $B'_1, B'_2, \ldots, B'_n$ of either I_0 or I_1. Since $S \cap B_1 = \emptyset$, we have that every block B'_i with origins inside I_1 is $\mathbf{C}$-small. Similarly, by Lemma 5.2.4, every block B'_i with origins inside I_0 is $\mathbf{C}$-small too. Moreover, since T is K-visit, we have that the number n of maximal output blocks of either I_0 or I_1 that are contained in S is at most $2K$. All together, this contradicts the assumption that S is $2K\mathbf{C}$-large. □

Lemma 5.2.6. *Let I be a loop of ρ. Then $flow_\rho(I)$ has at most one productive straight edge, and this edge must be* LR.

Proof. Suppose, by way of contradiction, that there are two productive straight edges in $flow_\rho(I)$, say e and f, with e before f in ρ. Suppose that we pump I twice, and let $I_1 < I_2$ be the copies of I in the pumped run ρ'. Let also e_1, e_2 (resp. f_1, f_2) be the corresponding copies of e (resp. f), so that e_j, f_j belong to the $flow_{\rho'}(I_j)$. It is easy to check the following properties:

- if e is an LR edge, then the sub-run witnessed by e_1 occurs in ρ' before the sub-run witnessed by e_2 (and the other way around if e is RL);
- the sub-runs witnessed by e_1 and e_2 occur in ρ' before the sub-runs witnessed by f_1, f_2.

Let us assume first that e is an RL edge. Then observe that (I_1, e_1, I_2, e_2) is an inversion in ρ'. But this contradicts T being inversion-free. Therefore, both e, f are LR edges. But then, (I_1, f_1, I_2, e_2) is an inversion in ρ', and we have again a contradiction. □

Using the above lemma, we prove our last technical lemma:

Lemma 5.2.7. *Let $I = I_1 \cdot I_2 \cdots I_n$, such that I is a loop and $flow_\rho(I) = flow_\rho(I_k)$ for all k. Then $bigout(I)$ can be decomposed as $B_1 \cdot J_1 \cdot B_2 \cdot J_2 \cdot \cdots \cdot J_{n-1} \cdot B_n$, where*

1. $B_k = \mathit{bigout}(I_k)$ *for all* k *(with* B_k *possibly empty);*
2. *for* $1 \leq k < n$*, the positions in* J_k *have origins inside* $I_k \cup I_{k+1}$ *and* J_k *is* $2K\mathbf{C}$*-small.*

Proof. By Lemma 5.2.6, we can assume that $\mathit{flow}_\rho(I) = \mathit{flow}_\rho(I_k)$ has a unique productive straight edge e, which is an LR edge. Let B'_k be the output block corresponding to e in $\mathit{flow}_\rho(I_k)$. Since $\mathit{flow}_\rho(I)$ is idempotent, any output block of I has one of the following shapes:

(a) A block $B = B'_1 \cdot J'_1 \cdot \ldots J'_{n-1} \cdot B'_n$, for some intervals $J'_1, \ldots, J'_{n-1}$ such that $\mathit{out}(I_k)$ is included in $J'_{k-1} \cdot B'_k \cdot J'_k$ for all $1 < k < n$,

(b) At most $2K$ output blocks $L_1, \ldots, L_p, R_1, \ldots, R_s$, where each L_i and R_j corresponds to an edge of $\mathit{flow}_\rho(I_1)$ and $\mathit{flow}_\rho(I_n)$, respectively: the blocks L_i, R_j appear before, respectively after the straight edge.

Moreover, the order of the output blocks of I is $L_1, \ldots, L_p, B, R_1, \ldots, R_s$.

Note that $B_k = \mathit{bigout}(I_k)$ is contained in $J'_{k-1} \cdot B'_k \cdot J'_k$ for all $1 < k < n$. Moreover, $B_1 = \mathit{bigout}(I_1)$ is contained in $L_1 \cdots L_p \cdot B'_1 \cdot J'_1$, and $B_n = \mathit{bigout}(I_n)$ is contained in $J'_{n-1} \cdot B'_n \cdot R_1 \cdots R_s$. Also by Lemma 5.2.3, B_j precedes B_{j+1} for all j.

If one of the L_k is $\mathbf{C}$-large, then B_1 is non-empty, hence $\mathit{bigout}(I)$ is non-empty and starts at the first position of B_1. Similarly, if one of the R_k is $\mathbf{C}$-large then B_n is non-empty, hence $\mathit{bigout}(I)$ is non-empty and ends with the last position of B_n. Otherwise, if all L_j, R_j are $\mathbf{C}$-small then $\mathit{bigout}(I)$ is either empty or equal to B. In all cases we can write $\mathit{bigout}(I) = B_1 \cdot J_1 \cdot B_2 \cdot J_2 \cdot \cdots \cdot J_{n-1} \cdot B_n$, with each J_k consisting of at most K $\mathbf{C}$-small blocks of I_k and K $\mathbf{C}$-small blocks of I_{k+1}, namely those left over after gathering the $\mathbf{C}$-large blocks into $\mathit{bigout}(I_k)$ and $\mathit{bigout}(I_{k+1})$, respectively. Therefore, each J_k is $2K\mathbf{C}$-small. □

In the proof of Theorem 5.3.1, we will use the previous Lemmas and an induction on the factorization tree of a run to build the desired resynchronizer.

5.3 Bounded-visit case

We call a bounded, regular resynchronizer *functional*, if for every $\hat{u} \in \mathsf{ipar}$, for every position $i, j, j' \in \mathrm{dom}(u)$, and for every output type τ, if $(\hat{u}, i, j) \vDash \mathsf{move}_\tau$ and $(\hat{u}, i, j') \vDash \mathsf{move}_\tau$, then $j = j'$. In other words, for a fixed annotated input, move_τ functionally determines the target origin based on the source origin. Note that this is the dual of the 1-bounded restriction (see page 73), where the target origin uniquely determines the source origin. Note that functionality here is a restriction on move_τ. A synchronized pair (u, v) can still be mapped to multiple synchronized pairs depending on the input annotations defined by ipar.

We now state and prove the characterization of one-way resynchronizability for bounded-visit 2NFTs.

Theorem 5.3.1. *For every bounded-visit 2NFT T, the following are equivalent:*

(1) T is one-way resynchronizable*, i.e, there exists a bounded, regular resynchronizer R such that $R(T)$ is one-way definable in the origin semantics,*

(2) the cross-width of T is finite,

(3) no successful run of T has inversions,

(4) there is a functional*,* 1-bounded*, regular resynchronizer R that is T-preserving and such that $R(T)$ is order-preserving.*

The complexity of one-way resynchronizability is PSPACE*-complete (assuming that the bound for visits is given in unary).*

Proof. First of all, observe that the implication 4. to 1. is straightforward.

Implication 1. to 2. Assume that there is a k-bounded, regular resynchronizer R that is T-preserving and such that $R(T)$ is order-preserving. Theorem 3.3.10 implies that R is bounded-traversal by t, for some constant t. We prove that T has cross-width bounded by $t + k$.

Consider two synchronized pairs (u, v) and (u, v') such that $(u, v) \in T$ and $((u, v), (u, v')) \in R$, and consider a cross (X_1, X_2) of (u, v). We denote by $orig(X)$ the set of input positions i such that $i = orig(v(x))$ for some position $x \in X$. These are the set of origins of positions in X in v. Similarly, we denote by $orig'(X)$ the set of origins of positions in X in v'. We claim that for the cross (X_1, X_2), either $|orig(X_1)|$ or $|orig(X_2)|$ is at most $t + k$.

Let $i_1 = \min(orig(X_1))$, $i'_1 = \max(orig'(X_1))$, and $i_2 = \max(orig(X_2))$, and $i'_2 = \min(orig'(X_2))$, where min and max is based on the input order. Since (X_1, X_2) is a cross, we have $i_1 > i_2$, and since (u, v') is order-preserving, we have $i'_1 \leq i'_2$.

Now, suppose $i'_1 < i_1$. This corresponds to the case that, for all positions of $x \in X_1$, $orig(v'(x)) < orig(v(x))$, i.e, the origins of all positions in X_1 are moved to the left of i_1. Therefore, then at least $|orig(X_1)| - k$ input positions from $orig(X_1)$ traverse i'_1 from the right. The $-k$ term is due to the fact that at most k input positions can be resynchronized to i'_1.

Symmetrically, if $i'_2 > i_2$, then at least $|orig(X_2)| - k$ input positions from $orig(X_2)$ traverse i_2 from the left. Note that this corresponds to the case when the origins of all positions of X_2 are moved to the right of i_2. In both these cases, we have $|orig(X_1)|$ (resp. $|orig(X_2)|$) to be less than $k + t$. Otherwise, (u, v) is not traversal bounded by t.

If $i'_1 \geq i_1$, and $i'_2 \leq i_2$, then we have $i'_1 \geq i_1 > i_2 \geq i'_2$, which contradicts that (u, v') is order-preserving. Therefore, $\min(|orig(X_1)|, |orig(X_2)|) \leq t + k$, as claimed.

Note that this part of the proof does not use the bounded-visit restriction in any way. The remaining implications however, rely on the assumption that T is bounded-visit.

Implication 2. to 3. We prove this implication by contradiction. Consider a successful run ρ of T on some input u and suppose there is an inversion: ρ has disjoint loops $I < I'$, whose flows contain productive straight edges, say e in $\mathit{flow}_\rho(I)$ and e' in $\mathit{flow}_\rho(I')$, such that e' precedes e in the run order. Let $u = u_1\, w\, u_2\, w'\, u_3$ so that w and w' are the factors of the input delimited by the loops I and I', respectively. Further let v and v' be the outputs produced along the edges e and e', respectively. Consider now the run ρ_k obtained from ρ by pumping the input an arbitrary number k of times on the loops I and I'. This run is over the input $u_1\,(w)^k\,u_2\,(w')^k\,u_3$, and in the output produced by ρ_k there are k (possibly non-consecutive) occurrences of v and v'. By Lemma 5.2.1 all occurrences of v' precede all occurrences of v. In particular, if X_1 (resp. X_2) is the set of positions corresponding to all the occurrences of v (resp. v') in the output produced by ρ_k, then (X_1, X_2) is a cross of width at least k.

Implication 3. to 4. This is the most interesting direction of the proof. In the absence of inversions, we construct a functional resynchronizer. The resynchronizer R will use input and output parameters to guess a successful run ρ of T on the input u and a corresponding *factorization tree* for ρ of height at most $H = 3|M_T|$. The existence of such a factorization tree linear in the size of the monoid is guaranteed by Simon's factorization theorem [Sim90]. The bound can be in fact be made $3|M_T|$ (see [Col07]). We build the resynchronizer inductively using the properties of the dominant output blocks proved earlier (see page 128).

Let (u, v) be a synchronized pair produced by a run ρ of T. Fix a factorization tree of the run ρ. For an output position $x \in \mathrm{dom}(v)$, and a level ℓ of the factorization tree of ρ, we denote by $p_{x,\ell}$ the unique node at level ℓ such that $I(p_{x,\ell})$ contains the source origin of x.

Input parameters. The successful run ρ together with its factorization tree of height at most $H = 3|M_T|$ can be encoded over the input using input parameters and the formula ipar using classical techniques. The parameters describe each input interval $I(p)$ and the label $\mathit{flow}(I(p))$ of each node p in the factorization tree. Formally, an input interval $I(p)$ is described by marking the begin and end with two distinguished parameters for the specific level. The label $\mathit{flow}(I(p))$ annotates every position inside $I(p)$. This accounts for $H(2 + |M_T|)$ input parameters. Correctness of the annotations with the above input parameters can be expressed by a formula ipar. In particular, on the leaves, ipar checks that every interval is a singleton of the

form $\{i\}$ and its flow is the one induced by the letter $u(i)$. On the internal nodes, ipar checks that the label of a node coincides with the monoid product of the labels of its children, which is a composition of flows. It also checks that for every node with more than two children, the node and the children are labeled by the same idempotent flow.

Output parameters. We also need to encode the run ρ on the output, because the resynchronizer will determine the target origin of an output position, not only on the basis of the flow at the source origin, but also on the basis of the productive transition that generated that particular position. The annotation that encodes the run ρ on the output is done using output parameters (one for each transition in Δ), and its correctness will be enforced by a suitable combination of the formulas opar, move_τ, and $\mathsf{next}_{\tau,\tau'}$.

Below, we explain how the resynchronizer works assuming the output is correctly annotated with the flow and the marked productive transition. We will explain how to check the correctness of the annotations later.

In a nutshell, origins are transformed below by a series of partial resynchronizers R_ℓ that "converge" in finitely many steps to a desired resynchronization, under the assumption that the output annotation correctly encodes the same run ρ that is represented in the input annotation.

Moving origins. Here we will work with a fixed successful run ρ and a factorization tree for it, that we assume are correctly encoded by the input and output annotations. For every level ℓ of the factorization tree, we will define a functional, bounded, regular resynchronizer R_ℓ. Each resynchronizer R_ℓ will be *partial*, in the sense that for some output positions it will not define source-target origin pairs. However, the set of output positions with associated source-target origin pairs increases with the level ℓ, and the top level resynchronizer R_* will specify source-target origin pairs for all output positions. The latter resynchronizer will almost define the resynchronization that is needed to prove item (4) of the theorem; we will only need to modify it slightly in order to make it 1-bounded and to check that the output annotation is correct.

To enable the inductive construction, we need the resynchronizer R_ℓ to satisfy the following properties, for every level ℓ of the factorization tree:

- the set of output positions for which R_ℓ defines target origins is the union of the dominant output intervals $\mathit{bigout}(p)$ of all nodes p at level ℓ;
- R_ℓ only moves origins within the same interval at level ℓ, that is, R_ℓ defines only pairs (y, z) of source-target origins such that $y, z \in I(p)$ for some node p at level ℓ;
- the target origins defined by R_ℓ are order-preserving within the same interval at level ℓ, that is, for all output positions $x < x'$, if R_ℓ defines

the target origins of x, x' to be z, z', respectively, and if $z, z' \in I(p)$ for some node p at level ℓ, then $z \leq z'$.

— R_ℓ is $\ell \cdot 4K\mathbf{C}$-bounded, namely, there are at most $\ell \cdot 4K\mathbf{C}$ distinct source origins that are moved by R_ℓ to the same target origin.

The inductive construction of R_ℓ will basically amount to defining appropriate formulas $\mathsf{move}_\tau(y, z)$.

Base case. The base case is $\ell = 0$, namely, when the resynchronization is acting at the leaves of the factorization tree. In this case, the regular resynchronizer R_ℓ is vacuous, as the input intervals $I(p)$ associated with the leaves p are singletons, and hence all dominant output intervals $\mathit{bigout}(p)$ are empty. Formally, for this resynchronizer R_ℓ, we simply let $\mathsf{move}_\tau(y, z)$ be false, independently of the underlying output type τ and of the source and target origins. This resynchronization is clearly functional, 0-bounded, and order-preserving.

Inductive step. For the inductive step, we explain how the origins of an output position $x \in \mathit{bigout}(p)$ are moved within the interval $I(p)$, where $p = p_{x,\ell}$ is the node at level ℓ that "generates" x. Even though we explain this by mentioning the node $p_{x,\ell}$, the definition of the resynchronization will not depend on it, but only on the level ℓ and the underlying input and output parameters. In particular, to describe how the origin of a τ-labeled output position x is moved, the formula $\mathsf{move}_\tau(y, z)$ has to determine the productive edge that generated x in the flow that labels the node $p_{x,\ell}$. This can be done by first determining from the output type τ the productive transition t_x that generated x, and then inspecting the annotation at the source origin y to "track" t_x inside the productive edges of the flow $\mathit{flow}(I_{p'})$, for each node p' along the unique path from the leaf $p_{x,0}$ to node $p_{x,\ell}$. In the case distinction below, we implicitly rely on this type of computation, which can be easily implemented in MSO using the output parameters.

1. ***$p_{x,\ell}$* is a binary node**. We first consider the case where $p = p_{x,\ell}$ is a binary node (the annotation on the source origin y will tell us whether this is the case). Let p_1, p_2 be the left and right children of p. If x belongs to one of the dominant output blocks $\mathit{bigout}(p_1)$ and $\mathit{bigout}(p_2)$ (again, this information is available at the input annotation), then the resynchronizer R_ℓ will inherit the source-target origin pairs associated with x from the lower level resynchronization $R_{\ell-1}$. Note that $\mathit{bigout}(p_1) < \mathit{bigout}(p_2)$ by Lemma 5.2.3, so R_ℓ is order-preserving at least for the output positions inside $\mathit{bigout}_{p_1}(\cup)\mathit{bigout}_{p_2}()$.

   We now describe the source-target origin pairs when $x \in \mathit{bigout}(p) \setminus (\mathit{bigout}(p_1) \cup \mathit{bigout}(p_2))$. The idea is to move the origin of x to one of the following three input positions, depending on the relative order between x and the positions in $\mathit{bigout}(p_1)$ and in $\mathit{bigout}(p_2)$:

   — the first position of $I(p_1)$, if $x < \mathit{bigout}(p_1)$;

   — the last position of $I(p_1)$, if $\mathit{bigout}(p_1) < x < \mathit{bigout}(p_2)$;

— the last position of $I(p_2)$, if $x > \mathit{bigout}(p_2)$.

Which of the above cases holds can be determined, again, by inspecting the output type τ and the annotation of the source origin y, in a way similar to the computation of the productive edge that generated x at level ℓ. So the described resynchronization can be implemented by an MSO formula $\mathsf{move}_\tau(y, z)$.

The resulting resynchronization R_ℓ is functional and order-preserving inside every interval at level ℓ. It remains to argue that R_ℓ is $\ell \cdot 4K\mathbf{C}$-bounded. To see why this holds, assume, by the inductive hypothesis, that $R_{\ell-1}$ is $(\ell-1)\cdot 4K\mathbf{C}$-bounded. Recall that the new source-target origin pairs that are added to R_ℓ are those associated with output positions in $\mathit{bigout}(p)\setminus(\mathit{bigout}(p_1)\cup\mathit{bigout}(p_2))$. Lemma 5.2.5 tells us that there are at most $4K\mathbf{C}$ distinct positions that are source origins of such positions. So, in the worst case, at most $(\ell-1)\cdot 4K\mathbf{C}$ source origins from $R_{\ell-1}$ and at most $4K\mathbf{C}$ new source origins from R_ℓ are moved to the same target origin. This shows that R_ℓ is $\ell \cdot 4K\mathbf{C}$-bounded.

2. $\boldsymbol{p_{x,\ell}}$ **is an idempotent node**. The case where $p = p_{x,\ell}$ is an idempotent node with children $p_1, p_2, \ldots, p_n$ follows a similar approach. For brevity, let $I_i = I(p_i)$ and $B_i = \mathit{bigout}(p_i)$. By Lemma 5.2.3, we have $B_1 < B_2 < \cdots < B_n$. Lemma 5.2.7 then provides a decomposition of $\mathit{bigout}(p)$ as $B_1 \cdot J_1 \cdot B_2 \cdot J_2 \cdot \cdots \cdot J_{n-1} \cdot B_n$, for some $2K\mathbf{C}$-small output intervals J_k, for $k = 1, \ldots, n-1$, that have origins inside $I_k \cup I_{k+1}$.

   As before, the resynchronizer R_ℓ behaves exactly as $R_{\ell-1}$ for the output positions inside the B_k's. For any other output position, say $x \in J_k$ for some $k = 1, 2, \ldots, n-1$, we first recall that the source origin y of x is either inside I_k or inside I_{k+1}. Depending on which of the two intervals contains y, the resynchronizer R_ℓ will define the target origin z to be either the last position of I_k or the first position of I_{k+1}. However, since we cannot determine using MSO the index k of the interval J_k that contains x, we proceed as follows.

   First observe that any block B_i can be identified by some flow edge at level $\ell-1$, and the latter edge can represented in MSO by suitable monadic predicates over the input. Let B, B' be the two consecutive blocks among $B_1, \ldots, B_n$ such that $B < x < B'$. Note that these blocks can be determined in MSO once the productive edge that generated x is identified. Further let I be the interval among $I_1, \ldots, I_n$ that contains the origin y of x. By the previous arguments, we have that the interval I contains either all the origins of B or all the origins B'. Moreover, which of the two sub-cases holds can again be determined in MSO by inspecting the annotations. The formula $\mathsf{move}_\tau(y, z)$ can thus define the target origin z to be
   — the last position of I, if I contains the origins of B;

— the first position of I, if I contains the origins of B'.

The above construction results in a functional regular resynchronization R_ℓ that associates with any two output positions $x < x'$ with source origins in the same interval $I(p)$, some target origins $z \leq z'$. In other words, the resynchronization R_ℓ is order-preserving in each interval at level ℓ.

It remains to show that R_ℓ is $\ell \cdot 4K\mathbf{C}$-bounded, under the inductive hypothesis that $R_{\ell-1}$ is $(\ell-1) \cdot 4K\mathbf{C}$-bounded. This is done using a similar argument as before, that is, by observing that the output positions in $\mathit{bigout}(p) \setminus \big(\bigcup_{1 \leq k \leq n} \mathit{bigout}(p_i)\big)$ belong to some J_k, and in the worst case all source origins y of positions from J_k are moved to the last position of I_k. By Lemma 5.2.7, there are at most $2K\mathbf{C}$ such positions y.

Top level resynchronizer. Let R_* be the resynchronizer R_ℓ obtained at the top level ℓ of the factorization tree. Based on the above constructions, R_* defines target origins for all output positions, unless the dominant output interval $\mathit{bigout}(p)$ associated with the root p is empty (this can indeed happen when the number of different origins in the output is at most $\mathbf{C}$, so not sufficient for having at least one $\mathbf{C}$-large output factor). In particular, if $\mathit{bigout}(p) \neq \emptyset$, then $\mathit{bigout}(p)$ is the whole output, and R_ℓ is basically the desired resynchronization, assuming that the output annotations are correct.

Let us now discuss briefly the degenerate case where $\mathit{bigout}(p) = \emptyset$, which of course can be detected in MSO. In this case, the appropriate resynchronizer R_* should be redefined so that it moves all source origins to the same target origin, say the first input position. Clearly, this gives a functional, regular resynchronizer that is order-preserving and $\mathbf{C}$-bounded.

Correctness of output annotation. Recall that the properties of the resynchronizers R_ℓ crucially rely on the assumption that every output position x is correctly annotated with the productive transition that generated it. This assumption cannot be guaranteed by the MSO sentence opar alone since the property intrinsically talks about a relation between input and output annotations. Below, we explain how to check correctness of the output annotation with the additional help of the formulas $\mathsf{move}_\tau(y, z)$ (that will be modified for this purpose) and $\mathsf{next}_{\tau,\tau'}(z, z')$.

Let ρ be the successful run as encoded by the input annotation. The idea is to check that the sequence of productive transitions t_x, where t_x is the transition that annotates position x in the output, is the maximal subsequence of ρ consisting only of productive transitions. Besides the straightforward conditions (concerning, for instance, the first and last productive transitions of ρ, or the possible multiple symbols that could be produced within a single transition), the important and difficult condition to be veri-

fied is the following:

> *Let ρ be the run that is annotated on the input. For every pair of consecutive output positions $x, x+1$ with source origins y, y', respectively, one can move from transition t_x at position y of ρ to transition t_{x+1} at position y' of ρ by using as intermediate steps only non-productive transitions.* (†)

The above property is easily expressible by an MSO formula $\varphi^{\dagger}_{\tau,\tau'}(y, y')$, assuming that τ, τ' are the output types of $x, x+1$ and the free variables y and y' are interpreted as the *source origins* of x and $x+1$, with x ranging over all output positions. This is very close to the type of constraints that can be enforced by the formula $\mathsf{next}_{\tau,\tau'}$ of a regular resynchronizer, with the only difference that the latter formula can only access the *target origins* z, z' of $x, x+1$.

We thus need a way to uniquely determine from the target origins z, z' of x the source origins y, y' of x. For this, we could rely on the formulas $\mathsf{move}_\tau(y, z)$, if only they were defining a partial bijection between y and z. Those formulas are in fact close to define partial bijections, as they are functional and k-bounded, for $k = H \cdot 4K\mathbf{C}$. The latter boundedness property, however, depends again on the assumption that the output annotation is correct. We overcome this problem by gradually modifying the resynchronizer R_* so as to make it functional and 1-bounded, independently of the output annotations.

We start by modifying the formulas $\mathsf{move}_\tau(y, z)$ to make them "syntactically" k-bounded. Formally, we construct from $\mathsf{move}_\tau(y, z)$ the formula

$$\begin{aligned}\mathsf{move}'_\tau(y, z) \;=\; & \mathsf{move}_\tau(y, z) \\ & \wedge \;\; \forall y_1, \ldots, y_k, y_{k+1} \, \Big(\bigwedge\nolimits_i \mathsf{move}_\tau(y_i, z) \Big) \rightarrow \Big(\bigvee\nolimits_{i \neq j} y_i = y_j \Big).\end{aligned}$$

Intuitively, the above formula is semantically equivalent to $\mathsf{move}_\tau(y, z)$ when there are at most k input positions y' that can be paired with z via the same formula move_τ, and it is false otherwise.

Let R'_* be the regular resynchronizer obtained from R_* by replacing the formulas move_τ by move'_τ, for every output type τ. By construction, R'_* is functional and k-bounded, independently of any assumption on the output annotations. We can then apply Lemma 3.3.4 and obtain from R'_* an equivalent regular resynchronizer $R''_* = (\overline{I}'', \overline{O}'', \mathsf{ipar}'', \mathsf{opar}'', (\mathsf{move}''_\tau)_\tau, (\mathsf{next}''_{\tau,\tau'})_{\tau,\tau'})$ that is functional and 1-bounded, and therefore, defines a partial bijection.

We are now ready to verify the correctness of the output annotation. Recall that the idea is to enforce the property (†) by exploiting the previously defined formula $\varphi^{\dagger}_{\tau,\tau'}(y, y')$ and the partial bijection between the source origins y, y' and the target origins z, z', as defined by $\mathsf{move}''_\tau(y, z)$ and

$\mathsf{move}''_{\tau'}(y', z')$. Formally, we define

$$\mathsf{next}'''_{\tau,\tau'}(z, z') \;=\; \mathsf{next}''_{\tau,\tau'}(z, z') \wedge \exists y, y'\, \mathsf{move}''_{\tau}(y, z) \wedge \mathsf{move}''_{\tau'}(y', z') \wedge \varphi^{\dagger}_{\tau,\tau'}(y, y').$$

To conclude, by replacing in R'' the formulas $\mathsf{next}''_{\tau,\tau'}$ with $\mathsf{next}'''_{\tau,\tau'}$, we obtain a regular resynchronizer R that is functional, 1-bounded, T-preserving and such that $R(T)$ is order-preserving. This completes the proof of the implication 3. to 4. of our Theorem 5.3.1.

Complexity. The effectiveness of Theorem 5.3.1 comes from condition (3) about inversions. The presence of inversions in a run can be guessed and verified in space exponential in the size of T. Indeed, to guess an inversion, the algorithm needs to guess the flows at loops $I < I'$ and check that it is indeed an inversion. To do this, it needs to be checked that there exist flows F_1, F_2, F_3 such that $F_1 \cdot \mathit{flow}_\rho(I) \cdot F_2 \cdot \mathit{flow}_\rho(I') \cdot F_3$ corresponds to a run, and there exist straight productive edges e and e' in $\mathit{flow}_\rho(I)$ and $\mathit{flow}_\rho(I')$ $e' < e$ in the juxtaposition. Note that even though the size of the flow monoid is exponential in the constant K bounding the number of visits, to detect inversions, one only needs to guess $O(1)$ flows. Since, each of them has size polynomial in the size of T and K, the algorithm only needs polynomial space. Therefore, we have an upper bound of PSpace.

The lower bound of PSpace comes from the emptiness problem for 2DFA, similar to the lower bound for one-way definability in the origin semantics (Proposition 5.1.4). Given a 2DFA accepting a regular language L, we can construct another 2DFA accepting $(L\#)^*$ which is the language of words of the form $w_1\# \dots w_n\#$, where $w_i \in L$. The reverse function on $(L\#)^*$ can be realized by a 2DFT, which will be one-way resynchronizable if and only if $(L\#)^*$ is empty. Note that here we require the reverse function on $(L\#)^*$ instead of L because the reverse function on L can be one-way resynchronizable if L is a finite language. □

5.4 General case

In this section, we prove Theorem 5.1.6, which says that one-way resynchronizability is decidable for arbitrary 2NFTs. The idea is to use the decidability result for the bounded-visit case, which was proved in Theorem 5.3.1.

The main obstacle towards dropping the bounded-visit restriction from Theorem 5.3.1, while maintaining the effectiveness of the characterization, is the lack of a bound on the number of flows. For a 2NFT T that is not necessarily bounded-visit, there is no bound on the number of emerging flows that encode successful runs of T. Note that the proofs of the implications 1. to 2. and 4. to 1. in Theorem 5.3.1 remain valid, even for a 2NFT T that is not bounded-visit.

Given a 2NFT T, we would like to construct:

1. a bounded-visit 2NFT $low(T)$ that is classically equivalent to T,
2. a functional, 1-bounded, regular resynchronizer R that is T-preserving and such that $R(T) =_o low(T)$.

Once we did so, we could apply our characterization of one-way resynchronizability in the bounded-visit case to the 2NFT $low(T)$. If $low(T)$ is one-way resynchronizable, then by Theorem 5.3.1 we obtain another functional, 1-bounded, regular resynchronizer R' that is $low(T)$-preserving and such that $R'(low(T)))$ is order-preserving. Thanks to Lemma 3.3.5, the resynchronizers R and R' can be composed, so we conclude that the original 2NFT T is one-way resynchronizable. Otherwise, if $low(T)$ is not one-way resynchronizable, we show that neither is T.

There are however some challenges in the approach described above. First, as T may output arbitrarily many symbols with origin in the same input position, and $low(T)$ is bounded-visit, we need $low(T)$ to be able to produce arbitrarily long outputs within a single transition. For this reason, we allow $low(T)$ to be a 2NFT with *regular outputs* (see page 9). The transition relation of such a 2NFT consists of finitely many tuples of the form (q, a, L, q'), with $q, q' \in Q$, $a \in \Sigma$, and $L \subseteq \Gamma^*$ a regular language over the output alphabet. Such a transition can output any word from L. We also need to use 2NFTs with common guess (see page 12). It can be checked that the proof of Theorem 5.3.1 in the bounded-visit case can be rather easily adapted to these features.

There is still another problem: we cannot always expect that there exists a bounded-visit 2NFT $low(T)$ classically equivalent to T. Consider, for instance, the 2NFT that performs an arbitrary number of passes on the input, and on each left-to-right pass, it reads the input position and copies the letter. Therefore, the output on an input u will be any word in $((u(1), 1)(u(2), 2) \dots (u(n), n))^*$. By standard pumping arguments, it is possible to show that no bounded-visit 2NFT can realize such a relation. Therefore, we first check a suitable condition on the 2NFT T under which it becomes possible to construct an equivalent bounded-visit 2NFT $low(T)$ such that T is one-way resynchronizable if and only if $low(T)$ is one-way resynchronizable. For this, we introduce the notion of vertical loop.

Vertical loop. Given a run ρ of a 2NFT T on an input u, a *vertical loop* is a sub-run of ρ that begins and ends with the same state at the same cut. For example, in Figure 5.6, the blue sub-run starts and ends in the same configuration$(q_1, 2)$ (recall that cut i is between positions i and $i + 1$). This is an example of a vertical loop. We use this run from the figure as a running example. The intuition for defining such a loop is that it can be pumped without having to pump the input.

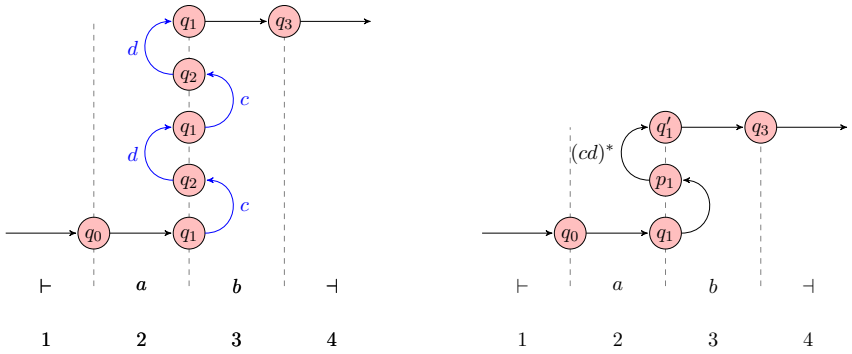

Figure 5.6 – A vertical loop and the summarized run

Given a vertical loop ρ', we say an input position i is an origin in ρ' if there is a productive transition in ρ' reading the i-th position. In our example, both position 2 and 3 are origins in the vertical loop.

Reachability order on an input. Given an input u and a 2NFT T, a *tagged transition* is any pair (t, i), where $t \in \Delta$ is a transition of T and i is a position of the input u, such that t occurs at position i in some successful run on u. The reachability preorder on tagged transitions is such that $(t, i) \preceq_u (t', i')$ whenever T has a run on u starting with the transition t reading the letter $u(i)$ and ending with the transition t' reading the letter $u(i')$. We can use this preorder to define an equivalence relation $\sim_u$ such that $(t, i) \sim_u (t', i')$ if and only if $(t, i) \preceq_u (t', i')$ and $(t', i') \preceq_u (t, i)$. Strictly speaking, this is not an equivalence relation on all tagged transitions, since it is possible to have $(t, i) \not\preceq_u (t, i)$. However, for tagged transitions that are part of a vertical loop, this is indeed a equivalence relation.

Intuitively, $(t, i) \sim_u (t', i')$ means that T can cycle an arbitrary number of times between these two tagged transitions on the input u. In particular, each equivalence class identifies a vertical loop in some run of T. In our example, let $t_1 = (q_1, b, c, q_2)$ and $t_2 = (q_2, a, d, q_1)$ be the transition marked in blue. Therefore, the tagged transitions $(t_1, 3)$ and $(t_2, 2)$ are in the same equivalence class defined by the word $u = \vdash ab \dashv$.

A $\sim_u$-equivalence class C is called *realizable on* u if there is a successful run on u that uses at least once a tagged transition from the class C. If a transition (t, i) from C is used in some run ρ, it can be easily extended to a run ρ' which uses all tagged transitions from C. For any $(t', i') \in C$, the run ρ can be changed to move from (t, i) to (t', i') and back to (t, i) witnessing realizability of (t', i').

Sparsity of vertical loops. We say that T is *k-sparse* if for every input u and every realizable $\sim_u$-equivalence class C, there are at most k tagged transitions in C that are productive (recall that a productive transition is

one that produces non-empty output). Intuitively, bounded sparsity means that the number of origins of outputs produced by vertical loops in successful runs of T is uniformly bounded.

When T is k-sparse, the productive tagged transitions from the same realizable $\sim_u$-equivalence class can be lexicographically ordered and distinguished by means of numbers from a fixed finite range, say $\{1, \ldots, k\}$. An important observation is that the equivalence $\sim_u$ is a regular property, in the sense that one can construct, for instance, an MSO formula $\varphi_{t,t'}^{\sim_u}(i, i')$ that holds on input u if and only if $(t, i) \sim_u (t', i')$. In particular, this implies that unbounded sparsity can be effectively tested. It suffices to construct the regular language consisting of every possible input u with a distinguished realizable $\sim_u$-equivalence class marked on it. The problem of checking unbounded sparsity therefore reduces to checking whether this language contains words with arbitrarily many marked positions that correspond to productive tagged transitions. If there are unboundedly many marked positions, then by standard pumping arguments, there must be a loop. Therefore, the problem reduces to checking loops with a marked position in the NFA for the language of inputs with marked equivalence class.

Lemma 5.4.1. *If T has unbounded sparsity, then T is not one-way resynchronizable.*

Proof. The assumption that T has unbounded sparsity and the definition of $\sim_u$ imply that, for every $n \in \mathbb{N}$, there exist an input u, a successful run ρ on u, and $2n$ tagged transitions $(t_1, i_1), \ldots, (t_n, i_n)$, $(t'_1, i'_1), \ldots, (t'_n, i'_n)$ such that the t_j's occur before the t'_ℓ in ρ and the i_j are to the right of the i'_ℓ. Since n can grow arbitrarily, this witnesses precisely the fact that T has unbounded cross-width. Thus, by the implication $1 \to 2$ of Theorem 5.3.1, which is independent of T being bounded-visit, we know that T is not one-way resynchronizable. □

We now construct from a given 2NFT with bounded-sparsity, a classically equivalent bounded-visit 2NFT $low(T)$ such that T is one-way resynchronizable if and only if $low(T)$ is one-way resynchronizable. We will use the features of regular outputs and common guess in $low(T)$.

Equivalent bounded-visit 2NFT. Assume T is a k-sparse 2NFT. Intuitively, $low(T)$ simulates successful runs of T on input u by shortcutting maximal vertical loops. The output of a vertical loop will be summarized by a regular language.

For a run ρ on input u, let ρ' be a vertical loop starting with the transition (t, i). By definition, all tagged transitions of ρ' (t', i') are $\sim_u$-equivalent to (t, i). Call this equivalence class $C_{(t,i)}$. Since T is k-sparse, there are at most k productive tagged transitions in $C_{(t,i)}$.

The productive transitions of $C_{t,i}$ can be seen as the vertices of a strongly connected graph, with edge from (t,i) to (t',i'), if there is a non-productive run on u starting with (t,i) and ending in (t',i'). By replacing the vertex (t,i) by L_t, where L_t is the language of output words in the transition t, we can obtain a NFA. Note that this is not immediately an NFA, since the words are at a state. However, by standard arguments, we can replace the vertices by an edge to obtain an NFA. By fixing a tagged transition (t,i) as the initial vertex, we obtain a regular language $L_{t,i}$. This language will be used to summarize the output of a vertical loop. Note that for any choice of (t,i), there are only finitely many languages $L_{t,i}$. This is because each of the languages corresponds to choosing a subset of the transitions of T, which there are finitely many. In the equivalence class in our example, the transitions $(t_1,3)$ and $(t_2,2)$ are in the same equivalence class. Therefore, the language starting with transition $(t_1,3)$ is $(cd)^*$.

Given a input u, a position $i \in \mathrm{dom}(u)$ can be annotated with two sets of transitions as follows. The first set corresponds to transitions that are part of some vertical loop at i, i.e, $(t,i) \sim_u (t,i)$ and the second set consists of transitions t such that $(t,i) \not\sim_u (t,i)$. Furthermore, for every transition t in the first set, the annotation also includes an NFA representing the language $L_{t,i}$. The correctness of the annotation of the sets can be easily checked. The partition of transitions into sets can be checked by a 2NFA that checks whether the transition is part of a vertical loop or not. The annotation of the language $L_{t,i}$ can in fact be built on-the-fly by checking the other (at most k) productive transitions in the equivalence class of (t,i) and building the NFA defined earlier. Therefore, using common guess in $low(T)$, we can assume that every position i carries as annotation the language $L_{t,i}$ for each transition t such that (t,i) is part of some vertical loop.

Shortcutting runs. Consider an arbitrary successful run ρ of T on u. Let $low(\rho)$ be the run obtained by replacing, from left to right, every maximal vertical loop at (t,i), where t starts from state q by the two transitions $t_1 = (q, u(i), \epsilon, p)$ and $t_2 = (p, u(i-1), L, q')$, where $L = L_{t,i}$ and q' is a copy of the state q. Here, maximality refers to the sub-run relation, i.e, the vertical loop cannot be extended to a bigger sub-run which is also a vertical loop. We require two transitions t_1 and t_2 because it needs to start and end in the same configuration which is not possible by a single transition. In our example, we do this by having two transitions (q_1,p_1) and (p_1,q_1'). Note that this introduces an intermediate state for every transition t, and a copy of every state of T. We call $low(\rho)$ the *normalization of* ρ and we observe that this is a successful, $2|Q| + |\Delta|$-visit run. This means that

— $low(\rho)$ can be finitely encoded on the input as a sequence of flows of height at most $2|Q| + |\Delta|$,

— the language consisting of inputs annotated with such encodings is

regular.

The 2NFT $low(T)$ guesses the encoding of a normalization $low(\rho)$ and uses it to simulate a possible run ρ of T. Recall that the annotations contain the information about whether a transition is a part of a vertical loop or not. For transitions that are not part of a vertical loop, $low(T)$ behaves exactly as before. However, every time $low(T)$ traverses a transition t starting from state q, that is part of a vertical loop, it outputs a word from the language $L_{t,i}$, moves to an intermediate state and comes back to a copy of q. Intuitively, the first copy of q allows to produce outputs that are part of a vertical loop and the second copy is used to simulate the part of the run without a vertical loop. However, in order to simplify later the construction of a resynchronizer R such that $R(T) =_o low(T)$, it is convenient that $low(T)$ outputs the word from $L_{t,i}$ in a origin possibly different from i. This new origin is uniquely determined by the $\sim_u$-equivalence class of (t, i).

Formally, we define the *anchor* of a $\sim_u$-equivalence class C, denoted $an(C)$, to be the leftmost input position j such that $(t', j) \in C$ for some transition t'. In our example (Figure 5.6), we have $an(C) = 2$ for the equivalence class $\{(t_1, 3), (t_2, 2)\}$.

After traversing a transition t from the flow at position i, and before outputting a word from $L_{t,i}$, the 2NFT $low(T)$ moves to the anchor $an([(t, i)]_{\sim_u})$. Then it outputs the appropriate word and moves back to position i, where it can resume the simulation of the normalized run $low(\rho)$. Note that the position i can be recovered from the anchor $an([(t, i)]_{\sim_u})$ since the elements inside the equivalence class $[(t, i)]_{\sim_u}$ can be identified by numbers from $\{1, \ldots, k\}$ (recall that T is k-sparse), and since the relationship between any two such elements is a regular property. Note that this will introduce several new states to move to the anchor and return to the original position. However, the 2NFT $low(T)$ still remains bounded-visit. It follows from the definitions that $low(T)$ is classically equivalent to T.

Building a resynchronizer. We now explain how to construct a functional, 1-bounded, regular resynchronizer R that is T-preserving and such that $R(T) =_o low(T)$. We proceed as in the construction of $low(T)$ by annotating the input word u with flows that encode the normalization $low(\rho)$ of a successful run ρ of T on u. Note that it is possible to define the flow of the normalization $low(\rho)$, since it is bounded-visit. As for the output word v, we annotate every position x of v with the productive transition $t = (q, a, v, q')$ of ρ that generated x. For short, we call t *the transition of* x. In addition, we fix an MSO-definable total ordering on tagged transitions (e.g. the lexicographic ordering). Then, we determine from each output position x the $\sim_u$-equivalence class $C = [(t, i)]_{\sim_u}$, where u is the underlying input, t is the productive transition that generated x, and i is its origin. We extend the annotation of x with the index of the element (t, i) inside the equivalence

class C, according to the fixed total ordering on tagged transitions. This number i is called *the index of* x. This is possible because there are at most k such transitions due to T being k-sparse.

The resynchronizer R needs to redirect the source origin i of any output position generated by a transition t to a target origin j that is the anchor of the $\sim_u$-equivalence class of (t, i). To simplify the explanation, we temporarily assume that the input and output are correctly annotated as described above. By inspecting the type τ of an output position x, the formula $\mathsf{move}_\tau(y, z)$ of R can determine the transition t of x, and enforce that $(t, y) \sim_u (t', z)$, for some transition t', and that $(t, y) \not\sim_u (t'', z')$, for all $z' < z$ and all transitions t''.

Under the assumption that the input and output annotations are correct, this would result in a bounded resynchronizer R. Indeed, for every position z, there exist at most $k \cdot |\Delta|$ positions y that, paired with some productive transition, turn out to be $\sim_u$-equivalent to (t', z) for some transition t'. In fact, we need to further constrain the relation $\mathsf{move}_\tau(y, z)$ so that it describes a partial bijection between source and target origins (this will be used later, similar to the bounded-visit case). For this, it suffices to additionally enforce that (t, y) is the ℓ-th element in its $\sim_u$-equivalence class, accordingly to the fixed total ordering on tagged transitions, where ℓ is the *index* specified in the output type τ of x. As a matter of fact, this latter modification also guarantees that ℓ is the correct index of x.

The above arguments crucially rely on the assumption that the input and output annotations are correct. However, we can apply the same trick that we used in the proof of Theorem 5.3.1, to make the resynchronizer R "syntactically" 1-bounded, even in the presence of badly-formed annotations. Formally, let $\mathsf{move}_\tau(y, z)$ be the formula that transforms the origins in the way described above, and define

$$\mathsf{move}'_\tau(y, z) \;=\; \mathsf{move}_\tau(y, z) \;\wedge\; \forall y' \,\big(\mathsf{move}_\tau(y', z) \to y' = y\big).$$

By construction, the above formula defines a partial bijection entailing the old relation move_τ (in the worst case, when the annotations are not correct, the above formula may not hold for some pairs of source and target origins). In addition, if the annotations are correct, then $\mathsf{move}'_\tau(y, z)$ is semantically equivalent to $\mathsf{move}_\tau(y, z)$, as desired. In this way, we obtain a regular resynchronizer $R = (\overline{I}, \overline{O}, \mathsf{ipar}, \mathsf{opar}, \mathsf{move}'_\tau, \mathsf{next})$ that is always 1-bounded, no matter how we define ipar, opar, and next.

Checking annotations. We now explain how to check that the annotations are correct. The input annotation does not pose any particular problem, since the language of inputs annotated with normalized runs is regular, and can be checked using the first formula ipar of the resynchronizer. As for the output annotation, correctness of the indices was already enforced by

the move'_τ relation. It remains to enforce correctness of the transitions. As before, this boils down to verifying the following property (†):

> *Let ρ be the run encoded on the input. For every pair of consecutive output positions $x, x+1$ with source origins y, y', respectively, if t, t' are the productive transitions specified in the output types of $x, x+1$, one can move from transition t at position y of ρ to transition t' at position y' of ρ by using as intermediate edges only non-productive transitions.* (†)

From here we proceed exactly as in the proof of Theorem 5.3.1. Observe that Property (†) is expressible by an MSO formula $\varphi^{\dagger}_{\tau,\tau'}(y, y')$, assuming that τ, τ' are the output types of $x, x+1$, that y, y' are interpreted by the source origins of $x, x+1$, and that x ranges over all output positions. We then recall that $\mathsf{move}_\tau(y, z)$ and $\mathsf{move}_\tau(y', z')$ describe partial bijections between source and target origins, and exploit this enforce (†) by means of the last formula of R:

$$\mathsf{next}_{\tau,\tau'}(z, z') \;=\; \exists y, y'\ \mathsf{move}_\tau(y, z) \;\wedge\; \mathsf{move}_{\tau'}(y', z') \;\wedge\; \varphi^{\dagger}_{\tau,\tau'}(y, y').$$

This guarantees that all annotations are correct, and proves that R is a functional, 1-bounded, regular resynchronizer satisfying $R(T) =_o \mathit{low}(T)$. It is also immediate to see that R is T-preserving.

Checking one-way resynchronizability. We finally prove that one-way resynchronizability of T reduces to one-way resynchronizability of $\mathit{low}(T)$, which can be effectively tested using Theorem 5.3.1 since $\mathit{low}(T)$ is bounded-visit.

Lemma 5.4.2. *For all 2NFTs T, T', with T' bounded-visit, and for every functional, 1-bounded, regular resynchronizer R that is T-preserving and such that $R(T) =_o T'$, T is one-way resynchronizable if and only if T' is one-way resynchronizable.*

Proof. For the right-to-left implication, suppose that $T' =_o R(T)$ is bounded-visit and one-way resynchronizable. Since T' is bounded-visit, we can use the condition 4. in Theorem 5.3.1 to get the existence of a functional, 1-bounded, T'-preserving, regular resynchronizer R' that is T'-preserving and such that $R'(T')$ is order-preserving. By Lemma 3.3.5, there is a bounded, regular resynchronizer R'' that is equivalent to $R' \circ R$. In particular, $R''(T)$ is order-preserving. It remains to verify that R'' is also T-preserving. Consider any synchronized pair $(u, v) \in [\![T]\!]_o$. Since R is T-preserving, (u, v) belongs to the domain of R', and hence $((u, v), (u, v')) \in R$ for some synchronized pair $(u, v') \in [\![T']\!]_o$. Since R is T'-preserving, (u, v') belongs to the domain of R, and hence there is $((u, v), (u, v'')) \in (R' \circ R) =$

R''. This shows that R'' is T-preserving, and hence T is one-way resynchronizable.

For the converse direction, suppose that T' is bounded-visit, but not one-way resynchronizable. We apply again Theorem 5.3.1, and obtain that T' has unbounded cross-width

We also recall that $R = (\overline{I}, \overline{O}, \mathsf{ipar}, (\mathsf{move}_\tau)_\tau, (\mathsf{next}_{\tau,\tau'})_{\tau,\tau})$ is functional and 1-bounded. In particular, this means that every formula $\mathsf{move}_\tau(y, z)$ defines a partial bijection from source to target positions. A useful property of every MSO-definable partial bijection is that, for every position t, it can only define boundedly many pairs (y, z) with either $y \leq t < z$ or $z \leq t < y$ for short, we say call such a pair (y, z) *t-separated*. This follows from compositional properties of regular languages. Indeed, let $\mathcal{A}$ be a deterministic automaton equivalent to the formula that defines the partial bijection. For every pair (y, z) in the partial bijection, let $q_{y,z}$ be the state visited at position t by the successful run of $\mathcal{A}$ on the input annotated with the pair (y, z). If $\mathcal{A}$ accepted more than $|Q|$ pairs that are t-separated, where Q is the state space of $\mathcal{A}$, then at least two of them, say (y, z) and (y', z'), would satisfy $q_{y,z} = q_{y',z'}$. But this would imply that the pair (y, z') is also accepted by $\mathcal{A}$, which contradicts the assumption that $\mathcal{A}$ defines a partial bijection. Note that for this proof, we require the resynchronizer to be functional, in order to get a partial bijection from move_τ.

We now exploit the above result to prove that the property of having unbounded cross-width transfers from T' to T. Consider a cross (X_1, X_2) of arbitrarily large width h in some synchronized pair $\sigma = (u, v)$ of T'. Without loss of generality, assume that all positions in $X_1 \cup X_2$ have the same type τ. Let $Z_i = \mathit{orig}(X_i)$, for $i = 1, 2$, and $t = \max(Z_2)$. By definition of cross, we have $X_1 < X_2$ and $Z_2 \leq t < Z_1$. Here, we write $X_1 < X_2$ to denote for every $x_1 \in X_1$ and $x_2 \in X_2$, $x_1 < x_2$, and $Z_2 \leq t < Z_1$ is defined similarly.

Recall that move_τ defines a partial bijection, and that this implies that there are only boundedly many pairs of source-target origins that are t-separated, say $(y_1, z_1), \ldots, (y_k, z_k)$ for a constant k that only depends on R. Moreover, since $R(T) =_o T'$, the positions in Z_i can be seen as target origins of the formula move_τ of R. Now, let $X_i' = X_i \setminus \mathit{orig}^{-1}(\{z_1, \ldots, z_k\})$, where $\mathit{orig}^{-1}(\{z_1, \ldots, z_k\})$ denotes the set of output positions with target origin z_j, and $Y_i' = \mathit{orig}'(X_i')$, for any synchronized pair $\sigma' = (u, v')$ such that $(\sigma, \sigma') \in R$.

By construction, we have $X_1' < X_2'$ and $Y_2' \leq t < Y_1'$ (the latter condition follows from the fact that the source origins from Y_i' can only be moved to target origins on the same side w.r.t. t). This means that (X_1', X_2') is a cross of width $h - k$. As h can be taken arbitrarily large and k is constant, this proves that T has unbounded cross-width as well.

Finally, by the contrapositive of the implication 1. to 2. of Theorem 5.3.1, we conclude that T is not one-way resynchronizable. □

The algorithm and complexity. Summing up, the algorithm that decides whether a given 2NFT T is one-way resynchronizable first verifies that T is k-sparse for some k. If so, the algorithm constructs a bounded-visit 2NFT $low(T)$ equivalent to T, and finally decides whether $low(T)$ is one-way resynchronizable. This concludes the proof of Theorem 5.1.6.

To recall, we summarize the steps of the algorithm.

1. As a first step, one has to check if T has a bounded sparsity. To do this, we annotate a possible input u of T with a distinguished $\sim_u$-equivalence class, and crossing sequences at positions which belong to this particular $\sim_u$-equivalence class. Note that we can assume the configuration only repeats at the anchor (leftmost) position and therefore the crossing sequences are bounded. Checking unbounded sparsity reduces to detecting the presence of a loop with respect to the annotated crossing sequence which contains a productive transition. This is a witness for unbounded sparsity as by pumping such a loop, one can obtain an input on which T is k-sparse, for any k. Standard techniques for two-way automata allow to decide presence of such loops in space that is polynomial in the size of T. Moreover, this also gives us a computable exponential bound to the largest constant k for which T can be k-sparse for T to have bounded sparsity. The exponential bound comes from the fact that the size of the crossing sequences are exponential in size of T.
2. Next, we construct from the k-sparse T, bounded-visit 2NFT $low(T)$ that is classically equivalent to T. A close inspection to the construction of $low(T)$ shows that, this can be done in space that is exponential in the size of T. Note that the construction of $low(T)$ introduces $|Q|+|\Delta|$ many new states. The $|\Delta|$ new states are needed to produce $L_{t,i}$ for some tagged transition in a $\sim_u$-equivalence class. However, the output languages $L_{t,i}$ introduced in $low(T)$ can have size polynomial in k, where k is a bound on the sparsity. Since k is exponential is T, $low(T)$ can have size exponential in size of T.
3. Finally, one has to decide one-way resynchronizability of $low(T)$ by detecting inversions in successful runs of $low(T)$. Since $low(T)$ is bounded-visit and has size exponential in the size of T, we obtain an upper bound of ExpSpace.

This concludes the proof of Theorem 5.1.6. The best lower bound we have is PSpace, same as in the bounded-visit case. Therefore, it is an open question if the upper bound of ExpSpace is optimal or not.

5.5 Conclusions

The one-way definability problem happens to be straightforward in origin semantics due to the characterization as order-preserving transductions

(cross-width 0) due to [Boj14]. In the classical semantics, the problem is decidable with 2-ExpSpace complexity for functional 2NFTs thanks to [BGMP18].

We showed that the problem of one-way resynchronizability is decidable in PSpace for bounded-visit 2NFTs (which contains the class of functional 2NFTs). In fact, the characterization of T having no inversions is also similar to the one of Theorem 3.6 from [BGMP18] for one-way definability in the classical semantics. The main difference is that in the classical semantics, one also needs to check some combinatorial properties of the inversions, which we are able to avoid in our result.

For the bounded-visit case, the bounded cross-width property generalizes the notion of order-preserving transductions, which corresponds to the case of cross-width 0. For the general case, every one-way resynchronizable 2NFT T has the bounded cross-width property. Whether the converse holds or not is not known. One approach to solve this would be to build a family of resynchronizers R_k, for every k, such whenever T has cross-width bounded by k, $R_k(T)$ is order preserving. In other words, R_k would undo crosses of width k.

Another point worth mentioning is that in our constructions, we always build a functional resynchronizer that defines a partial bijection between source and target origins. This shows that whenever T is one-way resynchronizable, it is resynchronizable using a functional resynchronizer. Lemma 5.4.2 shows that the class of one-way resynchronizable 2NFTs are closed under application of functional resynchronizer.

In both the bounded-visit and the general case, we give upper bounds of PSpace and ExpSpace for the one-way resynchronizability problem. We also give a lower bound of PSpace by a reduction from the emptiness problem for 2DFA.

Possible extensions to NSSTs. One of the motivation for studying the one-way resynchronizability problem is the connection to streamability. A NFT can produce the output without having to store the input, which is interesting for many applications. A 2NFT, on the other hand, has to store the entire input to produce the output. Therefore, one-way resynchronizability identifies classes of transductions which can be preprocessed to obtain streamability. In this case, the preprocessing step is the application of the appropriate resynchronizer.

The class of NSSTs was also introduced with a similar motivation of streamability. NSSTs process the input one letter at a time, but use registers to store parts of the output that are later combined to obtain the output. A NFT can be seen as a NSST with a single register that is allowed to append only to the right. Therefore, a generalization of the one-way resynchronizability problem would be to check whether a given 2NFT be

resynchronized to k-register NSST.

Other definability problems. Finally, there are analogous definability problems that emerge in the origin semantics. For instance, one could ask whether a given 2NFT can be resynchronized, through some bounded, regular resynchronization, to a relation that is origin-equivalent to a *first-order logical transduction*. A first-order transduction is a model of logical transduction, that is a restriction of the MSO-transduction defined by Courcelle[Cou94]. This can be seen as a relaxation of the first-order definability problem in the origin semantics. The first-order definability problem, i.e, checking whether a 2NFT is origin-equivalent to some first-order transduction was shown to be decidable for functional transductions in [Boj14]. In the classical semantics, the first-order definability problem is open for 2NFTs and NSSTs. A restriction of NSSTs, called aperiodic DSSTs, capturing first-order definable functional NSSTs was given in [FKT14], but checking whether a given NSST is equivalent to an aperiodic one remains open.

Bibliography

[AC10] Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. In Kamal Lodaya and Meena Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–12, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[AC11] Rajeev Alur and Pavol Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 599–610, New York, NY, USA, 2011. Association for Computing Machinery.

[AD11] Rajeev Alur and Jyotirmoy V. Deshmukh. Nondeterministic streaming string transducers. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming*, pages 1–20, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[ADD+13] R. Alur, L. DAntoni, J. Deshmukh, M. Raghothaman, and Y. Yuan. Regular functions and cost register automata. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 13–22, 2013.

[AFR14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[AFT12] Rajeev Alur, Emmanuel Filiot, and Ashutosh Trivedi. Regular transformations of infinite strings. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, page 65–74, USA, 2012. IEEE Computer Society.

[AL85] M.H. Albert and J. Lawrence. A proof of ehrenfeucht's conjecture. *Theoretical Computer Science*, 41:121 – 123, 1985.

[BCPS00] Marie-Pierre Béal, Olivier Carton, Christophe Prieur, and Jacques Sakarovitch. Squaring transducers: An efficient procedure for deciding functionality and sequentiality of transducers. In Gaston H. Gonnet and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics*, pages 397–406, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[BCPS03] Marie-Pierre Béal, Olivier Carton, Christophe Prieur, and Jacques Sakarovitch. Squaring transducers: an efficient procedure for deciding functionality and sequentiality. *Theor. Comput. Sci.*, 292:45–63, 2003.

[BDGP17] Mikolaj Bojańczyk, Laure Daviaud, Bruno Guillon, and Vincent Penelle. Which Classes of Origin Graphs Are Generated by Transducers. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 114:1–114:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BDK18] Mikolaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. Regular and first-order list functions. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 125–134, New York, NY, USA, 2018. Association for Computing Machinery.

[BDSW17] Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell. Polynomial automata: Zeroness and applications. In *Proceedings of the Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2017.

[BGMP18] Félix Baschenis, Olivier Gauwin, Anca Muscholl, and Gabriele Puppis. One-way definability of two-way word transducers. *Logical Methods in Computer Science*, Volume 14, Issue 4, December 2018.

[BH77] Meera Blattner and Tom Head. Single-valued a-transducers. *J. Comput. and System Sci.*, 15:310–327, 1977.

[BH79] Meera Blattner and Tom Head. The decidability of equivalence for deterministic finite transducers. *Journal of Computer and System Sciences*, 19(1):45 – 49, 1979.

[Bir73] Malcolm Bird. The equivalence problem for deterministic two-tape automata. *J. Comput. Syst. Sci.*, 7(2):218–236, April 1973.

[BKM+19] Sougata Bose, Shankara Narayanan Krishna, Anca Muscholl, Vincent Penelle, and Gabriele Puppis. On Synthesis of Resyn-

chronizers for Transducers. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*, volume 138 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 69:1–69:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BKMP21] Sougata Bose, S. N. Krishna, Anca Muscholl, and Gabriele Puppis. One-way resynchronizability of word transducers, 2021.

[BMPP18] Sougata Bose, Anca Muscholl, Vincent Penelle, and Gabriele Puppis. Origin-equivalence of two-way word transducers is in PSPACE. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'18)*, volume 122 of *LIPIcs*, pages 1–18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[Boj14] Mikolaj Bojańczyk. Transducers with origin information. In *International Colloquium on Automata, Languages and Programming (ICALP'14)*, number 8572 in LNCS, pages 26–37. Springer, 2014.

[Bol16] B. Bollig. One-counter automata with counter observability. In *FSTTCS*, 2016.

[Buc60] J Richard Buchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[CD15] Olivier Carton and Luc Dartois. Aperiodic Two-way Transducers and FO-Transductions. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*, volume 41 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 160–174, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[CFM13] Michaël Cadilhac, Alain Finkel, and Pierre McKenzie. Unambiguous constrained automata. *Int. J. Found. Comput. Sci.*, 24(7):1099–1116, 2013.

[Cho78] Christian Choffrut. *Contribution à l'étude de quelques familles remarquables de fonctions rationnelles.* PhD thesis, 1978.

[Cho03] Christian Choffrut. Minimizing subsequential transducers: a survey. *Theoretical Computer Science*, 292(1):131 – 143, 2003. Selected Papers in honor of Jean Berstel.

[CK86] Karel Culik II and Juhani Karhumäki. The equivalence of finite valued transducers (on HDT0L languages) is decidable. *Theor. Comput. Sci.*, 47:71–84, 1986.

[CK87] Karel Culik II and Juhani Karhumäki. The equivalence problem for single-valued two-way transducers (on NPDT0L languages) is decidable. *SIAM J. Comput*, 16(2):221–230, 1987.

[Col07] Thomas Colcombet. Factorisation forests for infinite words. In Erzsébet Csuhaj-Varjú and Zoltán Ésik, editors, *Fundamentals of Computation Theory*, pages 226–237, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Cou94] Bruno Courcelle. Monadic second-order definable graph transductions: a survey. *Theoretical Computer Science*, 126(1):53 – 75, 1994.

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Transformations: Foundations*, volume 1, pages 165–254. World Scientific, 1997.

[DFL18] Luc Dartois, Emmanuel Filiot, and Nathan Lhote. Logics for word transductions with synthesis. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 295–304, New York, NY, USA, 2018. Association for Computing Machinery.

[DGH16] Antoine Durand-Gasselin and Peter Habermehl. Regular transformations of data words through origin information. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 285–300, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[EH01] Joost Engelfriet and Hendrik Jan Hoogeboom. Mso definable string transductions and two-way finite-state transducers. *ACM Trans. Comput. Logic*, 2(2):216–254, April 2001.

[Eil74] Samuel Eilenberg. *Automata, Languages, and Machines.* Academic Press, Inc., USA, 1974.

[EM65] Calvin C. Elgot and Jorge E. Mezei. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9(1):47–68, 1965.

[Eng97] Joost Engelfriet. Context-free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 125–213. Springer, 1997.

[FG82] E. P. Friedman and S. Greibach. A polynomial time algorithm for deciding the equivalence problem for 2-tape deterministic finite state acceptors. *SIAM J. Comput.*, 11:166–183, 1982.

[FGL16] Emmanuel Filiot, Olivier Gauwin, and Nathan Lhote. Aperiodicity of Rational Functions Is PSPACE-Complete. In Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen, editors, *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016)*, volume 65 of *Leibniz International Proceedings in Informatics*

(LIPIcs), pages 13:1–13:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[FGRS13] Emmanuel Filiot, Olivier Gauwin, Pierre-Alain Reynier, and Frederic Servais. From two-way to one-way finite state transducers. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, page 468–477, USA, 2013. IEEE Computer Society.

[FJLW16] Emmanuel Filiot, Ismaël Jecker, Christof Löding, and Sarah Winter. On equivalence and uniformisation problems for finite transducers. In *ICALP'16*, volume 55 of *LIPIcs*, pages 125:1–125:14, 2016.

[FKT14] Emmanuel Filiot, Shankara Narayanan Krishna, and Ashutosh Trivedi. First-order Definable String Transformations. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 147–159, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[FMR18] Emmanuel Filiot, Nicolas Mazzocchi, and Jean-François Raskin. A pattern logic for automata with outputs. In Mizuho Hoshi and Shinnosuke Seki, editors, *Developments in Language Theory*, pages 304–317, Cham, 2018. Springer International Publishing.

[FMRT15] Emmanuel Filiot, Sebastian Maneth, Pierre-Alain Reynier, and Jean-Marc Talbot. Decision problems of tree transducers with origin. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming*, pages 209–221, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[FR68] Patrick C. Fischer and Arnold L. Rosenberg. Multi-tape one-way nonwriting automata. *J. Comput. and System Sci.*, 2:88–101, 1968.

[FR16] Emmanuel Filiot and Pierre-Alain Reynier. Transducers, logic and algebra for functions of finite words. *ACM SIGLOG News*, 3(3):4–19, August 2016.

[FR17] Emmanuel Filiot and Pierre-Alain Reynier. Copyful streaming string transducers. In Matthew Hague and Igor Potapov, editors, *Reachability Problems*, pages 75–86, Cham, 2017. Springer International Publishing.

[Gas19] Paul Gastin. Modular descriptions of regular functions. In Miroslav Ćirić, Manfred Droste, and Jean-Éric Pin, editors, *Al-*

gebraic Informatics, pages 3–9, Cham, 2019. Springer International Publishing.

[GI83] E. Gurari and O. Ibarra. A note on finite-valued and finitely ambiguous transducers. *Mathematical systems theory*, 16:61–66, 1983.

[Gri68] T. V. Griffiths. The unsolvability of the equivalence problem for lambda-free nondeterministic generalized machines. *J. ACM*, 15(3):409–413, 1968.

[Gub86] Victor S. Guba. Equivalence of infinite systems of equations in free groups and semigroups to finite subsystems. *Mathematical notes of the Academy of Sciences of the USSR*, 40:688–690, 1986.

[Gur82] Eitan M. Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. *SIAM Journal of Computing*, 448–452, 1982.

[Iba78] Oscar H. Ibarra. The unsolvability of the equivalence problem for e-free NGSM's with unary input (output) alphabet and applications. *SIAM J. of Comput.*, 7(4):524–532, 1978.

[Jec21] Ismaël Jecker. A ramsey theorem for finite monoids, 2021.

[KM20] Denis Kuperberg and Jan Martens. Regular Resynchronizability of Origin Transducers Is Undecidable. In Javier Esparza and Daniel Král, editors, *45th International Symposium on Mathematical Foundations of Computer Science (MFCS 2020)*, volume 170 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 58:1–58:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Koz77] Dexter Kozen. Lower bounds for natural proof systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 254–266, USA, 1977. IEEE Computer Society.

[Lot02] M. Lothaire. *Makanin's Algorithm*, page 387–442. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2002.

[Moo56] Edward F. Moore. *Gedanken-Experiments on Sequential Machines*, pages 129 – 154. Princeton University Press, Princeton, 31 Dec. 1956.

[MOS04] Markus Müller-Olm and Helmut Seidl. A note on karr's algorithm. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming*, pages 1016–1028. Springer Berlin Heidelberg, 2004.

[MP19a] Anca Muscholl and Gabriele Puppis. Equivalence of Finite-Valued Streaming String Transducers Is Decidable. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano

Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 122:1–122:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[MP19b] Anca Muscholl and Gabriele Puppis. The many facets of string transducers (invited talk). In *36th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 126 of *LIPIcs*, pages 2:1–2:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[Niv68] Maurice Nivat. Transduction des langages de Chomsky. *Annales de l'Institut Fourier*, 18:339–455, 1968.

[Pra20] M. Praveen. What You Must Remember When Transforming Datawords. In Nitin Saxena and Sunil Simon, editors, *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2020)*, volume 182 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 55:1–55:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959.

[Sch61] M.-P. Schützenberger. A remark on finite transducers. *Information and Control*, 4(2-3):185–196, 1961.

[Sch75] Marcel-Paul Schützenberger. Sur les relations rationnelles. In *Proc. of 2nd GI conference, Automata Theory and Formal Languages*, number 33 in LNCS, pages 209–213. Springer, 1975.

[SdS10] Jacques Sakarovitch and Rodrigo de Souza. Lexicographic decomposition of ¡i¿k¡/i¿-valued transducers. *Theor. Comp. Sys.*, 47(3):758–785, October 2010.

[She59] J. C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3(2):198–200, 1959.

[SHI85] R. E. Stearns and H. B. Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3):598–611, 1985.

[Sim90] Imre Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65 – 94, 1990.

[Tho97] Wolfgang Thomas. *Languages, Automata, and Logic*, page 389–455. Springer-Verlag, Berlin, Heidelberg, 1997.

[Var89] Moshe Y. Vardi. A note on the reduction of two-way automata to one-way automata. *Information Processing Letters*, 30(5):261–264, 1989.

[vDKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *J. Symb. Comput.*, 15(5–6):523–545, May 1993.

[Web96] Andreas Weber. Decomposing a k-valued transducer into k unambiguous ones. *ITA*, 30(5):379–413, 1996.

[WS91] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theor. Comput. Sci.*, 88(2):325–349, 1991.