Efficient Representation and Manipulation of Automata on Linear Orderings

Auteur : Clara, Tom
Promoteur(s) : Boigelot, Bernard
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en science des données, à finalité spécialisée
Année académique : 2024-2025
URI/URL : http://hdl.handle.net/2268.2/23281

UNIVERSITÉ DE LIÈGE
FACULTÉ DES SCIENCES APPLIQUÉES

Efficient Representation and Manipulation of Automata on Linear Orderings

Clara Tom

Thesis presented to obtain the degree of :
Master of Science in Data Science and Engineering

Thesis supervisor :
Boigelot Bernard

Academic year: **2024 - 2025**

Acknowledgments

I would like to thank my advisor, Bernard Boigelot, for all his advice, his valuable feedbacks and his unwavering support. In particular, I am very grateful for the numerous video conferences we had together when I was enjoying the sunny weather in Nice and he had to endure the rainy weather in Liège. Moreover, I was very happy to work with my friend Thomas Braipson throughout this year. Without the 473 hours of discussion we had on automata, I would never have understood them as deeply. Hopefully, we will keep working as a team in the next years, and I am looking forward to tackling new problems with him and Bernard Boigelot! Finally, I want to thank Pascal Fontaine and Michel Rigo, who accepted to join the jury of this master's thesis.

Abstract

Automata on linear orderings have been introduced by Bruyère and Carton as a broad generalization of finite-word, infinite-word and transfinite-word automata. Even though they were first seen as purely mathematical objects, automata on linear orderings can in fact serve as algorithmic tools for decision procedures, e.g., the monadic first-order theory of order on $\mathbb{R}$ or $\mathbb{Q}$. In this context, efficient algorithms and data structures need to be carefully designed to implement and then manipulate these automata. This is the problem addressed in this work. We first show that the most crucial operation to optimize is the product of automata on linear orderings, which consists in computing an automaton that simulates the joint behaviour of several automata. From a theoretical point of view, this problem is intrinsically hard. Nevertheless, we investigate several strategies to implement it efficiently in most practical cases, including in the context of decision procedures. These strategies include both theoretical and practical contributions, that range from the definition of an efficient representation mechanism for automata on linear orderings to an actual implementation, in the C programming language, of these automata and their product operation. We then demonstrate experimentally that it is indeed possible to efficiently manipulate automata on linear orderings algorithmically. This paves the way for an actual decision procedure for the monadic first-order theory of order on $\mathbb{R}$ or $\mathbb{Q}$, which could then lead to applications in verification and SMT solving. In parallel, we extended (together with Thomas Braipson and Bernard Boigelot) the formalism of automata on linear orderings, in order to allow their transitions to be labelled by the empty word. The resulting automata model, i.e., epsilon automata on linear orderings, is a powerful theoretical tool that could ease the developments of new methods involving automata on linear orderings.

Résumé

Les automates sur ordres linéaires ont été définis par Bruyère et Carton comme une généralisation importante des automates sur mots finis, infinis et transfinis. Bien que ces automates aient d'abord été considérés comme des objets purement mathématiques, ils peuvent en réalité servir d'outils algorithmiques dans le cadre de procédures de décision (par exemple, pour décider la théorie monadique de l'ordre, du premier ordre, sur $\mathbb{R}$ ou $\mathbb{Q}$). Dans ce contexte, il est nécessaire de concevoir des algorithmes et structures de données efficaces permettant d'implémenter puis de manipuler ces automates sur ordres linéaires. Nous abordons ce problème dans ce travail. D'abord, nous montrons que l'opération la plus importante à optimiser est le produit d'automates sur ordres linéaires, qui consiste à calculer un automate simulant le comportement conjoint et simultané de plusieurs autres automates. D'un point de vue théorique, ce problème est intrinsèquement difficile. Néanmoins, nous explorons plusieurs stratégies visant à le résoudre efficacement dans la plupart des cas pratiques, en particulier dans le contexte de procédures de décision. Ces stratégies incluent à la fois des contributions théoriques et pratiques, allant de la définition d'un mécanisme de représentation efficace des automates sur ordres à l'implémentation pratique, en C, de ces automates et de l'opération consistant à calculer leur produit. Ensuite, nous démontrons, preuves expérimentales à l'appui, qu'il est bel et bien possible de manipuler efficacement ces automates en tant qu'outils algorithmiques. Cela ouvre la porte à une implémentation pratique d'une procédure de décision pour la théorie monadique de l'ordre, du premier ordre, sur $\mathbb{R}$ ou $\mathbb{Q}$. Une telle procédure de décision pourrait ensuite déboucher sur des applications en vérification, ou étendre les possibilités des solveurs SMT. En parallèle, nous étendons (en collaboration avec Thomas Braipson et Bernard Boigelot) le formalisme des automates sur ordres linéaires, en permettant la présence de transitions sur le mot vide. Ceci mène à un nouveau type d'automates appelés *automates epsilon*, qui sont un outil théorique puissant permettant de simplifier le développement de nouvelles méthodes basées sur les automates sur ordres linéaires.

Contents

Introduction

Automata are expressive mathematical objects that can serve as powerful theoretical tools. Historically, they have already been used successfully for establishing various decidability results. For instance, *Büchi automata* have been introduced to demonstrate the decidability of the monadic second-order theory of one successor (S1S) [9]. Presburger arithmetic, i.e., the first-order additive arithmetic of integer numbers, can also be decided by building a finite automaton recognizing the models of a formula, and then by checking the non-emptiness of its accepted language [8].

Over the years, it became clear that automata can also be used as practical data structures to build effective decision procedures. Their good algorithmic properties make it possible to decide, for instance, temporal logics [19], which are of practical interest in the field of *computer-aided verification* [11]. The long-range goal of the research team in which the master's thesis was carried out is to develop a similar approach for the monadic first-order theory of order on $\mathbb{R}$ or $\mathbb{Q}$. Besides verification, a decision procedure for this logic could enrich the capabilities of modern *Satisfiability Modulo Theories (SMT)* solving tools [1].

Building a decision procedure for such an expressive logic fragment requires the use of a particular form of automata called *automata on linear orderings*. They have been introduced as purely mathematical objects in [7] and can be considered as a broad generalization of finite-word, infinite-word and transfinite-word automata. The first step towards the implementation of the decision procedure consists in developing tools and strategies for the representation, the implementation and the manipulation of these automata. This master's thesis tackles some of these problems, both at the theoretical and at the practical levels.

Another problem that must be solved is the design of a non-emptiness test for automata on linear orderings, on $\mathbb{R}$ or $\mathbb{Q}$. This question is addressed in the master's thesis of Thomas Braipson [6]. Since we both worked on automata on linear orderings, we wrote the first chapter of this manuscript together, as well as Appendix A. Furthermore, Chapter 2 presents the content of an article co-written with Thomas Braipson and Bernard Boigelot (see [3]).

In Chapter 1, we recall the notion of *linear ordering* and the concept of *word indexed by a linear ordering*. Then, we restate the syntax and the semantics of automata on linear orderings, as defined in [7]. These automata can be seen as finite-word automata augmented with *limit transitions*, that make it possible to read words indexed by arbitrary linear orderings.

In Chapter 2, we introduce *epsilon automata on linear orderings*, that generalize plain automata by allowing epsilon transitions. These transitions are powerful tools that can significantly simplify some proofs and developments involving automata on linear orderings. We illustrate their usefulness by correcting a small mistake found in [2]. Our contribution is theoretical, but can be considered as a step towards a broader use of automata on linear orderings in other (maybe more practical) contexts.

Next, in Chapter 3, we briefly recall the details of the decision procedure for the first-order theory of order and motivate the need for an efficient product algorithm for automata on linear orderings. The product of two automata is, intuitively, another automaton that simulates their joint behaviour. The product of automata on linear orderings is a difficult yet crucial operation, since it serves a building block for the intersection and the union operations, that translate the Boolean connectives $\wedge$ and $\vee$ of our logic fragment. In this chapter, we show that handling efficiently automata on linear orderings essentially amounts to designing an algorithm that computes their product efficiently (for most of the problem instances). We then demonstrate that a naive product algorithm does not work in practice, because it generates a prohibitively large number of limit transitions.

In Chapter 4, we start elaborating theoretical strategies to implement the product operation efficiently. We first investigate the idea of deleting some *useless* limit transitions. Then, we explore the possibility of representing automata on linear orderings more efficiently, thanks to *finite history automata*.

Chapter 5 describes formally and exhaustively algorithms and data structures (related to the product operation) that implement the principles developed in Chapter 4. This chapter also motivates the correctness of these algorithms. Even though this chapter presents several algorithmic contributions, the reader might want to skip it during its first reading of the manuscript.

Finally, Chapter 6 presents the implementation (in the C language) of the algorithms and data structures described in Chapter 5. This implementation is integrated to the LASH toolset (see [13]). This chapter also discusses a few important practical details and presents some experimental results, that tend to show that automata on linear orderings can indeed be manipulated efficiently in practice, especially in the context of the decision procedure.

Chapter 1

Automata on linear orderings

In this chapter, we present automata on linear orderings. These finite-state automata can be seen as a broad generalization of finite-word, infinite-word and transfinite-word automata.

1.1 Linear orderings

In this part, we recall elementary notions on linear orderings. Much more detail can be found in [17].

A *linear ordering* is a set J equipped with an order relation $<_J$ which is

- total (for all $j, k, \ell \in J$, either $j <_J k$ or $k <_J j$),
- irreflexive ($j \not<_J j$),
- antisymmetric ($j <_J k$ implies $k \not<_J j$), and
- transitive ($j <_J k$ and $k <_J \ell$ imply $j <_J \ell$).

When clear from the context, $<_J$ will be abbreviated by $<$. Two elements $j, k \in J$ are said to be *consecutive* if $j < k$ and if there does not exist any $\ell \in J$ such that $j < \ell < k$. In this case, j is the *predecessor* of k and k is the *successor* of j. A linear ordering K is a *subordering* of J if $K \subseteq J$ and if the order relation $<_K$ is a restriction of $<_J$ to the elements of K.

Two linear orderings J_1 and J_2 are *isomorphic* if there exists an order-preserving bijection between them, i.e., if there exists a bijective mapping $\tau : J_1 \to J_2$ such that $k <_{J_1} \ell$ iff $\tau(k) <_{J_2} \tau(\ell)$. The *order type* of a linear ordering J is the class of all linear orderings that are isomorphic to J. The order types of the usual linear orderings $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ are denoted by ω, ζ, η and λ, respectively. Even though there exists a bijection between $\mathbb{Q}$ and $\mathbb{N}$ (since $\mathbb{Q}$ is a countable set), the order types ω and η are different, because such a bijection cannot be order-preserving. This is intuitively why $\mathbb{Q}$ and $\mathbb{N}$ are two very different linear orderings (see Examples 1 and 2). For the same reason, ω and ζ are different order types as well.

A linear ordering K is *dense* if it does not contain any pair of consecutive elements. It is *dense in* J if it is a subordering of J such that for any $i, j \in J$ such that $i < j$, one has $i < k < j$ for some $k \in K$. If K does not contain any dense infinite subordering, then K is said to be *scattered*.

Given two linear orderings J and K, we denote by $J + K$ the juxtaposition of J and K, i.e., the linear ordering L obtained via the disjoint union of J and K and in which the order relation is defined as follows:

- If $\ell_1, \ell_2 \in J$, then $\ell_1 <_L \ell_2$ iff $\ell_1 <_J \ell_2$.
- If $\ell_1, \ell_2 \in K$, then $\ell_1 <_L \ell_2$ iff $\ell_1 <_K \ell_2$.
- If $\ell_1 \in J$ and $\ell_2 \in K$, then $\ell_1 <_L \ell_2$.

More generally, for any linear ordering J, the sum $\sum_{j \in J} K_j$ (where K_j is a linear ordering for each $j \in J$) denotes the linear ordering whose elements are all the pairs (k, j) where $j \in J$ and $k \in K_j$. The order relation $(k_1, j_1) <_L (k_2, j_2)$ holds iff $j_1 <_J j_2$ (i.e., K_{j_1} comes before K_{j_2} in the juxtaposition) or $j_1 = j_2$ and $k_1 <_{K_{j_1}} k_2$ (i.e., k_1 and k_2 belong to the same linear ordering $K_{j_1} = K_{j_2}$, and k_1 is lower than k_2 in this linear ordering). Finally, we denote by $-J$ the linear ordering J in which the order relation has been reversed. In other words, $-J$ and J are the same set, but equipped with different order relations: $j <_J k$ iff $k <_{-J} j$.

A *cut* of a linear ordering J is an ordered pair of sets (K, L), where K and L partition J and $k <_J \ell$ for any $k \in K$ and any $\ell \in L$. The set of all the cuts of J is denoted by $\widehat{J}$ and can be linearly ordered by the order relation $<_{\widehat{J}}$, defined as $(K, L) <_{\widehat{J}} (K', L')$ iff $K \subsetneq K'$. The least element of $\widehat{J}$ is the *first cut* $\widehat{J}_{min} = (\emptyset, J)$. Similarly, the greatest element of $\widehat{J}$ is the *last cut* $\widehat{J}_{max} = (J, \emptyset)$. A cut that is not the first one or the last one is a *non trivial* cut, and the set of non trivial cuts is denoted by $\widehat{J}^*$. A cut $(K, L) \in \widehat{J}$ is a *gap* if $K \neq \emptyset$ has no greatest element and $L \neq \emptyset$ has no least element. The linear ordering J is *complete* if it has no gap. For all linear orderings J, it can be shown that $\widehat{J}$ is complete (see [7] for details).

Given the two linear orderings J and $\widehat{J}$, a new linear ordering $J \cup \widehat{J}$ can be defined by extending the order relations of J and $\widehat{J}$: the elements of J and the elements of $\widehat{J}$ are still ordered in the same way internally, and for any $j \in J$ and any cut $c = (K, L) \in \widehat{J}$, one has $j < c$ (resp. $c < j$) iff $j \in K$ (resp. $j \in L$). The linear ordering $J \cup \widehat{J}^*$ is defined analogously ($J \cup \widehat{J}^*$ is actually obtained by removing the least and the greatest elements from $J \cup \widehat{J}$). It can be shown that both $J \cup \widehat{J}$ and $J \cup \widehat{J}^*$ are complete linear orderings. The semantics of automata on linear orderings that we will introduce in Section 1.3 strongly relies on the fact that in $J \cup \widehat{J}$, any element $j \in J$ admits one successor $c_j^+ = (K \cup \{j\}, L)$ and one predecessor $c_j^- = (K, L \cup \{j\})$, where $K = \{k \in J \mid k < j\}$ and $L = \{\ell \in J \mid j < \ell\}$.

Example 1. *The linear ordering $\mathbb{N} + (-\mathbb{N})$ can be visually represented by the following line, in which the elements of the linear ordering are sorted by increasing order:*

$$0_{\mathbb{N}}, 1_{\mathbb{N}}, 2_{\mathbb{N}}, 3_{\mathbb{N}}, \ldots \quad \ldots, 3_{-\mathbb{N}}, 2_{-\mathbb{N}}, 1_{-\mathbb{N}}, 0_{-\mathbb{N}}$$

The notations $x_{\mathbb{N}}$ and $x_{-\mathbb{N}}$ are used to distinguish the elements of $\mathbb{N}$ from the elements of $-\mathbb{N}$. The linear ordering $\mathbb{N} + (-\mathbb{N})$ is scattered because each element has a predecessor and a successor (except $0_{\mathbb{N}}$ which has no predecessor and $0_{-\mathbb{N}}$ which has no successor), hence it does not contain any dense subordering. It is not complete, because the cut $(\mathbb{N}, -\mathbb{N})$ is a gap (since $\mathbb{N}$ does not have a greatest element and $-\mathbb{N}$ does not have a least element). Adding a single additional element between $\mathbb{N}$ and $-\mathbb{N}$ would make the ordering complete, because $\mathbb{N}$ and $-\mathbb{N}$ are already complete. Note that $+$ is not commutative: $-\mathbb{N} + \mathbb{N}$ is isomorphic to $\mathbb{Z}$ and differs completely from $\mathbb{N} + (-\mathbb{N})$.

Example 2. *Throughout this example, we use the notation $\rbrack a, b \lbrack_J$ where J is a subordering of $\mathbb{R}$ and $a, b \in \mathbb{R} \cup \{-\infty, +\infty\}$. This is a shorthand for $\{x \in J \mid a <_{\mathbb{R}} x \ \wedge \ x <_{\mathbb{R}} b\}$. As usual, the orientation of the square brackets (closed or open) is used to accept or reject the left bound a or the right bound b. For instance, $\rbrack a, b \rbrack_J$ denotes the set $\{x \in J \mid a <_{\mathbb{R}} x \ \wedge \ \neg (b <_{\mathbb{R}} x)\}$.*

The set $\mathbb{R}$ of real numbers (equipped with its natural order relation) is a dense linear ordering. Cuts of $\mathbb{R}$ consist of the first cut $(\emptyset, \mathbb{R})$, the last cut $(\mathbb{R}, \emptyset)$ and two infinite sets of non trivial cuts

$$\left\{ \left(\left]-\infty, r\right[_{\mathbb{R}}, \left[r, +\infty\right[_{\mathbb{R}} \right) \mid r \in \mathbb{R} \right\}$$

and

$$\left\{ \left(\left]-\infty, r\right]_{\mathbb{R}}, \left]r, +\infty\right[_{\mathbb{R}} \right) \mid r \in \mathbb{R} \right\}.$$

None of these cuts is a gap, hence $\mathbb{R}$ is a complete linear ordering. In contrast, the set $\mathbb{Q}$ of rational numbers is not a complete linear ordering. In addition to the non trivial cuts

$$\left\{ \left(\left]-\infty, q\right[_{\mathbb{Q}}, \left[q, +\infty\right[_{\mathbb{Q}} \right) \mid q \in \mathbb{Q} \right\}$$

and

$$\left\{ \left(\left]-\infty, q\right]_{\mathbb{Q}}, \left]q, +\infty\right[_{\mathbb{Q}} \right) \mid q \in \mathbb{Q} \right\}$$

there are uncountably many cuts of the form $\left(\left]-\infty, r\right[_{\mathbb{Q}}, \left]r, +\infty\right[_{\mathbb{Q}} \right)$ where $r \in \mathbb{R} \setminus \mathbb{Q}$. These cuts are gaps, because if r is irrational, the set $\left]-\infty, r\right[_{\mathbb{Q}}$ has no greatest element and the set $\left]r, +\infty\right[_{\mathbb{Q}}$ has no least element. Interestingly, the set of cuts of $\mathbb{Q}$ is thus uncountable, even though $\mathbb{Q}$ is a countable set.

1.2 Words on linear orderings

We now introduce the notions related to words on linear orderings, borrowed from [7].

Given a linear ordering J, a *word of length J* (also called *word indexed by J*) is a function w from J to a finite alphabet Σ (whose elements are *letters* or *symbols*). The letter associated with the element $j \in J$ is denoted by $w(j)$ and the length of w by $|w|$. Intuitively, if $|w|$ is a finite linear ordering, w is simply a finite word, in the classical sense. Setting $|w|$ to $\mathbb{N}$ or $\mathbb{Z}$ leads to the well-known infinite or bi-infinite words, respectively. The *empty word* ε is the word of length $\emptyset$. This shows that words on linear orderings consist of a broad generalization of the concepts of finite, infinite and bi-infinite words.

Two words w_1 and w_2 are *isomorphic* if there exists an order-preserving bijection $\tau : |w_1| \to |w_2|$ such that $w_1(j) = w_2(\tau(j))$ for all $j \in |w_1|$. The isomorphism between words is obviously an equivalence relation. In the sequel, we do not distinguish two words that are isomorphic, i.e., that belong to the same equivalence class.[1] Examples 3 and 4 show that this choice is natural. Formally, this amounts to working with the equivalence classes of words rather than with the words themselves. Still, we keep using the concept of word throughout this thesis, while keeping in mind that isomorphic words are undistinguishable.

Example 3. *Consider the word $w_1 : \{1, 2, 3\} \to \{a, b\}$ defined as*

$$w_1(1) = a, \;\; w_1(2) = b, \;\; w_1(3) = a$$

and the word $w_2 : \{4, 5, 6\} \to \{a, b\}$ defined as

$$w_2(4) = a, \;\; w_2(5) = b, \;\; w_2(6) = a.$$

These two words are finite words that are isomorphic, and they should of course be considered as identical, since they can both be written aba.

Example 4. *Let $\mathbb{N}_p$ be the set of natural numbers that are multiple of $p > 0$. On the alphabet $\{a\}$, two words w_1 and w_2 of respective lengths $\mathbb{N}_2$ and $\mathbb{N}_3$ are isomorphic. These two words are infinite words that contain only the symbol a, and they should thus intuitively be considered as identical.*

[1] Strictly speaking, the length of a word should therefore be an order type rather than a linear ordering.

The *product*, or *concatenation*, of two words w_1 and w_2 is defined as the word $w = w_1 \cdot w_2$ (sometimes just written $w_1 w_2$) of length $|w_1| + |w_2|$ such that

$$w(j) = \begin{cases} w_1(j) & \text{if } j \in |w_1| \\ w_2(j) & \text{if } j \in |w_2| \end{cases}$$

This operation is the same as for finite words, and can be generalized to multiple operands: for any linear ordering J, the product $\prod_{k \in J} w_k$ is the word w of length $\sum_{k \in J} |w_k|$ such that $w(j) = w_k(j)$ iff $j \in |w_k|$. Note that nothing prevents J to be dense in this definition.

As in the case of finite or infinite words, rational expressions can be defined for languages of words on linear orderings. A language (i.e., a class of words on linear orderings) is *rational* if it can be denoted by a rational expression. In the sequel, we present these rational expressions. Note that a rational expression *represents* a language but is not equal to a language. By convenience, we thus introduce the operator $\mathscr{L}(.)$ that converts a rational expression into the language that it represents.

Let Σ be a finite alphabet. We first define the elementary rational expressions, from which all the other rational expressions can be built inductively. These elementary rational expressions are:

- ε, that denotes the language $\mathscr{L}(\varepsilon) = \{\varepsilon\}$.
- $\varnothing$, that denotes the empty language $\mathscr{L}(\varnothing) = \{\}$.
- σ, that denotes the language $\mathscr{L}(\sigma) = \{\sigma\}$, for all $\sigma \in \Sigma$.

Next, we define how a rational expression can be built from a simpler one. Let X be a rational expression. As in the case of finite words, one can define the *finite iteration* X^* as the rational expression that denotes the language

$$\mathscr{L}(X^*) = \left\{ w \mid \exists\, n \in \mathbb{N} \text{ and } w_1, w_2, \ldots, w_n \in \mathscr{L}(X) \text{ such that } w = w_1 w_2 ... w_n \right\}. \tag{1.1}$$

Similarly, the *ω-iteration* X^ω represents the language

$$\mathscr{L}(X^\omega) = \left\{ \prod_{i \in \mathbb{N}} w_i \mid w_i \in \mathscr{L}(X) \right\}. \tag{1.2}$$

Interestingly, the finite iteration and the ω-iteration are particular cases of a general iteration $X^{\mathcal{J}}$ that denotes the language

$$\mathscr{L}\left(X^{\mathcal{J}}\right) = \left\{ \prod_{j \in J} w_j \mid w_j \in \mathscr{L}(X) \text{ and } J \in \mathcal{J} \right\}$$

where $\mathcal{J}$ is a class of linear orderings. This general iteration can be used to define concisely all the iterations that are needed to define rational languages of words on linear orderings:

- Setting $\mathcal{J}$ equal to the class of all finite linear orderings yields the finite iteration X^* whose meaning is already defined in 1.1.
- Setting $\mathcal{J}$ to $\{\omega\}$, i.e., the class of all the orderings isomorphic to $\mathbb{N}$, brings back the ω-iteration X^ω, that represents the language previously described in 1.2.
- The *ordinal iteration* $X^\sharp$ can be obtained by setting $\mathcal{J}$ to the class of all ordinals.
- The reverse ω-iteration $X^{-\omega}$ can be defined by setting $\mathcal{J}$ to $\{-\omega\}$ (the class of linear orderings isomorphic to $-\mathbb{N}$).

- Finally, the reverse ordinal iteration $X^{-\sharp}$ can be defined by setting $\mathcal{J}$ to the class of all reverse ordinals (the class of all the orderings J such that $-J$ is an ordinal).

We obviously do not introduce the reverse finite iteration, because $-J$ and J are isomorphic if J is a finite linear ordering. Note that we do not allow $\mathcal{J}$ to be the class of all linear orderings, because such a fully general iteration will be a particular case of the *diamond operator*, introduced later in this section.

Example 5. *The language denoted by the rational expression $(ab)^\omega(ba)^{-\omega}$ contains a single word (recall that we do not distinguish isomorphic words), of length $\mathbb{N}+(-\mathbb{N})$. On the alphabet $\{a,b\}$, this is the only word of this length which starts and ends by an a and in which two identical symbols are never consecutive (i.e., one cannot find two consecutive elements of $\mathbb{N}+(-\mathbb{N})$ that are mapped to the same symbol). In contrast, the language denoted by the rational expression $(ba)^{-\omega}(ab)^\omega$ contains a single word of length $\mathbb{Z}$. In this word, one can find two consecutive symbols a, corresponding to the last symbol of $(ba)^{-\omega}$ and the first symbol of $(ab)^\omega$.*

We now introduce the rational expressions that combine two simpler rational expressions X_1 and X_2. The rational expression $X_1 \cdot X_2$ denotes the *product* or the *concatenation* of the languages denoted by X_1 and X_2:

$$\mathscr{L}(X_1 \cdot X_2) = \Big\{ w_1 \cdot w_2 \mid w_1 \in \mathscr{L}(X_1) \text{ and } w_2 \in \mathscr{L}(X_2) \Big\}.$$

The rational expression $X_1 + X_2$ denotes the *union* of the languages represented by X_1 and X_2:

$$\mathscr{L}(X_1 + X_2) = \mathscr{L}(X_1) \cup \mathscr{L}(X_2).$$

We now introduce an operator that makes it possible to create words of arbitrary length. The rational expression $X_1 \diamond X_2$ (where $\diamond$ is the *diamond operator*) represents the following language:

$$\mathscr{L}(X_1 \diamond X_2) = \left\{ \prod_{j \in J \cup \widehat{J}^*} w_j \mid J \neq \emptyset \text{ and } w_j \in \mathscr{L}(X_1) \text{ if } j \in J \text{ and } w_j \in \mathscr{L}(X_2) \text{ if } j \in \widehat{J}^* \right\}.$$

Intuitively, to generate a word in $\mathscr{L}(X_1 \diamond X_2)$ of length $J \cup \widehat{J}^*$, one needs to choose a non empty linear ordering J and to assign a word from $\mathscr{L}(X_1)$ to each element $j \in J$ and a word from $\mathscr{L}(X_2)$ to each element $j \in \widehat{J}^*$. In the particular case where $\mathscr{L}(X_2)$ contains only the empty word, $X_1 \diamond X_2$ denotes the language

$$\mathscr{L}(X_1 \diamond \varepsilon) = \mathscr{L}(X_1^\diamond) = \left\{ \prod_{j \in J} w_j \mid J \neq \emptyset \text{ and } w_j \in \mathscr{L}(X_1) \right\}.$$

The rational expression $X^\diamond$ therefore serves as a shorthand for a general iteration based on any non-empty linear ordering.

Example 6. *The language denoted by $\varepsilon + (a+b)^\diamond$ contains every possible word on the alphabet $\{a,b\}$.*

A last rational expression has been introduced in [2] when generalizing the theory of automata on linear orderings to dense orderings. The *shuffle* of $X_1, X_2, \ldots, X_n$ is written $sh(X_1, X_2, \ldots, X_n)$. A word w belongs to $\mathscr{L}(sh(X_1, X_2, \ldots, X_n))$ if w can be written as $\prod_{j \in J} w_j$ where the ordering J has the two following properties:

- J is a non empty, dense and complete linear ordering without first and last element.
- J can be partitioned into n sub-orderings $J_1, J_2, \ldots, J_n$ that are all dense in J, and such that $w_j \in \mathscr{L}(X_i)$ if $j \in J_i$.

Intuitively, the shuffle operator mixes up languages associated to $X_1, X_2, \ldots, X_n$ in a dense fashion.

Even though a rational expression is not equal to the set it represents, we will allow ourselves to forget this distinction, by convenience. For instance, we will often write sentences like "the language a^ω contains a single word", even though a^ω is not a language, strictly speaking.

Example 7. *The word* $w_1 : \mathbb{R} \to \{a, b\}$ *such that*

$$\forall j \in \mathbb{R} \quad w_1(j) = \begin{cases} a & \text{if } j \in \mathbb{Q} \\ b & \text{if } j \in \mathbb{R} \setminus \mathbb{Q} \end{cases}$$

belongs to the language $sh(a, b)$*, since:*

- *Its length* $\mathbb{R}$ *is non empty, dense and complete. Moreover,* $\mathbb{R}$ *does not have a first or a last element.*
- $\mathbb{Q}$ *is dense in* $\mathbb{R}$ *and* $\mathbb{R} \setminus \mathbb{Q}$ *as well.*

The word $w_2 :]-1, 1[\to \{a, b\}$ *defined as*

$$\forall j \in]-1, 1[: \quad w_2(j) = \begin{cases} a & \text{if } j \in \mathbb{Q} \\ b & \text{if } j \in \mathbb{R} \setminus \mathbb{Q} \end{cases}$$

is isomorphic to w_1*, and we do not make any distinction between them. Note that it is not immediate to find an order-preserving bijection between* $\mathbb{R}$ *and* $]-1, 1[$ *that maps each rational of* $\mathbb{R}$ *to a rational of* $]-1, 1[$ *and each irrational of* $\mathbb{R}$ *to an irrational of* $]-1, 1[$*, but such a bijection does exist. For instance, consider the function*

$$\tau(j) = \begin{cases} \left(1 - \left(\frac{1}{2}\right)^{\lfloor j \rfloor} + (j - \lfloor j \rfloor)\left(\frac{1}{2}\right)^{\lceil j \rceil}\right) & \text{if } j \geq 0 \\ -\tau(-j) & \text{if } j < 0 \end{cases}$$

where $\lfloor x \rfloor$ *denotes the greatest integer lower or equal to* x*, and* $\lceil x \rceil$ *the least integer larger or equal to* x*. This function* τ *is piecewise linear, and each linear piece has a rational slope and a rational intercept. This guarantees that the image of a rational (resp. an irrational) is rational (resp. irrational). The function* τ *is depicted in Figure 1, in which the colours are used to highlight the different linear pieces of the function.*

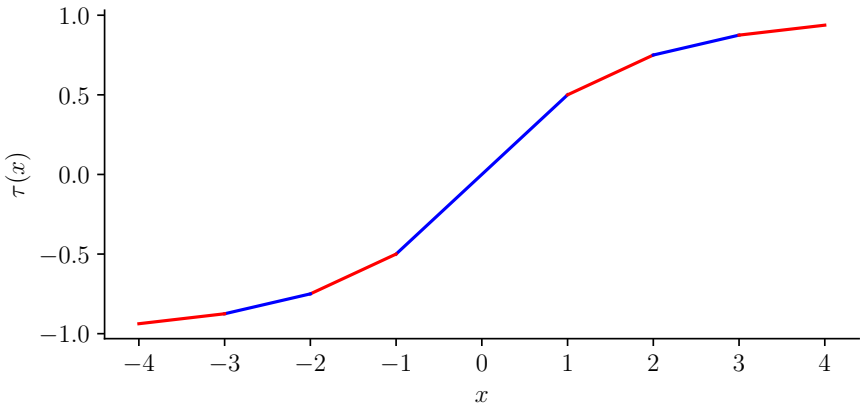


Figure 1: Bijection τ between $\mathbb{R}$ and $]1, 1[$. This bijection preserves the order, and maps each rational of $\mathbb{R}$ to a rational of $]-1, 1[$ and each irrational of $\mathbb{R}$ to an irrational of $]-1, 1[$.

Note that the language $sh(a, b)$ contains words whose length is not isomorphic to $\mathbb{R}$. For instance, the word

$$w_3 = \prod_{j \in \mathbb{R}} a \cdot w_1 \cdot a$$

has the length $\sum_{j\in\mathbb{R}}\{1\} + \mathbb{R} + \{1\}$, which is not isomorphic to $\mathbb{R}$ and yet does belong to $sh(a, b)$. The presence of the singletons $\{1\}$ around each copy of $\mathbb{R}$ is necessary to make $|w_3|$ complete.

Example 8. *The language $sh(a, b)$ does not contain any word of length $\mathbb{Q}$, since $\mathbb{Q}$ is not a complete linear ordering. However, the language $sh(a, \varepsilon)$ contains the word*

$$w = \prod_{j \in \mathbb{R}} w_j$$

where

$$w_j = \begin{cases} a & \text{if } j \in \mathbb{Q} \\ \varepsilon & \text{if } j \in \mathbb{R} \setminus \mathbb{Q} \end{cases} \qquad \forall j \in \mathbb{R}.$$

This word can be rewritten as

$$w = \prod_{j \in \mathbb{Q}} a \ ,$$

whose length is obviously $\mathbb{Q}$.

1.3 Automata on linear orderings

In this section, we present automata on linear orderings as defined by [7] and extended by [2]. They are a form of finite-state automata able to read words indexed by linear orderings as defined in Section 1.2. Their expressive power has been shown to coincide with the one of rational expressions [7, 2].

Definition 1. *An* automaton on linear orderings *is a tuple $(Q, \Sigma, \Delta, I, F)$ where:*

- *Q is a finite set of* states.
- *Σ is a finite* alphabet.
- *$\Delta \subseteq (Q \times \Sigma \times Q) \cup \left(Q \times 2^Q\right) \cup \left(2^Q \times Q\right)$ is a* transition relation.
- *$I \subseteq Q$ is a set of* initial states.
- *$F \subseteq Q$ is a set of* final states.

The transition relation Δ contains transitions of three types. Those belonging to the set $Q\times\Sigma\times Q$ are called *successor transitions* and can be written as $q_1 \xrightarrow{a} q_2$, for $q_1, q_2 \in Q$ and $a \in \Sigma$. We respectively call q_1, q_2 and a the *origin state*, the *destination state*, and the *label* of the transition. Transitions belonging to $Q \times 2^Q$ are called *right-limit transitions* and are noted $q \to P$, for $q \in Q$ and $P \subseteq Q$. In that case, q is the *origin state* of the transition. Symmetrically, *left-limit transitions* are those that belong to $2^Q \times Q$ and can be written as $P \to q$. In that case, q is the *destination state* of the transition.

We now present the semantics of such automata. Let $\mathcal{A}$ be an automaton on linear orderings.

Definition 2. *A* run *of $\mathcal{A}$ reading a word w of length J is a mapping $\rho : \widehat{J} \to Q$ such that:*

- *$\rho(\widehat{J}_{min}) \in I$.*
- *$\rho(\widehat{J}_{max}) \in F$.*

- *For every pair of cuts* $c_1 = (K_1, L_1)$ *and* $c_2 = (K_2, L_2)$ *such that* c_2 *is the successor of* c_1 *in* $\widehat{J}$, $(\rho(c_1), w(j), \rho(c_2)) \in \Delta$, *where* $j \in J$ *is such that* $K_2 = K_1 \cup \{j\}$.
- *For every cut* $c \neq \widehat{J}_{min}$ *which does not have any predecessor,* $(\lim_{c^-} \rho, \rho(c)) \in \Delta$, *where* $\lim_{c^-} \rho$ *is the* left-limit set *of* ρ *in* c, *defined as*

$$\lim_{c^-} \rho = \big\{q \in Q \mid (\forall c_1 < c)(\exists c_2)(c_1 < c_2 < c \wedge \rho(c_2) = q)\big\}.$$

- *For every cut* $c \neq \widehat{J}_{max}$ *which does not have any successor,* $(\rho(c), \lim_{c^+} \rho) \in \Delta$, *where* $\lim_{c^+} \rho$ *is the* right-limit set *of* ρ *in* c, *defined as*

$$\lim_{c^+} \rho = \big\{q \in Q \mid (\forall c_1 > c)(\exists c_2)(c < c_2 < c_1 \wedge \rho(c_2) = q)\big\}.$$

The word w is *accepted* by the automaton $\mathcal{A}$ iff there exists a run reading w in $\mathcal{A}$. The class of all words accepted by some automaton forms its *accepted language*. A state $q \in Q$ is *reachable* if it is reachable from an initial state, i.e., if there exists a word that can be read by starting in an initial state and by ending in q. Similarly, a state is *co-reachable* if a final state can be reached from it.

Intuitively, a run maps cuts to states, starting from an initial state and ending in a final one. Each pair of successive cuts corresponds to a unique element of the word's length. The run reads the letter associated to that element by taking a successor transition labelled accordingly.

Whenever a cut (different from the last one) does not have a successor, a right limit transition is used to reach the state to which it is mapped. The semantics is very close to a Muller acceptance condition [14]. That is, a jump can be followed after visiting exactly infinitely often some set of states. Left-limit transitions are symmetric.

Before illustrating the previous notions with the help of some examples, we define how automata on linear orderings are represented graphically. As usual with finite-state automata, we make use of a directed graph, whose vertices are states and edges are successor transitions. If there are several successor transitions sharing the same origin and destination states, they are grouped together into a single edge labelled by comma separated letters. Limit transitions are represented as annotations and can be grouped as well. For instance, given $q_0, q_1, q_2 \in Q$ and $P \subseteq Q$, the notation $q_0 \to P \to q_1, q_2$ is shorthand for $q_0 \to P$, $P \to q_1$, and $P \to q_2$. We will often say that P is the limit set of these limit transitions, even though this might be misleading (since a limit set must be a left-limit set $\lim_{c^-} \rho$ or a right-limit set $\lim_{c^+} \rho$ of the run ρ in c, as defined in Definition 2). Depending on the context, a limit set can therefore be the set of states associated to a limit transition in an automaton on linear orderings, or the limit set of a run ρ at a cut c.

Example 9. *The automaton in Figure 2a accepts the language* $(ab)^\omega$, *that contains a single word* $w : \mathbb{N} \to \{a, b\}$ *such that* $w(n) = a$ *if* n *is even and* $w(n) = b$ *if* n *is odd. Indeed, from the only initial state* q_0 *a letter* a *must be read, immediately followed by a letter* b, *itself immediately followed by the same sequence. In order to reach the only final state* q_2, *one must repeat that pattern infinitely often in order to follow the only limit transition* $\{q_0, q_1\} \to q_2$. *After this state has been reached, no other transition can be taken. If the reading of the whole word has come to an end, it is accepted. In this very simple setting this automaton behaves like a Muller automaton, as previously hinted.*

Example 10. *The automaton of Figure 2b can be understood as two copies of the previous automaton. The second copy has its transitions flipped and is concatenated to the first one. Therefore, its accepted language is* $(ab)^\omega (ba)^{-\omega}$. *Recall from Example 1 that the ordering indexing that language has a gap. This gap can be easily identified by the presence of state* q_2 *which is the origin of a right-limit transition and the destination of a left-limit transition. The non trivial cut* $(\mathbb{N}, -\mathbb{N})$ *is mapped to that state as it neither has a predecessor nor a successor.*

Example 11. *The automaton in Figure 2c accepts the language $(a^\omega)^{-\omega}$. Let us first analyse states q_1 and q_2, together with the associated successor transition from q_1 to itself and the left-limit transition from limit set $\{q_1\}$ to state q_2. For reasons very similar to those discussed in Example 9, this part reads words of length $\mathbb{N}$ whose elements are all mapped to the letter a (i.e., this part reads the language a^ω). Then, to reach that part from the initial state, one must take a right-limit transition towards a limit set containing q_1 and q_2. Exactly as visiting the loop formed by states q_3 and q_4 allowed to take the left-limit transition leading to a reverse ω-iteration in the previous example, the only way for a run to take this limit transition is then to visit infinitely often states q_1 and q_2 by reading a^ω-words infinitely often. These nested limit sets therefore lead to the doubly ω-iterated language announced: $(a^\omega)^{-\omega}$.*

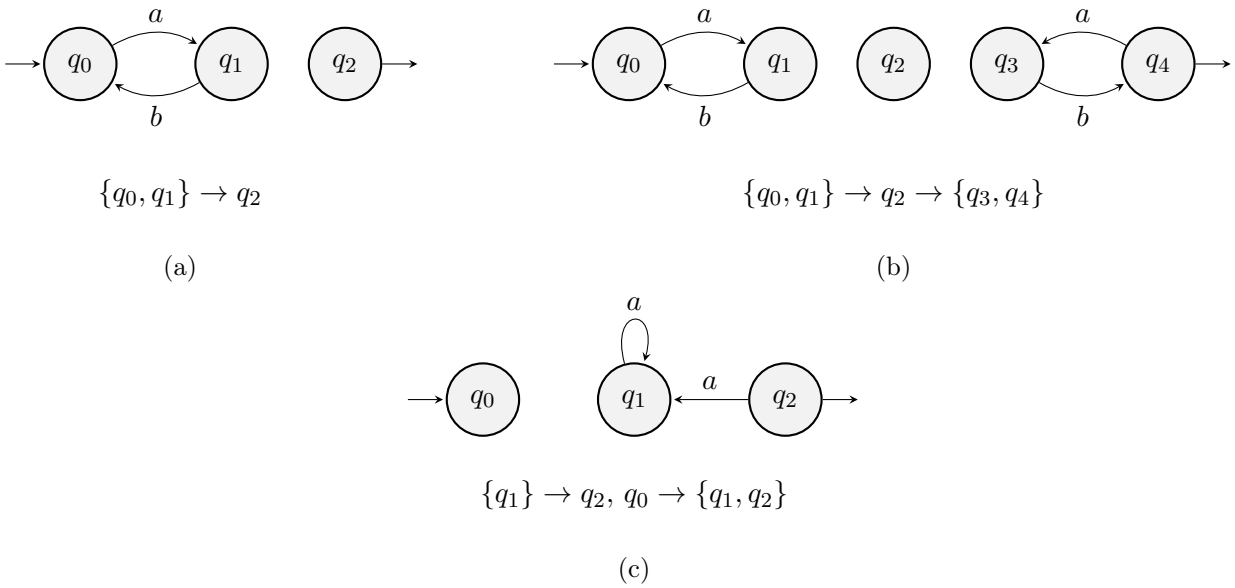


Figure 2: Automata on linear orderings accepting words of scattered lengths.

Example 12. *The automaton of Figure 3a accepts the language $sh(a, b)$. Recall from Example 7 that this language forms a dense and complete mixture of letters a and b, without a first or a last element. This is exactly what is enforced by this automaton. Indeed, from the initial state, one must take a right-limit transition, which requires to visit exactly infinitely often states q_1, q_2, q_3 and q_4. Moreover, states q_1 and q_3 can only be reached by following left-limit transitions. From these states, the only possible action is to follow a successor transition respectively labelled by a or b, which lead to states q_2 and q_3 respectively, themselves leading to the limit set by means of right-limit transitions. From this short analysis, several interesting things can be noticed:*

- *The ordering is complete since there is no gap.*
- *No letter admits a predecessor or a successor.*
- *The ordering does not have a first nor a last element.*
- *Readings of letters a and b have to be densely interleaved in order to satisfy requirements of limit transitions.*

This coincides with the definition of a shuffle.

Example 13. *Conversely, the automaton in Figure 3b accepts the language $sh(a, \varepsilon)$. Recall from Example 8 that such a language contains words indexed by $\mathbb{Q}$, which densely contain infinitely many gaps. The main difference with the previous example is that the successor transition formerly reading the letter b is now replaced by a single state, whose role is to be mapped to gaps.*

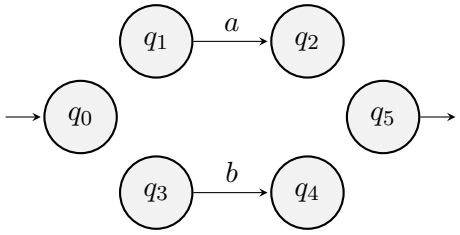
Example 14. *The automaton in Figure 3c accepts the language* $sh\,(a+b)$*. To give the intuition behind this, we show that every word consisting of a mapping from the set of real numbers to the alphabet* $\{a,b\}$ *is accepted by this automaton. Consider the function* $\rho : \widehat{\mathbb{R}} \to \{q_0, q_1\}$ *defined as*

$$\rho : \begin{cases} (\emptyset, \mathbb{R}) \mapsto q_0 \\ (\mathbb{R}, \emptyset) \mapsto q_1 \\ \left(\left]-\infty, r\right[_{\mathbb{R}}, \left[r, +\infty\right[_{\mathbb{R}} \right) \mapsto q_1, \quad \forall r \in \mathbb{R} \\ \left(\left]-\infty, r\right]_{\mathbb{R}}, \left]r, +\infty\right[_{\mathbb{R}} \right) \mapsto q_0, \quad \forall r \in \mathbb{R}. \end{cases}$$

Let us list properties of this function, to show that it complies with Definition 2.

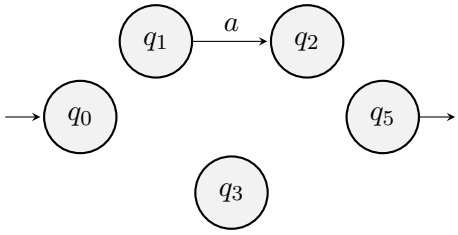
- *The first cut is mapped to an initial state and the last cut is mapped to a final state.*
- *Each pair of cuts* c_1*,* c_2 *where* c_1 *is the predecessor of* c_2 *is mapped to states connected by a successor transition reading the letter* a *or the letter* b*.*
- *Every cut of the form* $\left(\left]-\infty, r\right[_{\mathbb{R}}, \left[r, +\infty\right[_{\mathbb{R}} \right)$ *is not the first one, does not have a predecessor and every cut before it is mapped to* q_0 *or* q_1*.*
- *Every cut of the form* $\left(\left]-\infty, r\right]_{\mathbb{R}}, \left]r, +\infty\right[_{\mathbb{R}} \right)$ *is not the last one, does not have a successor and every cut after it is mapped to* q_0 *or* q_1*.*

Intuitively, this automaton consists of the one in Figure 3a, where states indexed by an even number are merged into state q_0 *and symmetrically, states indexed by an odd number are merged into state* q_1*.*



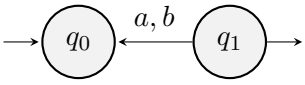
$q_0, q_2, q_4 \to \{q_1, q_2, q_3, q_4\} \to q_1, q_3, q_5$

(a)



$q_0, q_2, q_3 \to \{q_1, q_2, q_3\} \to q_1, q_3, q_5$

(b)



$q_0 \to \{q_0, q_1\} \to q_1$

(c)

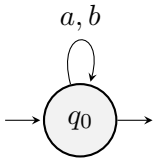
Figure 3: Automata on linear orderings accepting words of infinite, dense lengths.

Example 15. *The automaton in Figure 4 accepts any word indexed by a linear ordering on the alphabet* $\{a,b\}$*. Indeed, let us take a linear ordering* J *and show that the function* $\rho : \widehat{J} \to \{q_0\}$ *is a run reading a word of length* J *on the alphabet* $\{a,b\}$*, according to Definition 2.*

- *The first cut of* J *is mapped to an initial state and the last cut of* J *is mapped to a final state.*

- *For any pair of cuts $c_1, c_2 \in \widehat{J}$ such that c_1 is the predecessor of c_2, a successor transition from q_0 to itself can be followed to read the letter a or b.*
- *Every cut which is not the first one and which does not have a predecessor can be mapped to q_0 since every cut before it is mapped to q_0 as well, hence the left-limit transition $\{q_0\} \to q_0$ can be followed.*
- *Every cut which is not the last one and which does not have a successor can be mapped to state q_0 since every cut after it are mapped to q_0 as well, hence the right-limit transition $q_0 \to \{q_0\}$ can be followed.*

Therefore, this automaton accepts every word on the alphabet $\{a, b\}$.



$q_0 \to \{q_0\} \to q_0$

Figure 4: Automaton accepting every word on the alphabet $\{a, b\}$.

Chapter 2

Epsilon automata on linear orderings

In this chapter, we define a variant of automata on linear orderings that may contain epsilon transitions (i.e., transitions with an empty label) in their transition relation. With the appropriate choice of semantics, this new form of automata (that we call epsilon automata on linear orderings) keeps the same expressive power as automata on linear orderings (i.e., automata introduced in Definition 1). The use of epsilon transitions can significantly simplify some theoretical developments involving automata on linear orderings. The main motivation behind this work actually comes from the discovery of a (minor) error in [2] that, in our opinion, results from a construction made unnecessarily difficult by the lack of epsilon transitions. We illustrate the usefulness of epsilon automata by using them for correcting this erroneous construction.

2.1 Epsilon transitions

In this section, we motivate the need for epsilon transitions in automata on linear orderings, and introduce them with the help of an extended syntax and semantics.

2.1.1 Motivation

We mentioned in Chapter 1 that the languages that can be accepted by automata on linear orderings correspond exactly to those that can be described by rational expressions. This result has first been proved in [7] with a restriction to *countable and scattered* words, i.e., words w for which $|w|$ is a countable ordering that does not contain a dense infinite sub-ordering. It is then extended to unrestricted words in [2].

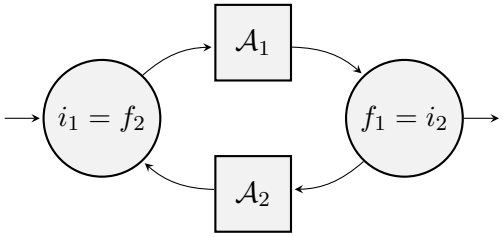
Recall from Chapter 1 that the rational expression $X_1 \diamond X_2$ represents the language

$$\mathscr{L}(X_1 \diamond X_2) = \left\{ \prod_{j \in J \cup \widehat{J}^*} w_j \mid J \neq \emptyset \text{ and } w_j \in \mathscr{L}(X_1) \text{ if } j \in J \text{ and } w_j \in \mathscr{L}(X_2) \text{ if } j \in \widehat{J}^* \right\}.$$

In the remainder of this chapter, it will be more convenient to only deal with languages, rather than with rational expressions. Therefore, we start by formally redefining the symbol $\diamond$ as a binary operator acting on languages of words indexed by linear orderings.

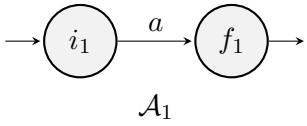
Definition 3. *Let L_1, L_2 be languages of words indexed by linear orderings, sharing the same alphabet Σ. The language $L = L_1 \diamond L_2$ contains all the words of the form $w = \prod_{j \in J \cup \widehat{J}^*} w_j$, where J is any non-empty linear ordering, and one has $w_j \in L_1$ for each $j \in J$ and $w_j \in L_2$ for each $j \in \widehat{J}^*$.*

Intuitively, $L_1 \diamond L_2$ is obtained by considering all the non-empty linear orderings J, and by replacing all their elements by words from L_1 and all their non-extreme cuts by words from L_2.

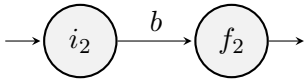


$$i_1 \to \{i_1, f_1, \ldots\} \to f_1$$

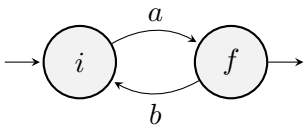
Figure 5: Automaton accepting $L_1 \diamond L_2$.



$\mathcal{A}_1$



$\mathcal{A}_2$



$$i \to \{i, f\} \to f$$

$\mathcal{A}$

Figure 6: Automata accepting $\{a\}$, $\{b\}$, and (erroneously) $\{a\} \diamond \{b\}$.

A construction building an automaton $\mathcal{A}$ accepting $L_1 \diamond L_2$ from automata $\mathcal{A}_1$ and $\mathcal{A}_2$ accepting respectively L_1 and L_2 is outlined in [7, 2]. It consists in first expressing each $\mathcal{A}_k$, for $k \in \{1, 2\}$, as a *normalised automaton*, which has a single initial state i_k and a single final state f_k such that $i_k \neq f_k$. In addition, i_k (resp. f_k) cannot be the destination (resp. the origin) of a successor or limit transition.

The next step is to combine $\mathcal{A}_1$ and $\mathcal{A}_2$ to form the automaton $\mathcal{A}$. The combination scheme is shown in Figure 5, in the particular case where neither $\mathcal{A}_1$ nor $\mathcal{A}_2$ accepts the empty word. In this construction, the limit transitions of the resulting automaton are those of $\mathcal{A}_1$ and $\mathcal{A}_2$, with additional transitions $i_1 \to P$ and $P \to f_1$ for all supersets P of $\{i_1, f_1\}$.

This construction is proved to be correct in [2], for the particular case of words restricted to be indexed by countable and scattered linear orderings. The claim made in [7] that it also applies to the general case is unfortunately invalid. Consider for instance the automata in Figure 6, accepting the languages $L_1 = \{a\}$ and $L_2 = \{b\}$. Let us show that the resulting automaton $\mathcal{A}$ accepts a language that differs from $L_1 \diamond L_2$. Indeed, $\mathcal{A}$ accepts in particular the word $w = \prod_{j \in \mathbb{R}} b$, i.e., the word indexed by $\mathbb{R}$ that only contains occurrences of the symbol b: a run ρ reading w can be obtained by setting $\rho((\emptyset, \mathbb{R})) = i$, $\rho((\mathbb{R}, \emptyset)) = f$, and for every $j \in \mathbb{R}$, $\rho((\mathbb{R}_{<j}, \mathbb{R}_{\geq j})) = f$ and $\rho((\mathbb{R}_{\leq j}, \mathbb{R}_{>j})) = i$ (remember Example 14). It is clear however that w does not belong to $\{a\} \diamond \{b\}$, since a occurs in every word of this language.

The error is caused by the fact that the construction in Figure 5 makes it possible to visit infinitely often all the states of $\mathcal{A}$ without ever following the successor transition labelled by a. This situation only happens when the limit transitions added by the construction can be followed without visiting a state of $\mathcal{A}_1$ distinct from i_1 and f_1.

A simple correction of this problem would thus consist in making sure that an internal state of $\mathcal{A}_1$ must necessarily be visited in order to enable the added limit transitions. However, this cannot be

achieved with our current definition of automata on linear orderings, since it follows from Definitions 1 and 2 that, in an automaton accepting $\{a\}$, there must be a successor transition from an initial state to a final one, labelled by a. This motivates the need for a more general form of automaton on linear orderings, offering the possibility of defining internal states linked by transitions with an empty label. We introduce such automata in the next section. The correction of the construction of an automaton accepting $L_1 \diamond L_2$ is addressed in Section 2.2.

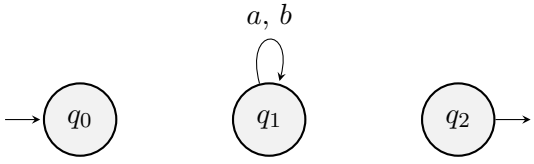
2.1.2 Epsilon automata on linear orderings

We define our extended form of automata over linear orderings as follows.

Definition 4. *An* epsilon automaton on linear orderings *is a tuple* $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ *where the set of states* Q*, the alphabet* Σ*, the sets of initial and final states* I *and* F *are as in Definition 1, and the relation transition* Δ *is of the form* $\Delta \subseteq (Q \times (\Sigma \cup \{\varepsilon\}) \times Q) \cup (Q \times 2^Q) \cup (2^Q \times Q)$.

Compared with Definition 1, the only difference is the possibility of having successor transitions (q_1, ε, q_2) with an empty label ε.

Designing an appropriate semantics for epsilon automata is not immediate. Following a successor transition labelled by ε should naturally make a run move from a state to another without reading any symbol in its accepted word. The question is to decide whether following an infinite combination of epsilon transitions should enable limit transitions. In the positive case, adding the transition (q_1, ε, q_1) to the automaton in Figure 7 would make it also accept all finite words over the alphabet $\{a, b\}$. In the negative case, its accepted language would not be affected. Our choice is to opt for the latter approach,



$q_0 \to \{q_1\} \to q_2$

Figure 7: Automaton accepting all the bi-infinite words on the alphabet $\{a, b\}$.

that conservatively minimises the impact of epsilon transitions on the semantics of automata. This leads to a semantics in which the states visited by a run reading a word w are still associated to the cuts of $|w|$, but with the property that following epsilon transitions does not modify the current cut.

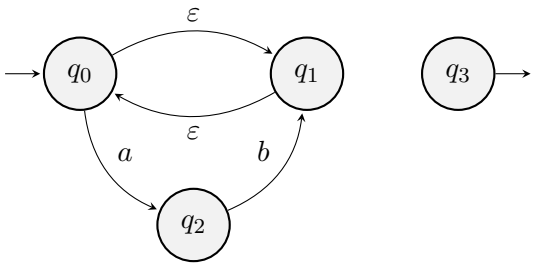
Formally, given an epsilon automaton $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$, we define the set $S \subseteq Q \times 2^Q \times Q$ of all the triples (q_1, U, q_2) such that $\mathcal{A}$ admits a finite path from q_1 to q_2 visiting exactly the states in U, entirely composed of successor transitions labelled by ε. In other words, such a path must be of the form $s_0 \xrightarrow{\varepsilon} s_1 \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} s_n$ with $n \geq 0$, $s_0 = q_1$, $s_n = q_2$, $U = \{s_0, s_1, \ldots, s_n\}$, and for every $0 \leq k < n$, $(s_k, \varepsilon, s_{k+1}) \in \Delta$. For each $\sigma = (q_1, U, q_2) \in S$, we define $\mathit{first}(\sigma) = q_1$, $\mathit{last}(\sigma) = q_2$, and $\mathit{states}(\sigma) = U$.

Definition 5. *A* run *of* $\mathcal{A}$ *reading a word* w *is a mapping* $\rho : \widehat{J} \to S$*, where* $J = |w|$*, that satisfies the following conditions:*

- $\mathit{first}(\rho(\widehat{J}_{min})) \in I$.
- $\mathit{last}(\rho(\widehat{J}_{max})) \in F$.

- *For every* $c_1 = (K_1, L_1)$, $c_2 = (K_2, L_2) \in \widehat{J}$ *such that* $K_2 = K_1 \cup \{j\}$ *for some* $j \in J$, *one has* $(last(\rho(c_1)), w(j), first(\rho(c_2))) \in \Delta$.
- *For every* $c \neq \widehat{J}_{min} \in \widehat{J}$ *that does not have a predecessor, one has* $\lim_{c^-} \rho \to first(\rho(c)) \in \Delta$, *where* $\lim_{c^-} \rho = \{q \in Q \mid (\forall c_1 < c)(\exists c_2)(c_1 < c_2 < c \wedge q \in states(\rho(c_2)))\}$.
- *For every* $c \neq \widehat{J}_{max} \in \widehat{J}$ *that does not have a successor, one has* $last(\rho(c)) \to \lim_{c^+} \rho \in \Delta$, *where* $\lim_{c^+} \rho = \{q \in Q \mid (\forall c_1 > c)(\exists c_2)(c < c_2 < c_1 \wedge q \in states(\rho(c_2)))\}$.

Intuitively, instead of mapping the cuts of $|w|$ onto states of the automaton as in Definition 2, those cuts are now associated with finite paths of epsilon successor transitions, in which the only relevant information is their origin and destination states, together with the set of all the states that they visit.



$\{q_0, q_1\} \to q_3$, $\{q_0, q_1, q_2\} \to q_3$

Figure 8: Example of epsilon automaton on linear orderings.

Example 16. *An example of epsilon automaton on linear orderings is given in Figure 8. This automaton accepts the language* $(ab)^\omega$, *composed of a single word* w *mapping* $\mathbb{N}$ *onto* $\{a, b\}$, *such that* $w(j) = a$ *for even values of* j, *and* $w(j) = b$ *for odd* j. *Note that this automaton does not accept the empty word. Indeed, the empty word has a single cut* $(\emptyset, \emptyset)$, *and this cut cannot be mapped to a triple* σ *such that* $first(\sigma)$ *is initial and* $last(\sigma)$ *is final, since the final state* q_3 *cannot be reached from the initial state* q_0 *by following only* ε*-transitions.*

2.1.3 Expressiveness

We now establish that, with the choices made in Section 2.1.2, adding epsilon transitions to automata on linear orderings does not increase their expressive power.

Theorem 1. *A language of words indexed by linear orderings can be accepted by an epsilon automaton on linear orderings iff it can be accepted by an automaton on linear orderings.*

Proof. One direction is immediate: by Definitions 1 and 4, an automaton on linear orderings $\mathcal{A}$ can also be seen as an epsilon automaton $\mathcal{A}_\varepsilon$ without any epsilon transition. Every run ρ of $\mathcal{A}$ can be turned into a run of $\mathcal{A}_\varepsilon$ reading the same word w, by replacing $\rho(c)$ by $(\rho(c), \{\rho(c)\}, \rho(c))$ for each $c \in \widehat{|w|}$.

For the other direction, we prove a stronger result by providing an algorithm that turns an epsilon automaton $\mathcal{A}_\varepsilon$ into an equivalent automaton $\mathcal{A}$ on linear orderings. Let $\mathcal{A}_\varepsilon = (Q_\varepsilon, \Sigma_\varepsilon, \Delta_\varepsilon, I_\varepsilon, F_\varepsilon)$. The automaton $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ is built as follows:

- $Q \subseteq Q_\varepsilon \times 2^{Q_\varepsilon} \times Q_\varepsilon$ is the set of all triples (q_1, U, q_2) such that $\mathcal{A}_\varepsilon$ admits a finite path of epsilon successor transitions from q_1 to q_2, visiting exactly the states that belong to U. (This corresponds to the definition of the set S in Section 2.1.2).
- $\Sigma = \Sigma_\varepsilon$.

- Δ contains
  - all $(\sigma_1, a, \sigma_2) \in Q \times \Sigma \times Q$ such that $(\mathit{last}(\sigma_1), a, \mathit{first}(\sigma_2)) \in \Delta_\varepsilon$,
  - all $(P, \sigma) \in 2^Q \times Q$ such that $\left(\bigcup_{\sigma' \in P} \mathit{states}(\sigma'), \mathit{first}(\sigma)\right) \in \Delta_\varepsilon$,
  - all $(\sigma, P) \in Q \times 2^Q$ such that $\left(\mathit{last}(\sigma), \bigcup_{\sigma' \in P} \mathit{states}(\sigma')\right) \in \Delta_\varepsilon$.
- $I = \{\sigma \in Q \mid \mathit{first}(\sigma) \in I_\varepsilon\}$.
- $F = \{\sigma \in Q \mid \mathit{last}(\sigma) \in F_\varepsilon\}$.

It follows from Definition 5 that every run of $\mathcal{A}_\varepsilon$ describes a run of $\mathcal{A}$ reading the same word. □

The algorithm given in the proof of Theorem 1 can incur an exponential blowup in the number of states of the automaton, and a double exponential blowup in the number of limit transitions. This is not problematic, since we expect epsilon automata on linear orderings to be mainly useful for theoretical developments.

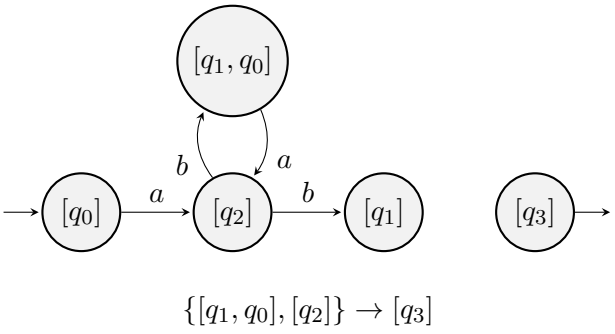
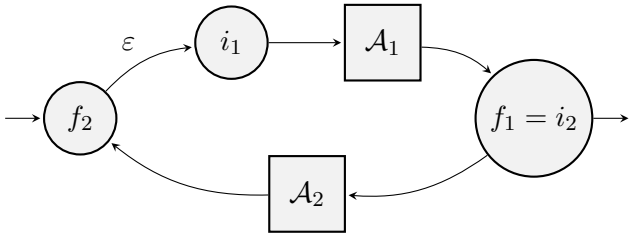


Figure 9: Automaton after eliminating epsilon transitions.

We illustrate this result by showing in Figure 9 the construction of an automaton on linear orderings equivalent to the one in Figure 8. We use the notation $[q_i]$ as shorthand for $(q_i, \{q_i\}, q_i)$, for $i \in \{0, 1, 2, 3\}$, and write $[q_1, q_0]$ in place of $(q_1, \{q_0, q_1\}, q_0)$. To keep the picture simple, some states and transitions that can not be visited or followed by any run are omitted.

2.2 Application

We are now ready to correct the construction of an automaton accepting $L_1 \diamond L_2$ with the help of epsilon automata. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be automata (either epsilon automata or plain automata on linear orderings) accepting respectively L_1 and L_2. Like in Section 2.1.1, we assume w.l.o.g. that $\mathcal{A}_1$ and $\mathcal{A}_2$ are normalised. Two remarks are in order. Firstly, epsilon transitions make it easy to normalise an arbitrary automaton: this operation amounts to replacing the initial states by a fresh state with outgoing epsilon transitions to them, and the final states by a fresh state with incoming epsilon transitions from them. Secondly, it is possible for a normalised epsilon automaton to accept the empty word: all that is needed is an epsilon transition linking its initial to its final states. This greatly simplifies the developments compared to [7], where languages that contain the empty word have to be handled as particular cases.



$$f_2 \to \{i_1, f_1, f_2, \ldots\} \to i_2$$

Figure 10: Automaton accepting $L_1 \diamond L_2$.

The construction of an automaton $\mathcal{A}$ accepting $L = L_1 \diamond L_2$ is shown in Figure 10. It consists in

- merging the initial state i_2 of $\mathcal{A}_2$ with the final state f_1 of $\mathcal{A}_1$ into a single state, which becomes the only final state of $\mathcal{A}$,
- making f_2 the only initial state of $\mathcal{A}$,
- adding an epsilon transition from f_2 to i_1,
- adding right-limit transitions from f_2 and left-limit transitions to i_2, for all sets of states containing i_1, i_2 and f_2.

Compared to the construction in Figure 5, the difference is that the presence of the epsilon transition (f_2, ε, i_1) now forces the state i_1, and therefore the branch associated to $\mathcal{A}_1$, to be visited in order to enable one of the added limit transitions. Let us show that this new construction is correct.

Theorem 2. *The automaton in Figure 10 accepts $L = L_1 \diamond L_2$.*

Proof. We first show that every word in $L_1 \diamond L_2$ is accepted by the constructed automaton $\mathcal{A}$. For any such word w, there exist a non-empty linear ordering J and words $w_j \in L_1$ for each $j \in J$ and $w_j \in L_2$ for each $j \in \widehat{J}^*$ such that $w = \prod_{j \in J \cup \widehat{J}^*} w_j$. For each $j \in J$ (resp. $j \in \widehat{J}^*$), let ρ_j be a run of $\mathcal{A}_1$ (resp. $\mathcal{A}_2$) that reads w_j. In order to obtain a run ρ of $\mathcal{A}$ reading w, one has to map every cut c of $|w|$ onto a triple $(q_1, U, q_2) \in Q \times 2^Q \times Q$, where Q is the set of states of $\mathcal{A}$. There are several cases to consider:

- If c is an internal cut of $|w_j|$ for some $j \in J \cup \widehat{J}^*$, i.e., $c \in \widehat{|w_j|}^*$, then we set $\rho(c) = \rho_j(c)$.
- Otherwise,
  - If $c = \widehat{|w|}_{min}$, $c = \widehat{|w_j|}_{min}$ for some $j \in J$, or $c = \widehat{|w_k|}_{max}$ for some $k \in \widehat{J}$, then we set $\rho(c) = \{f_2, \{f_2, i_1\}, i_1\}$.
  - If $c = \widehat{|w|}_{max}$, $c = \widehat{|w_j|}_{max}$ for some $j \in J$ or $c = \widehat{|w_k|}_{min}$ for some $k \in \widehat{J}$, then we set $\rho(c) = \{i_2, \{i_2\}, i_2\}$.

The ordering $J \cup \widehat{J}^*$ is complete, hence this list of cases is exhaustive. Let us show that the mapping ρ is a run reading w. We call a *segment* $[c_1, c_2]$ of ρ, with $c_1, c_2 \in \widehat{|w|}$, its restriction to the cuts $c \in \widehat{|w|}$ such that $c_1 \leq c \leq c_2$. Such a segment originates in the state $\mathit{first}(\rho(c_1))$ and ends in $\mathit{last}(\rho(c_2))$. It reads a subword of the word w read by ρ.

The run ρ starts at the initial state f_2 and ends at the final state i_2. It is composed of segments π_j from i_1 to $f_1 = i_2$ reading subwords w_j, with $j \in J$, by following runs of $\mathcal{A}_1$, and segments π_k from i_2 to f_2 reading subwords w_k, with $k \in \widehat{J}^*$, by following runs of $\mathcal{A}_2$. These segments are joined together according to the ordering $J \cup \widehat{J}^*$: for each $k \in \widehat{J}^*$, there are two possibilities for continuing ρ after the segment π_k:

- If k has a successor j in $J \cup \widehat{J}^*$, then the last state f_2 reached by π_k is linked to the first state i_1 of π_j by the epsilon transition (f_2, ε, i_1).
- Otherwise, one of the right-limit transitions $f_2 \to \{i_1, f_1, f_2, \ldots\}$ added by the construction can be followed from f_2, in order to reach further segments.

The same mechanisms allow to start the run ρ from its initial state f_2. Similarly, for each $k \in \widehat{J}^*$, the segment π_k can be reached either if its first state i_2 corresponds to the last state of the segment π_j associated to the predecessor j of k, or after following a left-limit transition to i_2 added by the construction. By the same principles, the run ends in the final state i_2.

The second part of the proof is to show that every word accepted by $\mathcal{A}$ belongs to the language $L_1 \diamond L_2$. Let ρ be a run of $\mathcal{A}$. This run can be decomposed into a set Γ_1 of segments that originate in f_2 and end in i_2 after going through i_1 and then $\mathcal{A}_1$, and a set Γ_2 of segments that originate in i_2 and end in f_2 after going through $\mathcal{A}_2$. Each segment in Γ_1 (resp. Γ_2) reads a word in L_1 (resp. L_2). The set $\Gamma_1 \cup \Gamma_2$ can be turned into a linear ordering by setting $\pi < \pi'$ iff the segment π comes before the segment π' in the run ρ. We now enumerate three properties that come directly from the structure of $\mathcal{A}$. Firstly, in the ordering $\Gamma_1 \cup \Gamma_2$, each element of Γ_1 that is not the first or the last one has a successor and a predecessor that both belong to Γ_2. Secondly, the ordering $\Gamma_1 \cup \Gamma_2$ is complete. Finally, for all $\pi_2 < \pi_2' \in \Gamma_2$, there exists $\pi_1 \in \Gamma_1$ such that $\pi_2 < \pi_1 < \pi_2'$, since π_2 and π_2' cannot be consecutive in $\Gamma_1 \cup \Gamma_2$, and $\Gamma_1 \cup \Gamma_2$ does not admit an infinite dense subordering that only contains elements of Γ_2. This last property is ensured by the presence of the state i_1 and the transition (f_2, ε, i_1).

We now show that those properties imply that the ordering $\Gamma_1 \cup \Gamma_2$ is isomorphic to $\Gamma_1 \cup \widehat{\Gamma_1}^*$, i.e., that there exists a one-to-one correspondence between these orderings that preserves order. This together with Definition 3 establishes that the word read by ρ belongs to $L_1 \diamond L_2$. Consider the mapping $\tau : \Gamma_1 \cup \Gamma_2 \to \Gamma_1 \cup \widehat{\Gamma_1}^*$ defined as $\tau(\pi_1) = \pi_1$ for all $\pi_1 \in \Gamma_1$ and $\tau(\pi_2) = \big(\{\pi_1 \in \Gamma_1 : \pi_1 < \pi_2\}, \{\pi_1 \in \Gamma_1 : \pi_2 < \pi_1\}\big)$ for all $\pi_2 \in \Gamma_2$. The fact that this mapping is an order-preserving bijection is a consequence of the following properties:

- Each element of Γ_1 does not belong to Γ_2, and is mapped by τ onto itself.
- For all segments $\pi_2, \pi_2' \in \Gamma_2$ such that $\pi_2 < \pi_2'$, one has $\tau(\pi_2) < \tau(\pi_2')$. This follows from the definition of τ and the existence of a segment π_1 between any two elements of Γ_2.
- For all $(K, L) \in \widehat{\Gamma_1}^*$, there exists $\pi_2 \in \Gamma_2$ such that $\tau(\pi_2) = (K, L)$, and $\pi_1 < \pi_2 < \pi_1'$ for all $\pi_1 \in K$ and $\pi_1' \in L$. Indeed, if K (resp. L) has a greatest element K_{max} (resp. a least element L_{min}), then π_2 can be chosen equal to the successor of K_{max} (resp. the predecessor of L_{min}) in $\Gamma_1 \cup \Gamma_2$. Otherwise, (K, L) is a gap in Γ_1, and the completeness property of $\Gamma_1 \cup \Gamma_2$ ensures the existence of π_2.

□

Chapter 3

Product of automata on linear orderings

The goal of this chapter is to introduce the main problem addressed in this master's thesis: the design of an algorithm that computes the product of automata on linear orderings. Section 3.1 first explains briefly the long-term goal of the research group in which this thesis has been carried out. This section can be seen as a description of the context in which our algorithms have been designed and tested. Section 3.2 then explains why the problem of computing efficiently the product of automata on linear orderings is important. This problem is also properly defined in this section. Next, Section 3.3 presents a first naive solution to this problem. Finally, Section 3.4 enumerates several ideas (explored later) that improve this naive solution.

Note that in this chapter and in the next ones, we do not use epsilon automata on linear orderings introduced in Chapter 2. In fact, we expect these automata to be mainly useful in theoretical developments. In the sequel, we address more practical problems for which epsilon automata are not helpful.

3.1 An automata-based decision procedure for the monadic first-order theory of order on $\mathbb{R}$ and $\mathbb{Q}$

The research group[1] in which this thesis has been carried out aims at building a decision procedure for the first-order theory of order over $\mathbb{R}$ and $\mathbb{Q}$, with uninterpreted unary predicates (see, for instance, [4]). The goal of such a procedure is to decide whether a quantified logical formula with variables in $\mathbb{R}$ or $\mathbb{Q}$, uninterpreted unary predicates, and the order relation $<$ (which can be considered as a special binary predicate) is satisfiable. This amounts to checking whether such a formula admits at least one *model*, i.e., an assignment of its unquantified variables and uninterpreted predicates that makes it true.

Example 17. *Consider the following logical formula Φ (taken from [4]), in which x, y, u and v are variables, and P is an uninterpreted unary predicate:*

$$\Phi : \ \forall x \, \forall y \, \exists u \, \exists v. \ \ x < y \implies \big(x < u < y \ \ \wedge \ \ x < v < y \ \ \wedge \ \ P(u) \ \ \wedge \ \ \neg P(v)\big)$$

This formula states that in any open interval $]x, y[$ of the domain, there exists a point for which P is true and a point for which P is false. This property is called chaoticity*. All the first-order variables of this formula are quantified and P is its only uninterpreted predicate. A model of Φ should thus simply define the predicate P. If the domain of the formula is $\mathbb{R}$ (i.e., the variables are interpreted over $\mathbb{R}$), such a model exists: setting*

$$P(x) = \begin{cases} \textit{True if } x \in \mathbb{Q} \\ \textit{False if } x \in \mathbb{R} \setminus \mathbb{Q} \end{cases}$$

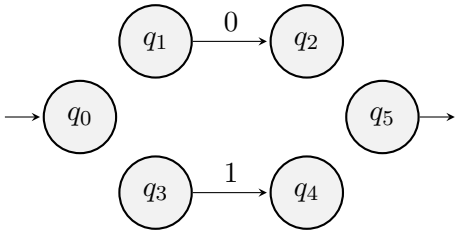
[1] Bernard Boigelot, Pascal Fontaine and Baptiste Vergain

makes the formula Φ true. This shows that the formula has at least one model and is thus satisfiable on $\mathbb{R}$.

For simpler domains like $\mathbb{N}$, a decision procedure already exists for such a logic fragment. This decision procedure translates a logical formula into a Büchi automaton that recognizes its models, and then checks whether the language accepted by this automaton is non-empty [9]. To solve the problem over $\mathbb{R}$ or $\mathbb{Q}$, the idea remains the same, except that Büchi automata must be replaced by more expressive automata. Intuitively, since the domains of interest are $\mathbb{R}$ and $\mathbb{Q}$, these automata must be able to read words of length $\mathbb{R}$ or $\mathbb{Q}$. Automata on linear orderings are thus a natural choice.

In order to build the decision procedure, the first step is to define how a model is encoded into a word. A natural idea is to use the so-called *unary notation*, as explained in [4]. A predicate P over $\mathbb{R}$ is encoded as a word $w_P : \mathbb{R} \to \{0,1\}$ such that $w_P(x) = 1$ if $P(x)$ and $w_P(x) = 0$ if $\neg P(x)$, for any $x \in \mathbb{R}$. A first-order variable v over $\mathbb{R}$ is encoded as a word $w_v : \mathbb{R} \to \{0,1\}$ such that $w_v(x) = 1$ iff $x = v$. A similar encoding is used when the domain is $\mathbb{Q}$, the only difference being that words of length $\mathbb{Q}$ are used. If the formula involves more than one uninterpreted predicate or unquantified first-order variable, then the words of the model are stacked to form a single word $w : \mathbb{R} \to \{0,1\}^T$, where T is the total number of uninterpreted predicates and unquantified first-order variables. In an automaton $\mathcal{A}$ reading w, each label is thus a tuple $(a_1, a_2, \ldots, a_T)$, where $a_i \in \{0,1\}$ refers to the i-th variable or predicate of the model. Such an automaton is said to read T *tapes* simultaneously. If the word w is accepted by the automaton $\mathcal{A}$, the model encoded into w is said to be *recognized* by $\mathcal{A}$. If the language accepted by $\mathcal{A}$ is the class of words that encode all the models of a formula, we will often say that the automaton $\mathcal{A}$ recognizes this formula (even though, strictly speaking, an automaton only recognizes the models of a formula).

Example 18. *Consider again the formula Φ on the domain $\mathbb{R}$ (see Example 17). The model that sets $P(x)$ to True iff $x \in \mathbb{Q}$ is such that $T = 1$ and is thus encoded as the word of length $\mathbb{R}$ that maps each rational number to 1 and each irrational number to 0. Now consider the automaton $\mathcal{A}_\Phi$ in Figure 11. This automaton accepts the language $sh(0,1)$ (see Example 7). A word w of length $\mathbb{R}$ is accepted iff the*



$q_0, q_2, q_4 \to \{q_1, q_2, q_3, q_4\} \to q_1, q_3, q_5$

Figure 11: Automaton $\mathcal{A}_\Phi$ that recognizes the models of Φ.

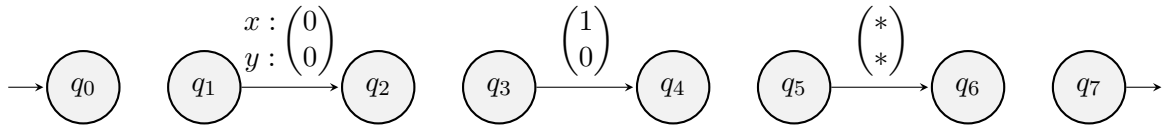
linear orderings $\{r_0 \in \mathbb{R} \mid w(r_0) = 0\}$ and $\{r_1 \in \mathbb{R} \mid w(r_1) = 1\}$ are both dense in $\mathbb{R}$. In other words, the set of points in which P is true and the set of points in which P is false must both be dense in $\mathbb{R}$. It is not difficult to see that $\mathcal{A}_\Phi$ thus recognizes the models of Φ on $\mathbb{R}$. Since $\mathcal{A}_\Phi$ accepts at least one word of length $\mathbb{R}$, the formula Φ is satisfiable. The goal of the decision procedure is to perform these two steps (building $\mathcal{A}_\Phi$ and checking the non-emptiness of its accepted language on $\mathbb{R}$ or $\mathbb{Q}$) algorithmically, for an arbitrary formula Φ given as input.

Building the automaton $\mathcal{A}_\Psi$ that recognizes an arbitrary formula Ψ can be done by structural induction. First, each atomic sub-formula (i.e., each sub-formula that cannot be split into smaller sub-formulas) is converted into an automaton (that we call an *atom*) that recognizes it. Then, the atoms

are progressively transformed and combined together, following the structure of the formula Ψ. This structural induction ends when an automaton $\mathcal{A}_\Psi$ that recognizes Ψ is obtained.

We now explain the details of this structural induction. Let us start by enumerating the atoms. The structure of these atoms will become important later. There are only 3 types of atomic sub-formulas:

- The sub-formulas that express an order constraint between two first-order variables (e.g., $x < y$ or $\neg(x < y)$, in $\mathbb{R}$ or in $\mathbb{Q}$). As an example, the sub-formula $x < y$ where $x, y \in \mathbb{R}$ is recognized by the automaton depicted in Figure 12. This atom has $T = 2$ tapes. The first one corresponds



$$q_0, q_2 \to \{q_1, q_2\} \to q_1, q_3$$

$$q_4, q_6 \to \{q_5, q_6\} \to q_5, q_7$$

Figure 12: Atom recognizing $x < y$ on $\mathbb{R}$.

  to x and the second one to y (the first label is annotated accordingly). The wildcard symbol $*$ is used by convenience when it does not matter to follow a transition labelled by 0 or by 1. For instance, the transition labelled by $\binom{*}{*}$ between q_5 and q_6 in Figure 12 represents the 4 transitions

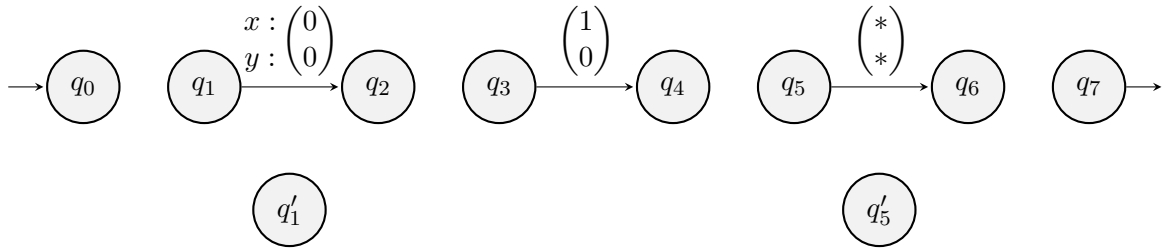
$$q_5 \xrightarrow{\binom{0}{0}} q_6, \quad q_5 \xrightarrow{\binom{1}{0}} q_6, \quad q_5 \xrightarrow{\binom{0}{1}} q_6 \quad \text{and} \quad q_5 \xrightarrow{\binom{1}{1}} q_6.$$

  Intuitively, given a word w on the alphabet $\{0,1\}^2$ that encodes the variables x and y, this automaton checks whether the symbol $\binom{1}{0}$ appears in w and all the previous symbols are $\binom{0}{0}$. If it the case, this means that $x < y$ and w is accepted, provided that its length is $\mathbb{R}$. Importantly, notice that the automaton in Figure 12 accepts words that are not necessarily of length $\mathbb{R}$. But since we are only interested in words of length $\mathbb{R}$ and $\mathbb{Q}$ in the context of the decision procedure, those words are ignored.

  The sub-formula $\neg(x < y)$ (or, equivalently, $y \leq x$), with $x, y \in \mathbb{R}$, can of course be encoded by the same automaton, except that the label $\binom{1}{0}$ must be replaced by $\binom{*}{1}$. If the domain is not $\mathbb{R}$ but $\mathbb{Q}$, the only modification to make is to add an isolated state q_1' and an isolated state q_5' to which the gaps of $\mathbb{Q}$ can be mapped. This yields the atom represented in Figure 13.

- The sub-formulas that state that a predicate is true (or false) in a point (e.g., $P(x)$ or $\neg P(x)$, in $\mathbb{R}$ or in $\mathbb{Q}$). These sub-formulas are encoded using the same principles as before. An atom that recognizes the formula $P(x)$ where $x \in \mathbb{R}$ must check that there is a 1 on the P-tape when there is a 1 of the x-tape. Assuming that the variable x has been encoded correctly, there should be a single 1 on the x-tape, which means that the automaton must simply check the presence of a 1 on the P-tape when the first 1 of the x-tape is read. This is exactly what the automaton in Figure 14 does. Encoding the sub-formula $\neg P(x)$ is similar, except that there must be a 0 on the P-tape when the first 1 of the x-tape is read. Working on $\mathbb{Q}$ simply amounts to adding two states in this automaton, as in Figure 13.

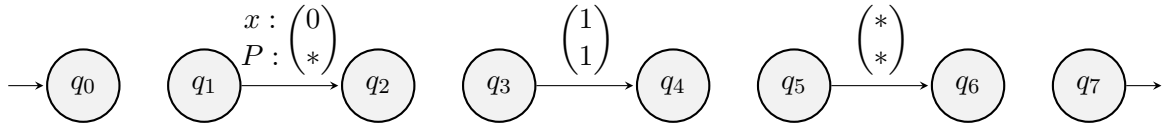
- The sub-formulas that state that a variable is real or rational (e.g., $x \in \mathbb{R}$ or $x \in \mathbb{Q}$). These sub-formulas are not directly present in the formula given as input to the decision procedure, but they implicitly appear once the domain has been fixed to $\mathbb{R}$ or $\mathbb{Q}$. An automaton recognizing



$$q_0, q_2, q'_1 \to \{q_1, q_2, q'_1\} \to q_1, q_3, q'_1$$

$$q_4, q_6, q'_5 \to \{q_5, q_6, q'_5\} \to q_5, q_7, q'_5$$

Figure 13: Atom recognizing $x < y$ on $\mathbb{Q}$.

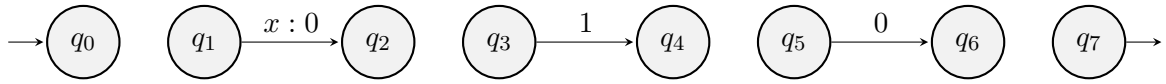


$$q_0, q_2 \to \{q_1, q_2\} \to q_1, q_3$$

$$q_4, q_6 \to \{q_5, q_6\} \to q_5, q_7$$

Figure 14: Atom recognizing $P(x)$ on $\mathbb{R}$.

$x \in \mathbb{R}$ is represented in Figure 15. It simply checks that the word encoding x contains a single 1, and that the length of this word is $\mathbb{R}$. Once again, one could obtain the automaton recognizing $x \in \mathbb{Q}$ by adding two states, as in Figure 13.



$$q_0, q_2 \to \{q_1, q_2\} \to q_1, q_3$$

$$q_4, q_6 \to \{q_5, q_6\} \to q_5, q_7$$

Figure 15: Atom recognizing the atomic sub-formula $x \in \mathbb{R}$.

Once each atomic sub-formula is associated to an atom, one can start to combine the atoms together and to transform them, following the syntactic structure of the formula Ψ. During this step, the automata undergo three types of transformation:

- Boolean operations: given two automata on linear orderings $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$ that recognize the formulas ψ_1 and ψ_2 respectively, the automaton $\mathcal{A}_{\psi_1 \wedge \psi_2}$ recognizing $\psi_1 \wedge \psi_2$ is obtained by computing the *intersection* of $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$. Similarly, one computes the automaton $\mathcal{A}_{\psi_1 \vee \psi_2}$ recognizing $\psi_1 \vee \psi_2$ via the *union* of $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$. Note that the automata $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$ can be either atoms, or more complicated automata that serve as intermediate steps between the atoms and the final automaton $\mathcal{A}_\Psi$.
- Existential quantification: given an automaton $\mathcal{A}_{\psi(x)}$ that recognizes the formula $\psi(x)$ (we write $\psi(x)$ rather than ψ here, to emphasize the fact that the formula ψ contains a free first-order

variable x, i.e., a first-order variable x that is not quantified), it is possible to build an automaton $\mathcal{A}_{\exists x.\psi(x)}$ that recognizes the formula $\exists x.\psi(x)$, by removing the x-tape from $\mathcal{A}_{\psi(x)}$ (see [5] for more details). This operation is not difficult, because it simply amounts to rewriting the labels of the successor transitions of $\mathcal{A}_{\psi(x)}$.

- Universal quantification: similarly, it seems possible to compute an automaton $\mathcal{A}_{\forall x.\psi(x)}$ that recognizes the formula $\forall x.\psi(x)$ (but it still needs to be confirmed). However, this operation is more complicated than existential quantification. The development of an algorithm that computes $\mathcal{A}_{\forall x.\psi(x)}$ from $\mathcal{A}_{\psi(x)}$ is still ongoing. A solution already exists when the domain is $\mathbb{N}$ (see [5]), i.e., for infinite-word automata (more precisely, for Büchi automata). To be able to work with the domains $\mathbb{R}$ and $\mathbb{Q}$, this solution must now be generalized to automata on linear orderings. Note that universal quantification can be reduced to existential quantification and negations (i.e., $\forall x.\psi(x)$ is equivalent to $\neg\exists x.\neg\psi(x)$), but the solution proposed in [5] does not rely on this equivalence, for two reasons. First, computing the automaton recognizing $\neg\psi(x)$ is hard, because it requires to complement the Büchi automaton $\mathcal{A}_{\psi(x)}$, which is known to be a difficult operation (see, e.g., [20]). Secondly, generalizing such a solution to automata on linear orderings is infeasible in general, because automata on linear orderings cannot be complemented.[2] Given an automaton on linear orderings $\mathcal{A}$ that accepts a language L on the alphabet Σ, the existence of an automaton $\mathcal{A}^C$ that accepts all the words on the alphabet Σ except the words of L is not guaranteed. In other words, unrestricted rational languages of words on linear orderings are not closed under complementation (see [16] for more details).

After having computed the automaton $\mathcal{A}_\Psi$, the last step of the decision procedure is to check whether this automaton accepts at least one word of length $\mathbb{R}$ or $\mathbb{Q}$, depending on the chosen domain. Implementing efficiently such a non-emptiness test is at the core of the master's thesis of Thomas Braipson [6].

Example 19. *Consider again the formula Φ from Examples 17 and 18. The first step of the decision procedure is to rewrite the implication $\Longrightarrow$ using only negations and Boolean connectives ($\wedge$ and $\vee$), and then push the negations inwards. This yields the following formula:*

$$\Phi: \quad \forall x\,\forall y\,\exists u\,\exists v.\ \neg(x<y)\ \vee\ \big(x<u<y\ \ \wedge\ \ x<v<y\ \ \wedge\ \ P(u)\ \ \wedge\ \ \neg P(v)\big).$$

Next, each atomic sub-formula is associated to an atom. In this case, there are 7 atomic sub-formulas:

$$\neg(x<y),\quad x<u,\quad u<y,\quad x<v,\quad v<y,\quad P(u),\quad \neg P(v),$$

Then, the automaton $\mathcal{A}_{\varphi(x,y,u,v)}$ recognizing the formula

$$\varphi(x,y,u,v): \quad \neg(x<y)\ \vee\ \big(x<u<y\ \ \wedge\ \ x<v<y\ \ \wedge\ \ P(u)\ \ \wedge\ \ \neg P(v)\big)$$

is computed via intersections and unions of automata. Finally, two applications of the existential quantification on automata followed by two applications of the universal quantification on automata (the order matters, since the order of the quantifiers in a logical formula is important) yield the automaton $\mathcal{A}_\Phi$ recognizing the formula

$$\Phi: \quad \forall x\,\forall y\,\exists u\,\exists v.\ \varphi(x,y,u,v).$$

Ideally, this process should output the automaton from Example 18. However, in practice nothing will guarantee that the decision procedure builds the simplest automaton recognizing Φ.

[2] This means that before translating a formula Ψ into an automaton that recognizes its models, all the negations in Ψ must first be pushed inwards, i.e., at the level of the atomic sub-formulas. Complementing an atom is not problematic, since we saw previously that it is trivial to compute the atom that recognizes the atomic sub-formula $\neg(x<y)$ or $\neg P(x)$.

3.2 Product of automata on linear orderings: motivation and definition of the problem

For the decision procedure sketched in Section 3.1 to be effective, automata on linear orderings must first be carefully implemented with the help of an adequate data structure. Appendix A, which is common with Thomas Braipson, is dedicated to this part. Then, all the operations that these automata must support have to be implemented efficiently. Amongst these operations, the union and the intersection of automata are critical, for the following reasons. First, those operations are needed to translate the effect of the Boolean connectives $\wedge$ and $\vee$, which are ubiquitous in the logic that we want to decide. Secondly, the union and intersection of automata are very common operations, used as subroutines in more complicated algorithms or applications (even outside the scope of the decision procedure). For instance, the universal quantification algorithm will probably need them, even though this is not clear yet. Finally, we will see that the way of implementing the union and the intersection of automata on linear orderings strongly impacts the memory consumption of the decision procedure. The latter is critical, since automata-based decision procedures are often bottlenecked by memory.

We now define properly the main problem addressed in this thesis. Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ be $n \geq 2$ automata on linear orderings sharing the same alphabet Σ, and $L_1, L_2, \ldots, L_n$ their accepted languages, respectively. The *n-ary intersection of automata on linear orderings* takes as inputs the n automata $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ (that we call *operands*) and outputs an automaton on linear orderings accepting the language

$$L_1 \cap L_2 \cap \cdots \cap L_n.$$

Similarly, the *n-ary union of automata on linear orderings* yields an automaton accepting the language

$$L_1 \cup L_2 \cup \cdots \cup L_n.$$

Our goal is to design an algorithm that computes the n-ary intersection operation and the n-ary union operation efficiently (both in terms of time and memory) for most of the inputs $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$. In particular, this algorithm should be efficient when $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ are typical automata manipulated by the decision procedure (e.g., atoms).

By convenience, we introduce the following notation: $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_n$ denotes the automaton on linear orderings obtained by applying the n-ary intersection operation to the operands $\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_n$ (that we will also called, in words, the n-ary intersection of $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$). Note that we do not define $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_n$ as *the* automaton accepting $L_1 \cap L_2 \cap \cdots \cap L_n$, since there exist infinitely many automata that accept this language. Instead, we simply use $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_n$ as shorthand for the automaton outputted by the n-ary intersection algorithm. This is slightly abusive, since we will build different versions of this algorithm throughout this master's thesis, hence $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_n$ might not always refer to the same automaton. The expression $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_n$ should therefore be understood as "the automaton yielded by the current version of the n-ary intersection algorithm", unless otherwise stated. The meaning of $\mathcal{A}_1 \cup \mathcal{A}_2 \cup \cdots \cup \mathcal{A}_n$ is similar.

Implementing the n-ary intersection or n-ary union operation is essentially the same problem. Conceptually, the solution consists in simulating the joint behaviour of each $\mathcal{A}_i$ ($i \in \{1, 2, \ldots, n\}$) by means of a product automaton $\mathcal{A}_{product}$. One then defines the final states of $\mathcal{A}_{product}$ according to the Boolean operation that needs to be computed. For an intersection, a run in $\mathcal{A}_{product}$ must simulate a run in each operand $\mathcal{A}_i$. For a union, a run in $\mathcal{A}_{product}$ must simulate a run in at least one operand $\mathcal{A}_i$. It is actually possible to implement an arbitrary combination of unions and intersections by following the same method. For instance, in the case where $n = 3$, computing an automaton accepting $(L_1 \cap L_2) \cup L_3$ simply amounts to choosing the final states of $\mathcal{A}_{product}$ in such a way that a run in $\mathcal{A}_{product}$ simulates

a run in $\mathcal{A}_1$ and in $\mathcal{A}_2$, or in $\mathcal{A}_3$. Since all these problems are solved in a similar way, we will focus without loss of generality on the n-ary intersection operation in the remainder of this master's thesis, for two reasons. First, the n-ary intersection automaton $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_n$ is well defined compared to the product automaton $\mathcal{A}_{product}$, which only serves as an intermediate computation step and does not strictly obey the definition of an automaton on linear orderings (for instance, the final states of $\mathcal{A}_{product}$ are not fixed, since they depend on whether $\mathcal{A}_{product}$ serves as an intermediate step for an intersection or a union operation[3]). Secondly, explaining the algorithm that computes n-ary intersection of automata is slightly easier, because concise examples can be used. However, the reader should keep two things in mind: the product of automata on linear orderings *is* the central problem studied in this master's thesis (the difficulty of the intersection and the union operations comes exclusively from the difficulty of simulating the joint behaviour of several automata on linear orderings), and all the discussions that follow about the n-ary intersection operation can be straightforwardly adapted to the n-ary union operation, or even to an arbitrary combination of unions and intersections, with very few modifications (if any). These modifications are outlined at the end of Chapter 5, after the complete description of an intersection algorithm.

Note that we phrased the problem in a general way: the number n of operands in the n-ary intersection operation is not fixed. An alternative would have been to set $n = 2$ and to reduce the n-ary intersection problem to $n - 1$ successive binary intersection operations. But in Chapter 4, we will introduce a new automata model that we will use internally during multiple successive intersection operations. We will see that this strategy can be highly beneficial when n is large. This explains why we do not want to restrict the problem to the case $n = 2$.

3.3 A naive solution

In this section, we present a naive algorithm that computes the n-ary intersection of automata on linear orderings. We then show that this algorithm is correct, but impossible to use in practice.

Let us first briefly recall how the intersection of finite-word automata can be computed. A finite-word automaton can be defined as an automaton on linear orderings (see Definition 1) without limit transitions, hence it makes sense to consider the intersection of finite-word automata as a first step towards the intersection of automata on linear orderings. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be finite-word automata and let Q_1 and Q_2 be their sets of states, respectively. Let Σ be the common alphabet of $\mathcal{A}_1$ and $\mathcal{A}_2$. The automaton $\mathcal{A} = \mathcal{A}_1 \cap \mathcal{A}_2$ can be obtained as follows:

- The first step is to define the set of states of $\mathcal{A}$ as $Q = Q_1 \times Q_2$. Intuitively, being in the state $q = (q_1, q_2) \in Q$ amounts to being simultaneously in the state q_1 in $\mathcal{A}_1$ and in the state q_2 in $\mathcal{A}_2$.
- The state $(q_1, q_2) \in Q$ is initial (resp. final) in $\mathcal{A}$ iff q_1 is initial (resp. final) in $\mathcal{A}_1$ and q_2 is initial (resp. final) in $\mathcal{A}_2$.
- The last step is to add successor transitions in $\mathcal{A}$. The successor transition $(q_1, q_2) \xrightarrow{\sigma} (q_1', q_2')$ (with σ a symbol of the alphabet Σ) is added in $\mathcal{A}$ iff $q_1 \xrightarrow{\sigma} q_1'$ and $q_2 \xrightarrow{\sigma} q_2'$ belong to the the transition relation of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Following the transition $(q_1, q_2) \xrightarrow{\sigma} (q_1', q_2')$ in $\mathcal{A}$ thus amounts to following simultaneously the transition $q_1 \xrightarrow{\sigma} q_1'$ in $\mathcal{A}_1$ and the transition $q_2 \xrightarrow{\sigma} q_2'$ in $\mathcal{A}_2$.

[3] Depending on the implementation, the transition relation Δ of the product $\mathcal{A}_{product}$ may also depend on whether the product is used as a subroutine for an intersection or a union operation.

Let now generalize this method to automata on linear orderings with limit transitions. We keep the same notations as above, but we now assume that $\mathcal{A}_1$ and $\mathcal{A}_2$ contain such limit transitions. To handle them, we need to introduce a new definition. Let $P_1 \subseteq Q_1, P_2 \subseteq Q_2$ be limit sets in the automata on linear orderings $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. A limit set $P \subseteq Q$ *covers exactly* P_1 if

$$P_1 = \big\{q_1 \mid (q_1, q_2) \in P\big\}$$

and similarly, P covers exactly P_2 if

$$P_2 = \big\{q_2 \mid (q_1, q_2) \in P\big\}.$$

Intuitively, if P covers exactly P_1, visiting infinitely often the states of P in $\mathcal{A}$ amounts to visiting infinitely often the states of P_1 in $\mathcal{A}_1$. More formally, if $\rho : \widehat{J} \to Q$ is a run in $\mathcal{A}$ associated to a word w of length J, and if $\lim_{c^-} \rho$ or $\lim_{c^+} \rho$ is equal to P for some cut $c \in \widehat{J}$, then it is possible to find a run $\rho_1 : \widehat{J} \to Q_1$ associated to w in $\mathcal{A}_1$ such that $\lim_{c^-} \rho_1$ or $\lim_{c^+} \rho_1$ is equal to P_1. If P covers exactly both P_1 and P_2, and if $\mathcal{A}_1$ and $\mathcal{A}_2$ contain the limit transitions $P_1 \to q_1$ and $P_2 \to q_2$ respectively, then following in $\mathcal{A}$ a limit transition $P \to (q_1, q_2)$ simulates how $P_1 \to q_1$ and $P_2 \to q_2$ can be followed simultaneously in each operand $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively.

We now describe formally a naive method (already mentioned in [10]) to compute the n-ary intersection of automata on linear orderings. The first step consists in rewriting the n-ary intersection as $n-1$ successive binary intersections. Then, each intermediate binary intersection is computed by means of the following method, that we call *naive binary intersection*. Let $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$ be the two operands. The intersection $\mathcal{A}_1 \cap \mathcal{A}_2$ is the automaton on linear orderings $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ such that:

- $Q = Q_1 \times Q_2$.
- $I = I_1 \times I_2$.
- $F = F_1 \times F_2$.
- The successor transition $(q_1, q_2) \xrightarrow{\sigma} (q_1', q_2')$ with $\sigma \in \Sigma$ belongs to Δ iff $q_1 \xrightarrow{\sigma} q_1'$ belongs to Δ_1 and $q_2 \xrightarrow{\sigma} q_2'$ belongs to Δ_2.
- The left-limit transition $P \to q$ belongs to Δ iff there exist $P_1 \to q_1$ in Δ_1 and $P_2 \to q_2$ in Δ_2 such that $q = (q_1, q_2)$ and P covers exactly P_1 and P_2.
- Similarly, the right-limit transition $q \to P$ belongs to Δ iff there exist $q_1 \to P_1$ in Δ_1 and $q_2 \to P_2$ in Δ_2 such that $q = (q_1, q_2)$ and P covers exactly P_1 and P_2.

Note that this method simply consists in computing $\mathcal{A}_1 \cap \mathcal{A}_2$ as if $\mathcal{A}_1$ and $\mathcal{A}_2$ were finite-word automata, and then in adding the necessary limit transitions. These two phases are completely independent. We will see later than interleaving these two phases brings several advantages.

Regarding the handling of the limit transitions, it is possible to rewrite the naive algorithm differently by means of a new operation that we call *limit set conjunction*. This operation allows to nicely combine the limit sets from $\mathcal{A}_1$ and $\mathcal{A}_2$.

Definition 6. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two automata on linear orderings, with set of states Q_1 and Q_2 respectively. Let $P_1 \subseteq Q_1$ and $P_2 \subseteq Q_2$ be limit sets of the automata $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. The* limit set conjunction

$$\langle P_1, P_2 \rangle \subseteq 2^{Q_1 \times Q_2}$$

is defined as the set that contains the subsets of $Q_1 \times Q_2$ covering exactly P_1 and P_2. This operation can be generalized to $n \geq 2$ operands: the set $\langle P_1, P_2, \dots P_n \rangle \subseteq 2^{Q_1 \times Q_2 \times \cdots \times Q_n}$ contains the subsets of $\prod_{i=1}^{n} Q_i$ covering exactly P_i, for all $i \in \{1, 2, \dots, n\}$.

Example 20. *Consider the limit set conjunction* $\left\langle \{2,3\}, \{2',3'\} \right\rangle$ *(that we will use later in Example 21). To compute it, we need to enumerate all the subsets of* $\{2,3\} \times \{2',3'\}$ *that cover exactly* $\{2,3\}$ *and* $\{2',3'\}$*. This yields*

$$\left\langle \{2,3\}, \{2',3'\} \right\rangle = \Big\{ \{(2,2'),(3,2'),(2,3'),(3,3')\}, \{(2,2'),(3,2'),(2,3')\}, \{(2,2'),(3,2'),(3,3')\}, \\ \{(2,2'),(2,3'),(3,3')\}, \{(3,2'),(2,3'),(3,3')\}, \{(2,2'),(3,3')\}, \{(3,2'),(2,3')\} \Big\}$$

The set $\{(2,2'),(2,3')\}$ *does not belong to the limit set conjunction, since it does not cover exactly* $\{2,3\}$*.*

As the name suggests, limit set conjunctions can help to formulate the naive intersection method differently. The set of left-limit transitions of the intersection $\mathcal{A} = \mathcal{A}_1 \cap \mathcal{A}_2$ can actually be concisely written as

$$\Big\{ P \to (q_1, q_2) \mid P_1 \to q_1 \text{ belongs to } \Delta_1 \ \wedge \ P_2 \to q_2 \text{ belongs to } \Delta_2 \ \wedge \ P \in \langle P_1, P_2 \rangle \Big\}$$

and similarly, the set of right-limit transitions takes the form

$$\Big\{ (q_1, q_2) \to P \mid q_1 \to P_1 \text{ belongs to } \Delta_1 \ \wedge \ q_2 \to P_2 \text{ belongs to } \Delta_2 \ \wedge \ P \in \langle P_1, P_2 \rangle \Big\}.$$

Before proving that the naive intersection method is correct, we illustrate it by means of an example.

Example 21. *Consider the two operands depicted in Figure 16. The first one accepts the language* $(ba^*a)^{-\omega}$*, that contains all the words* $w : -\mathbb{N} \to \{a,b\}$ *such that:*

- *the last symbol is the symbol* a*,*
- *there are infinitely many symbols* b*, and*
- *the symbol* b *is always followed by at least one symbol* a*.*

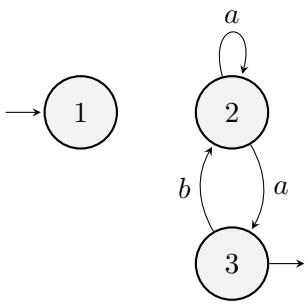
The second one accepts the language $(ab^*a)^{-\omega}$*. This language contains all the words* $w : -\mathbb{N} \to \{a,b\}$ *such that*

- *the last symbol is the symbol* a*,*
- *there are infinitely many symbols* a*, and*
- *the number of symbols* a *between two symbols* b *is even.*

The naive intersection method applied to these operands yields the automaton in Figure 17 (in which the states that are not reachable are not represented). Each operand has a single left-limit transition. The limit transitions of the automaton resulting from their intersection thus take the form $(1,1') \to P$ *for all* $P \in \left\langle \{2,3\}, \{2',3'\} \right\rangle$*. Therefore, there are* 7 *limit transitions (see Example 20).*

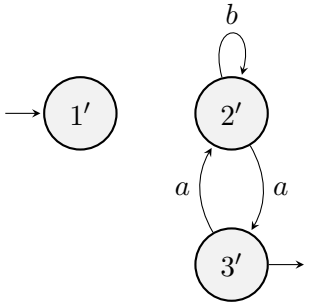
The only limit transition that can be followed in a run is $(1,1') \to \{(2,2'),(3,2'),(2,3')\}$*. In any other limit transition, the states of the limit set cannot be visited infinitely often. For instance, it is impossible to find a run* ρ *and a cut* c *such that* $\lim_{c^+} \rho = \{(2,3'),(3,2')\}$*, since* $(2,3')$ *cannot be reached from* $(3,2')$ *without visiting a state that does not belong to* $\{(2,3'),(3,2')\}$*. The automaton in Figure 17 thus accepts the language* $((aa)^*(aa)b)^{-\omega}(aa)^*a$*. This language contains the words* $w : -\mathbb{N} \to \{a,b\}$ *such that*

- *the symbols* a *and* b *appear infinitely often,*



$1 \to \{2, 3\}$

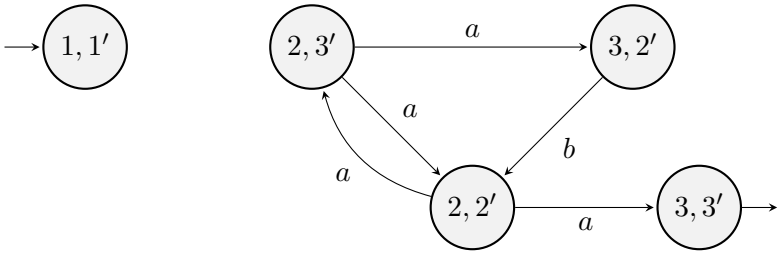
(a) First operand.



$1' \to \{2', 3'\}$

(b) Second operand.

Figure 16: The operands of the naive intersection method.



$(1, 1') \to \{(2, 2'), (3, 2'), (2, 3'), (3, 3')\}$
$(1, 1') \to \{(2, 2'), (3, 2'), (3, 3')\}$
$(1, 1') \to \{(3, 2'), (2, 3'), (3, 3')\}$
$(1, 1') \to \{(3, 2'), (2, 3')\}$

$(1, 1') \to \{(2, 2'), (3, 2'), (2, 3')\}$
$(1, 1') \to \{(2, 2'), (2, 3'), (3, 3')\}$
$(1, 1') \to \{(2, 2'), (3, 3')\}$

Figure 17: Result of the naive intersection method.

- *the symbol b is always followed by at least one symbol a,*
- *the number of symbols a between two symbols b is even,*
- *and the last symbol is the symbol a.*

Theorem 3. *The naive binary intersection method is correct.*

Proof. Let L_1 and L_2 be the languages respectively accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$. We first show that any word $w \in L_1 \cap L_2$ is accepted by $\mathcal{A}$. Let ρ_1 and ρ_2 be runs associated to w in $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, and let J denote the length of w. Consider the following function $\rho : \widehat{J} \to Q$:

$$\rho(c) = (\rho_1(c), \rho_2(c)) \quad \forall c \in \widehat{J}. \tag{3.1}$$

This function defines a run of $\mathcal{A}$ reading w. This follows easily from the way $\mathcal{A}$ has been built. The only non-trivial part is about limit transitions. Consider a cut c and the limit sets $\lim_{c^-} \rho$, $\lim_{c^-} \rho_1$ and $\lim_{c^-} \rho_2$, in the particular case where ρ follows 3.1:

$$\lim_{c^-} \rho = \{(q_1, q_2) \in Q \mid (\forall c' < c)(\exists c'')(c' < c'' < c \wedge \rho_1(c'') = q_1 \wedge \rho_2(c'') = q_2)\} \tag{3.2}$$

$$\lim_{c^-} \rho_1 = \{q_1 \in Q_1 \mid (\forall c' < c)(\exists c'')(c' < c'' < c \wedge \rho_1(c'') = q_1)\} \tag{3.3}$$

$$\lim_{c^-} \rho_2 = \{q_2 \in Q_2 \mid (\forall c' < c)(\exists c'')(c' < c'' < c \wedge \rho_2(c'') = q_2)\} \tag{3.4}$$

Clearly, $\lim_{c^-} \rho$ covers exactly $\lim_{c^-} \rho_1$ and $\lim_{c^-} \rho_2$. If the non-trivial cut c does not have any predecessor in $\widehat{J}$, then the left-limit transition $(\lim_{c^-} \rho) \to \rho(c)$ belongs to Δ, since $(\lim_{c^-} \rho_1) \to \rho_1(c)$ and $(\lim_{c^-} \rho_2) \to \rho_2(c)$ belong to Δ_1 and Δ_2, respectively. A similar argument can be used if the non-trivial cut c does not have any successor in $\widehat{J}$. This shows that ρ is indeed a run of $\mathcal{A}$ reading w. In other words, $\mathcal{A}$ accepts w.

Now let ρ be a run associated to w in $\mathcal{A}$. We have to show that w is accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$. One can easily find two runs ρ_1 and ρ_2 associated to w in $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively: for each $c \in \widehat{J}$, we set $\rho_1(c) = q_1$ and $\rho_2(c) = q_2$, where q_1 and q_2 are such that $\rho(c) = (q_1, q_2)$. Again, the non-trivial part is to check that ρ_1 and ρ_2 are valid runs regarding the limit transitions. Let $c \in \widehat{J}$ be a non-trivial cut without predecessor in $\widehat{J}$. The limit transition $\lim_{c^-} \rho \to \rho(c)$ belongs to Δ since ρ is a run. Looking at how Δ was built from Δ_1 and Δ_2, this implies that the limit transitions

$$\left\{ q_1 \mid (q_1, q_2) \in \lim_{c^-} \rho \right\} \to \rho_1(c)$$

$$\left\{ q_2 \mid (q_1, q_2) \in \lim_{c^-} \rho \right\} \to \rho_2(c)$$

belong to Δ_1 and Δ_2, respectively. Thanks to 3.2, 3.3 and 3.4, it is not difficult to see that

$$\left\{ q_1 \mid (q_1, q_2) \in \lim_{c^-} \rho \right\} = \lim_{c^-} \rho_1$$

$$\left\{ q_2 \mid (q_1, q_2) \in \lim_{c^-} \rho \right\} = \lim_{c^-} \rho_2$$

and therefore, $\lim_{c^-} \rho_1 \to \rho_1(c)$ (resp. $\lim_{c^-} \rho_2 \to \rho_2(c)$) is a limit transition of $\mathcal{A}_1$ (resp. $\mathcal{A}_2$). The same argument holds if the non-trivial cut c does not have any successor. This shows that ρ_1 and ρ_2 are indeed runs, and thus $w \in L_1 \cap L_2$. □

Example 21 illustrates the main weakness of the naive intersection method: the number of limit transitions in the resulting automaton becomes very large. The core of the problem is that the number of limit sets in $\langle P_1, P_2 \rangle$ grows prohibitively with $|P_1|$ and $|P_2|$. Assume that we want to compute the intersection between 10 automata $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_{10}$ that are similar to the operands used in Example 21, i.e., each automaton $\mathcal{A}_i$ has a single left-limit transition $P_i \to q_i$ with $|P_i| = 2$. According to the naive intersection method, one needs to compute 9 successive binary intersections. Assume that the 8 first intersections result in the automaton $\mathcal{A}$, and that the final step is to compute $\mathcal{A} \cap \mathcal{A}_{10}$. One of the limit transitions of $\mathcal{A}$ is

$$P_1 \times P_2 \times \cdots \times P_9 \to (q_1, q_2, \ldots, q_9)$$

since $P_1 \times P_2 \times \cdots \times P_9 \in \langle P_1, P_2, \ldots, P_9 \rangle$. Therefore, the computation of $\mathcal{A} \cap \mathcal{A}_{10}$ involves the enumeration of all the limit sets that belong to

$$\Big\langle P_1 \times P_2 \times \cdots \times P_9,\ P_{10} \Big\rangle.$$

This operation is completely out of reach for any computer. To prove it, let us first compute a lower bound for the cardinality of $\langle P, Q \rangle$. Assume without loss of generality that $|P| \geq |Q|$ and consider all the surjections from P to Q. Each surjection s can be turned into a different element of $\langle P, Q \rangle$ via the mapping

$$s \mapsto \big\{ (p, q) \mid p \in P \ \ \wedge \ \ q = s(p) \big\}.$$

Therefore, $|\langle P, Q \rangle|$ is lower bounded by the number of surjections between P and Q. The cardinality of $\big\langle P_1 \times P_2 \times \cdots \times P_9,\ P_{10} \big\rangle$ is thus lower bounded by the number of surjections between a set of 2^9 elements and a set of 2 elements, which is equal to $2^{2^9} - 2$. In other words, it is impossible to compute the intersection between $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_{10}$ with this approach, even though this task is small compared to the ones that would arise in the context of the decision procedure.

3.4 Possible improvements

In this section, we briefly introduce and motivate some ideas to improve the naive intersection method described in the previous section. These ideas are explored in the remainder of this master's thesis.

The first possible improvement is already suggested in Example 21. It consists in deleting limit transitions that can never be followed. To apply this idea, one should be able to decide whether a limit set can or cannot be visited infinitely often. We will see that this problem boils down to computing strongly connected components in the transition graph of an automaton.

Another promising idea is to design a new representation of limit transitions. In the naive n-ary intersection method, each intermediate binary intersection step yields an automaton on linear orderings satisfying Definition 1. But nothing prevents us from using another formalism internally during the computation, and to convert the result into an automaton on linear orderings at the end of the operation. Of course, this strategy does not reduce the size of the final automaton (i.e., if the intersection automaton produced by the naive n-ary intersection method is prohibitively large, using an internal representation during the intersection does not solve the problem, since the final automaton remains the same). However, such a strategy can improve the time complexity of the algorithm, and also decrease the peak memory usage. In many practical cases, the size of the final automaton is small even though the peak memory usage is high. For instance, consider the task of computing $\mathcal{A} = \mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_{10} \cap \mathcal{A}_{11}$ where

- $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_{10}$ are automata similar to the operands used in Example 21, i.e., each automaton has one right-limit transition whose limit set contains only 2 states.
- $\mathcal{A}_{11}$ does not contain any limit transition.

In Section 3.3, we saw that the task of computing $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_{10}$ via the naive intersection method is intractable. Computing $\mathcal{A} = \mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_{10} \cap \mathcal{A}_{11}$ by first computing $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \cdots \cap \mathcal{A}_{10}$ is thus intractable as well. However, since $\mathcal{A}_{11}$ does not have any limit transition, the automaton $\mathcal{A}$ does not have any limit transition either. Storing $\mathcal{A}$ on a computer is thus feasible (it contains at most 2^{11} states). This is a pathological example where the peak memory utilization is the bottleneck of the naive intersection method. The internal representation of the limit transitions that we will develop soon will allow us to dramatically lower this peak. The task of computing $\mathcal{A}$ will actually become trivial and almost as efficient as an intersection of finite-word automata.

Alternatively, a way of computing $\mathcal{A}$ efficiently would be to optimize the order of the operands. Since the operator $\cap$ is associative, one could compute $\mathcal{A}$ by successively computing $\mathcal{A}_{10} \cap \mathcal{A}_{11}$, $\mathcal{A}_9 \cap \mathcal{A}_{10} \cap \mathcal{A}_{11}$, $\mathcal{A}_8 \cap \mathcal{A}_9 \cap \mathcal{A}_{10} \cap \mathcal{A}_{11}$, etc. Since $\mathcal{A}_{11}$ does not have any limit transition, the task of computing $\mathcal{A}$ essentially reduces to an intersection of finite-word automata. In the sequel, we will not explicitly explore this idea of optimizing the order in which the operands are aggregated. Nevertheless, we will build an algorithm that takes quickly all the operands into account and postpones as much as possible the treatment of the limit transitions. This strategy involves benefits that are similar to those brought by optimizing the order of the operands.

Finally, we could also try to avoid instantiating useless states. In the naive binary intersection method, the set of states of the intersection automaton is set to $Q_1 \times Q_2$, even though some of these states could be useless (e.g., not reachable). Efficient implementations of the intersection of finite-word automata explore methodically the states of the operands and build the intersection automaton incrementally, by adding states one by one. This guarantees that any state that is added is reachable. Similarly, it is possible to make sure that all the states that remain in the final result are co-reachable. The algorithm that we will come up with at the end of the day will be similar, i.e., almost each state of an automaton

resulting from an n-ary intersection will be both reachable and co-reachable. Such a property is critical, because instantiating less states makes it possible to consider less limit transitions, as we will see. But this will make the intersection algorithm more complicated, since knowing whether a state is reachable requires to take the limit transitions into consideration. This means that we will have to interleave the phase where successor transitions are added with the phase where limit transitions are added. This will raise interesting questions.

Even if exploring these ideas will definitely improve the naive intersection method, some instances of the problem will remain intractable to solve, despite being small. For instance, one can find a few automata of reasonable size (i.e., with a reasonable number of states and limit transitions) whose intersection yields an automaton with a reasonable number of states but too many limit transitions to be stored in the memory of a computer. The problem of computing the intersection of automata on linear orderings is intrinsically hard, mostly because the number of different limit sets in an automaton with N states is $O(2^N)$ in the worst case.

Therefore, the purpose of the following chapters is not to design an algorithm that is efficient in the worst case, but rather to make sure that it is reasonably efficient in most of the practical scenarios that are expected (e.g., in the context of the decision procedure). The automata that recognize the logical formulas that we want to study are often sparse, i.e., there have few successor transitions. This is true for the atoms, and can be verified empirically for unions and intersections of atoms. This has an important consequence. Considering that an automaton can be seen as a graph (the states are the vertices and the successor transitions are the arcs), the automata that will typically arise in the context of the decision procedure have small strongly connected components. This will be exploited in the following chapters.

Chapter 4

Efficient intersection

This chapter introduces two ways of improving the naive intersection method for automata on linear orderings: the deletion of useless limit sets, and a concise representation of limit transitions. These two ideas are discussed theoretically and independently from each other. In the next chapter, they will be applied in parallel in the form of practical data structures and algorithms.

4.1 Deleting useless limit sets

This section thoroughly explores the idea of detecting and then deleting useless limit sets during the intersection (or, more generally, the product) operation of automata on linear orderings. In Section 4.1.1, we first define formally the notion of useless limit set. Then, the problem of detecting whether a limit set is useless is solved in Section 4.1.2. Section 4.1.3 presents a limit-set deletion algorithm, which is then adapted in Section 4.1.4 to run in the context of an intersection operation.

4.1.1 Useless limit sets

Let us start by defining a useless limit set.

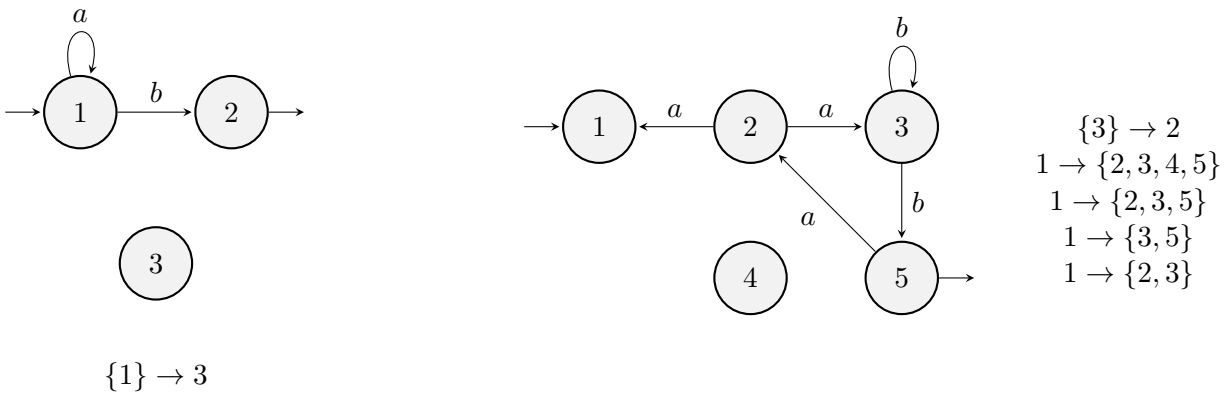
Definition 7. *Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be an automaton on linear orderings and let P be one of its limit sets (i.e., there exists $q \in Q$ such that $P \to q$ or $q \to P$ belongs to Δ). The limit set P is said to be* useless *if for any accepted word w of length J and for any run ρ reading w in $\mathcal{A}$, ρ does not follow a limit transition whose limit set is P. A limit set that is not useless is said to be* useful.

It follows from this definition that deleting a useless limit set of $\mathcal{A}$, i.e., removing from Δ all the limit transitions that involve it, does not change the language accepted by $\mathcal{A}$. We now illustrate the concept of useless limit set by means of examples.

Example 22. *Consider the automaton on linear orderings $\mathcal{A}_1$ in Figure 18a. State 3 is not co-reachable, hence the limit set $\{1\}$ is useless since it only appears in the limit transition $\{1\} \to 3$. The automaton $\mathcal{A}_2$ from Figure 18b is more interesting. Let us examine its limit sets one by one:*

- *State 4 is not reachable, and therefore there cannot exist a run that maps a cut to this state. The limit set $\{2, 3, 4, 5\}$ is thus useless.*
- *States 2, 3 and 5 are connected by successor transitions that form a cycle, i.e., they form a strongly connected component in the transition graph of $\mathcal{A}_2$. Therefore, it is possible to visit exactly these states infinitely often (given that they are reachable and co-reachable). Hence the limit transition $1 \to \{2, 3, 5\}$ can be followed in a run. The limit set $\{2, 3, 5\}$ is thus useful.*
- *The limit set $\{3, 5\}$ is a subset of $\{2, 3, 5\}$ but is useless, since it is impossible to visit exactly states 3 and 5 infinitely often without visiting state 2.*

- *Even though the limit set* $\{2, 3\}$ *might seem similar to* $\{3, 5\}$*, is is useful because the limit transition* $\{3\} \to 2$ *makes it possible to visit exactly states* 2 *and* 3 *infinitely often.*



(a) Automaton $\mathcal{A}_1$. (b) Automaton $\mathcal{A}_2$.

Figure 18

Example 22 shows that there exist two kinds of useless limit sets: limit sets that appear in a limit transition $P \to q$ (resp. $q \to P$) where q is not co-reachable (resp. not reachable), and limit sets whose states cannot be visited infinitely often. In this section, we focus on the second kind of useless limit sets, that can be handled with simple and effective strategies. We will develop an easy criterion that will detect *some* useless limit sets of the second kind. The goal is to avoid accumulating too many limit sets during successive binary intersections. Therefore, it is not required to remove *all* the useless limit sets. Removing most of them would already be a significant improvement.

4.1.2 Reachability graphs

Example 22 suggests that useless limit sets of the second kind can be detected by computing the strongly connected components of the transition graph of the automaton, i.e., the graph in which the vertices are the nodes of the automaton, the arcs[1] are its successor transitions (in which the labels are discarded) and the limit transitions are ignored. However, the case of the limit set $\{2, 3\}$ in Figure 18b also shows that the limit transitions must be taken into consideration before stating that a limit set is useless. This motivates the need of a tool that is slightly more complex than a transition graph and that we call *reachability graph*. We first define what *reachability* means.

Definition 8. *Given an automaton on linear orderings* $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$*, a state* q_2 *is* reachable from *a state* q_1 *if there exists a word* $w_{q_1 \to q_2}$ *that can be read in* $\mathcal{A}$ *by starting in* q_1 *and ending in* q_2*, in other words, the automaton* $\big(Q, \Sigma, \Delta, \{q_1\}, \{q_2\}\big)$ *accepts at least one word. This definition generalizes the notions of reachable and co-reachable states already introduced in Chapter 1.*

Definition 9. *Given an automaton on linear orderings* $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$*, an* reachability graph *is a tuple* $G = (V, E, R)$ *where:*

- V *is a set of vertices. Each vertex represents a state of* $\mathcal{A}$*.*
- $E \subseteq V \times V$ *is a set of arcs. Each arc of* E *represents a successor transition of* $\mathcal{A}$*.*
- $R \subseteq V \times V$ *is another set of arcs that do not necessarily represent successor transitions of* $\mathcal{A}$*, but guarantee the* reachability property*. This property states that if a state* $q_2 \in Q$ *is reachable from*

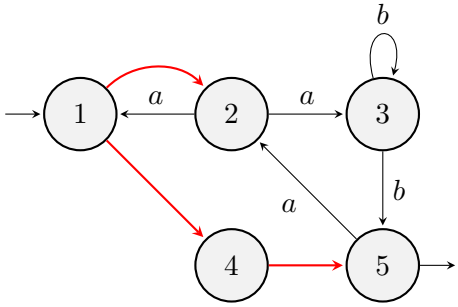
[1] In this master's thesis, an *arc* is a directed edge in a graph. We follow the convention used in [12].

a state $q_1 \in Q$, there exists a path *(i.e., a sequence of vertices linked by arcs from E or R) in G from v_1 to v_2, where v_1 and v_2 represent the states q_1 and q_2, respectively.*

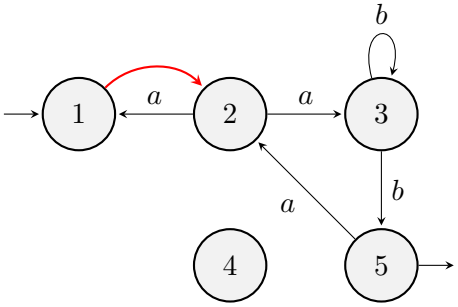
Informally, a reachability graph is simply a graph that represents an automaton, in which arcs from R have been added to guarantee the reachability property. We will often represent concisely an automaton together with its reachability graph, by simply adding (in red, by convention) the arcs of R to the representation of the automaton. In other words, we will allow us to represent a reachability graph by annotating the associated automaton. We will sometimes even forget the distinction between an automaton and a reachability graph (e.g., we will consider that $V = Q$). However, the reader should always keep in mind that:

- The arcs from R (that we will often call "red arcs") are *not* transitions, i.e., these arcs do not have any impact on the language accepted by an automaton.
- A *path* will always refer to a sequence of vertices connected by arcs from E or R, i.e., this term is used to describe a graph, not an automaton.
- The notion of *reachability* is reserved for automata.

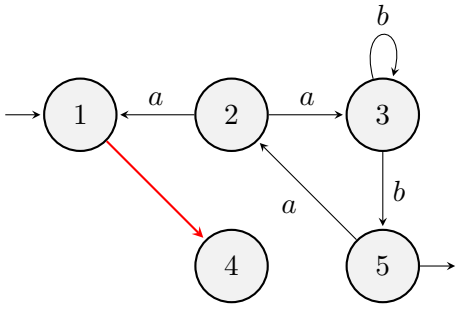
Example 23. *Figure 19 shows three examples: the two first ones are reachability graphs for the automaton $\mathcal{A}_2$ of Example 22, the third one is not. The limit transitions of $\mathcal{A}_2$ are represented only once in Figure 19d.*



(a) Valid reachability graph.



(b) Valid reachability graph.



(c) Incomplete reachability graph.

$$\{3\} \to 2$$
$$1 \to \{2,3,4,5\}$$
$$1 \to \{2,3,5\}$$
$$1 \to \{3,5\}$$
$$1 \to \{2,3\}$$

(d) Limit transitions.

Figure 19: Examples of valid and invalid reachability graphs.

Figures 19a and 19b show two different reachability graphs for $\mathcal{A}_2$. It can be checked that two states q_1 and q_2 such that q_2 is reachable from q_1 are connected by a path. For instance, State 5 is reachable from State 1, which is reflected by the presence of the path $1 \to 2 \to 3 \to 5$.

Note that the reachability graph from Figure 19a admits the path $1 \to 4$*, even though State* 4 *is not reachable from State* 1 *in the automaton* $\mathcal{A}_2$*. If* q_2 *is not reachable from* q_1*, then the reachability property does not say anything about the existence of a path from* q_1 *to* q_2*. This means that it is always possible to build a trivial reachability graph by adding red arcs between any pair of states. Of course, such a reachability graph is not interesting. In the sequel, we will try to minimize the number of added red arcs, as in the reachability graph from Figure 19b. In some sense, this reachability graph is "minimal", because any state* q_2 *which is not reachable from a state* q_1 *is not connected to it by a path.*

Figure 19c is not an example of reachability graph, since State 2 *is reachable from State* 1 *but there does not exist any path from* 1 *to* 2*. The reachability property is violated.*

4.1.3 Limit-set deletion algorithm

We are now ready to detect and delete some useless limit sets in an automaton $\mathcal{A}$. The idea is the following. We scan the limit sets in increasing order of cardinality. For each limit set, we check whether it is included in a strongly connected component[2] of the reachability graph. If it is not the case, then the limit set is useless and will be discarded. Otherwise, the effect of the limit transitions that involve this limit set is taken into account by adding red arcs in the reachability graph. Intuitively, examining the limit sets in increasing order of cardinality guarantees that a limit set P is examined after all the limit sets P' that are strictly included in P. Therefore, if visiting infinitely often the states of P requires to follow a limit transition whose limit set is strictly included in P, then the effect of this limit transition has been considered previously by red arcs added to the reachability graph.

Let us describe an algorithm based on this idea. It will soon appear that this algorithm is not entirely correct, but we will correct it later. Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be an automaton on linear orderings and $P_1, P_2, \ldots, P_m$ its limit sets, sorted in such a way that $P_j \not\subseteq P_i$ if $j > i$. A simple way to achieve that is to sort the limit sets in increasing order of cardinality. Let $G_0 = (V, E, R_0)$ be a reachability graph, where V corresponds to the set of states of $\mathcal{A}$, E to the successor transitions of $\mathcal{A}$, and R_0 is empty. Formally, G_0 is the reachability graph of the automaton $\mathcal{A}$ in which all the limit transitions are ignored. The goal is to detect some useless limit sets amongst $P_1, P_2, \ldots, P_m$. In order to do so, we repeat the following steps for each $i \in \{1, 2, \ldots, m\}$:

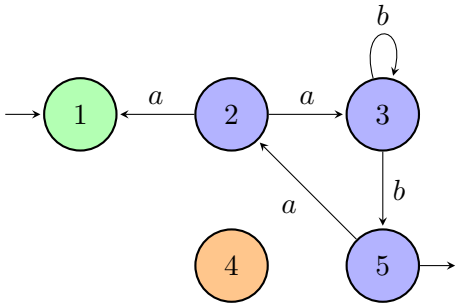
1. Compute the strongly connected component of G_{i-1}.
2. Check whether P_i is included in a strongly connected component of G_{i-1}.
   - If no, the limit set P_i is useless and all the limit transitions that rely on it can be removed from Δ. We can then set $G_i = G_{i-1}$.
   - If yes, then one must add red arcs in the reachability graph. Initially, set $R_i = R_{i-1}$ and choose an arbitrary state $p \in P_i$ that we call *representative* of P_i. Then, for all left-limit transitions $P_i \to q$ in Δ, an arc (p, q) is added in R_i. Similarly, for all right-limit transitions $q \to P_i$ in Δ, an arc (q, p) is added in R_i. Finally, G_i is defined as (V, E, R_i).

Example 24. *Consider the automaton from Example 23. We first sort its limit sets in increasing order of cardinality:*

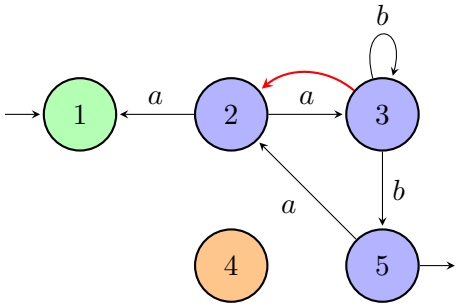
$$P_1 = \{3\}, \quad P_2 = \{2, 3\}, \quad P_3 = \{3, 5\}, \quad P_4 = \{2, 3, 5\}, \quad P_5 = \{2, 3, 4, 5\}.$$

The reachability graph G_0 *is depicted in Figure 20a. Colours are used to highlight the strongly connected components. The limit set* P_1 *is included in the blue strongly connected component. It is thus not*

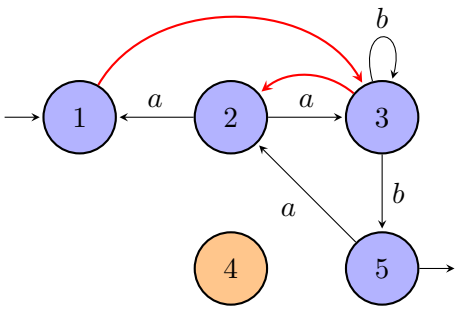
[2] A strongly connected component is a maximal subset of vertices such that any vertex is connected to any other vertex by a path. When we will need to refer to a (non necessarily maximal) set of vertices such that any vertex is connected to any other vertex by a path, we will use the notion of *non maximal strongly connected component*.



(a) Reachability graph G_0.



(b) Reachability graph G_1.



(c) Reachability graph $G_2 = G_3 = G_4 = G_5$.

Figure 20: Successive graphs obtained during the limit-set deletion algorithm.

deleted and the arc $3 \to 2$ *is added in the reachability graph. This yields* G_1 *(see Figure 20b). The strongly connected components remain the same as in* G_0*.*

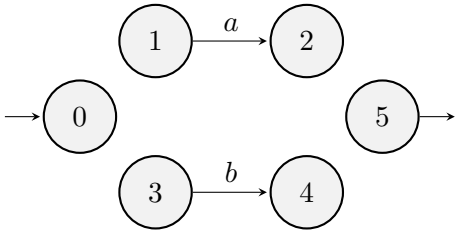
The limit sets P_2*,* P_3 *and* P_4 *are all included in the blue strongly connected component of* G_1*. None of them is deleted. The arc* $1 \to 3$ *is then added to the reachability graph. Since state* 3 *belongs to* P_2, P_3 *and* P_4*, this state is a common representative of these three limit sets, hence this single red arc suffices to translate the limit transitions* $1 \to P_2$*,* $1 \to P_3$ *and* $1 \to P_4$*. The resulting reachability graph* $G_2 = G_3 = G_4$ *is depicted in Figure 20c. The green and the blue strongly connected components have been merged.*

The limit set P_5 *spans the blue and the orange strongly connected components. Therefore,* P_5 *is useless and the limit transition* $1 \to P_5$ *is removed from* Δ*. The reachability graph* G_5 *is equal to* G_4*.*

The output of the algorithm is an automaton that accepts the same language as the initial one, but with one less limit transition, since $1 \to P_5$ *has been deleted.*

Example 24 shows a case where the limit-set deletion algorithm works nicely. Intuitively, a red arc encodes the effect of a limit transition whose limit set has not been deleted. If this limit transition is $P \to q$, then the red arc connects the strongly connected component in which P is included to the one in which q is included (and similarly for a limit transition $q \to P$).

However, our algorithm is not entirely correct. Consider the automaton in Figure 21. We saw in Chapter 1 that this automaton accepts the language $sh(a, b)$. It has a single limit set $\{1, 2, 3, 4\}$ that is not included in a strongly connected component of G_0. Our algorithm would thus wrongly remove the limit transitions $0, 2, 4 \to \{1, 2, 3, 4\} \to 1, 3, 5$. Therefore, it would output an automaton that accepts the empty language. The problem comes from the fact that the limit set $P = \{1, 2, 3, 4\}$ can be visited infinitely often by following limit transitions whose limit set is P. Informally, this is a kind



$$0, 2, 4 \to \{1, 2, 3, 4\} \to 1, 3, 5$$

Figure 21: Counter-example.

of auto-reference ("the usefulness of P is a consequence of the usefulness of P"). For this reason, such a limit set is said to be *auto-referencing* in the sequel. Currently, our algorithm does not handle this case, because it checks the usefulness of P_i based on G_{i-1}. Before correcting this mistake, we need to understand when a limit set is auto-referencing. Theorem 4 guarantees that in some circumstances, auto-referencing does not occur.

Theorem 4. *Let $\mathcal{A}$ be an automaton on linear orderings and $P_1, P_2, \ldots, P_m$ its limit sets, sorted in such a way that $P_j \not\subseteq P_i$ if $j > i$. Let G_{k-1} be a reachability graph taking into account the limit transitions that use the $k-1$ first limit sets, for some $k \in \{1, 2, \ldots, m\}$. Assume that*

- *the limit set P_k appears in right-limit transitions or in left-limit-transitions, but not both in right-limit and left-limit transitions,*
- *and P_k is not included in a strongly connected component of G_{k-1}.*

Then, P_k is a useless limit set.

Proof. The index k of P_k is omitted to simplify the notations. Without loss of generality, we assume that P appears in left-limit transitions and not in right-limit transitions. The proof that follows can be easily adapted for the dual case.

If $\mathcal{A}$ does not accept any word, then P is useless (see Definition 7). If $\mathcal{A}$ accepts a non-empty language, consider an accepted word w of length J and a run ρ reading w in $\mathcal{A}$. We want to show that the set

$$\widehat{J}_{lim} = \left\{ c \in \widehat{J} \mid \lim_{c^-} \rho = P \right\}$$

is empty.

We start by proving the following intermediate result. Let $\widehat{J}_P$ be a subordering of $\widehat{J}$ such that

$$\forall c_1 < c_2 \in \widehat{J}_P : \quad \big\{ \rho(c) \mid c_1 \leq c \leq c_2 \big\} = P. \tag{4.1}$$

It can be shown that this subordering has a least element. Assume that it it not the case. Let us consider the following other subordering of $\widehat{J}$

$$\widehat{J}_{left} = \big\{ c \in \widehat{J} \mid c < c_P \text{ for all } c_P \in \widehat{J}_P \big\}$$

and the cut (K, L) of $\widehat{J}$ defined as

$$K = \widehat{J}_{left}, \quad L = \{ c \in \widehat{J} \mid c_P \leq c \text{ for some } c_P \in \widehat{J}_P \}.$$

Since $\widehat{J}_P$ does not have a least element, L does not have a least element either. Knowing that $\widehat{J}$ is complete for all linear orderings J, this implies that $\widehat{J}_{left}$ must have a greatest element (otherwise, (K, L) would be a gap of $\widehat{J}$). Let us denote by c_{max} this greatest element. Note that c_{max} cannot have a successor, otherwise this successor would be the least element of L. Therefore, the run ρ leaves the state $\rho(c_{max})$ by following a right-limit transition. From the definitions of $\widehat{J}_P$ and c_{max}, it follows that

$$\lim_{c_{max}^+} \rho = P,$$

which is a contradiction, since P does not appear in a right-limit transition of $\mathcal{A}$. Therefore, $\widehat{J}_P$ must have a least element.

We now go back to the proof of Theorem 4. Assume that the set

$$\widehat{J}_{lim} = \left\{ c \in \widehat{J} \mid \lim_{c^-} \rho = P \right\}$$

has at least one element. The idea is to partition this set into subsets that all have Property 4.1 as $\widehat{J}_P$. Formally, we define a relation $\sim_P$ between elements of $\widehat{J}_{lim}$ as follows:

$$c_1 \sim_P c_2 \iff c_1 = c_2 \quad \vee \quad \big\{\rho(c) \mid \min(c_1, c_2) \leq c \leq \max(c_1, c_2)\big\} = P$$

It can be checked that $\sim_P$ is an equivalence relation. Since $\widehat{J}_{lim}$ has at least one element, the relation $\sim_P$ induces at least one equivalence class. Let S be a set of elements that all belong to the same equivalence class. We know that S is a subordering of $\widehat{J}$ that satisfies Property 4.1. Therefore, S contains a least element c_{min} such that $\lim_{c_{min}^-} \rho = P$. This implies that one can find $c_1 \in \widehat{J}$ such that $c_1 < c_{min}$ and

$$P = \big\{\rho(c) \mid c_1 \leq c < c_{min}\big\}.$$

Moreover, for all c such that $c_1 \leq c < c_{min}$, we have $\lim_{c^-} \rho \neq P$. Otherwise there would exist an element of $\widehat{J}_{lim}$ that it smaller than c_{min} and belongs to the same equivalence class (which would imply that c_{min} is not the least element of S). In other words, we proved that the states contained in P can be visited infinitely often without following a limit transition in which P is a left-limit set.

This implies that it is possible to visit exactly the states of P infinitely often by only following limit transitions whose limit set is strictly included in P. Therefore, for all $q, q' \in P$, there exists a path linking q to q' in G_{k-1}. We conclude that P is included in a strongly connected component of G_{k-1}, which is a contradiction. □

The limit-set deletion algorithm can now be corrected. Assume that the $k-1$ first limit sets have been studied and that we are now examining the limit set P_k by means of the reachability graph G_{k-1}. Previously, the first step was to compute the strongly connected components of G_{k-1}. We now add a preliminary step that consists in checking whether P_k could be auto-referencing as in Figure 21. This might be the case only if there exist at least two limit transitions $P_k \to q$ and $q' \to P_k$ such that $q, q' \in P_k$. Indeed, by Theorem 4, P_k must appear both in a right-limit and in a left-limit transition. Moreover, the source (resp. the destination) of the right-limit (resp. left-limit) transition must belong to P_k, otherwise this transition cannot be followed when visiting exactly the states in P_k.

Assume now that two such limit transitions exist. Before computing the strongly connected components of G_{k-1}, one needs to add red arcs as if P_k was known to be useful. This yields a new reachability graph G'_{k-1}. Afterwards, the strongly connected components of G'_{k-1} can be computed. If P_k is not included in a strongly connected component, G_k is set to G_{k-1} and P_k is deleted. Otherwise, G_k is set to G'_{k-1} and P_k is not deleted. Regarding the automaton from Example 24, this new algorithm behaves as the old incorrect one. But the automaton from Figure 21 is now correctly handled, as the next example shows.

Example 25. *Consider again the automaton from Figure 21. The updated algorithm starts by checking whether the limit set* $\{1,2,3,4\}$ *might be auto-referencing. This might be the case, since the limit transitions* $2 \to \{1,2,3,4\}$ *and* $\{1,2,3,4\} \to 1$ *belong to* Δ *and* $1,2 \in \{1,2,3,4\}$*. Therefore, we have to choose a representative* p *of* $\{1,2,3,4\}$ *and add the red arcs*

$$0 \to p, \quad 2 \to p, \quad 4 \to p$$

$$p \to 1, \quad p \to 3, \quad p \to 5.$$

Let us choose $p = 1$ *(the conclusions would be the same if* p *was chosen to be equal to* 2, 3 *or* 4*). We get the reachability graph* G'_0 *represented in Figure 22. The limit set* $\{1,2,3,4\}$ *is included in the blue*

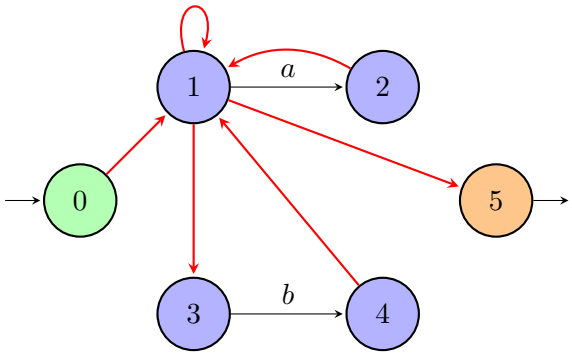


Figure 22: Reachability graph G'_0.

strongly connected component. Therefore, the limit transitions $0,2,4 \to \{1,2,3,4\} \to 1,3,5$ *are not removed from* Δ *and* G_1 *is set to* G'_0*.*

The next result, together with Theorem 4, guarantees that, under some assumptions, a limit set that is deleted by our updated algorithm is useless.

Theorem 5. *Let* $\mathcal{A}$ *be an automaton on linear orderings and* $P_1, P_2, \ldots, P_m$ *its limit sets, sorted in such a way that* $P_j \not\subseteq P_i$ *if* $j > i$*. Let* G_{k-1} *be a reachability graph taking into account the limit transitions that use the* $k-1$ *first limit sets, for some* $k \in \{1,2,\ldots,m\}$*. Let* G'_{k-1} *be another graph obtained by first choosing a representative* p *of* P_k *and then by adding a red arc* $p \to q$ *(resp.* $q \to p$*) for each limit transitions* $P_k \to q$ *(resp.* $q \to P_k$*) in* Δ*. Assume that* P_k *is not included in a strongly connected component of* G'_{k-1}*. Then,* P_k *is a useless limit set.*

Proof sketch. By assumption, P_k spans at least two strongly connected components of G'_{k-1}. Let $q_1, q_2 \in Q$ be two states that belong to different strongly connected components of G'_{k-1}. Note that P_k spans at least two strongly connected components of G_{k-1} as well.

Assume that P_k is useful. Therefore, q_1 is reachable from q_2 and conversely. We will show that G'_{k-1} admits a path from q_1 to q_2. If there exists such a path in G_{k-1}, then this path exists in G'_{k-1} as well, and we are done. If the reachability graph G_{k-1} does not contain such a path, then this means that reading a word $w_{q_1 \to q_2}$ in $\mathcal{A}$ by starting in q_1 and ending in q_2 requires to follow limit transitions where P_k is involved. From Theorem 4, we know that this is only possible if both left-limit and right-limit transitions involving P_k are followed (since P_k is not included in a strongly connected component of G_{k-1}). Now recall that all the sources of the limit transitions involving P_k are connected by a red arc to the representative of P_k, which itself is connected by a red arc to all the destinations of these limit transitions. It is then possible to build a path from q_1 to q_2 in G'_{k-1}.

Similarly, one can find a path in G'_{k-1} from q_2 to q_1. This means that q_1 and q_2 both belong to the same strongly connected component of G'_{k-1}, which is a contradiction. □

We can now describe formally a correct limit-set deletion algorithm. It consists in following these steps, starting from an automaton $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ with m limit sets:

1. Sort the limit sets $P_1, P_2, \ldots, P_m$ in such a way that $P_j \not\subseteq P_i$ if $j > i$. Note that there could exist several ways to do that.
2. Define $G_0 = (V, E, R_0)$ where each vertex in V corresponds to a state of $\mathcal{A}$, each arc in E corresponds to a successor transition in Δ and R_0 is empty.
3. For each $i \in \{1, 2, \ldots, m\}$, do the following:
   (a) If there exist limit transitions $q \to P_i$ and $P_i \to q'$ such that $q, q' \in P_i$:
      i. Choose a representative p of P_i and add a red arc $q \to p$ (resp. $p \to q$) for each right-limit (resp. left-limit) transition in Δ. This yields a graph G'_{i-1}.
      ii. Compute the strongly connected components of G'_{i-1}.
      iii. If P_i is included in a strongly connected component, set G_i to G'_{i-1}. Otherwise, remove from Δ all the limit transitions that involve P_i and set G_i to G_{i-1}.
   (b) Otherwise:
      i. Compute the strongly connected components of G_{i-1}.
      ii. If P_i is included in a strongly connected component, choose a representative p of P_i and add a red arc $q \to p$ (resp. $p \to q$) for each right-limit (resp. left-limit) transition in Δ. This yields a graph G_i. Otherwise, remove from Δ all the limit transitions that use P_i and set $G_i = G_{i-1}$.

At the end of the algorithm, we get an automaton that accepts the same language as $\mathcal{A}$.

Theorem 6. *The limit-set deletion algorithm is correct.*

Proof sketch. It must be shown that any limit set that is deleted (i.e., such that the limit transitions in which it is involved are deleted) is useless. The proof proceeds in two steps:

- Assume that for some $i \in \{1, 2, \ldots, m\}$, G_{i-1} is a reachability graph for the automaton $\mathcal{A}$ where only the first $i-1$ limit sets are taken into account. It follows from Theorem 4 and 5 that P_i is deleted only if it is useless.
- It can then be shown by induction that for all $k \in \{0, 1, 2, \ldots, m-1\}$, G_k is indeed a reachability graph for the automaton $\mathcal{A}$ where only the first k limit sets are taken into account. This is true for $k = 0$. Assuming that it holds for G_{k-1} with $k > 0$, it also holds for G_k, for the following reasons:
  - If P_k is deleted, then $G_k = G_{k-1}$. Since P_k is useless in this case, G_k can indeed be chosen equal to G_{k-1}.
  - If P_k is not deleted, then G_k is obtained by adding red arcs to G_{k-1}. It is guaranteed that P_k is included in a strongly connected component of G_k (and even in a strongly connected component of G_{k-1} if P_k is not auto-referencing). Furthermore, all the sources of the right-limit transitions involving P_k are connected to the representative of P_k via a red arc. Similarly, all the destinations of the left-limit transitions involving P_k can be reached from this representative via a red arc. By induction hypothesis, it then follows that two states q_1, q_2 such that q_2 is reachable from q_1 by following successor transitions and limit transitions involving $P_1, P_2, \ldots, P_k$ are connected through a path in G_k.

□

4.1.4 Deleting limit sets during the intersection operation

In this part, we discuss how the ideas introduced previously can be applied in the context of computing the intersection of automata on linear orderings. We also motivate some choices that have been made in the limit-set deletion algorithm.

In Chapter 3, we saw that the intersection of automata on linear orderings yields an automaton with a prohibitively large number of limit sets. Deleting useless limit sets could partially solve this problem. However, the limit-set deletion algorithm described in Section 4.1.3 needs as input an exhaustive list of limit sets. Some of them are then deleted, but *after* being enumerated. In other words, this algorithm does not really solve the problem of the large number of limit sets in the intersection automaton, since it needs to run after all the limit transitions have been computed.

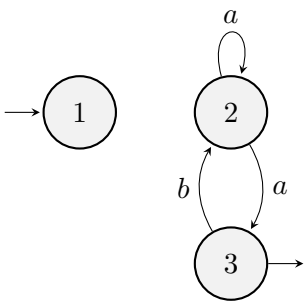
Therefore, to be useful in practice, the limit-set deletion algorithm must be adapted to work in tandem with the intersection algorithm for automata. This adaptation is conceptually straightforward. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the operands. We saw in Chapter 3 that the first step of the naive binary intersection algorithm is to compute the intersection of $\mathcal{A}_1$ and $\mathcal{A}_2$ as if they were finite-word automata. Next, instead of enumerating the limit sets of the automaton $\mathcal{A}$ and deleting some of them afterwards, we will enumerate the strongly connected components of its current reachability graph and, for each of them, compute the limit sets that are included in it. Intuitively, this amounts to computing the strongly connected components and filling them with limit sets, rather than enumerating the limit sets and checking whether each of them is included in a strongly connected component of the current reachability graph.

Let us now describe this idea more formally. Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be the automaton resulting from the intersection of $\mathcal{A}_1$ and $\mathcal{A}_2$ where the limit transitions have been ignored, and let $G_0 = (V, E, R_0)$ be the reachability graph of $\mathcal{A}$. We introduce a function Λ that maps a subset of Q (or equivalently, a subset of V) to the set of limit sets that should be included in Q in the final intersection automaton. We do not discuss now how such a function can be implemented (this question is addressed in Chapter 5). For now, we simply explain how such a function can be used in the adapted limit-set deletion algorithm. This algorithm repeats the following steps, starting with $i = 0$:

1. Compute the strongly connected components of G_i.
2. For each strongly connected component $S \subseteq Q$ of G_i:
   (a) Compute $\Lambda(S)$.
   (b) Add all the limit sets from $\Lambda(S)$ to the automaton $\mathcal{A}$, as well as the corresponding limit transitions.
   (c) Add the corresponding red arcs in G_i (without recomputing the strongly connected components).
3. Let G_{i+1} be the reachability graph G_i in which the red arcs from Step 2 have been added. Compute the strongly connected components of G_{i+1}. If these are the same as in Step 1, the algorithm terminates.

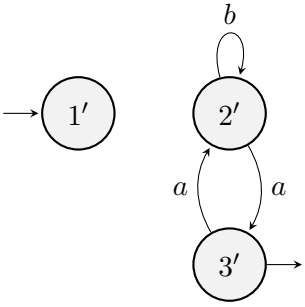
Let us illustrate this adapted algorithm by means of the example already used in Chapter 3.

Example 26. *Consider again the two automata in Figure 23. Computing their intersection as if they were finite-word automata yields the automaton whose reachability graph G_0 is depicted in Figure 24. The reachability graph G_0 has 3 strongly connected components. Each of them is examined separately:*



$1 \to \{2, 3\}$

(a) First operand.



$1' \to \{2', 3'\}$

(b) Second operand.

Figure 23

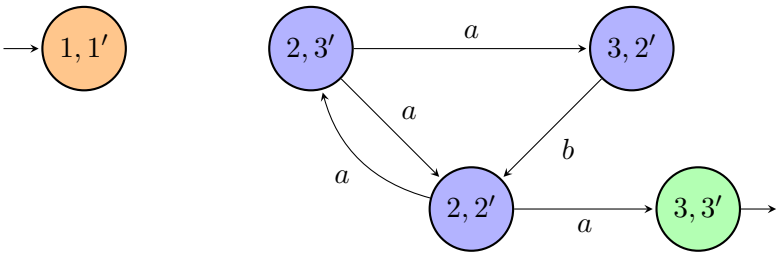


Figure 24: Reachability graph G_0.

- *It is clear that no limit set can be included in the orange strongly connected component* $\{(1, 1')\}$. *In other words,* $\Lambda\Big(\big\{(1, 1')\big\}\Big) = \emptyset$.
- *Similarly, no limit set can be included in the green strongly connected component* $\big\{(3, 3')\big\}$.
- *Finally, one needs to compute the limit sets that are included in the blue strongly connected component* $\big\{(2, 3'), (3, 2'), (2, 2')\big\}$. *We know that they must cover exactly* $\{2, 3\}$ *and* $\{2', 3'\}$, *hence these limit sets are*

$$\Lambda\Big(\big\{(2, 3'), (3, 2'), (2, 2')\big\}\Big) = \Big\{\big\{(2, 3'), (3, 2')\big\}, \big\{(2, 3'), (3, 2'), (2, 2')\big\}\Big\}.$$

Therefore, the limit transitions $(1, 1') \to \big\{(2, 3'), (3, 2')\big\}$ *and* $(1, 1') \to \big\{(2, 3'), (3, 2'), (2, 2')\big\}$ *need to be added to the transitions relation* Δ. *The reachability graph* G_1 *is obtained by adding the red arc* $(1, 1') \to (2, 3')$ *(this single red arc suffices to translate the effect of the two added limit transitions). It is shown in Figure 25. The strongly connected components in* G_1 *are the same as in* G_0. *The algorithm thus terminates.*

Interestingly, the automaton resulting from the intersection has only 2 *limit transitions. In Example 21, the naive intersection method produced an automaton with* 7 *limit transitions.*

We now introduce a new intersection problem that is slightly more complex, and that we will keep as running example in the sequel.

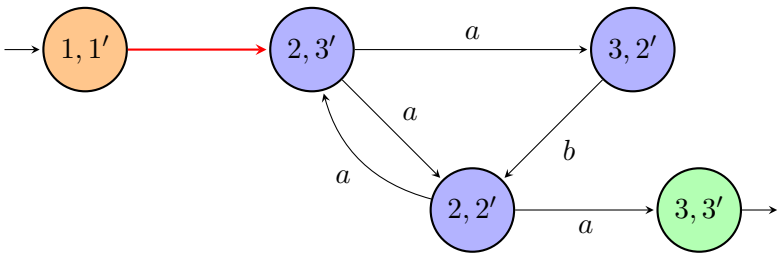
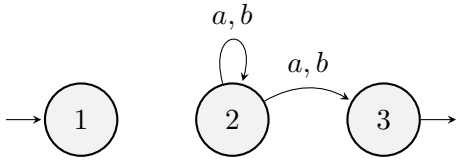


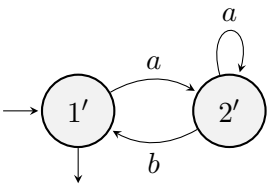
Figure 25: Reachability graph G_1.



$3 \to \{2\}$

$1 \to \{2, 3\}$

(a) First operand.



$1' \to \{1', 2'\}$

(b) Second operand.

Figure 26

Example 27. *Let us apply our algorithm to compute the intersection of the automata in Figure 26. The first operand accepts the language* $\left((a+b)^{-\omega}\right)^{-\omega}$ *that contains all the words of length* $\sum_{j\in(-\mathbb{N})}(-\mathbb{N})$ *on the alphabet* $\{a, b\}$*. The second operand accepts the language* $(aa^*b)^{-\sharp}$ *(this automaton can be obtained by first considering an automaton accepting* aa^*b*, and then applying the construction outlined in [7] to add the effect of the reverse ordinal iteration). Since the linear ordering* $\sum_{j\in(-\mathbb{N})}(-\mathbb{N})$ *is a reverse ordinal, the intersection of these two operands should accept the language* $\left((aa^*b)^{-\omega}\right)^{-\omega}$*.*

The reachability graph G_0 *is depicted in Figure 27 (as always, it is obtained by first computing the intersection of the operands as if they were finite-word automata).* G_0 *has 4 strongly connected components. Clearly, the only one in which a limit set can be included is the blue one, and it contains the limit set* $\{(2, 1'), (2, 2')\}$ *(that covers exactly* $\{2\}$ *and* $\{1', 2'\}$*). This means that the limit transition* $(3, 1') \to \{(2, 1'), (2, 2')\}$ *is added in the transition relation, and that a red arc* $(3, 1') \to (2, 2')$ *is added in the reachability graph. This yields* G_1*, represented in Figure 28. The strongly connected components in* G_1 *are not the same as in* G_0*, hence the algorithm continues. The blue strongly connected components now contains the limit set* $\{(2, 1'), (2, 2'), (3, 1')\}$ *(that covers exactly* $\{2, 3\}$ *and* $\{1', 2'\}$*). The limit*

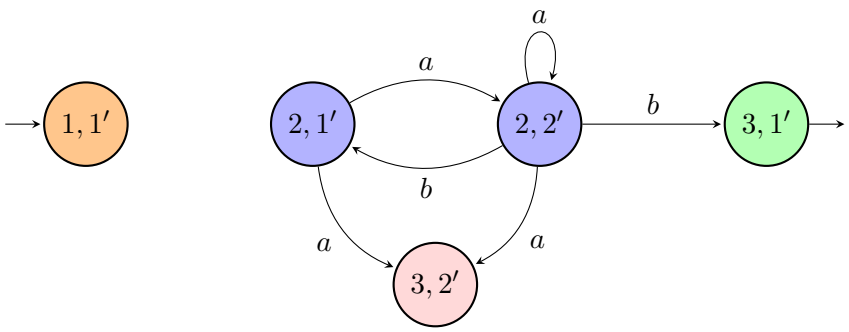


Figure 27: Reachability graph G_0.

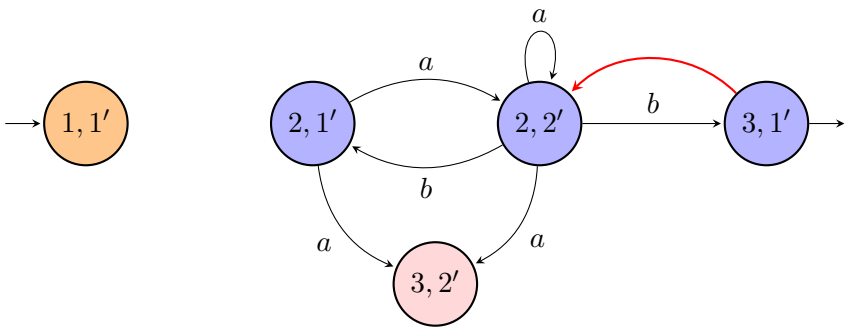


Figure 28: Reachability graph G_1.

transition $(1,1') \to \{(2,1'),(2,2'),(3,1')\}$ *is thus added in* Δ*, and the a red arc* $(1,1') \to (2,1')$ *is added in the reachability graph, which results in* G_2 *(see Figure 29). The strongly connected components have not been modified by this newly added arc, hence the algorithm terminates. We built an automaton* $\mathcal{A}$ *that has only two limit transitions:* $(3,1') \to \{(2,1'),(2,2')\}$ *and* $(1,1') \to \{(2,1'),(2,2'),(3,1')\}$*. It can be checked that this automaton accepts the language* $\left((aa^*b)^{-\omega}\right)^{-\omega}$

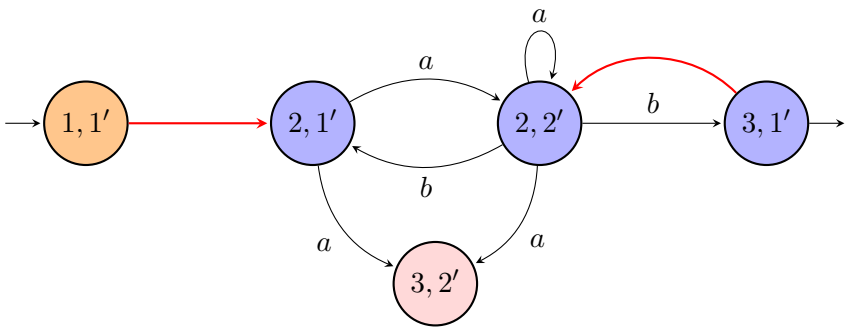


Figure 29: Reachability graph G_2.

Two remarks are in order. First, the adapted algorithm that we illustrated in Examples 26 and 27 does not include any mechanism that handles the auto-referencing limit sets correctly. We will see later how this problem can be solved in practice. Secondly, throughout this section about limit-set deletion, we used the fact that a useful limit set must be included in a strongly connected component of a reachability graph. In fact, this statement could be easily strengthened: a useful limit set must be *exactly equal* to a (possibly non maximal) strongly connected component, i.e., for all states q_1, q_2 in the limit set, it should be possible to read a word by starting in q_1, ending in q_2 and visiting only states that belong to the limit set. Applying this criterion would allow us to delete more limit sets. For instance, in Example 26, the useless limit set $\{(2,3'),(3,2')\}$ would be deleted (on the contrary, our algorithm does not delete it, because it is included in the blue strongly connected component). We did not use such a strengthened criterion, for the following reasons:

- Such a criterion is more difficult to check, since it is not sufficient to compute once for all the maximal strongly connected components of a reachability graph. Instead, it becomes necessary to enumerate all the subsets of states that form a (possibly non maximal) strongly connected component. This is significantly more difficult, and we do not want to carry out such an operation too often. We will see that the algorithm that deletes some useless limit sets is used extensively, hence it it is preferable to keep it efficient.

- For this criterion to be applied correctly, adding red arcs in reachability graphs does not suffice. Intuitively, it would be necessary to know that state q_1 is reachable from state q_2 by only visiting

states from some set S. But this information is not captured by red arcs, since they only focus on reachability.

4.2 Concise representation of limit transitions

In this section, we tackle the problem of representing limit transitions concisely. Section 4.2.1 introduces finite base automata and discusses the advantages of this representation. Then, Section 4.2.2 presents a formalism that is inspired from finite base automata, but is better suited for computing efficiently the intersection of automata on linear orderings.

4.2.1 Finite base automata

In [15], Rabinovich introduces a new form of automata called *finite base automata* (that we will often simply call *base automata*). In [15], these automata are proven to be equivalent to usual automata on linear orderings introduced in Definition 1. In the sequel, automata satisfying Definition 1 are sometimes called *plain automata*, to avoid any confusion with finite base automata.

Finite base automata are mostly used in [15] for their theoretical properties. But in this master's thesis, we consider them as an alternative way of representing or encoding automata on linear orderings.

Definition 10. *A* finite base automaton *is a tuple* $(B, Q, \Sigma, \Delta, I, F)$ *where:*

- B *is a finite set of* base elements. *The set* B *is the* base *of the base automaton.*
- $Q \subseteq 2^B$ *is a finite set of states. Each state is a subset of the base* B*.*
- Σ *is a finite alphabet.*
- $\Delta \subseteq (Q \times \Sigma \times Q) \cup \big(Q \times 2^B\big) \cup \big(2^B \times Q\big)$ *is a transition relation. It contains successor transitions* $(q, \sigma, q') \in Q \times \Sigma \times Q$*,* base right-limit *transitions* $(q, P_B) \in Q \times 2^B$ *and* base left-limit *transitions* $(P_B, q) \in 2^B \times Q$*.*
- $I \subseteq Q$ *is a set of initial states.*
- $F \subseteq Q$ *is a set of final states.*

As usual, the transitions from $Q \times \Sigma \times Q$ are denoted by $q \xrightarrow{\sigma} q'$, where $q, q' \in Q$ and $\sigma \in \Sigma$. The transitions from $Q \times 2^B$ (resp. $2^B \times Q$) are base right-limit transitions (resp. base left-limit transitions), denoted by $q \to P_B$ (resp. $P_B \to q$) where $q \in Q$ and $P_B \in 2^B$. We will often refer to P_B as a *base right-limit set* or *base left-limit set.*

There are two main differences compared to Definition 1. First, the states are now defined as sets of base elements. Secondly, the usual left-limit and right-limit transitions are replaced by base left-limit and base right-limit transitions. The semantics of base automata is given in Definition 11.

Definition 11. *A* run *of a finite base automaton* $\mathcal{A} = (B, Q, \Sigma, \Delta, I, F)$ *reading a word* w *of length* J *is a mapping* $\rho : \widehat{J} \to Q$ *such that:*

- $\rho(\widehat{J}_{min}) \in I$*.*
- $\rho(\widehat{J}_{max}) \in F$*.*
- *For every pair of cuts* $c_1 = (K_1, L_1)$ *and* $c_2 = (K_2, L_2)$ *such that* c_2 *is the successor of* c_1 *in* $\widehat{J}$*,* $(\rho(c_1), w(j), \rho(c_2)) \in \Delta$*, where* $j \in J$ *is such that* $K_2 = K_1 \cup \{j\}$*.*

- *For every cut* $c \neq \widehat{J}_{min}$ *which does not have any predecessor,* $\left(\lim^B_{c^-} \rho, \rho(c)\right) \in \Delta$*, where* $\lim^B_{c^-} \rho$ *is the* base left-limit set *of* ρ *in* c*, defined as*

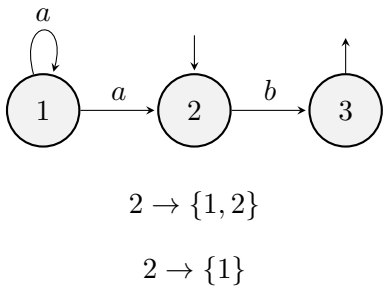
$$\lim^B_{c^-} \rho = \Big\{ b \in B \mid \Big(\forall c_1 < c\Big)\Big(\exists c_2\Big)\Big(c_1 < c_2 < c \wedge \big((\forall c_3)\ \ c_2 < c_3 < c \implies b \in \rho(c_3)\big)\Big)\Big\}.$$

- *For every cut* $c \neq \widehat{J}_{max}$ *which does not have any successor,* $\left(\rho(c), \lim^B_{c^+} \rho\right) \in \Delta$*, where* $\lim^B_{c^+} \rho$ *is the* base right-limit set *of* ρ *in* c*, defined as*

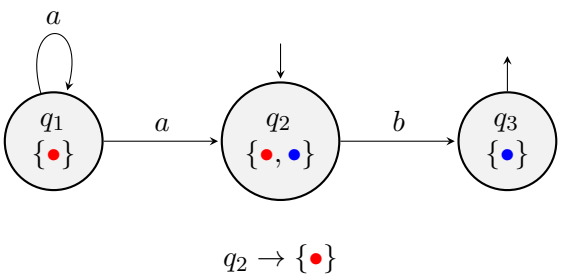
$$\lim^B_{c^+} \rho = \Big\{ b \in B \mid \Big(\forall c_1 > c\Big)\Big(\exists c_2\Big)\Big(c_1 > c_2 > c \wedge \big((\forall c_3)\ \ c_2 > c_3 > c \implies b \in \rho(c_3)\big)\Big)\Big\}.$$

Intuitively, the base left-limit transition $P_B \to \rho(c)$ can be followed if P_B contains all the base elements that are common to each state that is visited arbitrarily close to c. Informally, $P_B \to \rho(c)$ is enabled iff P_B is the intersection of the states (considered as subsets of the base) that belong to the left-limit set of ρ in c, in the sense of Definition 2. This link between base limit transitions and usual limit transitions also holds for right-limit transitions. In some sense, base automata are less explicit than usual automata on linear orderings, because base limit sets are not simply sets of states. Instead, a base limit set is the set of base elements that are common to each state appearing infinitely often at the left or at the right of a cut, and arbitrarily close to it.

Example 28. *Figure 30 shows two equivalent automata. The first one is a plain automaton* $\mathcal{A}$ *that accepts the language* $(a^{-\omega})^{-\omega}\, b + (a^{-\omega})^*\, b$*, that can also be concisely written* $(a^{-\omega} + \varepsilon)^{-\omega}\, b$*. The second automaton is a finite base automaton* $\mathcal{A}_B$ *that accepts the same language. Its base is* $\{\bullet, \bullet\}$*. Each of its states is a subset of the base. To make the correspondence between the two automata explicit, the state* $i \in \{1, 2, 3\}$ *in* $\mathcal{A}$ *corresponds to the state labelled by* q_i *in* $\mathcal{A}_B$*. This label is not strictly necessary, since a state in* $\mathcal{A}_B$ *is entirely defined by a subset of the base.*



(a) Plain automaton $\mathcal{A}$.



(b) Finite base automaton $\mathcal{A}_B$.

Figure 30: Two equivalent automata accepting the language $(a^{-\omega} + \varepsilon)^{-\omega}\, b$.

The two limit transitions $2 \to \{1, 2\}$ *and* $2 \to \{1\}$ *of* $\mathcal{A}$ *are encoded into a single base limit transition* $q_2 \to \{\bullet\}$ *in* $\mathcal{A}_B$*. Note that* $q_2 \to \{\bullet\}$ *should actually be written* $\{\bullet, \bullet\} \to \{\bullet\}$*, but this notation is ambiguous because it does not make explicit the fact that* $\{\bullet, \bullet\}$ *is a state and that* $\{\bullet\}$ *is a subset of the base that does not need to be a state. Hence we use the label* q_2 *rather than* $\{\bullet, \bullet\}$*, to emphasize this distinction.*

One can check that both automata accept the same language. Let us compute all the possible intersections between states of $\mathcal{A}_B$*, in order to translate the base limit transitions into plain limit transitions:*

$$q_1 \cap q_2 \cap q_3 = \{\ \}$$
$$q_1 \cap q_3 = \{\ \}$$
$$q_1 = \{\bullet\}$$
$$q_3 = \{\bullet\}$$

$$q_1 \cap q_2 = \{\bullet\}$$
$$q_2 \cap q_3 = \{\bullet\}$$
$$q_2 = \{\bullet, \bullet\}$$

Two of these intersections yield $\{\bullet\}$*:* $q_1 \cap q_2$ *and* q_1*. The base limit transition* $q_2 \to \{\bullet\}$ *thus indeed encodes the two limit transitions* $2 \to \{1, 2\}$ *and* $2 \to \{1\}$*.*

Note that there are other ways of encoding the plain automaton $\mathcal{A}$ *as a finite base automaton. For instance, we could have set* $q_1 = \{\bullet, \bullet\}$*,* $q_2 = \{\bullet\}$ *and* $q_3 = \{\ \}$*. In this case, it would have been impossible to encode the limit transitions* $2 \to \{1, 2\}$ *and* $2 \to \{1\}$ *into a single base limit transition.*

The main practical advantage of using base automata is the easiness of computing their intersection. Let us describe formally how to compute the intersection $\mathcal{A}_B = (B, Q, \Sigma, \Delta, I, F)$ of two base automata $\mathcal{A}_{B_1} = (B_1, Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $\mathcal{A}_{B_2} = (B_2, Q_2, \Sigma, \Delta_1, I_2, F_2)$:

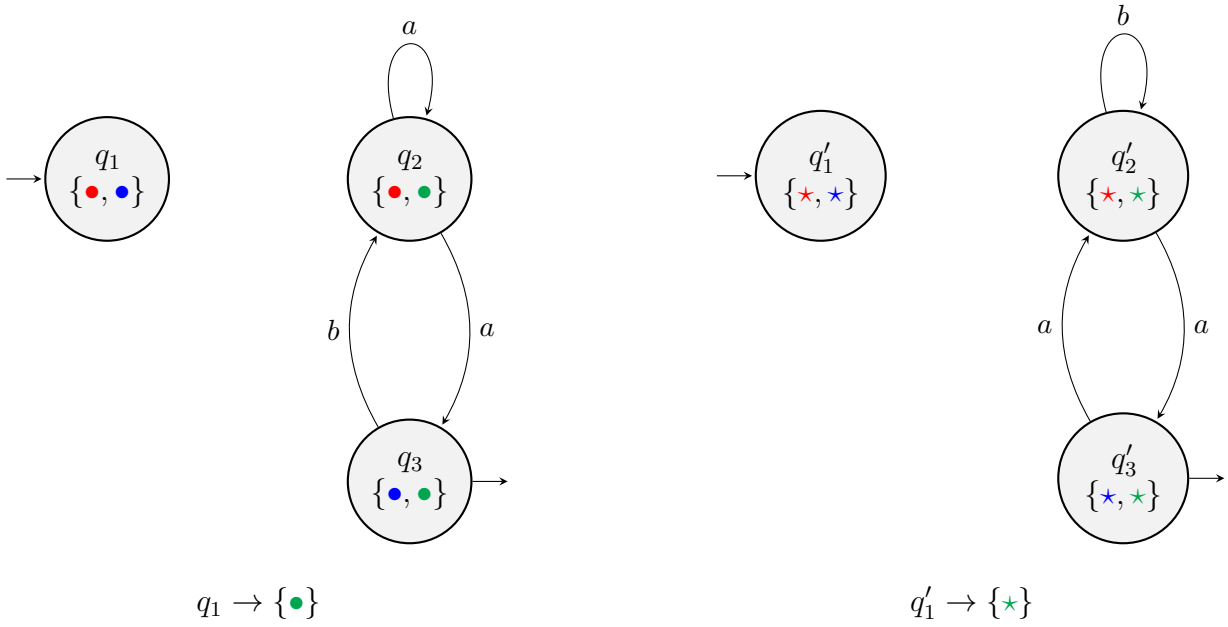
- $B = B_1 \cup B_2$.
- $Q \subseteq 2^B$.
- $I = \{q_1 \cup q_2 \mid q_1 \in I_1 \wedge q_2 \in I_2\}$.
- $F = \{q_1 \cup q_2 \mid q_1 \in F_1 \wedge q_2 \in F_2\}$.
- The successor transition $(q_1 \cup q_2) \xrightarrow{\sigma} (q_1' \cup q_2')$ with $\sigma \in \Sigma$ belongs to Δ iff $q_1 \xrightarrow{\sigma} q_1'$ belongs to Δ_1 and $q_2 \xrightarrow{\sigma} q_2'$ belongs to Δ_2.
- The base left-limit transition $P_B \to q$ belongs to Δ iff there exists $P_{B_1} \to q_1$ in Δ_1 and $P_{B_2} \to q_2$ in Δ_2 such that $q = q_1 \cup q_2$ and $P_B = P_{B_1} \cup P_{B_2}$.
- Similarly, the base right-limit transition $q \to P_B$ belongs to Δ iff there exists $q_1 \to P_{B_1}$ in Δ_1 and $q_2 \to P_{B_2}$ in Δ_2 such that $q = q_1 \cup q_2$ and $P_B = P_{B_1} \cup P_{B_2}$.

Intuitively, simulating the joint behaviour of two base automata is as simple as computing unions of base elements:

- The state $q \in Q$ that encodes the possibility of being simultaneously in state q_1 in $\mathcal{A}_{B_1}$ and in state q_2 in $\mathcal{A}_{B_2}$ is $q = q_1 \cup q_2$.
- If $\rho : \widehat{J} \to Q$ is a run in $\mathcal{A}_B$ that simulates two runs $\rho_1 : \widehat{J} \to Q_1$ and $\rho_2 : \widehat{J} \to Q_2$ in $\mathcal{A}_{B_1}$ and $\mathcal{A}_{B_2}$ respectively, then $\lim_{c^-}^{B} \rho = \lim_{c^-}^{B_1} \rho_1 \cup \lim_{c^-}^{B_2} \rho_2$ and $\lim_{c^+}^{B} \rho = \lim_{c^+}^{B_1} \rho_1 \cup \lim_{c^+}^{B_2} \rho_2$.

Example 29 illustrates the intersection of finite base automata.

Example 29. *Consider the two base automata depicted in Figure 31. Their bases are* $\{\bullet, \bullet, \bullet\}$ *and* $\{\star, \star, \star\}$*, respectively. These automata are equivalent to those in Example 21: the base right-limit transition* $q_1 \to \{\bullet\}$ *translates into the right-limit transition* $q_1 \to \{q_2, q_3\}$*, since* $\{\bullet, \bullet\} \cap \{\bullet, \bullet\} = \{\bullet\}$ *and no other intersection yields* $\{\bullet\}$*. Similarly, the base right-limit transitions* $q_1' \to \{\star\}$ *translates into the right-limit transition* $q_1' \to \{q_2', q_3'\}$*. The intersection of these two base automata is the base automaton* $\mathcal{A}$ *represented in Figure 32. For convenience, each state is annotated by a tuple* (q_i, q_j')*, in order to make explicit the correspondence with Figure 17. Recall however that a state in a base automaton is entirely defined by a subset of the base. Intuitively, the base automaton* $\mathcal{A}$ *has been computed in the following way:*



(a) First operand. (b) Second operand.

Figure 31: Finite base automata used as operands for intersection.

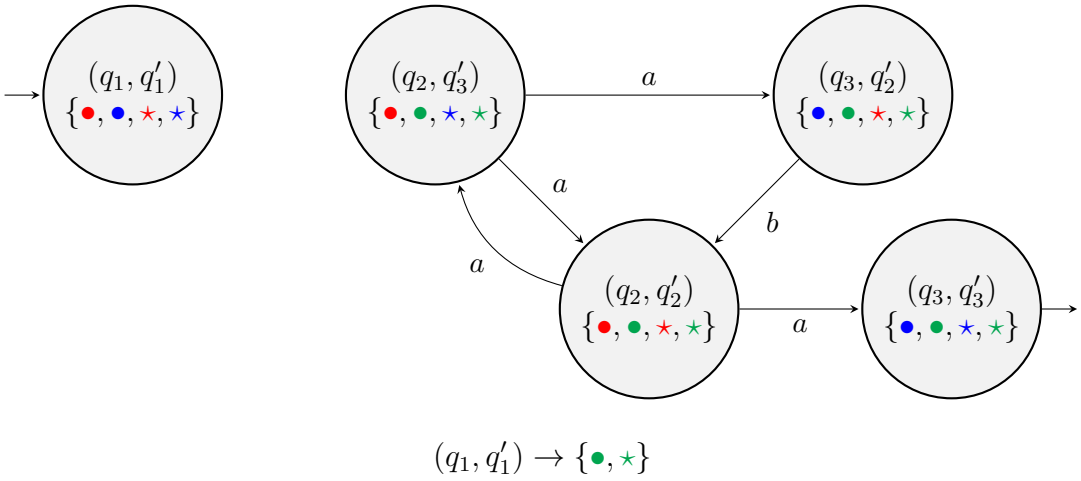


Figure 32: Base automata $\mathcal{A}$ obtained after intersection.

- *The state annotated by* (q_i, q'_j) *is defined as* $q_i \cup q'_j$*. For instance, the state annotated by* (q_2, q'_3) *corresponds to* $\{\bullet, \bullet\} \cup \{\star, \star\} = \{\bullet, \bullet, \star, \star\}$*.*
- *The base limit sets of the operands are simply combined together by taking their union. The base limit transitions* $q_1 \to \{\bullet\}$ *and* $q'_1 \to \{\star\}$ *are thus combined together into* $(q_1, q'_1) \to \{\bullet, \star\}$*.*

Using base automata greatly simplifies the computation of the limit transitions. In Example 21, we saw that combining the limit transitions $q_1 \to \{q_2, q_3\}$ *and* $q'_1 \to \{q'_2, q'_3\}$ *resulted in* 7 *limit transitions. Here, these* 7 *limit transitions are represented concisely by the single base limit transition* $(q_1, q'_1) \to \{\bullet, \star\}$*. One can check that it is indeed correct by enumerating all the sets* P *of states of* $\mathcal{A}$ *such that the base elements common to all the states of* P *are* $\{\bullet, \star\}$*. This yields the* 7 *limit sets of Figure* 16*.*

We saw that base automata provide a great way of representing automata on linear orderings. Computing intersections is much easier with base automata, thanks to their ability to represent many limit transitions by means of a single base limit transition. However, recall that the objective of this master's thesis is to design an efficient algorithm that computes the intersection of plain automata. Using finite base automata to represent plain automata thus creates two problems:

- Converting operands that are plain automata into finite base automata is not trivial. As Example 28 mentioned, there are several ways to convert a plain automaton into a finite base automaton. Finding the smallest finite base automaton (e.g., the automaton with the smallest base B) equivalent to an arbitrary plain automaton seems to be a difficult problem. Yet, the efficiency of the intersection algorithm strongly relies on how the operands were encoded into finite base automata.
- More importantly, converting a finite base automaton $\mathcal{A}_B$ back to a plain automaton $\mathcal{A}$ is time-consuming. For each base limit transition of $\mathcal{A}_B$, one needs to compute all the limit transitions of $\mathcal{A}$ that it represents. In most cases, this cannot be done efficiently, because it requires to enumerate the subsets of states of $\mathcal{A}$.

In the sequel, we try to design a representation of automata on linear orderings that keeps the advantages of base automata, and that circumvents these two problems.

4.2.2 Finite history automata

In this section, we define a new form of automata that we call *finite history automata*. They can be used to represent efficiently automata on linear orderings, and they are inspired from finite base automata.

Intuitively, the power of finite base automata comes from the fact that limit sets are encoded descriptively. In other words, multiple limit sets $P_1, P_2, \ldots, P_k$ can be represented by a single base limit set if for each $i \in \{1, 2, \ldots, k\}$, the set of base elements that are common to each state in P_i is equal to the base limit set. This condition can be considered as a description or a characterisation of $P_1, P_2, \ldots, P_k$. This mechanism makes the intersection of finite base automata almost trivial, since combining two descriptions of limit sets is easy: in the case of finite base automata, it just amounts to computing the union of the base limit sets.

Finite history automata are also based on this idea of describing the limit sets rather than enumerating them. However, our objective is to design a formalism that remains close to plain automata, for two reasons:

- We saw in the previous section that converting plain automata into finite base automata and vice-versa is not immediate. Opting for a formalism that is as close as possible to plain automata could solve this problem.
- In Section 4.1, we built an algorithm that is able to delete some useless limit transitions. This algorithm should still be applicable with an improved formalism.

Intuitively, finite history automata are a symbolic way of representing the intersection of d multiple plain automata. Their strength relies on the fact that they represent the set $\langle P_1, P_2, \ldots, P_d \rangle$ by the tuple $(P_1, P_2, \ldots, P_d)$ (which is, as in finite base automata, a description of multiple limit sets) and not by explicitly enumerating the limit sets covering exactly $P_1, P_2, \ldots P_d$. Definitions 12 and 13 formalize this idea.

Definition 12. *A* finite history automaton *is a tuple* $(d, \mathcal{H}, Q, \Sigma, \Delta, \Gamma, I, F)$ *where:*

- $d \in \mathbb{N}_{>0}$ *is the* dimension *of the finite history automaton.*

- $\mathcal{H} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_d\}$ *is the* history *of* $\mathcal{A}$*. It is a set of* d *plain automata (i.e., automata satisfying Definition 1) sharing the same alphabet* Σ*. Let* $\mathcal{A}_k$ *be equal to* $(Q_k, \Sigma, \Delta_k, I_k, F_k)$*, for all* $k \in \{1, 2, \ldots, d\}$*.*
- $Q \subseteq Q_1 \times Q_2 \times \cdots \times Q_d$ *is the set of states of* $\mathcal{A}$*.*
- Σ *is a finite alphabet that is shared with all the automata that belong to* $\mathcal{H}$*.*
- $\Delta \subseteq (Q \times \Sigma \times Q) \cup \left(Q \times \prod_{1 \leq i \leq d} 2^{Q_i}\right) \cup \left(\prod_{1 \leq i \leq d} 2^{Q_i} \times Q\right)$ *is a transition relation. It contains:*
  - *successor transitions* $(q, \sigma, q') \in Q \times \Sigma \times Q$*,*
  - historic right-limit transitions $(q, P_1, P_2, \ldots, P_d) \in Q \times \prod_{1 \leq i \leq d} 2^{Q_i}$*, where* P_k *is a limit set in* $\mathcal{A}_k$ *for all* $k \in \{1, 2, \ldots, d\}$*,*
  - *and* historic left-limit transitions $(P_1, P_2, \ldots, P_d, q) \in \prod_{1 \leq i \leq d} 2^{Q_i} \times Q$*, where each* P_k *is again a limit set in* $\mathcal{A}_k$*.*

  As usual, a successor transition (q, σ, q') *is also denoted by* $q \xrightarrow{\sigma} q'$*. An historic right-limit transition* $(P_1, P_2, \ldots P_d, q)$ *is also denoted by* $(P_1, P_2, \ldots, P_d) \to q$*. The notation for historic left-limit transitions is similar. In an historic right-limit or left-limit transition, the tuple* $(P_1, P_2, \ldots, P_d)$ *is an* historic limit set*.*
- $\Gamma : \prod_{1 \leq i \leq d} 2^{Q_i} \to 2^Q$ *is a function that maps each historic limit set to a subset of the states of the automaton. The purpose of this function will be explained later.*
- $I \subseteq Q$ *is a set of initial states.*
- $F \subseteq Q$ *is a set of final states.*

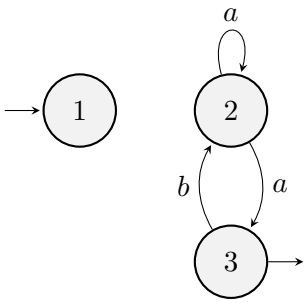
Note that a plain automaton $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ can be seen as a finite history automaton where $d = 1$ and $\mathcal{H} = \{\mathcal{A}\}$ (the function Γ being ignored for now).

Let us now define implicitly the semantics of finite history automata by converting an arbitrary finite history automaton into the plain automaton that it represents.

Definition 13. *Let* $\mathcal{A}_\mathcal{H} = (d, \mathcal{H}, Q, \Sigma, \Delta, \Gamma, I, F)$ *be a finite history automaton. This finite history automaton* $\mathcal{A}_\mathcal{H}$ *represents the plain automaton* $\mathcal{A}$ *defined as follows:*

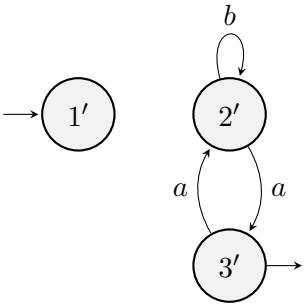
- *The set of states of* $\mathcal{A}$*, its alphabet, its successor transitions and its initial and final states are the same as* $\mathcal{A}_\mathcal{H}$*.*
- *The left-limit transitions of* $\mathcal{A}$ *are all the tuples* (P, q) *such that*
  - $(P_1, P_2, \ldots, P_d, q)$ *is an historic left-limit transition of* $\mathcal{A}_\mathcal{H}$*,*
  - *and* $P \in \langle P_1, P_2, \ldots, P_d \rangle$*.*
- *Similarly, the right-limit transitions of* $\mathcal{A}$ *are all the tuples* (q, P) *such that*
  - $(q, P_1, P_2, \ldots, P_d)$ *is an historic right-limit transition of* $\mathcal{A}_\mathcal{H}$*,*
  - *and* $P \in \langle P_1, P_2, \ldots, P_d \rangle$*.*

Example 30. *Consider once again the operands* $\mathcal{A}$ *and* $\mathcal{A}'$ *depicted in Figure 33. The finite history automaton* $\mathcal{A}_\mathcal{H}$ *from Figure 33c is a concise representation of the intersection between* $\mathcal{A}$ *and* $\mathcal{A}'$*. The* 7 *limit transitions from Figure 17 are denoted by the single historic limit transition* $(1, 1') \to \big(\{2, 3\}, \{2', 3'\}\big)$*. As finite base automata, finite history automata store limit transitions in a condensed manner.*



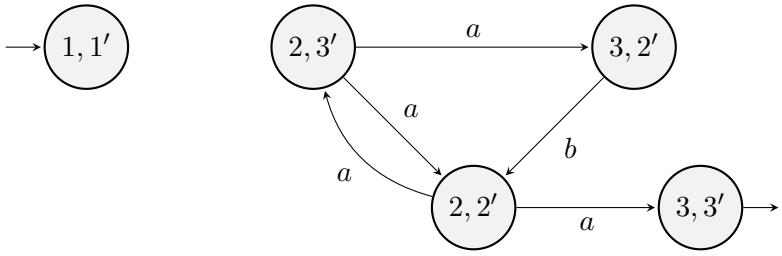
$1 \to \{2, 3\}$

(a) Automaton $\mathcal{A}$.



$1' \to \{2', 3'\}$

(b) Automaton $\mathcal{A}'$.



$(1, 1') \to \big(\{2, 3\}, \{2', 3'\}\big)$

(c) Finite history automaton $\mathcal{A}_{\mathcal{H}}$ with $\mathcal{H} = \{\mathcal{A}, \mathcal{A}'\}$.

Figure 33

As already suggested in Example 30, the intersection of finite history automata is a trivial operation. Let $\mathcal{A}_{\mathcal{H}_1} = (d_1, \mathcal{H}_1, Q_1, \Sigma, \Delta_1, \Gamma_1, I_1, F_1)$ and $\mathcal{A}_{\mathcal{H}_2} = (d_2, \mathcal{H}_2, Q_2, \Sigma, \Delta_2, \Gamma_2, I_2, F_2)$ be two finite history automata sharing the same alphabet Σ. We use the notation $x \cdot y$ to denote the concatenation of the tuples x and y (for instance, if $x = (1, 2)$ and $y = (3, 4, 5)$, then $x \cdot y = (1, 2, 3, 4, 5)$). The intersection of $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$ is the finite history automaton $\mathcal{A}_{\mathcal{H}} = (d, \mathcal{H}, Q, \Sigma, \Delta, \Gamma, I, F)$ defined as follows:

- $d = d_1 + d_2$.
- $\mathcal{H} = \mathcal{H}_1 \cup \mathcal{H}_2$ and Q follows from $\mathcal{H}$.
- $I = \big\{(q_1 \cdot q_2) \mid q_1 \in I_1 \wedge q_2 \in I_2\big\}$.
- $F = \big\{(q_1 \cdot q_2) \mid q_1 \in F_1 \wedge q_2 \in F_2\big\}$.
- The successor transition $(q_1 \cdot q_2) \xrightarrow{\sigma} (q_1' \cdot q_2')$ with $\sigma \in \Sigma$ belongs to Δ iff $q_1 \xrightarrow{\sigma} q_1'$ belongs to Δ_1 and $q_2 \xrightarrow{\sigma} q_2'$ belongs to Δ_2.
- The historic left-limit transition $P_{\mathcal{H}} \to q$ belongs to Δ iff there exists $P_{\mathcal{H}_1} \to q_1$ in Δ_1 and $P_{\mathcal{H}_2} \to q_2$ in Δ_2 such that $q = (q_1 \cdot q_2)$ and $P_{\mathcal{H}} = (P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2})$.
- Similarly, the historic right-limit transition $q \to P_{\mathcal{H}}$ belongs to Δ iff there exists $q_1 \to P_{\mathcal{H}_1}$ in Δ_1 and $q_2 \to P_{\mathcal{H}_2}$ in Δ_2 such that $q = (q_1 \cdot q_2)$ and $P_{\mathcal{H}} = (P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2})$.
- The method for the computation of Γ from Γ_1 and Γ_2 will be described later.

Example 31. *It can be checked that converting the automata $\mathcal{A}$ and $\mathcal{A}'$ from Example 30 into finite history automata and then computing their intersection yields the finite history automaton $\mathcal{A}_\mathcal{H}$ depicted in Figure 33c.*

As previously hinted, finite history automata behave in the same way as finite base automata: combining limit sets amounts to performing a simple operation (concatenating tuples for finite history automata, computing unions for finite base automata). In some sense, combining limit sets now boils down to a merely syntactic operation.

We now tackle the problem of recovering a plain automaton from a finite history automaton resulting from one or several intersections. Let $\mathcal{A}_\mathcal{H}$ be a finite history automaton $(d, \mathcal{H}, Q, \Sigma, \Delta, \Gamma, I, F)$. In order to convert efficiently $\mathcal{A}_\mathcal{H}$ into a plain automaton, we need to find a tractable way of turning any historic limit transition $(P_1, P_2, \dots P_d) \to q$ or $q \to (P_1, P_2, \dots P_d)$ of $\mathcal{A}_\mathcal{H}$ into limit transitions of the form $P \to q$ or $q \to P$, with $P \subseteq Q$. There are two main strategies to solve this problem:

- One could apply Definition 13, i.e., enumerate the limit sets P from $\langle P_1, P_2, \dots, P_d \rangle$. We saw in Chapter 3 that this strategy is intractable, even for small values of d.
- Alternatively, we could use an approach similar to the conversion of a finite base automaton into a plain automaton, i.e., we could enumerate all the subsets P of Q that might be useful limit sets, and then check whether $P \in \langle P_1, P_2, \dots, P_d \rangle$. But for large sets of states, this is intractable as well.

These strategies are unlikely to work in practice.

The solution to this problem relies on the function Γ. Recall that this function maps an historic limit set to a subset of Q. In any finite history automaton $\mathcal{A}_\mathcal{H}$, we will enforce the following property: for any historic limit set $(P_1, P_2, \dots, P_d)$ such that $(P_1, P_2, \dots, P_d, q) \in \Delta$ or $(q, P_1, P_2, \dots, P_d) \in \Delta$, one has

$$\Gamma(P_1, P_2, \dots, P_d) \supseteq P \qquad \forall P \subseteq Q \text{ such that } P \in \langle P_1, P_2, \dots P_d \rangle.$$

Intuitively, the function Γ thus maps an historic limit set to a subset of Q in which all the subsets P that cover exactly $P_1, P_2, \dots, P_d$ are included. This implies that applying the second strategy to convert $(P_1, P_2, \dots, P_d, q) \in \Delta$ or $(q, P_1, P_2, \dots, P_d) \in \Delta$ into limit transitions $P \to q$ or $q \to P$ now requires to enumerate the subsets of $\Gamma(P_1, P_2, \dots, P_d)$ only (rather than the subsets of Q). This can be done efficiently if $\Gamma(P_1, P_2, \dots, P_d)$ is a small subset of Q, or at least if the useful limit sets of $\Gamma(P_1, P_2, \dots, P_d)$ can be efficiently enumerated.

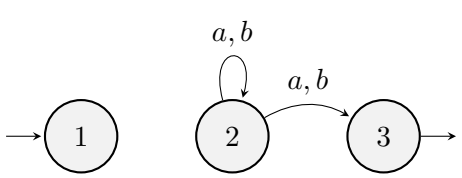
In other words, thanks to the new representation offered by finite history automata, the computation of the n-ary intersection of automata on linear orderings reduces to 3 successive sub-problems:

1. Converting the n operands (that all satisfy Definition 1) into finite history automata. We have already seen that this task is trivial. The only non trivial part will be to define carefully the function Γ for each operand.
2. Computing the intersection of the n finite history automata, as explained above. This is not difficult, since the intersection of finite history automata is almost as easy as the intersection of finite-word automata (at least in theory). However, we will have to find a way of computing the function Γ of the intersection automaton from the functions Γ_1 and Γ_2 of the operands. This will be one of the main problems addressed in the next chapters.

3. Finally, the finite history automaton $\mathcal{A}_{\mathcal{H}}$ resulting from the n-ary intersection operation has to be turned back into a plain automaton. This is not difficult, provided that the useful limit sets in $\Gamma(P_1, P_2, \dots P_d)$ can be efficiently enumerated, for all historic limit sets $(P_1, P_2, \dots, P_d)$. In practice, this will be the case most of the time.

Example 32 illustrates these 3 steps by means of the automata already introduced in Example 27.

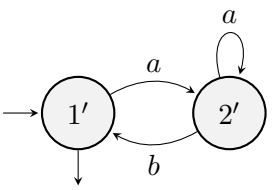
Example 32. *Consider the two plain automata $\mathcal{A}$ and $\mathcal{A}'$ depicted in Figure 34. These two operands*



$3 \to \{2\}$

$1 \to \{2, 3\}$

(a) Automaton $\mathcal{A}$.

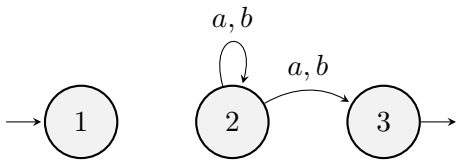


$1' \to \{1', 2'\}$

(b) Automaton $\mathcal{A}'$.

Figure 34: Two operands for the intersection operation.

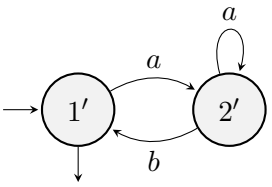
can be easily turned into finite history automata, as shown in Figure 35. We have set $\Gamma_{\{\mathcal{A}\}}(P) = P$ for each limit set P of $\mathcal{A}$, and similarly for $\Gamma_{\{\mathcal{A}'\}}$. We will see later that in practice, we will have to refine this choice.



$3 \to \{2\}$ and $\Gamma_{\{\mathcal{A}\}}\big(\{2\}\big) = \big\{2\big\}$

$1 \to \{2, 3\}$ and $\Gamma_{\{\mathcal{A}\}}\big(\{2, 3\}\big) = \big\{2, 3\big\}$

(a) Finite history automaton with history $\{\mathcal{A}\}$.



$1' \to \{1', 2'\}$ and $\Gamma_{\{\mathcal{A}'\}}\big(\{1', 2'\}\big) = \big\{1', 2'\big\}$

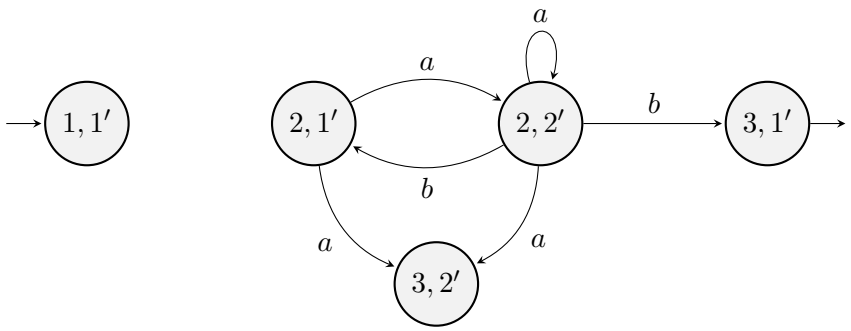
(b) Finite history automaton with history $\{\mathcal{A}'\}$.

Figure 35: Finite history automata equivalent to the operands in Figure 34.

The intersection of these two finite history automata is shown in Figure 36. The resulting finite history automaton has two historic right-limit transitions. We do not explain here how the function $\Gamma_{\{\mathcal{A},\mathcal{A}'\}}$ has been computed (this is one of the subjects covered in Chapter 5).

Finally, this finite history automaton must be converted into a plain automaton:

- *The first step is to translate the historic limit transition $(3, 1') \to \big(\{2\}, \{1', 2'\}\big)$ into limit transitions. Since $\Gamma_{\{\mathcal{A},\mathcal{A}'\}}\big(\{2\}, \{1', 2'\}\big) = \big\{(2, 1'), (2, 2')\big\}$, we need to enumerate all the subsets of $\big\{(2, 1'), (2, 2')\big\}$ that cover exactly $\{2\}$ and $\{1', 2'\}$ and that can be visited infinitely often. Clearly, the only subset that satisfies these conditions is $\big\{(2, 1'), (2, 2')\big\}$ itself. We thus get the single limit transition $(3, 1') \to \big\{(2, 1'), (2, 2')\big\}$.*



$$(3,1') \to \big(\{2\},\{1',2'\}\big) \quad \text{and} \quad \Gamma_{\{\mathcal{A},\mathcal{A}'\}}\big(\{2\},\{1',2'\}\big) = \big\{(2,1'),(2,2')\big\}$$

$$(1,1') \to \big(\{2,3\},\{1',2'\}\big) \quad \text{and} \quad \Gamma_{\{\mathcal{A},\mathcal{A}'\}}\big(\{2,3\},\{1',2'\}\big) = \big\{(2,1'),(2,2'),(3,1')\big\}$$

Figure 36: Finite history automaton with history $\{\mathcal{A},\mathcal{A}'\}$, resulting from the intersection.

- *We now translate the historic limit transition* $(1,1') \to \big(\{2,3\},\{1',2'\}\big)$. *We know that*

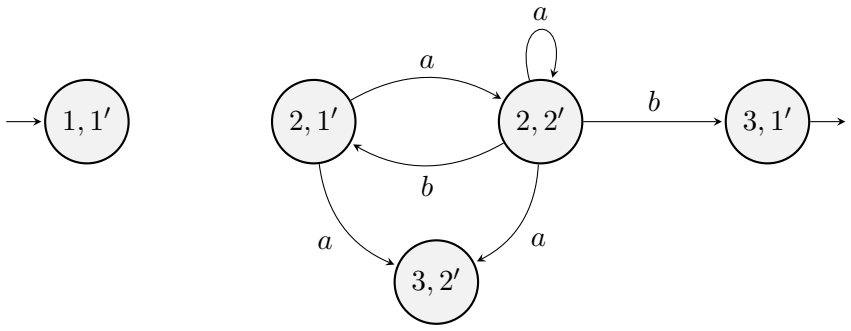
$$\Gamma_{\{\mathcal{A},\mathcal{A}'\}}\big(\{2,3\},\{1',2'\}\big) = \big\{(2,1'),(2,2'),(3,1')\big\}$$

  . *The subsets of* $\big\{(2,1'),(2,2'),(3,1')\big\}$ *that cover exactly* $\{2,3\}$ *and* $\{1',2'\}$ *are:*

  - $\{(2,2'),(3,1')\}$, *but this is impossible to visit exactly this set of states infinitely often. The only way of leaving state* $(3,1')$ *is to use the limit transition* $(3,1') \to \big\{(2,1'),(2,2')\big\}$, *but this would require to visit the state* $(2,1')$. *In other words, this limit set is useless.*
  - $\big\{(2,1'),(2,2'),(3,1')\big\}$, *and it is possible to visit infinitely often this set of states.*

  Therefore, we obtain the single limit transition $(1,1') \to \big\{(2,1'),(2,2'),(3,1')\big\}$.

Note that even though the set of states $\big\{(2,1'),(2,2'),(3,2')\big\}$ *covers exactly* $\{2,3\}$ *and* $\{1',2'\}$, *it has not been enumerated, because* $\big\{(2,1'),(2,2'),(3,2')\big\} \notin \Gamma_{\{\mathcal{A},\mathcal{A}'\}}\big(\{2,3\},\{1',2'\}\big)$. *This illustrates the purpose of the function* $\Gamma_{\{\mathcal{A},\mathcal{A}'\}}$*: decreasing the number of subsets of states that must be examined. In this toy example, this decrease is not important. But in practice, the use of a function like* $\Gamma_{\{\mathcal{A},\mathcal{A}'\}}$ *could dramatically accelerate the conversion from a finite history automaton into a plain automaton.*



$$(3,1') \to \big\{(2,1'),(2,2')\big\}$$

$$(1,1') \to \big\{(2,1'),(2,2'),(3,1')\big\}$$

Figure 37: Plain automaton resulting from the intersection of $\mathcal{A}$ and $\mathcal{A}'$.

In this example, only $n = 2$ operands were considered, and therefore using finite history automata as internal representation is not really relevant. But in practice, n will often be much larger. In this case, using finite history automata as internal representation during the $n - 1$ binary intersections is really beneficial.

Chapter 5

Algorithms

This chapter translates the ideas introduced in Chapter 4 into data structures and algorithms. One of our goals is to explain how the functions Λ and Γ mentioned previously can be implemented. In Section 5.1, we present the data structure that is at the core of all the algorithms described later. Section 5.2 incrementally builds an algorithm for the intersection of finite history automata. Finally, Section 5.3 explains how a finite history automaton can be converted into a plain automaton. At the end of this chapter, we also briefly discuss the few modifications that are involved to compute the union of automata on linear orderings.

Even though this chapter is more practical than the previous one, it presents all the algorithms at a very abstract level. For instance, all the details related to the actual implementation of plain automata and finite history automata are completely abstracted away in this chapter.

5.1 Strongly connected components hierarchy

5.1.1 Definition

We saw in Chapter 4 that some useless limit sets of an automaton could be deleted by studying the strongly connected components of its reachability graph. Moreover, these strongly connected components were sometimes merged under the action of new red arcs. Furthermore, finite history automata are equipped with a function Γ that localizes each historic limit set into a subset of states of the automaton. A very natural idea is to choose the strongly connected components as containers for the historic limit sets, since they are already needed for the limit-set deletion algorithm.

We now describe the *strongly connected components hierarchy* data structure (that we will often call SCC hierarchy). It is able to keep track of the strongly connected components hierarchically, and to store historic limit sets into them. In some sense, this data structure is the combination of several types of reachability graphs. Note that we temporarily assume that auto-referencing limit sets (that we defined in Chapter 4) do not exist (we will come back to this question in Section 5.2.5).

Definition 14. *Given a finite history automaton $\mathcal{A}_\mathcal{H}$, an SCC hierarchy is a tuple $(\mathcal{G}, \mathcal{T})$ where $\mathcal{G}$ and $\mathcal{T}$ are defined as follows:*

- *$\mathcal{G}$ is a graph that is syntactically indistinguishable from a reachability graph of $\mathcal{A}_\mathcal{H}$ (see Section 4.1.2), except that its red arcs now have a label.*
- *$\mathcal{T}$ is a tree (that we often call* SCC tree*), i.e., an acyclic undirected graph.*[1] *Its vertices are called* nodes *and can store a payload. Each node has a* level *greater or equal to* 0*. A* descendant *of a*

[1] We do not enforce this graph to be connected. Technically, $\mathcal{T}$ is thus more a forest than a tree.

node n is another node n' that is located below n (i.e., at a lower level) and is connected to it by a sequence of adjacent edges. A child *of n is a descendant of n that is located one level below n. A* leaf *is a node that does not have any child.*

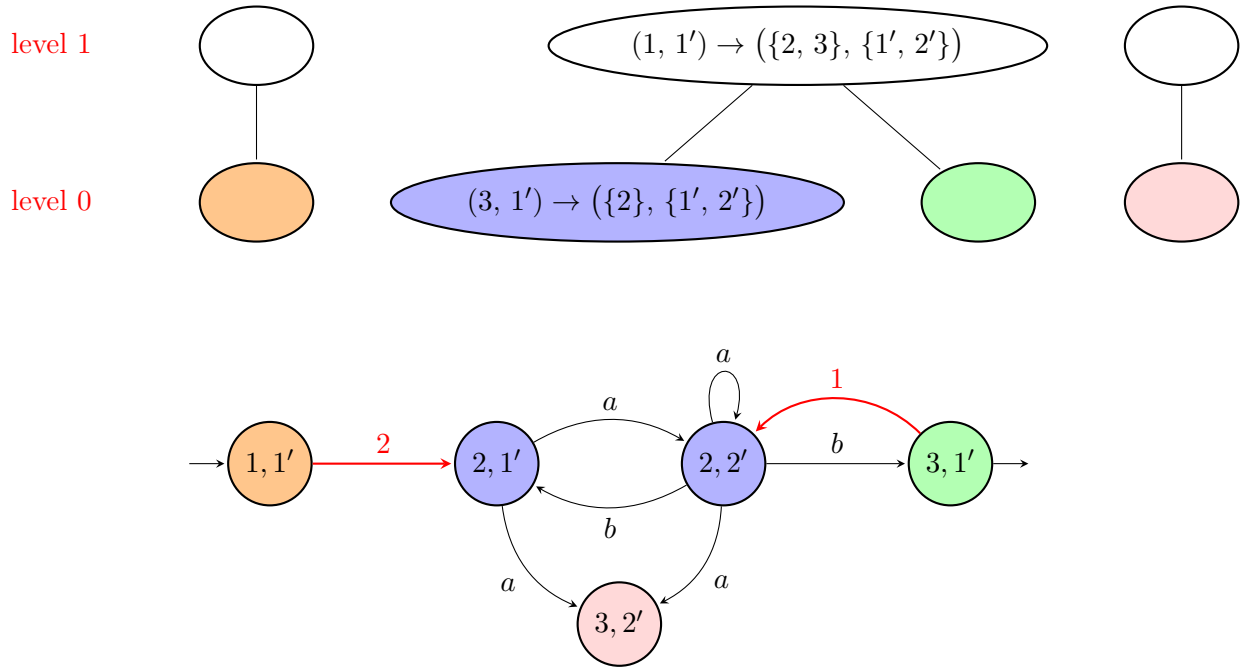


Figure 38: Example of SCC hierarchy for the intersection automaton obtained in Examples 27 and 32.

An example of SCC hierarchy (related to the intersection automaton obtained in Examples 27 and 32) is depicted in Figure 38. We now define informally the semantics and the properties of SCC hierarchies by means of this example. Intuitively, the graph $\mathcal{G}$ (see the lower part of Figure 38) combines the information of different types of reachability graphs. Each colour refers to a strongly connected component of the reachability graph $G_0 = (V, E, R_0)$ where R_0 is empty. In other words, each of these strongly connected components is based on arcs from E only (i.e., on successor transitions). These arcs are said to be of *level* 0. Similarly, these strongly connected components are called *level*-0 *strongly connected components*. They are represented by the leaves of the tree $\mathcal{T}$ depicted in the upper part of Figure 38. Each leaf can *store* one or several historic limit sets of $\mathcal{A}_{\mathcal{H}}$ as payload. A leaf representing a strongly connected component $S \subseteq Q$ (where Q is the set of states of $\mathcal{A}_{\mathcal{H}}$) stores an historic limit set $(P_1, P_2, \ldots, P_d)$ iff

$$\exists T \subseteq S : T \in \langle P_1, P_2, \ldots, P_d \rangle.$$

If $(P_1, P_2, \ldots, P_d)$ is stored in some leaf, we will also often say (indistinguishably) that this leaf *stores the historic limit transitions* that involve $(P_1, P_2, \ldots, P_d)$.

For instance, the blue leaf (that represents the blue strongly connected component) stores the historic limit transition $(3, 1') \to (\{2\}, \{1', 2'\})$, since $\{(2, 1'), (2, 2')\}$ is a subset of the blue SCC that covers exactly $\{2\}$ and $\{1', 2'\}$. On the other hand, $(3, 1') \to (\{2\}, \{1', 2'\})$ is not stored in the orange leaf, because there does not exist any subset of $\{(1, 1')\}$ that covers exactly $\{2\}$ and $\{1', 2'\}$.

Importantly, each red arc is now labelled by a natural number strictly greater than 0, that we call once again the *level* of the red arc. Moreover, there are nodes the tree $\mathcal{T}$ that are not leaves and that represent strongly connected components of *level* greater than 0. We now define inductively what it means for a red arc or a strongly connected component to be of level $\ell > 0$, and also what it means for an historic limit set to be *stored* in a node representing such a strongly connected component.

- A red arc $p \to q$ of level $\ell > 0$ translates an historic limit transition $(P_1, P_2, \ldots, P_d) \to q$ such that $(P_1, P_2, \ldots, P_d)$ is stored in a node representing a strongly connected component of level $\ell - 1$, and p belongs to this strongly connected component.[2]
- A red arc $q \to p$ of level $\ell > 0$ is defined similarly.
- A strongly connected component of level $\ell > 0$ is a subset $S \subseteq Q$ such that:
  - For all states s_1, s_2 in S, there exists a path from s_1 to s_2 in $\mathcal{G}$ that only uses arcs of level lower or equal to ℓ.
  - There exist s_1, s_2 in S such that all paths connecting s_1 to s_2 in $\mathcal{G}$ use at least one arc of level ℓ.

  In the SCC hierarchy, a strongly connected component S of level ℓ is represented by a node of level ℓ connected via edges to its children, which represent strongly connected components of level $\ell - 1$ whose union is S.

  Observe that this definition also holds for the base case $\ell = 0$, except that a node representing a SCC of level 0 (i.e., a leaf) does not have any child.
- An historic limit set $(P_1, P_2, \ldots, P_d)$ is *stored* in a strongly connected component S (or, equivalently, in the node representing S) of level $\ell > 0$ iff
  - there exists $T \subseteq S$ such that $T \in \langle P_1, P_2, \ldots, P_d \rangle$, and
  - there does not exist any descendant of S that has this property.

  This definition is also valid for the base case $\ell = 0$, since a leaf does not have any descendant. Once again, if $(P_1, P_2, \ldots, P_d)$ is stored in some node, we will allow us to say that this node *stores the historic limit transitions* that involve $(P_1, P_2, \ldots, P_d)$.

For instance, the red arc $(3, 1') \to (2, 2')$ is of level 1 because it translates the historic limit transition $(3, 1') \to \big(\{2\}, \{1', 2'\}\big)$, that is stored in a strongly connected component of level 0. The red arc $(1, 1') \to (2, 1')$ is of level 2, since it encodes the historic limit transition $(1, 1') \to \big(\{2, 3\}, \{1', 2'\}\big)$, that is stored in the node of level 1 whose children are the blue and the green leaves. This historic limit transition must be stored in this node, since there exists at least one subset of $\big\{(2, 1'), (2, 2'), (3, 1')\big\}$ (actually, two) that covers exactly $\{2, 3\}$ and $\{1', 2'\}$, and the blue and the green strongly connected components do not have this property. Note that we did not represent the strongly connected components of level 2 in Figure 38, since they would be identical to those of level 1.

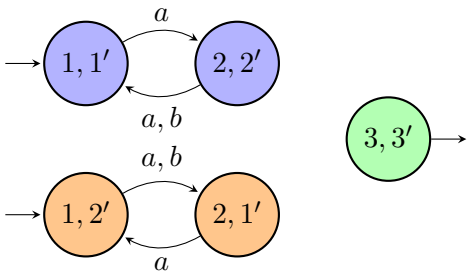
Intuitively, an SCC hierarchy can be seen in two different ways:

- It can be considered as a superposition of several types of reachability graphs, the lowermost one being the reachability graph G_0 that we already mentioned many times in Chapter 4. We saw in this chapter that it is correct to assume that any useful limit set is included in a strongly connected component of sufficiently high level. This explains why we enforce the historic limit sets to be stored in an SCC. This amounts to running the limit-set deletion algorithm implicitly.

[2] Note that in Chapter 4, p was defined as a representative of a limit set. But we are now dealing with historic limit sets, hence p is now defined as a representative of the strongly connected component in which the historic limit set is stored. Since red arcs only encode reachability, this modification does not raise any issue.

If $P \in \langle P_1, P_2, \ldots, P_d \rangle$ is a useful limit set, then there exists an SCC in which P is included, and therefore we have the guarantee that $(P_1, P_2, \ldots, P_d)$ is stored in some SCC of the SCC hierarchy. However, note that if $(P_1, P_2, \ldots, P_d)$ is stored in the SCC hierarchy, it does not always mean that there exists a useful limit set $P \in \langle P_1, P_2, \ldots, P_d \rangle$.

- An SCC hierarchy can also be considered as a way of implementing the function Γ from Chapter 4. Given an historic limit set $(P_1, P_2, \ldots, P_d)$, one can compute $\Gamma(P_1, P_2, \ldots, P_d)$ by simply collecting all the strongly connected components that store $(P_1, P_2, \ldots, P_d)$. Note that even though it is not frequent in practice, an historic limit set can be stored in more than one strongly connected component. For instance, consider the finite history automaton in Figure 39, which can be obtained via an intersection of finite history automata. If we were to build its SCC hierarchy, we would have to store $\big(\{1,2\}, \{1',2'\}\big)$ both in the blue and in the orange SCC.



$$\big(\{1,2\}, \{1',2'\}\big) \to (3,3')$$

Figure 39: Finite history automaton whose SCC hierarchy stores the same historic limit set in two different strongly connected components (or, equivalently, in two different nodes).

5.1.2 Supported operations

We now introduce the two main operations supported by any SCC hierarchy. We assume that we have an object `scch` that represents an SCC hierarchy.

- Let T be a set of states (that can form a strongly connected component or not). Then, `query`$(\texttt{scch}, T)$ returns a set of historic limit sets, computed as follows:

  1. First, find the lowest level strongly connected component S in which T is included.
  2. If this SCC exists, then return the set of historic limit sets that are stored in S or in its descendants. Otherwise, return the empty set.

  Consider once again the SCC hierarchy in Figure 38. If $T = \big\{(2,2'), (3,1')\big\}$, then

  $$\texttt{query}(\texttt{scch}, T) = \Big\{\big(\{2\}, \{1',2'\}\big), \big(\{2,3\}, \{1',2'\}\big)\Big\}.$$

  On the other hand, if $T = \big\{(3,2'), (3,1')\big\}$, then $\texttt{query}(\texttt{scch}, T) = \emptyset$, since there does not exist any SCC that includes both $(3,2')$ and $(3,1')$.

- Let ℓ be a level, i.e., a natural number. Then, `build`$(\texttt{scch}, \ell)$ computes the strongly connected components of level ℓ based on the current reachability graph and adds them in the tree of the SCC hierarchy. For instance, if the SCC hierarchy is empty, executing `build`$(\texttt{scch}, 0)$ computes its leaves. Importantly, the `build` function does not recompute the strongly connected components already present in the hierarchy. In other words, all the states that are already included in a

SCC of level ℓ are ignored during the execution of $\texttt{build}(\texttt{scch}, \ell)$. When an SCC is added in the tree, it is added permanently and cannot be modified. As we will see in Section 5.2, this is really constraining, but guarantees that SCC hierarchies are built efficiently.

In all cases, the output of the function $\texttt{build}(\texttt{scch}, \ell)$ is the set of strongly connected components that have been added to the hierarchy. This set can be empty. For instance, if we execute $\texttt{build}(\texttt{scch}, 0)$ twice in a row, the second function call returns the empty set.

In addition to these two main operations, we will need other simpler operations and notations to deal with set of states, strongly connected components and historic limit sets of a finite history automaton $\mathcal{A}_\mathcal{H}$ of dimension d:

- Given an historic limit set $P_\mathcal{H}$, the set of strongly connected components in which $P_\mathcal{H}$ is stored is $\Gamma(\texttt{scch}, P_\mathcal{H})$.
- Given a strongly connected component S, the set of historic limit transitions stored in S is $\Lambda(\texttt{scch}, S)$. In some sense, Λ and Γ are antagonist functions.
- $\texttt{level}(S)$ returns the level of the strongly connected component S.
- $\texttt{height}(\texttt{scch})$ returns the number of levels present in the SCC hierarchy $\texttt{scch}$. For instance, the height of the SCC hierarchy in Figure 38 is 2.
- $R(\texttt{scch})$ denotes the set of red arcs of the SCC hierarchy $\texttt{scch}$.
- Let $T \in Q_1 \times Q_2 \times \cdots \times Q_d$ be a set of states (that can form a strongly connected component) of $\mathcal{A}_\mathcal{H}$ and $P_\mathcal{H} = (P_1, P_2, \ldots, P_d)$ one of its historic limit sets.
  - The function $\texttt{select}(T, P_\mathcal{H})$ returns the subset of T defined as
  $$\Big\{(q_1, q_2, \ldots, q_d) \in T \mid q_i \in P_i \quad \forall\, 1 \leq i \leq d\Big\}.$$
  Intuitively, it "selects" the states of T that "belong" to $P_\mathcal{H}$.
  - The function $\texttt{covers}(T, P_\mathcal{H})$ returns a Boolean value indicating whether T *covers* $P_\mathcal{H}$, i.e., whether $\texttt{select}(T, P_\mathcal{H})$ belongs to $\langle P_\mathcal{H}\rangle$, where $\langle P_\mathcal{H}\rangle$ is shorthand for $\langle P_1, P_2, \ldots, P_d\rangle$. If T is in fact a strongly connected component, then $P_\mathcal{H}$ can be stored in T only if $\texttt{covers}(T, P_\mathcal{H})$ returns True. Note that the function $\texttt{covers}(T, P_\mathcal{H})$ can be implemented efficiently, it does not require to enumerate the elements of $\langle P_\mathcal{H}\rangle$.

  For instance, if $P_\mathcal{H} = \big(\{1,2\}, \{1'\}\big)$ and $T = \big\{(1,1'), (2,1'), (1,2')\big\}$, then $\texttt{select}(T, P_\mathcal{H}) = \big\{(1,1'), (2,1')\big\} \in \langle P_\mathcal{H}\rangle$, hence T covers $P_\mathcal{H}$.
- If $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$ are finite history automata with set of states Q_1 and Q_2 respectively, we already know that the set of states of their intersection is $Q = \{q_1 \cdot q_2 \mid q_1 \in Q_1 \ \wedge \ q_2 \in Q_2\}$. Then, if $T \subseteq Q$, we will often use the notations $\pi_{Q_1}(T)$ and $\pi_{Q_2}(T)$, that respectively denote the sets
  $$\big\{q_1 \in Q_1 \mid \exists q \in T \text{ such that } q = q_1 \cdot q_2 \text{ for some } q_2 \in Q_2\big\}$$
  and
  $$\big\{q_2 \in Q_2 \mid \exists q \in T \text{ such that } q = q_1 \cdot q_2 \text{ for some } q_1 \in Q_1\big\}.$$
  The function π can be considered as a "projection" operator.
- The function $\texttt{repres}(T)$ returns a representative of T, i.e., an arbitrary state included in T. Two calls to this function with the same input T yield identical outputs (i.e., $\texttt{repres}$ is a function in the mathematical sense).

- The notations $\texttt{orig}(P_{\mathcal{H}})$ and $\texttt{dest}(P_{\mathcal{H}})$ denote, respectively, the origin states and destination states of the historic limit transitions that involve $P_{\mathcal{H}}$. Similarly, given a state q, $\texttt{succ}(q)$ denotes the set of states that can be reached from q by following a successor transition, and $\texttt{rightlim}(q)$ denotes the historic limit sets involved in the historic right-limit transitions leaving q.

5.2 Intersection algorithm

5.2.1 Automata data structures

In this section, we define how plain automata and finite history automata will be manipulated in the algorithms that follow. The data structures and algorithms introduced here are completely independent from the actual implementation of these automata.

Regarding plain automata, we will stick as much as possible to Definition 1. In our algorithms, a plain automaton is a tuple $(Q, \Delta^s, \Delta^{\lim}, I, F)$. For clarity, we distinguish the set Δ^s of successor transitions from the set $\Delta^{\lim}$ of limit transitions. The alphabet Σ is always implicitly defined as the common alphabet of all the operands of the intersection. $Q, \Delta^s, \Delta^{\lim}, I$ and F remain sets in the mathematical sense, i.e., an instruction that adds a state q in Q will be written as

$$Q \leftarrow Q \cup \{q\}$$

Similarly, adding a successor transition or a limit transition can be done, for instance, via

$$\Delta^s \leftarrow \Delta^s \cup \left\{q_1 \xrightarrow{\sigma} q_2\right\}$$

and

$$\Delta^{\lim} \leftarrow \Delta^{\lim} \cup \left\{q_1 \rightarrow \{q_2, q_3\}\right\}.$$

The finite history automata data structure slightly deviates from Definition 12. It consists of a tuple $(d, Q, \Delta^s, \texttt{scch}, I, F)$ where:

- d, Q, I and F satisfy Definition 12.
- Δ^s is the set of successor transitions of the finite history automaton.
- $\texttt{scch}$ is an SCC hierarchy. It contains all the information about the historic limit transitions (this explains why we do not need $\Delta^{\lim}$ in the data structure), the strongly connected components and the red arcs. It is always automatically synchronised with Q and Δ^s: when a new state or a new successor transition is added to Q or Δ^s, the reachability graph of the SCC hierarchy (i.e., the lower part of Figure 38) is updated accordingly.

5.2.2 Assumptions

The intersection algorithm described in the next sections takes as inputs finite history automata $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$ equipped with an SCC hierarchy. Then, it computes their intersection $\mathcal{A}_{\mathcal{H}}$ progressively and simultaneously builds the SCC hierarchy of $\mathcal{A}_{\mathcal{H}}$. Later, $\mathcal{A}_{\mathcal{H}}$ can thus become the operand of another intersection operation, since its SCC hierarchy is available.

However, the main problem tackled in this master's thesis is the intersection of plain automata on linear orderings. Therefore, a preliminary step is to convert each plain automaton into a finite history automaton (we saw in Chapter 4 that this operation is immediate) and to compute its SCC hierarchy (this is less trivial). In this work, we do not want to address the problem of computing the SCC hierarchy of a finite state automaton. In fact, this problem is not really difficult (it is easier than the intersection problem that we will solve in the remainder of this chapter) and we let it for some future work.

In other words, we assume that the finite state automata given as inputs are magically equipped with a valid SCC hierarchy. In fact, this assumption is not really constraining. Recall that in the context of the decision procedure, the input operands are atoms. It is extremely easy to compute by hand the SCC hierarchy of each type of atom, and to reuse it when it is necessary. We do not really need an algorithm that computes the SCC hierarchy of an arbitrary finite history automaton.

5.2.3 Initial version

Algorithm 1 computes the binary intersection of two finite history automata $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$. It is a direct translation of the ideas introduced in Chapter 4, but does not handle properly the auto-referencing limit sets. We will slightly modify it in Section 5.2.5 to solve this issue.

Lines 3 to 14 simply create an empty finite history automaton $\mathcal{A}_{\mathcal{H}}$ and compute its states, initial states and final states. It also computes the successor transition relation Δ^s.

The remainder of the algorithm consists in computing the SCC hierarchy level by level (starting with level 0, obviously), and in adding historic limit transitions to it. For a given level ℓ, the algorithm computes the set $\mathscr{S}$ of all the strongly connected components of level ℓ. For each $S \in \mathscr{S}$, it considers all the pairs $(P_{\mathcal{H}_1}, P_{\mathcal{H}_2})$ where $P_{\mathcal{H}_1}$ is an historic limit transition of $\mathcal{A}_{\mathcal{H}_1}$ and $P_{\mathcal{H}_2}$ is an historic limit transition of $\mathcal{A}_{\mathcal{H}_2}$. In Line 20, it then checks whether

- $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ is not already stored in S or in one of its descendants,
- and S covers $\langle P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2} \rangle$.

If these two conditions are satisfied, all the historic limit transitions involving $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ are added in the SCC hierarchy. Red arcs of level $\ell + 1$ are added to it as well, to take into account the effect of the new historic limit transitions.

Next, the algorithm increases the level ℓ by 1 and repeats the same operations until $\mathscr{S} = \mathscr{S}_{\text{old}}$, i.e., until the strongly connected components of two consecutive levels are the same (which means that there does not exist any new historic limit set $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ that can enter the SCC hierarchy).

We now briefly argue that this algorithm is correct.

- For each strongly connected component, all the pairs $(P_{\mathcal{H}_1}, P_{\mathcal{H}_2})$ are always examined. Moreover, if $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ is not stored in the SCC hierarchy at the end of the algorithm, then it means that there does not exist any strongly connected component S (even of high level) that includes an element of $\langle P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2} \rangle$ (i.e., that covers $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$). Thanks to the results of Chapter 4, all the elements of $\langle P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2} \rangle$ can thus be safely deleted. In other words, if $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ is not stored in the SCC hierarchy, each $P \in \langle P_{\mathcal{H}_1}, P_{\mathcal{H}_2} \rangle$ is useless.
- Checking that $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2} \notin \texttt{query}(\texttt{scch}, S)$ in Line 20 ensures that there does not exist any descendant of S that covers $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$. The algorithm thus builds a correct SCC hierarchy for the intersection automaton.

5.2.4 Examining less pairs of historic limit sets

Algorithm 1 can be significantly improved by decreasing the number of pairs $(P_{\mathcal{H}_1}, P_{\mathcal{H}_2})$ examined for each strongly connected component S. Intuitively, knowing the states contained in S makes it possible to discard a lot of pairs $(P_{\mathcal{H}_1}, P_{\mathcal{H}_2})$ for which it is obvious that they cannot be stored in S.

Algorithm 1: Intersection of finite history automata (first version).

1 FHA_binary_intersection ($\mathcal{A}_{\mathcal{H}_1} = (d_1, Q_1, \Delta_1^s, \text{scch}_1, I_1, F_1)$, $\mathcal{A}_{\mathcal{H}_2} = (d_2, Q_2, \Delta_2^s, \text{scch}_2, I_2, F_2)$):

2 $\quad \mathcal{A}_{\mathcal{H}} \leftarrow$ new empty finite history automaton $\big(d_1 + d_2, Q = \emptyset, \Delta^s = \emptyset,$

3 $\qquad\qquad\qquad\qquad\qquad\qquad \text{scch} = \text{scch\_new\_empty}(), I = \emptyset, F = \emptyset\big)$

4 $\quad Q \leftarrow \Big\{q_1 \cdot q_2 \mid q_1 \in Q_1 \ \wedge \ q_2 \in Q_2\Big\}$

5 $\quad$ **for** $q \in Q$ **do**

6 $\quad\quad$ Let $q_1 \in Q_1$ and $q_2 \in Q_2$ be such that $q = q_1 \cdot q_2$

7 $\quad\quad$ **if** $q_1 \in I_1$ **and** $q_2 \in I_2$ **then**

8 $\quad\quad\quad I \leftarrow I \cup \{q\}$

9 $\quad\quad$ **if** $q_1 \in F_1$ **and** $q_2 \in F_2$ **then**

10 $\quad\quad\quad F \leftarrow F \cup \{q\}$

11 $\quad\quad$ **for** $(q_1', q_2') \in \text{succ}(q_1) \times \text{succ}(q_2)$ **do**

12 $\quad\quad\quad$ **for** $\sigma \in \Sigma$ **do**

13 $\quad\quad\quad\quad$ **if** $q_1 \xrightarrow{\sigma} q_1' \in \Delta_1^s$ **and** $q_2 \xrightarrow{\sigma} q_2' \in \Delta_2^s$ **then**

14 $\quad\quad\quad\quad\quad \Delta^s \leftarrow \Delta^s \cup \big\{q_1 \cdot q_2 \xrightarrow{\sigma} q_1' \cdot q_2'\big\}$

15

16 $\quad \ell \leftarrow 0, \quad \mathscr{S} \leftarrow \text{build}(\text{scch}, 0), \quad \mathscr{S}_{\text{old}} \leftarrow \emptyset$

17 $\quad$ **while** $\mathscr{S} \neq \mathscr{S}_{\text{old}}$ **do**

18 $\quad\quad$ **for** $S \in \mathscr{S}$ **do**

19 $\quad\quad\quad$ **for** each $(P_{\mathcal{H}_1}, P_{\mathcal{H}_2})$ such that $P_{\mathcal{H}_1}$ stored in scch_1 and $P_{\mathcal{H}_2}$ stored in scch_2 **do**

20 $\quad\quad\quad\quad$ **if** $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2} \notin \text{query}(\text{scch}, S)$ **and** $\text{covers}(S, P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2})$ **then**

21 $\quad\quad\quad\quad\quad$ **for** $(q_1, q_2) \in \text{orig}(P_{\mathcal{H}_1}) \times \text{orig}(P_{\mathcal{H}_2})$ **do**

22 $\quad\quad\quad\quad\quad\quad \Lambda(\text{scch}, S) \leftarrow \Lambda(\text{scch}, S) \cup \Big\{\big(q_1 \cdot q_2 \to P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}\big)\Big\}$

23 $\quad\quad\quad\quad\quad\quad R(\text{scch}) \leftarrow R(\text{scch}) \cup \big\{(q_1 \cdot q_2) \xrightarrow{\ell+1} \text{repres}(S)\big\}$

24 $\quad\quad\quad\quad\quad$ **for** $(q_1, q_2) \in \text{dest}(P_{\mathcal{H}_1}) \times \text{dest}(P_{\mathcal{H}_2})$ **do**

25 $\quad\quad\quad\quad\quad\quad \Lambda(\text{scch}, S) \leftarrow \Lambda(\text{scch}, S) \cup \Big\{\big(P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2} \to q_1 \cdot q_2\big)\Big\}$

26 $\quad\quad\quad\quad\quad\quad R(\text{scch}) \leftarrow R(\text{scch}) \cup \big\{\text{repres}(S) \xrightarrow{\ell+1} (q_1 \cdot q_2)\big\}$

27 $\quad\quad \mathscr{S}_{\text{old}} \leftarrow \mathscr{S}$

28 $\quad\quad \ell \leftarrow \ell + 1, \quad \mathscr{S} \leftarrow \text{build}(\text{scch}, \ell)$

29 $\quad$ **return** $\mathcal{A}_{\mathcal{H}}$

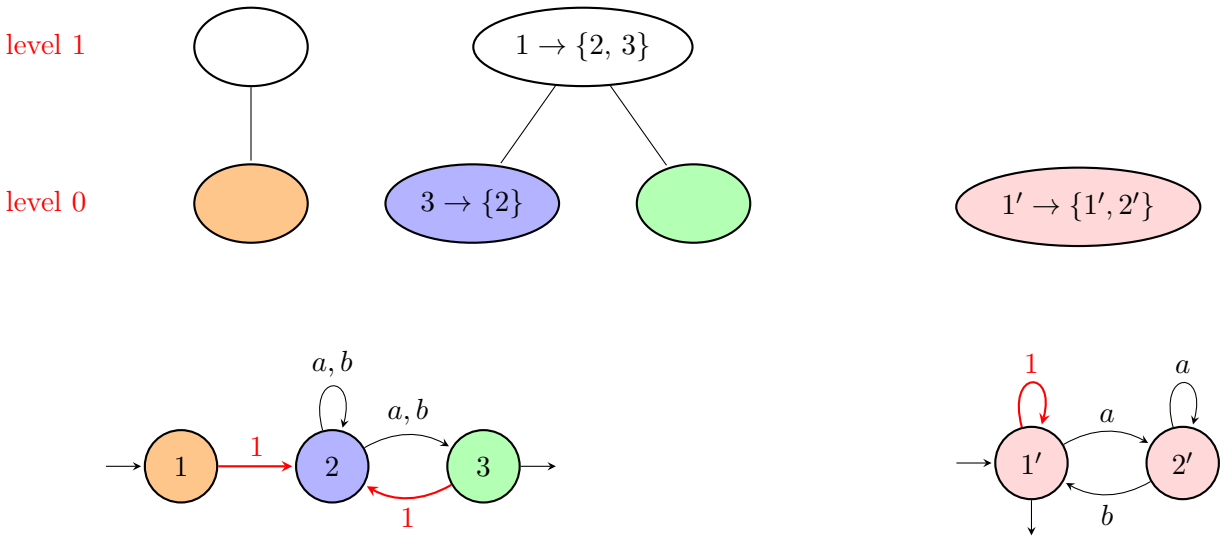
Let us formalize this idea. If $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ is stored in S, then S covers $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ (because of Line 20). This implies that $\pi_{Q_1}(S)$ covers $P_{\mathcal{H}_1}$ and $\pi_{Q_2}(S)$ covers $P_{\mathcal{H}_2}$. Thanks to the properties of the SCC hierarchies of $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$, it can then be shown easily that $P_{\mathcal{H}_1} \in \texttt{query}\big(\texttt{scch}_1, \pi_{Q_1}(S)\big)$ and $P_{\mathcal{H}_2} \in \texttt{query}\big(\texttt{scch}_2, \pi_{Q_2}(S)\big)$. The proof is almost immediate: $P_{\mathcal{H}_1}$ is covered by the lowest strongly connected component S_1 that includes $\pi_{Q_1}(S)$, and since $P_{\mathcal{H}_1}$ is stored in the lowest strongly connected component that covers it, it is stored in S_1 or in one of its descendants. The exact same argument works for $P_{\mathcal{H}_2}$.

Therefore, Line 19 can in fact be replaced by

$$\textbf{for } (P_{\mathcal{H}_1}, P_{\mathcal{H}_2}) \in \Big(\texttt{query}\big(\texttt{scch}_1, \pi_{Q_1}(S)\big) \times \texttt{query}\big(\texttt{scch}_2, \pi_{Q_2}(S)\big)\Big) \textbf{ do},$$

which is dramatically more efficient in practice. We illustrate this modification, as well as the full algorithm, in Example 33.

Example 33. *We consider once again the automata already introduced in Examples 27 and 32. They are represented in Figure 40, together with their SCC hierarchy. They look like plain automata, but here we consider them as finite history automata for which $d = 1$. In order to compute their intersection,*



(a) $\mathcal{A}_{\mathcal{H}_1}$ and its SCC hierarchy.

(b) $\mathcal{A}_{\mathcal{H}_2}$ and its SCC hierarchy.

Figure 40

the first step is to instantiate an empty finite history automaton $\mathcal{A}_{\mathcal{H}}$ and then to compute Q, I, F and Δ^s (Lines 3 to 14 in Algorithm 1). Next, one computes the level 0 of the SCC hierarchy of $\mathcal{A}_{\mathcal{H}}$. This yields the hierarchy depicted in Figure 41. Some states that are clearly not reachable or not co-reachable are not represented.

The next step is to store historic limit transitions in the strongly connected components of level 0. The orange, green and pink strongly connected components do not store any historic limit transition, because

$$\Big(\texttt{query}\big(\texttt{scch}_1, \pi_{Q_1}(S)\big) \times \texttt{query}\big(\texttt{scch}_2, \pi_{Q_2}(S)\big)\Big)$$

is the empty set for $S = \{(1, 1')\}, \{(3, 1')\}$ or $\{(3, 2')\}$. But for the blue strongly connected component, one has

$$\texttt{query}\big(\texttt{scch}_1, \pi_{Q_1}(S)\big) \times \texttt{query}\big(\texttt{scch}_2, \pi_{Q_2}(S)\big) = \Big\{\big(\{2\}, \{1', 2'\}\big)\Big\}.$$

Since the blue strongly connected component covers $\big(\{2\}, \{1', 2'\}\big)$ and does not have any descendant, the historic limit transition $(3, 1') \to \big(\{2\}, \{1', 2'\}\big)$ is stored inside it, and a red arc $(3, 1') \xrightarrow{1} (2, 2')$ is added to the reachability graph, assuming that the representative of the blue strongly connected component is $(2, 2')$. Finally, the level 1 of the hierarchy is computed. This yields the hierarchy in Figure 42. The blue and the green strongly connected components of level 0 have been merged into a strongly connected

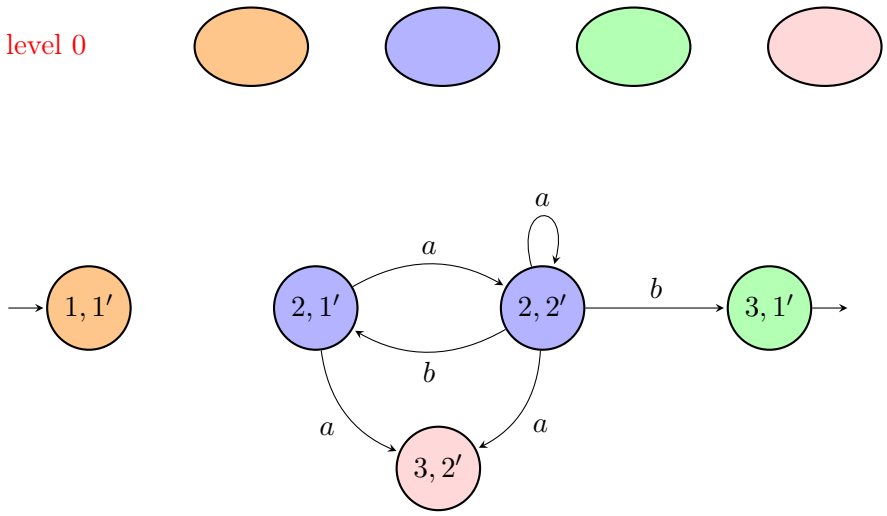


Figure 41: SCC hierarchy after computing the SCC of level 0.

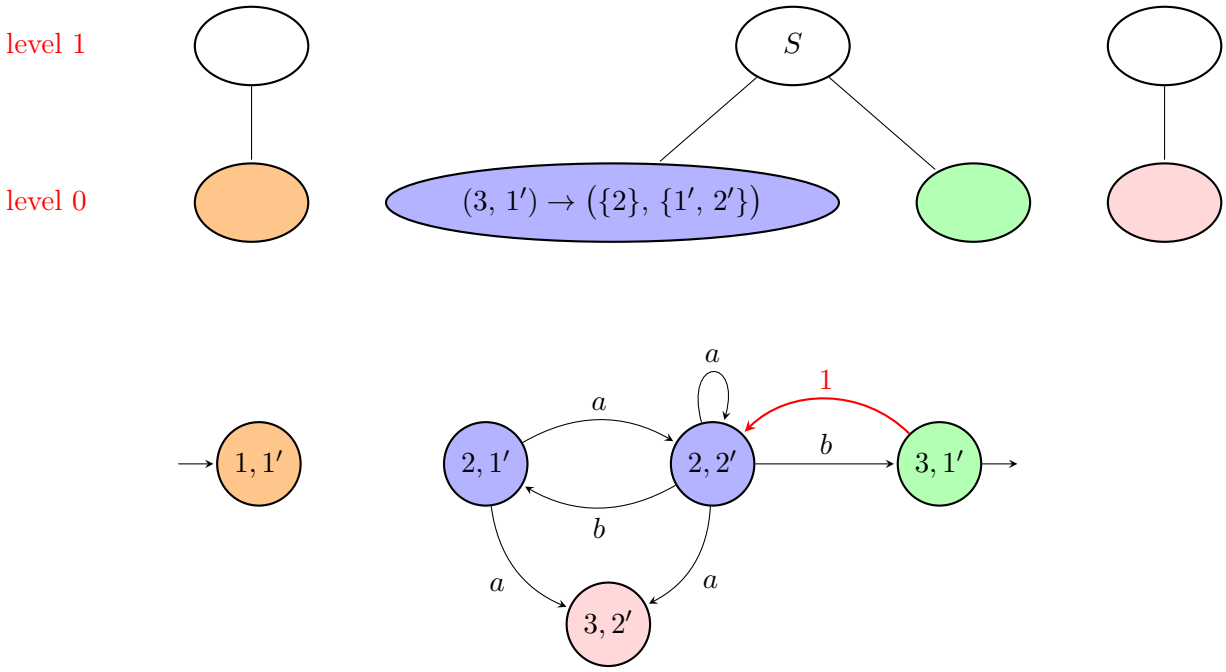


Figure 42: SCC hierarchy after computing the SCC of level 1.

component of level 1. *The only strongly connected component of level* 1 *that can store historic limit transitions is obviously* S. *We have* $\pi_{Q_1}(S) = \{2, 3\}$ *and* $\pi_{Q_2}(S) = \{1', 2'\}$. *Then, one computes*

$$\mathtt{query}(\{2, 3\}) = \{\{2, 3\}, \{2\}\}$$

and

$$\mathtt{query}(\{1', 2'\}) = \{\{1', 2'\}\}.$$

Therefore, two historic limit sets must be examined: $(\{2, 3\}, \{1', 2'\})$ *and* $(\{2\}, \{1', 2'\})$. *The latter is already stored in a descendant of* S *and is thus discarded. The former is covered by* S, *hence the historic*

limit transition $(1,1') \to \big(\{2,3\},\{1',2'\}\big)$ *is stored in* S*, and a red arc* $(1,1') \xrightarrow{2} (2,1')$ *is added to the graph. The algorithm then terminates, since the level* 2 *of the hierarchy has the same strongly connected components as the level* 1*. The SCC hierarchy at the end of the algorithm was actually represented previously in Figure 38.*

Note that there might exist more sophisticated ways of decreasing the number of pairs $(P_{\mathcal{H}_1}, P_{\mathcal{H}_2})$ examined for each strongly connected component. We opted for a simple and natural way of doing it, but there remains some space for optimization (perhaps via more evolved data structures).

5.2.5 Auto-referencing historic limit sets

In this section, we explain how a SCC hierarchy handles auto-referencing historic limit sets. An historic limit set $P_{\mathcal{H}}$ is auto-referencing if at least one element of $\langle P_{\mathcal{H}} \rangle$ is auto-referencing. Up to now, these historic limit sets are not handled properly by the intersection algorithm.

Let us start by slightly extending the formalism of SCC hierarchies in order to take auto-referencing limit sets into account. Figure 43 shows an example of an augmented SCC hierarchy, related to an automaton accepting the language $sh(a \cdot sh(b) \cdot a)$. In addition to red arcs of level ≥ 1, we added *shuffle red arcs*. These arcs are labelled by *sh* and are the only red arcs that have the level 0 (i.e., the level of the successor transitions of the automaton). For an SCC hierarchy to be valid, shuffle red arcs should verify the following property: for any historic limit set $P_{\mathcal{H}}$ stored in a SCC S of the hierarchy, if both $\texttt{select}(\texttt{orig}(P_{\mathcal{H}}), P_{\mathcal{H}})$ and $\texttt{select}(\texttt{dest}(P_{\mathcal{H}}), P_{\mathcal{H}})$ are non-empty (which means that $P_{\mathcal{H}}$ might be auto-referencing), then all the states of $\texttt{select}(\texttt{orig}(P_{\mathcal{H}}), P_{\mathcal{H}})$ must be linked to a representative of S via a shuffle red arc, and this representative should be linked to all the states of $\texttt{select}(\texttt{dest}(P_{\mathcal{H}}), P_{\mathcal{H}})$ via a shuffle red arc as well.

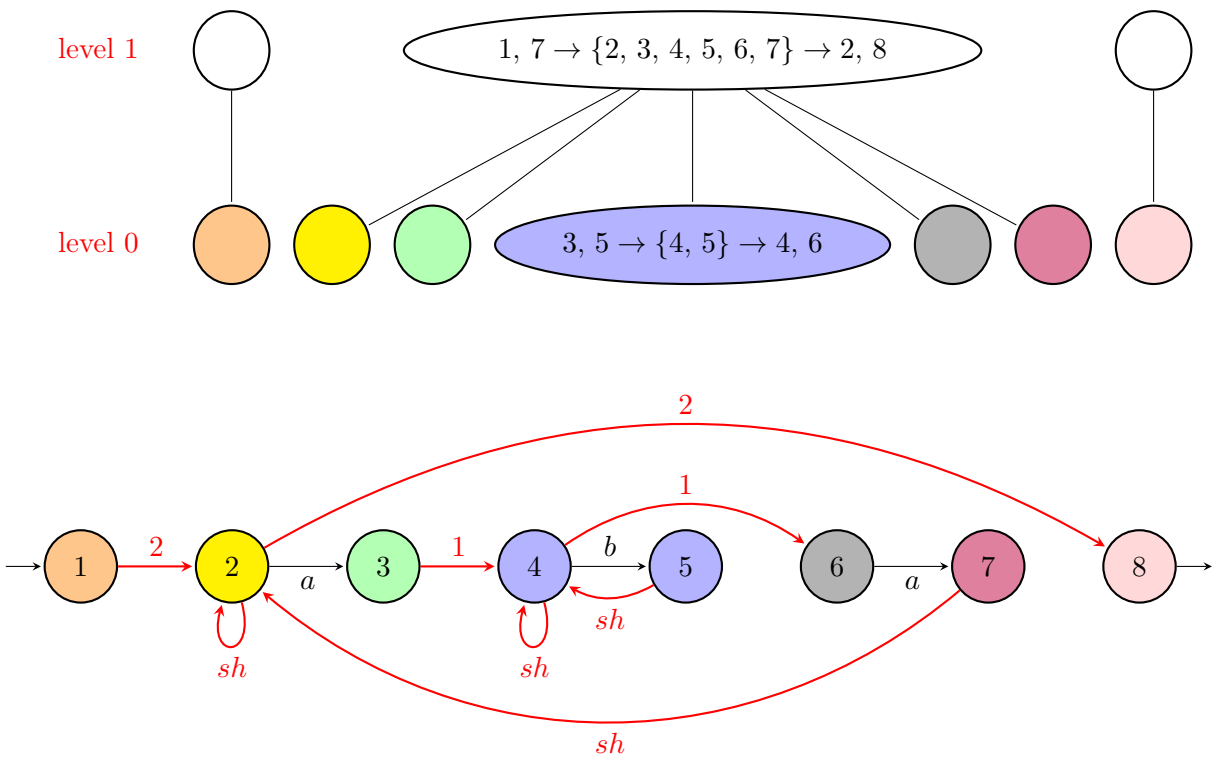


Figure 43: Example of SCC hierarchy for auto-referencing limit transitions.

In fact, the shuffle red arcs play the role of the red arcs added anticipatively in Theorem 5. Hence, thanks to shuffle red arcs, if a limit set that belongs to $\langle P_{\mathcal{H}} \rangle$ is useful in $\mathcal{A}_{\mathcal{H}}$, then it is included in a strongly connected component (even if it is auto-referencing) and $P_{\mathcal{H}}$ is thus stored in the hierarchy. Moreover, since shuffle red arcs are red arcs of level 0, they do not have any influence on the level of the strongly connected connected component in which $P_{\mathcal{H}}$ is stored.

Consider for instance the limit transitions $3,5 \to \{4,5\} \to 4,6$ in Figure 43. The limit set $\{4,5\}$ (that is considered here as an historic limit set for which $d = 1$) might be auto-referencing, since

$$\texttt{select}\big(\{3,5\},\{4,5\}\big) \neq \emptyset$$

and

$$\texttt{select}\big(\{4,6\},\{4,5\}\big) \neq \emptyset$$

This explains the presence of the shuffle red arc $5 \xrightarrow{sh} 4$ and $4 \xrightarrow{sh} 4$ (the representative is 4 in this case). As a consequence, the limit set $\{4,5\}$ is stored in the blue strongly connected component, whose level is 0. Similarly, the auto-referencing historic limit set $\{2,3,4,5,6,7\}$ is stored in a strongly connected component of level 1, thanks to the red arcs of level 1 related to the transitions $3,5 \to \{4,5\} \to 4,6$ and thanks to the shuffle red arcs $2 \xrightarrow{sh} 2$ and $7 \xrightarrow{sh} 2$. Note that the name "shuffle red arc" comes from the fact that auto-referencing limit sets are almost always related to a shuffle construct in practice.

Our intersection algorithm should now be modified in order to deal with extended SCC hierarchies. Given two operands $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$ whose SCC hierarchies contain shuffle red arcs, one should determine how shuffle red arcs must be added to the SCC hierarchy of $\mathcal{A}_{\mathcal{H}}$. Importantly, these shuffle red arcs must be added before the construction of the SCC tree, since the computation of the strongly connected components rely on them.

The solution to this problem consists in virtually adding a special symbol sh to the alphabet Σ, and to compute the intersection as before. In other words, if $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$ contain the shuffle red arcs $q_1 \xrightarrow{sh} q_1'$ and $q_2 \xrightarrow{sh} q_2'$ respectively, then a shuffle red arc should be added in $\mathcal{A}_{\mathcal{H}}$ between $q_1 \cdot q_2$ and $q_1' \cdot q_2'$. This small modification of the first phase of the algorithm ensures that an auto-referencing historic limit set $P_{\mathcal{H}} = P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ is correctly treated later. Indeed, assuming that $P_{\mathcal{H}}$ is auto-referencing, one has

$$\begin{cases} \texttt{select}(\texttt{orig}(P_{\mathcal{H}}), P_{\mathcal{H}}) \neq \emptyset \\ \texttt{select}(\texttt{dest}(P_{\mathcal{H}}), P_{\mathcal{H}}) \neq \emptyset. \end{cases}$$

Since

$$\begin{cases} \texttt{orig}(P_{\mathcal{H}}) = \{q_1 \cdot q_2 \mid q_1 \in \texttt{orig}(P_{\mathcal{H}_1}) \wedge q_2 \in \texttt{orig}(P_{\mathcal{H}_2})\} \\ \texttt{dest}(P_{\mathcal{H}}) = \{q_1 \cdot q_2 \mid q_1 \in \texttt{dest}(P_{\mathcal{H}_1}) \wedge q_2 \in \texttt{dest}(P_{\mathcal{H}_2})\} \end{cases}$$

it follows that

$$\begin{cases} \texttt{select}(\texttt{orig}(P_{\mathcal{H}_1}), P_{\mathcal{H}_1}) \neq \emptyset \\ \texttt{select}(\texttt{dest}(P_{\mathcal{H}_1}), P_{\mathcal{H}_1}) \neq \emptyset \end{cases} \quad \text{and} \quad \begin{cases} \texttt{select}(\texttt{orig}(P_{\mathcal{H}_2}), P_{\mathcal{H}_2}) \neq \emptyset \\ \texttt{select}(\texttt{dest}(P_{\mathcal{H}_2}), P_{\mathcal{H}_2}) \neq \emptyset \end{cases}$$

and therefore, the origin states of $P_{\mathcal{H}_1}$ are all linked via shuffle red arcs to a representative, which is itself linked by such arcs to the destinations of $P_{\mathcal{H}_1}$ (since $\mathcal{A}_{\mathcal{H}_1}$ is equipped with a valid SCC hierarchy, by assumption). The same conclusion holds for $P_{\mathcal{H}_2}$, and thus for $P_{\mathcal{H}}$ as well (thanks to the modification of the algorithm). All the shuffle red arcs that must be present in the hierarchy of $\mathcal{A}_{\mathcal{H}}$ are thus indeed added to it.

Note that the shuffle red arcs do not influence the language accepted by an automaton. They should not be considered as successor transitions. As all red arcs, their purpose is only to encode reachability.

5.2.6 Deleting some states that are not reachable or not co-reachable

Our intersection algorithm yields a finite history automaton equipped with an SCC hierarchy. The reachability graph of the SCC hierarchy offers a simple solution to remove some states that are not reachable or not co-reachable. Indeed, we know that if a state is reachable, there exists a path connecting an initial state to it in the reachability graph (and a similar property holds for co-reachable states). Therefore, all the states for which such a path does not exist can be deleted.

Formally, one can introduce a function `filter_states`($\mathcal{A}_{\mathcal{H}}$) that exploits this idea. This function proceeds in 4 steps:

1. Add red arcs from each final state to each initial state of $\mathcal{A}_{\mathcal{H}}$. The level of these arcs does not matter.
2. Compute the strongly connected components of the reachability graph, taking into account all the red arcs (whatever their levels).
3. Keep the states that are included in a strongly connected component that contains an initial and a final state. Remove all the other states.
4. Remove the red arcs added in Step 1.

It is easy to see that if a state q is deleted, then there does not exist any path from an initial state to q, or there does not exist any path from q to a final state. Indeed, if we assume that two such paths exist, then

- there exists a path from i to q, where i is initial,
- there exists a path from q to f, where f is final, and
- there exists a path from f to i, thanks to the red arcs added in Step 1.

Hence, q, i and f are in the same strongly connected component, and q is thus not deleted.

We did not describe the `filter_states` function in pseudo-code, since it would require to define precisely what it means to "delete" a state (it is not trivial to do that without corrupting the SCC hierarchy). We will see in Chapter 6 how it can be done in practice.

5.2.7 Instantiating less states

Currently, the intersection algorithm instantiates all the states $q_1 \cdot q_2$ where q_1 and q_2 are states of $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$, respectively. But in practice, many of these states are likely to be useless (e.g., not reachable) and should not be instantiated. This is crucial, since instantiating less states makes it possible to ignore some historic limit sets.

In the case of finite-word automata, it is possible to design an algorithm that only instantiates reachable states. Such an algorithm proceeds as follows:

1. Given two finite-word automata $\mathcal{A}_1$ and $\mathcal{A}_2$ (with sets of states Q_1 and Q_2, respectively), the algorithm first instantiates an empty finite-word automaton $\mathcal{A}$.

2. A set Q_{explore} is created. It contains all the pairs of states $(q_1, q_2) \in Q_1 \times Q_2$ that need to be instantiated and explored in $\mathcal{A}$. Initially, Q_{explore} contains all the pairs (q_1, q_2) such that q_1 and q_2 are initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Then, the following operations are repeated until Q_{explore} is empty:
   (a) A pair (q_1, q_2) is removed from Q_{explore}, and the state $q = (q_1, q_2)$ is instantiated in $\mathcal{A}$.
   (b) All the transitions $(q_1, q_2) \xrightarrow{\sigma} (q_1', q_2')$ such that $q_1 \xrightarrow{\sigma} q_1'$ and $q_2 \xrightarrow{\sigma} q_2'$ are transitions in $\mathcal{A}_1$ and $\mathcal{A}_2$ (for some $\sigma \in \Sigma$) are added to $\mathcal{A}$. If (q_1', q_2') is not yet instantiated in $\mathcal{A}$, it is added to Q_{explore}.

This algorithm can be generalized to automata on linear orderings. Currently, Algorithm 1 can be divided into two main blocks: it first instantiates all the states, and then adds historic limit transitions. The idea is to interleave these two phases. The *expansion phase* consists in expanding the intersection automaton by instantiating new states that can be reached by successor transitions or right-limit transitions. When this phase ends (i.e., when Q_{explore} is empty), the *limit set phase* starts. During this phase, the SCC hierarchy is built, historic limit transitions and red arcs are added to it, and new states that can be reached by a left-limit transition are added to Q_{explore}. Next, the expansion phase starts again. The algorithm terminates when the two uppermost levels of the SCC hierarchy are identical (as in Algorithm 1). Intuitively, such an algorithm can be seen as a sequence of intersections of finite-word automata (i.e., expansion phases), interleaved with limit set phases.

The expansion phase is described in Algorithm 2. It takes as argument the operands $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$, the intersection automaton $\mathcal{A}_{\mathcal{H}}$ (that can be empty or partially computed) and the set of states Q_{explore} from which the expansion should begin. Lines 3 to 17 straightforwardly translate the intersection algorithm for finite-word automata, except that it handles the shuffle red arcs in Lines 16 and 17.

As opposed to the case of finite-word automata, one should also take into account the possibility of following a right-limit transition simultaneously in each operand. This is done in Lines 18 to 20: for each pair $(q_1 \to P_{\mathcal{H}_1}, q_2 \to P_{\mathcal{H}_2})$ of historic right-limit transitions, the function `handle_right_limit_trans` computes a subset $Q_{P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}} \subseteq Q$ whose instantiation guarantees that $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ will be stored in the SCC hierarchy during a future limit set phase, if an element of $\langle P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2} \rangle$ is useful. The states from $Q_{P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}}$ that are not yet instantiated are then added to Q_{explore} in Line 20.

The function `handle_right_limit_trans` is described in Algorithm 3. It takes as arguments $\mathcal{A}_{\mathcal{H}_1}$, $\mathcal{A}_{\mathcal{H}_2}$, $\mathcal{A}_{\mathcal{H}}$ (which is only partially computed), a state $q_1 \cdot q_2$ and two historic limit sets $P_{\mathcal{H}_1}$ and $P_{\mathcal{H}_2}$ such that $q_1 \to P_{\mathcal{H}_1}$ and $q_2 \to P_{\mathcal{H}_2}$ are historic right-limit transitions in $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$, respectively. In Lines 9 to 12, one first retrieves all the pairs (S_1, S_2) of strongly connected components such that $P_{\mathcal{H}_1}$ is stored in S_1 and $P_{\mathcal{H}_2}$ in S_2. Then, for each pair, one adds to the set $Q_{P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}}$ (initially empty) all the states $q_1 \cdot q_2$ such that $q_1 \in \texttt{select}(S_1, P_{\mathcal{H}_1})$ and $q_2 \in \texttt{select}(S_2, P_{\mathcal{H}_2})$. In other words, all the states in $\mathcal{A}_{\mathcal{H}}$ that could potentially belong to a useful limit set of $\langle P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2} \rangle$ are added to $Q_{P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}}$. Lines 3 to 7 simply check whether $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ has already been added to the hierarchy during a previous limit set phase. If it is the case, historic right-limit transitions $q_1 \cdot q_2 \to P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ need to be added in each strongly connected component that already stores $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$, and red arcs must translate these transitions.

We are now ready to write the final version of the intersection algorithm for finite history automata. Algorithm 4 combines all the improvements discussed so far: the reduction of the number of pairs $(P_{\mathcal{H}_1}, P_{\mathcal{H}_2})$ examined in each strongly connected component, the use of shuffle red arcs, the deletion of some states that are not reachable or not co-reachable, and the interleaving of expansion and limit set phases. The limit set phase spans Lines 16 to 28. It is almost identical to what was already done in

Algorithm 2: Expansion sub-routine.

```
1  expansion (A_{H1} = (d1, Q1, Δ1^s, scch1, I1, F1), A_{H2} = (d2, Q2, Δ2^s, scch2, I2, F2),
2             A_H = (d1 + d2, Q, Δ^s, scch, I, F), Q_start ⊆ {q1 · q2 | q1 ∈ Q1 ∧ q2 ∈ Q2}):
3    while Q_explore ≠ ∅ do
4      Let q1 · q2 ∈ Q_explore (with q1 ∈ Q1, q2 ∈ Q2).
5      Q ← Q ∪ {q1 · q2}   // Instantiation of q1 · q2
6      if q1 ∈ I1 and q2 ∈ I2 then
7        I ← I ∪ {q1 · q2}
8      if q1 ∈ F1 and q2 ∈ F2 then
9        F ← F ∪ {q1 · q2}
10     for (q1', q2') ∈ succ(q1) × succ(q2) do
11       for σ ∈ Σ do
12         if q1 -σ-> q1' ∈ Δ1^s and q2 -σ-> q2' ∈ Δ2^s then
13           Δ^s ← Δ^s ∪ {q1 · q2 -σ-> q1' · q2'}
14           if q1' · q2' ∉ Q then
15             Q_explore ← Q_explore ∪ {q1' · q2'}
16       if q1 -sh-> q1' ∈ R(scch1) and q2 -sh-> q2' ∈ R(scch2) then
17         R(scch) ← R(scch) ∪ {q1 · q2 -sh-> q1' · q2'}
18     for (P_{H1}, P_{H2}) ∈ rightlim(q1) × rightlim(q2) do
19       Q_{P_{H1}·P_{H2}} ← handle_right_lim_trans(q1 · q2, P_{H1}, P_{H2}, A_{H1}, A_{H2}, A_H)
20       Q_explore ← Q_explore ∪ (Q_{P_{H1}·P_{H2}} \ Q)
21     Q_explore ← Q_explore \ {q1 · q2}
22   return
```

Algorithm 1, except that it considers less pairs $(P_{\mathcal{H}_1}, P_{\mathcal{H}_2})$ and adds states in $\mathrm{Q}_{\text{explore}}$. The expansion phase is carried out by the `expansion` function in Line 9. The remainder of the algorithm consists in interleaving correctly the two phases, and in deciding when the algorithm must stop.

Recall that the `build` function used in Line 16 only adds strongly connected components of level ℓ to the hierarchy, it does not recompute the strongly connected components of level ℓ that are already present. It means that the instruction `build(scch,` ℓ`)` can be executed only if all the necessary successor transitions and red arcs of level $\leq \ell$ have been added before. For instance, it would be incorrect to execute the instruction `build(scch, 1)` when the current set of states in $\mathcal{A}_{\mathcal{H}}$ is Q, and then to add a red arc of level 1 between two states of Q. Indeed, it would require to recompute the strongly connected components of level 1, which is impossible to do with the function `build`.

In fact, it can be shown that this problem cannot occur. Intuitively, Algorithm 4 builds the SCC hierarchy as a wall. The level ℓ is completed only if all the lower levels have been completed before. When a new state that can be reached via a left-limit transition is discovered (Line 25), a new expansion phase must start. New states are thus instantiated, and the level 0 of the SCC hierarchy must be completed again. Once it is done, level 1 can be completed, etc.

Algorithm 3: Right limit transitions sub-routine.

```
1  handle_right_lim_trans (q1 · q2, P_{H1}, P_{H2}, A_{H1} = (d1, Q1, Δ1^s, scch1, I1, F1),
2                          A_{H2} = (d2, Q2, Δ2^s, scch2, I2, F2), A_H = (d1 + d2, Q, Δ^s, scch, I, F)):
3      if P_{H1} · P_{H2} already stored in scch then
4          for S ∈ Γ(scch, P_{H1} · P_{H2}) do
5              ℓ ← level(S)
6              R(scch) ← R(scch) ∪ {q1 · q2 --ℓ+1--> repres(S)}
7              Λ(scch, S) ← Λ(scch, S) ∪ {(q1 · q2) → P_{H1} · P_{H2}}
8
9      Q_{P_{H1}·P_{H2}} ← ∅
10     for (S1, S2) ∈ Γ(scch1, P_{H1}) × Γ(scch2, P_{H2}) do
11         for (q1, q2) ∈ (select(S1, P_{H1}), select(S2, P_{H2})) do
12             Q_{P_{H1}·P_{H2}} ← Q_{P_{H1}·P_{H2}} ∪ {q1 · q2}
13     return Q_{P_{H1}·P_{H2}}
```

Formally, the algorithm has the property stated in Theorem 7, which ensures that the strongly connected components added when running `build(scch, ℓ)` will never have to be recomputed. They are permanent.

Theorem 7. *Let Q_t be the set of states already instantiated at time t where the instruction `build(scch, ℓ)` is executed. After this execution, Algorithm 4 does not add any additional successor transitions or red arcs of level $\leq \ell$ that leaves a state of Q_t.*

Proof. Observe that Algorithm 4 maintains a set Q_{explore} of states that need to be instantiated and explored by the `expansion` function. At the beginning of each iteration of the while loop, one checks whether this set is empty. If no, the `expansion` function is called and the level ℓ is reset to 0. Otherwise, ℓ is incremented by 1. This implies that if the instruction `build(scch, ℓ)` is executed with $\ell > 1$, the instruction `build(scch, ℓ − 1)` was executed during the previous iteration of the loop. This can be exploited for an induction.

First, consider the case where $\ell = 0$. The successor transitions and the red arcs of level ≤ 0 (i.e., the shuffle red arcs) are added in $\mathcal{A}_{\mathcal{H}}$ via the `expansion` function only. Hence, a successor transition or a red arc of level ≤ 0 leaving a state of Q_t cannot appear in the future, because it must have been added to $\mathcal{A}_{\mathcal{H}}$ by the expansion function previously.

Now consider the instruction `build(scch, k)` with $k > 0$. We know that `build(scch, k − 1)` was executed during the previous iteration of the loop. Moreover, no state was instantiated during these two instructions (since any call to `expansion` is followed by $\ell \leftarrow 0$). Therefore, by induction hypothesis, after the execution of `build(scch, k)` the algorithm does not add any successor transition or red arc of level $\leq k - 1$ that leaves a state of Q_t. Moreover, after the execution of `build(scch, k)`, there are only two ways of adding a red arc of level k, and in each case the origin state of the red arc cannot belong to Q:

- The first way is to reach Line 6 of Algorithm 3. But in this case, the origin of the red arc is a state that was instantiated right before, in Algorithm 2. Therefore, it cannot belong to Q_t.

Algorithm 4: Intersection of finite history automata (final version).

```
1  FHA_binary_intersection (A_H1 = (d1, Q1, Δ1^s, scch1, I1, F1), A_H2 = (d2, Q2, Δ2^s, scch2, I2, F2)):
2      A_H ← new empty finite history automaton (d1 + d2, Q = ∅, Δ^s = ∅,
3                                                scch = scch_new_empty(), I = ∅, F = ∅)
4      Q_explore ← {q1 · q2 | q1 ∈ I1 ∧ q2 ∈ I2}   // assumed non empty
5      starting_new_level ← True,   modified ← True
6      while modified or (¬ starting_new_level) do
7          modified ← False
8          if Q_explore ≠ ∅ then
9              expansion(Q_explore, A_H1, A_H2, A_H)   // Q_explore is empty after this
10             modified ← True
11             ℓ ← 0
12         else
13             ℓ ← ℓ + 1
14         starting_new_level ← (ℓ == height(scch))
15
16         S ← build(scch, ℓ)
17         for S ∈ S do
18             for (P_H1, P_H2) ∈ (query(scch1, π_Q1(S)) × query(scch2, π_Q2(S))) do
19                 if P_H1 · P_H2 ∉ query(scch, S) and covers(S, P_H1 · P_H2) then
20                     for (q1, q2) ∈ orig(P_H1) × orig(P_H2) do
21                         if (q1 · q2) ∈ Q then
22                             Λ(scch, S) ← Λ(scch, S) ∪ {(q1 · q2) → P_H1 · P_H2}
23                             R(scch) ← R(scch) ∪ {(q1 · q2) --ℓ+1--> repres(S)},   modified ← True
24                     for (q1, q2) ∈ dest(P_H1) × dest(P_H2) do
25                         if (q1 · q2) ∉ Q then
26                             Q_explore ← Q_explore ∪ {q1 · q2}
27                         Λ(scch, S) ← Λ(scch, S) ∪ {P_H1 · P_H2 → (q1 · q2)}
28                         R(scch) ← R(scch) ∪ {repres(S) --ℓ+1--> (q1 · q2)},   modified ← True
29     filter_states(A_H)
30     return A_H
```

- The second way is to reset the level to 0 after an expansion, and then to increase it progressively until $k-1$. Red arcs of level k can then be added in Line 23 or 28 of Algorithm 4.
  - In Line 28, the origin of the red arc ($\texttt{repres}(S)$) cannot belong to Q because S is a strongly connected whose states are not in Q (otherwise, S would have been examined earlier, i.e., before the level was reset to 0).

– In Line 23, it can also be shown that the origin $q_1 \cdot q_2$ of the red arc does not belong to Q_t. Since $(q_1, q_2) \in \texttt{orig}(P_{\mathcal{H}_1}) \times \texttt{orig}(P_{\mathcal{H}_2})$, the states from the set $Q_{P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}}$ are instantiated (in Algorithm 3) right after $q_1 \cdot q_2$, i.e., during the same expansion phase. Next, define the *highest level* of a state as the level of the highest strongly connected component that already exists in the hierarchy and that includes this state. It is easy to check that the states from $Q_{P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}}$ and $q_1 \cdot q_2$ always share the same highest level (intuitively, they are instantiated simultaneously and then their highest levels remain equal). When the instruction in Line 23 is executed, there exists at least one state in $Q_{P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}}$ whose highest level is $k-1$ (otherwise, the condition in Line 19 would never be satisfied). The highest level of $q_1 \cdot q_2$ must thus be equal to $k-1$ as well. This implies that $q_1 \cdot q_2 \notin Q_t$, since the highest level of the states of Q_t is $\geq k$.

□

Algorithm 4 exits the main loop when a new level of the hierarchy has been built from scratch, without yielding any modification (i.e., new red arcs). This condition is tracked thanks to the Boolean variables `modified` and `starting_new_level`. The variable `modified` is set to False at the beginning of each iteration of the loop, and set to True afterwards only if an expansion phase occurs, or if at least one red arc is added during the limit set phase. The variable `starting_new_level` is recomputed before each limit set phase and set to True only if the current level ℓ is equal to the height of the SCC hierarchy, i.e., if the limit set phase that follows will compute the level ℓ of the hierarchy from scratch. If it is the case and modifications do not occur, the condition in Line 6 is not satisfied, and the loop ends. It would be useless to remain in the loop, since building a new level of the hierarchy would not result in any new strongly connected component (it would simply result in building a new level with the exact same strongly connected components). The final step is to delete some states that are not reachable or not co-reachable thanks to the function `filter_states`.

Now that the full algorithm has been analysed, it is worth mentioning that historic right-limit sets can be handled more efficiently in Algorithm 3. In fact, it is possible to compute a smaller set $Q_{P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}}$, while keeping the guaranty that instantiating the states from this set ensures that $P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}$ will be stored in the hierarchy during a future limit set phase, if a limit set from $\langle P_{\mathcal{H}} \rangle$ is useful. This is important, since decreasing the number of states instantiations makes the SCC hierarchy smaller.

To understand how it is possible, consider the example of two historic right-limit transitions $1 \to \{2,3\}$ and $1' \to \{1',2'\}$, that belong to $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$, respectively (these right-limit transitions are directly taken from Example 33). These historic limit transitions are such that $d = 1$, but the explanations that follow can be generalized to any d. Assume that $\{2,3\}$ is stored in a single strongly connected component S_1, and $\{1',2'\}$ in a single strongly connected component S_2. Then, the instruction $\texttt{handle\_right\_lim\_trans}\big((1,1'), \{2,3\}, \{1',2'\}, \mathcal{A}_{\mathcal{H}_1}, \mathcal{A}_{\mathcal{H}_2}, \mathcal{A}_{\mathcal{H}}\big)$ (with $\mathcal{A}_{\mathcal{H}}$ the intersection automaton that is currently computed) returns

$$Q_{P_{\mathcal{H}_1} \cdot P_{\mathcal{H}_2}} = \big\{(2,1'), (2,2'), (3,1'), (3,2')\big\}.$$

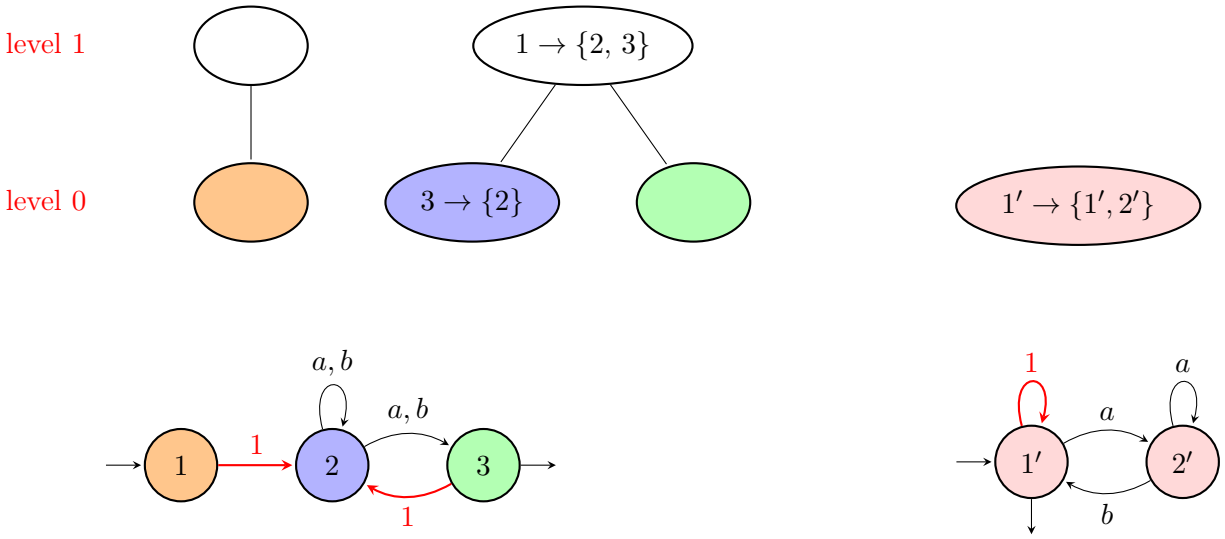
However, it suffices to instantiate, for instance, the states $(2,1')$ and $(2,2')$, since any limit set in $\langle \{2,3\}, \{1',2'\} \rangle$ must include at least one of these two states (otherwise, 2 would not be covered). Therefore, if there exists $P \in \langle \{2,3\}, \{1',2'\} \rangle$ that is useful, instantiating $(2,1')$ and $(2,2')$ guarantees that later (after potentially many expansion and limit set phases), P will be included in a strongly connected component, and therefore $(\{2,3\}, \{1',2'\})$ will be stored in the hierarchy.

This idea can be straightforwardly generalized to historic right-limit sets with $d > 1$. We do not enter into the details here, but the implementation of Algorithm 3 exploits successfully this idea.

5.2.8 Recapitulative examples

Let us illustrate the final intersection algorithm via two examples that we already used throughout this chapter.

Example 34. *Consider once again the two finite history automata depicted in Figure 44. Let us apply Algorithm 4 to compute their intersection. Initially,* $Q_{explore}$ *is set to* $\{(1, 1')\}$*, since the only initial state of the first (resp. second) operand is* 1 *(resp.* $1'$*). The first expansion phase thus instantiates* $(1, 1')$*, and then handles the right-limit transitions* $1 \to \{2, 3\}$ *and* $1' \to \{1', 2'\}$*. We showed in Section 5.2.7 that instantiating* $(2, 1')$ *and* $(2, 2')$ *suffices. Next, going through the entire expansion phase leads to Figure 45. After this phase,* $Q_{explore}$ *is empty. Next, a limit set phase starts with* $\ell = 0$*. The historic*



(a) First operand.

(b) Second operand.

Figure 44

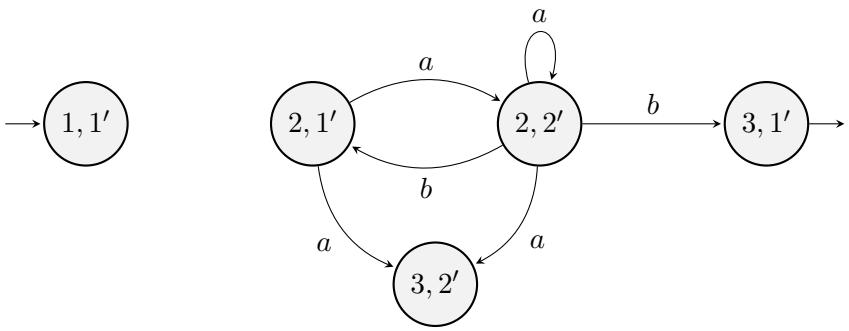


Figure 45: Intersection automaton after the first expansion phase.

limit set $(\{2\}, \{1', 2'\})$ *is stored in the blue strongly connected component (see Figure 46), and a red arc* $(3, 1') \xrightarrow{1} (2, 2')$ *is added to the hierarchy (these steps were already discussed in Example 33). After this limit set phase,* $Q_{explore}$ *is still empty, hence no expansion phase is required and a new limit set phase with* $\ell = 1$ *begins. The blue and the green strongly connected components are merged into a strongly connected component of level* 1*, which stores* $(1, 1') \to \big(\{2, 3\}, \{1', 2'\}\big)$*. A red arc* $(1, 1') \xrightarrow{2} (2, 1')$ *is*

added. Once again, $Q_{explore}$ is still empty and a limit set phase with $\ell = 2$ thus starts immediately. Note that:

- *The level 2 of the hierarchy has not been computed before, i.e., this limit set phase builds this level from scratch.*
- *This phase does not add any red arc to the hierarchy.*

Therefore, one exits the main loop. The resulting intersection automaton is identical to the one already obtained in Example 33, except that Algorithm 4 deletes State $(3, 2')$ at the end, since this state is not co-reachable.

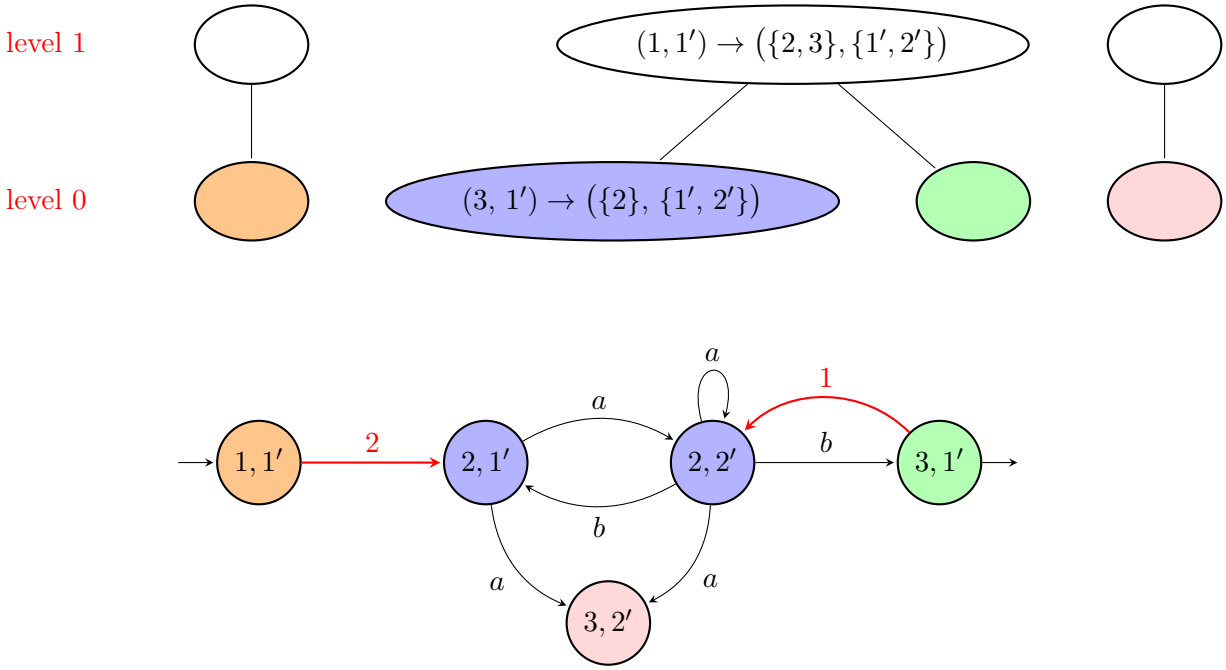
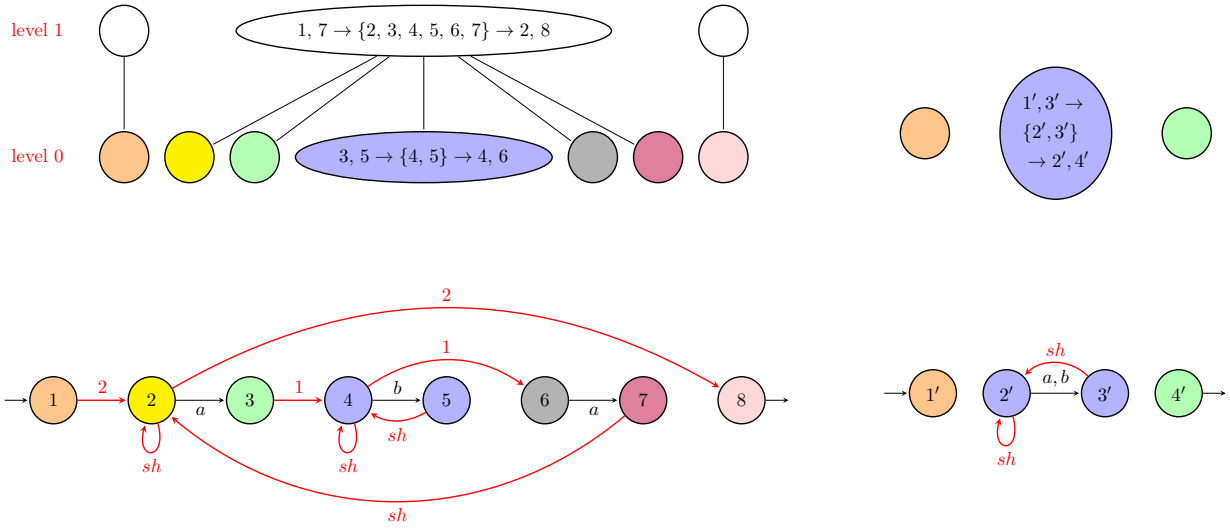


Figure 46: Intersection automaton after two limit set phases.

Example 35. *Let us compute the intersection between two automata accepting the languages $sh(a \cdot sh(b) \cdot a)$ and $sh(a + b)$, respectively. These automata are represented in Figure 47, along with their SCC hierarchies. Note that the first automaton was already used in Section 5.2.5 to illustrate the concept of shuffle red arc.*

Initially, $\mathrm{Q}_{explore}$ contains the state $(1, 1')$ only. The first expansion phase (which is summarized in Figure 48) starts by handling the right limit transitions $1 \to \{2, 3, 4, 5, 6, 7\}$ and $1' \to \{2', 3'\}$. It suffices to add in $\mathrm{Q}_{explore}$ the states $(2, 2')$ and $(2, 3')$. The latter is actually unreachable (it can be easily checked "by hand") and is thus not represented in Figure 48, to keep the following pictures simple. This state would in fact be deleted at the end of the algorithm with the function `filter_states`*. Next, the expansion phase discovers the state $(3, 3')$ and must then handle the right limit transitions $3 \to \{4, 5\}$ and $3' \to \{2', 3'\}$, by instantiating $(4, 2')$ and $(4, 3')$, which is once again unreachable. Finally, the expansion function discovers $(5, 3')$ and, importantly, adds the shuffle red arc $(5, 3') \xrightarrow{sh} (4, 2')$.*

A limit set phase then starts, with $\ell = 0$. The only strongly connected components of level 0 that can store an historic limit set is the blue one, which stores $\big(\{4, 5\}, \{2', 3'\}\big)$. Consequently, red arcs of level



(a) First operand. (b) Second operand.

Figure 47

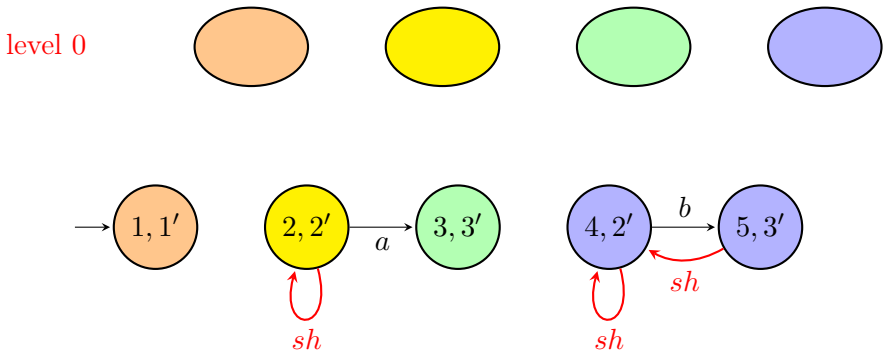


Figure 48: Intersection automaton after the first expansion phase and the computation of the SCC of level 0.

1 *are added between the origin states of* $(\{4,5\},\{2',3'\})$ *that are instantiated and the representative* $(4,2')$*: these red arcs are* $(3,3') \xrightarrow{1} (4,2')$ *and* $(5,3') \xrightarrow{1} (4,2')$*. The latter has no impact (since the red arc* $(5,3') \xrightarrow{sh} (4,2')$ *already exists), hence it is not represented in Figure 49. Next, the destinations of* $(\{4,5\},\{2',3'\})$ *that are not yet instantiated are added in* $\mathrm{Q}_{explore}$*. Since* $(4,2')$ *is already instantiated and* $(4,4')$*,* $(6,4')$ *are not co-reachable, we assume that only* $(6,2')$ *is added to* $\mathrm{Q}_{explore}$ *(once again, for convenience). The red arcs* $(4,2') \xrightarrow{1} (4,2')$ *and* $(4,2') \xrightarrow{1} (6,2')$ *are then added to the hierarchy. Once again, since* $(4,2') \xrightarrow{sh} (4,2')$ *already exists, the red arc* $(4,2') \xrightarrow{1} (4,2')$ *is not represented.*

The set $\mathrm{Q}_{explore}$ *is not empty, hence an expansion phase is necessary and* ℓ *is reset to* 0*. This expansion phase simply discovers* $(7,3')$*, since this state can be reached from* $(6,2')$ *by a successor transition. It also adds the shuffle red arc* $(7,3') \xrightarrow{sh} (2,2')$ *The current intersection automaton is represented in Figure 49. The next limit set phase (of level* 0*) adds the gray and purple strongly connected components to the SCC tree, but does not add red arcs or states in* $\mathrm{Q}_{explore}$*.*

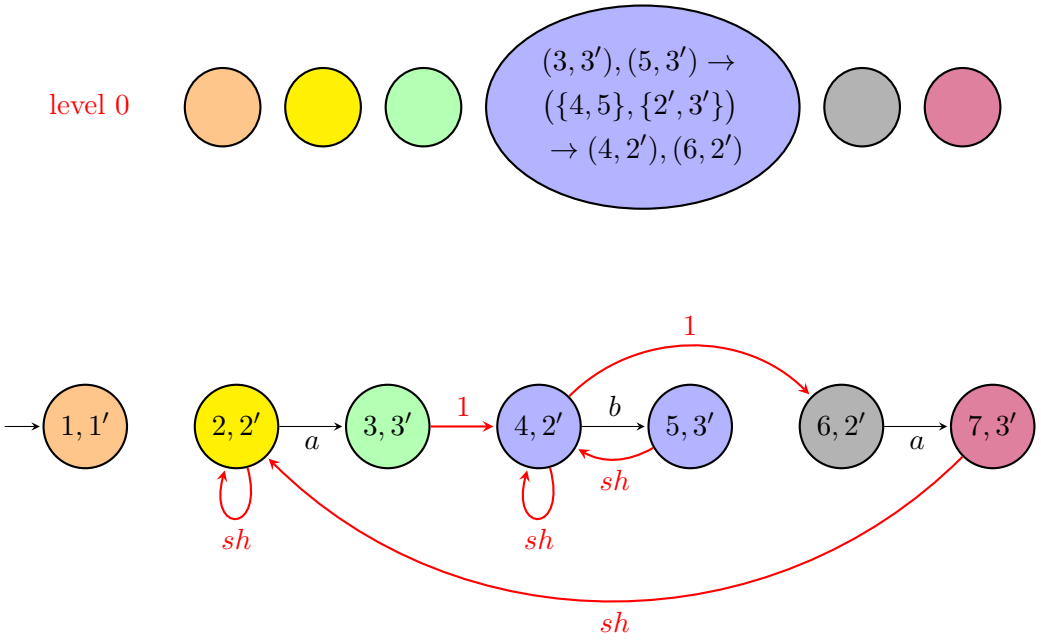


Figure 49: Intersection automaton after the second expansion phase and the computation of the new SCC of level 0.

Another limit set phase with $\ell = 1$ thus starts immediately. It discovers the level 1 strongly connected component

$$\{(2,2'),(3,3'),(4,2'),(5,3'),(6,2'),(7,3')\},$$

which stores the historic limit set $(\{2,3,4,5,6,7\},\{2',3'\})$. Amongst its potential origin states $(1,1')$, $(1,3')$, $(7,1')$ and $(7,3')$, only $(1,1')$ and $(7,3')$ are already instantiated. Since $(7,3') \xrightarrow{2} (2,2')$ does not have any impact, we only add the red arc $(1,1') \xrightarrow{2} (2,2')$ in Figure 50. Then, the destination states must be handled: $(2,2')$ is already instantiated, $(2,4')$ and $(8,2')$ are not co-reachable, hence only $(8,4')$ is added in $\mathrm{Q}_{explore}$ (and observe that this state is final). Red arcs $(2,2') \xrightarrow{2} (2,2')$ and $(2,2') \xrightarrow{2} (8,4')$ are added, but the former is not represented since $(2,2') \xrightarrow{sh} (2,2')$ already exists.

Since a new state has been added to $\mathrm{Q}_{explore}$, an expansion phase must start. It does not add anything in the hierarchy. Next, ℓ is reset to 0, and a limit set phase of level 0 simply adds the pink SCC in the SCC tree (see Figure 50). The level is increased by 1. A new limit set phase of level 1 is carried out, but it does not add any red arc. Finally, limit set phase of level 2 starts. It does compute the level 2 of the hierarchy from scratch, and does not add any red arc. Hence, the algorithm terminates.

Formally, one should now run the function `filter_states`*. But we actually did it on the fly, by instantiating only the states that are reachable and co-reachable. The output of the algorithm is thus exactly depicted in Figure 50. It is not difficult to see that this automaton is equivalent to the first operand in Figure 47, which accepts $sh(a \cdot sh(b) \cdot a)$. In fact, this shows that $sh(a+b)$ contains words of length $\sum_{j\in\mathbb{R}} \{1\} + \mathbb{R} + \{1\}$. We already mentioned a similar property in Example 7, without giving a formal proof.*

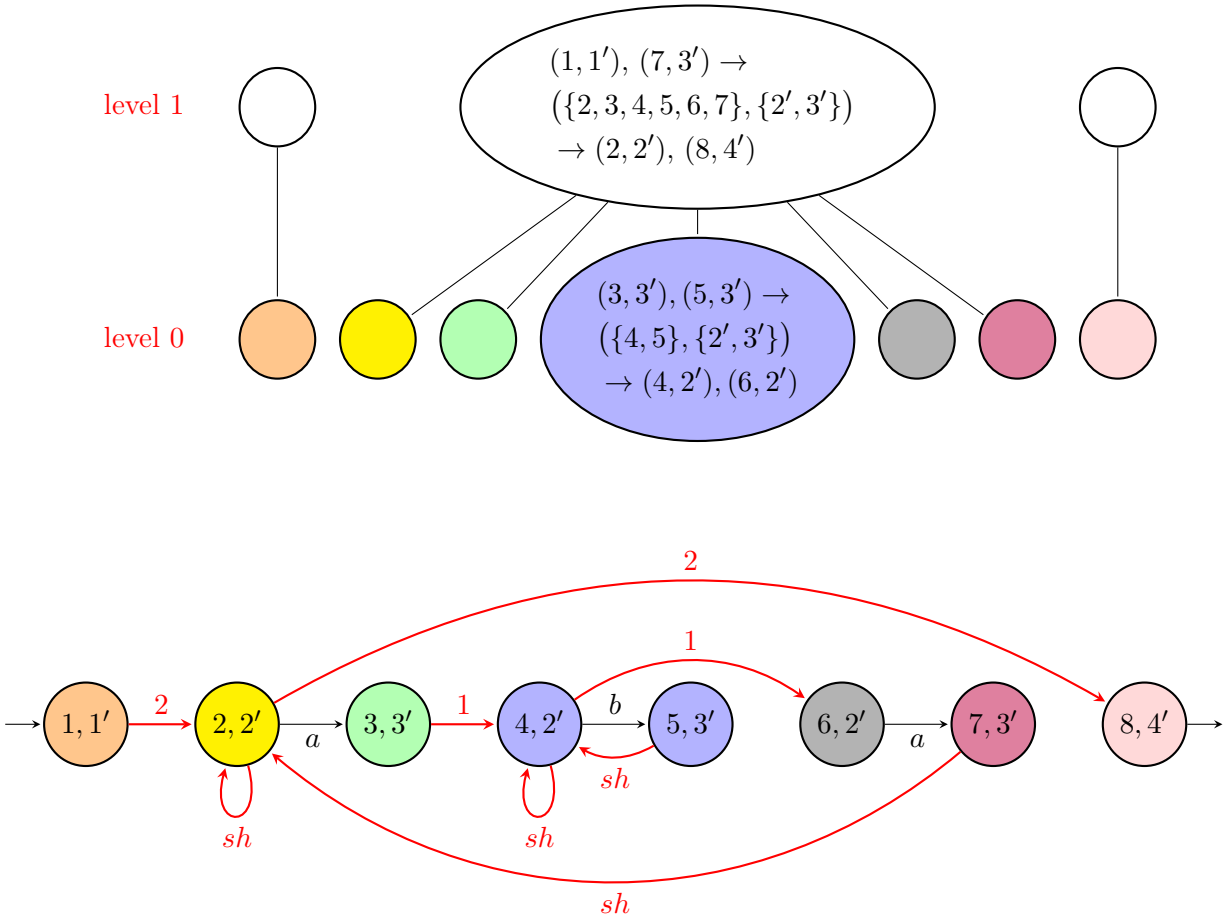


Figure 50: Intersection automaton after the last expansion phase and the computation of the new SCC of level 0.

5.3 Conversion from a finite history to a plain automaton

We now tackle the problem of converting a finite history automaton into a plain automaton. In Section 5.3.1, we introduce a simple but non-optimal solution to this problem. In Section 5.3.2, we discuss how it could be improved.

5.3.1 Simple solution

Algorithm 5 takes as input a finite history automaton $\mathcal{A}_\mathcal{H}$ and starts by creating an empty plain automaton $\mathcal{A}$. Its states (including the initial and the final ones) are identical to those of $\mathcal{A}_\mathcal{H}$, as well as its successor transition relation. Next, we enumerate all the strongly connected components (of any level) of the SCC hierarchy of $\mathcal{A}_\mathcal{H}$. For each strongly connected component S, the historic limit sets that it stores are scanned and converted into usual limit sets. This simply amounts to enumerating all the subsets of S that belong to $\langle P_\mathcal{H} \rangle$. (In fact, only the subsets of $\texttt{select}(S, P_\mathcal{H})$ should be considered, since the others cannot belong to $\langle P_\mathcal{H} \rangle$.) The subsets that satisfy this property become limit sets of $\mathcal{A}$.

For a large strongly connected component S, enumerating all the subsets of $\texttt{select}(S, P_\mathcal{H})$ might be infeasible. A possible improvement is to exploit the fact that if $P \in \texttt{select}(S, P_\mathcal{H})$ and $P \notin \langle P_\mathcal{H} \rangle$, then for any $P' \subseteq P$, one has $P' \notin \langle P_\mathcal{H} \rangle$. This explains why the subsets P of $\texttt{select}(S, P_\mathcal{H})$ are handled

Algorithm 5: Conversion from a finite history to a plain automaton (simple version).

```
1  FHA_to_plain_automata (A_H = (d, Q_H, Δ^s_H, scch, I_H, F_H)):
2      A ← new empty plain automaton (Q = ∅, Δ^s = ∅, Δ^lim = ∅, I = ∅, F = ∅)
3      Q ← Q_H,   I ← I_H,   F ← F_H,   Δ^s ← Δ^s_H
4
5      for ℓ ∈ {0, 1, 2, ..., height(scch)} do
6          for each strongly connected component S of level ℓ do
7              for each P_H in a transition of Λ(scch, S) do
8                  for each P ⊆ select(S, P_H), sorted in decreasing order of cardinality do
9                      if P ∈ ⟨P_H⟩ then
10                         for q ∈ orig(P_H) do
11                             Δ^lim ← Δ^lim ∪ {q → P}
12                         for q ∈ dest(P_H) do
13                             Δ^lim ← Δ^lim ∪ {P → q}
14                     else
15                         Discard all the subsets of P.
16     return A
```

in decreasing order of cardinality in Line 8. It allows us to discard all its subsets if $P \notin \langle P_{\mathcal{H}} \rangle$, i.e., it decreases the number of subsets that must be enumerated.

5.3.2 Possible improvements

Algorithm 5 does not check whether a subset P is strongly connected, i.e., whether it is possible to go from any state of P to any other state of P by visiting only states of P. As a consequence, some limit sets accepted (i.e., added to Δ^{lim}) by this algorithm are probably useless. In theory, it is not a problem, since adding useless limit transitions to an automaton does not modify its accepted language. But in practice, we would like to get a plain automaton that is as small as possible.

A possible workaround is to implement a function `is_strongly_connected` that checks whether a subset P is strongly connected, and to use it to accept less limit sets. However, this solution raises several challenges. First, as already mentioned in Chapter 4, checking whether a subset of states is strongly connected cannot be carried out with simple red arcs. Instead, we need *extended red arcs*, i.e., red arcs labelled by a limit set. For instance, if the limit transition $P \to q$ is added to the plain automaton $\mathcal{A}$, it is necessary to add an extended red arc $\texttt{repres}(P) \xrightarrow{P} q$ that encodes the possibility of going from $\texttt{repres}(P)$ to q, but after visiting the states of P.

Thanks to such arcs, checking whether a subset P is strongly connected is possible and easy: if suffices to check that P is included in a strongly connected component that relies only on successor transitions and extended red arcs whose label is a subset of P. However, for this method to be correct, all the extended red arcs labelled by a subset of P must have been added before P is examined. This has an important consequence: given a strongly connected component S and an historic limit set $P_{\mathcal{H}}$, converting $P_{\mathcal{H}}$ into limit sets requires to enumerate the subsets of $\text{select}(S, P_{\mathcal{H}})$ in increasing order of

cardinality. For instance, consider the strongly connected component S of level 0 depicted in Figure 51. Assume that it only stores the historic limit transition $\big(\{1,2\},\{1',2'\}\big) \to (1,2')$. Observe that it is

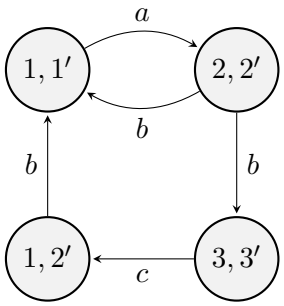


Figure 51: Strongly connected component S of level 0.

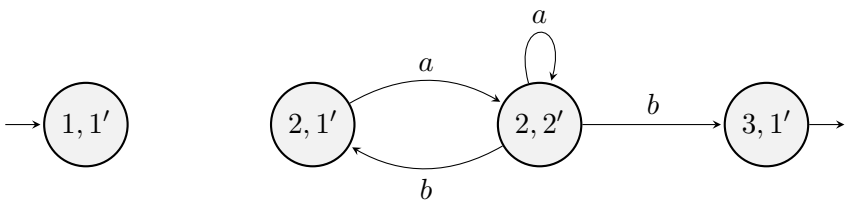
crucial in this case to examine the subset $\big\{(1,1'),(2,2')\big\}$ before the subset $\big\{(1,1'),(2,2'),(1,2')\big\}$. Both are strongly connected, but $\texttt{is\_strongly\_connected}\Big(\big\{(1,1'),(2,2'),(1,2')\big\}\Big)$ outputs True only if an extended red arc from $(1,1')$ (or $(2,2')$) to $(1,2')$ labelled by $\big\{(1,1'),(2,2')\big\}$ has been added before.

As a consequence, using the function `is_strongly_connected` in Algorithm 5 is not trivial, because it becomes impossible to enumerate the subsets of states as in Line 8. However, there are particular cases where an easy workaround exists: for instance, if $\texttt{orig}(P_{\mathcal{H}}) \cap \texttt{select}(S, P_{\mathcal{H}}) = \emptyset$ and $\texttt{dest}(P_{\mathcal{H}}) \cap \texttt{select}(S, P_{\mathcal{H}}) = \emptyset$, then the problem highlighted in Figure 51 does not occur, and therefore the order in which the subsets of states are enumerated is not constrained. In this particular case, enumerating them in decreasing order of cardinality is doubly worthwhile, for the following reason: if $P \notin \langle P_{\mathcal{H}} \rangle$ or P is not strongly connected, then the subsets of P can be discarded.

Another challenge is to handle the auto-referencing historic limit sets. As in Chapter 4, the solution is probably to add extended red arcs before handling such historic limit sets, in order to make sure that all the limit sets P that are useful are indeed found by the algorithm.

The question of designing an improved version of Algorithm 5 is left open. We only mentioned a few ideas that pave the way for improvements. However, even though it is really simple, Algorithm 5 works nicely in practice (see Chapter 6).

Example 36. *In Example 34, we obtained Figure 46 as final intersection automaton (where State* $(3,2')$ *should be deleted). Applying the simple conversion algorithm yields the following results, summarized in Figure 52:*

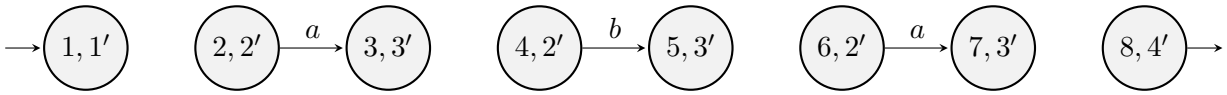


$(3,1') \to \big\{(2,1'),(2,2')\big\}$ $\qquad$ $(1,1') \to \big\{(2,1'),(2,2'),(3,1')\big\}$
$(1,1') \to \big\{(2,2'),(3,1')\big\}$

Figure 52: Plain automaton obtained via Algorithm 5.

- *The historic limit set* $(\{2\}, \{1', 2'\})$ *is converted into a single limit set* $\big\{(2, 1'), (2, 2')\big\}$*, which is useful.*
- *The historic limit set* $(\{2, 3\}, \{1', 2'\})$ *is converted into two limit sets:* $\big\{(2, 1'), (2, 2'), (3, 1')\big\}$ *and* $\big\{(2, 2'), (3, 1')\big\}$*. The former is useful but the latter is useless. An improved conversion algorithm should be able to discard it.*

Example 37. *Recall that we obtained Figure 50 as final result in Example 35. Applying the conversion algorithm yields an "optimal" result (see Figure 53) in this case, i.e., all the accepted limit sets are useful. Moreover, even though there are* 6 *states in the SCC* $\big\{(2, 2'), (3, 3'), (4, 2'), (5, 3'), (6, 2'), (7, 3')\big\}$*, only a few subsets are enumerated: those whose cardinality is at least* 5*. This comes from the fact that no subset of cardinality* 5 *belongs to* $\langle\{2, 3, 4, 5, 6, 7\}, \{2', 3'\}\rangle$*, hence the subsets of cardinality* 4 *are all discarded.*



$$(3, 3'), (5, 3') \to \big\{(4, 2'), (5, 3')\big\} \to (4, 2'), (6, 2')$$

$$(1, 1'), (7, 3') \to \big\{(2, 2'), (3, 3'), (4, 2'), (5, 3'), (6, 2'), (7, 3')\big\} \to (2, 2'), (8, 4')$$

Figure 53: Plain automaton obtained via Algorithm 5.

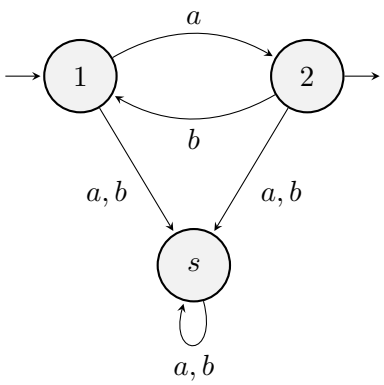
5.4 Union of automata on linear orderings

In this section, we briefly outline the few modifications of the intersection algorithm that are necessary to implement the union of automata on linear orderings.

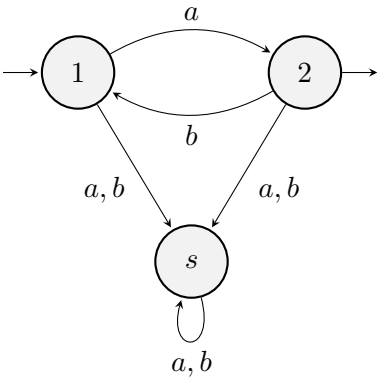
Let us start by a small reminder about the union of finite-word automata. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two finite-word automata, and $\mathcal{A}$ the automaton obtained by computing their union. Intuitively, a run in $\mathcal{A}$ must represent a run in $\mathcal{A}_1$ or in $\mathcal{A}_2$. It implies that there could exist some states of $\mathcal{A}$ that represent the fact of being in $\mathcal{A}_1$ only, or in $\mathcal{A}_2$ only. For instance, if the successor transition $q_1 \xrightarrow{\sigma} q_1'$ exists in $\mathcal{A}_1$ and if q_2 cannot be left by a successor transition reading σ, one should still add a successor transition leaving the state (q_1, q_2) in $\mathcal{A}$. This successor transition is labelled by σ and reaches a state (q_1, s_2) that represents the fact of being simultaneously in state q_1 in $\mathcal{A}_1$ and "nowhere" in $\mathcal{A}_2$. The state s_2 can actually be thought of as a very special state of $\mathcal{A}_2$ from which the only reachable state is s_2 itself. For this reason, s_2 is often called a *sink state*. Symmetrically, $\mathcal{A}_1$ also has its sink state s_1.

Formally, the union of finite-word automata $\mathcal{A}_1$ and $\mathcal{A}_2$ can be computed as follows:

1. Add a sink state s_1 in $\mathcal{A}_1$ and a sink state s_2 in $\mathcal{A}_2$. These sink states are not initial or final.
2. In $\mathcal{A}_1$, add all the successor transitions (labelled by any symbol) from any state (including s_1) to s_1, and do the same in $\mathcal{A}_2$. This yields two automata $\mathcal{A}_1^s$ and $\mathcal{A}_2^s$. An example of how a finite-word automaton can be extended with a sink state is depicted in Figure 54a.
3. Compute the intersection of $\mathcal{A}_1^s$ and $\mathcal{A}_2^s$. This yields an automaton $\mathcal{A}^s$.
4. Redefine the initial states of $\mathcal{A}^s$ as the states (q_1, q_2) such that q_1 or q_2 is initial, and similarly for the final states.



(a) Finite-word automaton (with $\Sigma = \{a, b\}$) extended with a sink state s.



$\{1, 2\} \to 2 \qquad s \to \{s\} \to s$

(b) Plain automaton on linear orderings (with $\Sigma = \{a, b\}$) extended with a sink state s.

Figure 54

It can be checked that the automaton $\mathcal{A}^s$ accepts the union of the languages of $\mathcal{A}_1$ and $\mathcal{A}_2$. However, in practice we do not add explicitly all the successor transitions reaching the sink state. Instead, we assume that there are implicitly present and we exploit them only when they are necessary, i.e., when a successor transition $q_1 \xrightarrow{\sigma} q_1'$ can be followed in $\mathcal{A}_1$ but not in $\mathcal{A}_2$, or conversely.

Let us first adapt this idea for plain automata on linear orderings. Intuitively, the sink state of a finite-word automaton makes it "non-blocking", since it is possible to reach the sink state from any other state, and any word can then be read from the sink state. In the case of automata on linear orderings, this idea can be generalized by simply adding implicitly the limit transitions $s \to \{s\} \to s$, which allows to read any word of any length from the sink state. An example of plain automaton on linear orderings extended with a sink state is represented in Figure 54b.

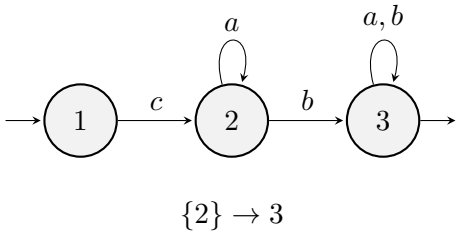
We are now ready to extend a finite history automaton with a sink state. Let $\mathcal{A}_\mathcal{H}$ be such an automaton, and let $\mathcal{H} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_d\}$ be its history. Assume that each plain automaton $\mathcal{A}_i$ has been extended with a sink state s_i. Then, we define the *historic sink state* and the *historic sink limit set* of $\mathcal{A}_\mathcal{H}$ as follows:

- The *historic sink state* of $\mathcal{A}_\mathcal{H}$ is the state $s = (s_1, s_2, \ldots, s_d)$. All the states of $\mathcal{A}_\mathcal{H}$ (including itself) are linked to it by a successor transition labelled by all the symbols, as in the case of finite-word automata. An historic sink state is not initial or final.
- The *historic sink limit set* of $\mathcal{A}_\mathcal{H}$ is the historic limit set $P^s_\mathcal{H} = \big(\{s_1\}, \{s_2\}, \ldots, \{s_d\}\big)$. It appears in the historic limit transitions $s \to P^s_\mathcal{H} \to s$.

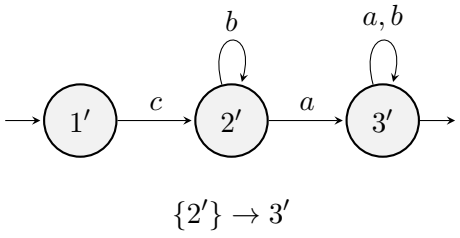
Note that setting $d = 1$ in these definitions yields a plain automaton on linear orderings, extended with a sink state as in Figure 54b. Intuitively, the limit transitions $s \to P^s_\mathcal{H} \to s$ will be used when a limit transition can be followed in one operand but not in the other. These transitions have, in some sense, the same purpose as the successor transitions linking each state to the historic sink state.

In order to handle correctly the historic sink state and the historic sink limit set, we must now adapt the algorithms developed previously. Let $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$ be two finite history automata, extended as explained before. Computing their union automaton $\mathcal{A}_\mathcal{H}$ requires to make the following modifications to Algorithms 2 and 4:

- Obviously, a state $q_1 \cdot q_2$ is initial (resp. final) iff q_1 or q_2 is initial (resp. final).
- As in the case of the union of finite-word automata, one should add additional successor transitions in $\mathcal{A}_{\mathcal{H}}$, compared to the intersection case. When exploring the state $q_1 \cdot q_2$ in the expansion function (i.e., Algorithm 2), if $q_1 \xrightarrow{\sigma} q_1'$ is a successor transition of $\mathcal{A}_{\mathcal{H}_1}$ but q_2 cannot be left by a successor transition labelled by σ, then one adds the successor transition $q_1 \cdot q_2 \xrightarrow{\sigma} q_1' \cdot s_2$, where s_2 is the historic sink state of $\mathcal{A}_{\mathcal{H}_2}$. Then, one instantiates and explores $q_1' \cdot s_2$, if it has not been instantiated yet.
- Similarly, in the expansion function one should make it possible to follow an historic right-limit transition in only one of the two operands. When we explore the state $q_1 \cdot q_2$, if q_1 can be left by the historic limit transition $q_1 \to P_{\mathcal{H}_1}$ but q_2 cannot be left by an historic limit transition, then we consider that $q_1 \cdot q_2$ can be left by the historic right-limit transition $q_1 \cdot q_2 \to P_{\mathcal{H}_1} \cdot P^s_{\mathcal{H}_2}$, where $P^s_{\mathcal{H}_2}$ is the historic sink limit set of $\mathcal{A}_{\mathcal{H}_2}$. Handling this right-limit transition can be done with the function `handle_right_lim_trans` (Algorithm 3), without any change.
- The last modification is less intuitive. It is linked to the treatment of the historic limit sets during the limit set phase. Currently, they are not handled correctly. To understand why, consider the two operands in Figure 55a and 55b, in which the historic sink state and the historic limit set are implicit (note that here, we consider these operands as finite history automata for which $d = 1$). Applying the current union algorithm (i.e., applying the 3 modifications introduced above) yields



(a) First operand.



(b) Second operand.

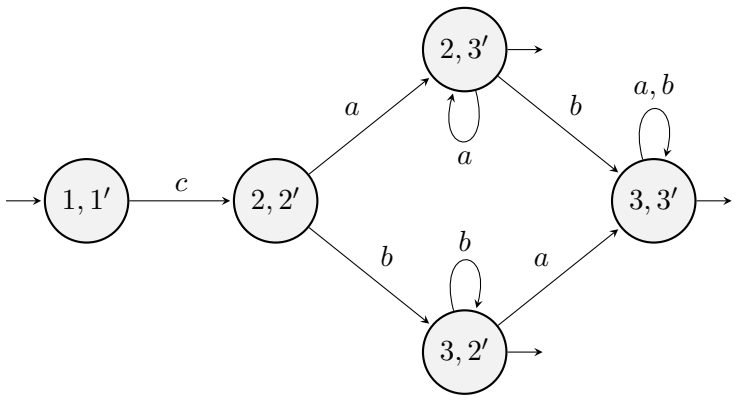
Figure 55: Operands for the union operation.

the finite history automaton represented in Figure 56. This automaton does not accept any word of length $\mathbb{N}$, since the historic limit transition $\big(\{1\}, \{1'\}\big) \to (2, 2')$ cannot be followed (given that State $(1, 1')$ is not linked to itself). But each operand accepts a word of this length. Currently, the union algorithm is thus incorrect.

Intuitively, the problem is the following: in the finite history automaton shown in Figure 56, it is impossible to follow only the limit transition $\{2\} \to 3$ (or only $\{2'\} \to 3'$). Ideally, the historic limit sets $\big(\{2\}, \{s'\}\big)$ and $\big(\{s\}, \{2'\}\big)$ should be naturally added to the union automaton. But the expansion phase did not instantiate States $(2, s')$ and $(s, 2')$. Therefore, the historic limit sets $\big(\{2\}, \{s'\}\big)$ and $\big(\{s\}, \{2'\}\big)$ were not examined during a limit set phase. A solution to this problem is to enforce the presence of $(2, s')$ and $(s, 2')$.

This can be done by slightly extending the limit set phase. When a strongly connected component S of level 0 is examined, if $\texttt{query}\big(\texttt{scch}_1, \pi_{Q_1}(S)\big) \times \texttt{query}\big(\texttt{scch}_2, \pi_{Q_2}(S)\big)$ is non empty (see Line 18 of Algorithm 4) and if S does not contain any state $q_1 \cdot q_2$ where q_1 or q_2 is an historic sink state, then:

1. We first handle each $(P_{\mathcal{H}_1}, P_{\mathcal{H}_2}) \in \texttt{query}\big(\texttt{scch}_1, \pi_{Q_1}(S)\big) \times \texttt{query}\big(\texttt{scch}_2, \pi_{Q_2}(S)\big)$ as before.



$\big(\{2\}, \{2'\}\big) \to (3, 3')$

Figure 56: Wrong result.

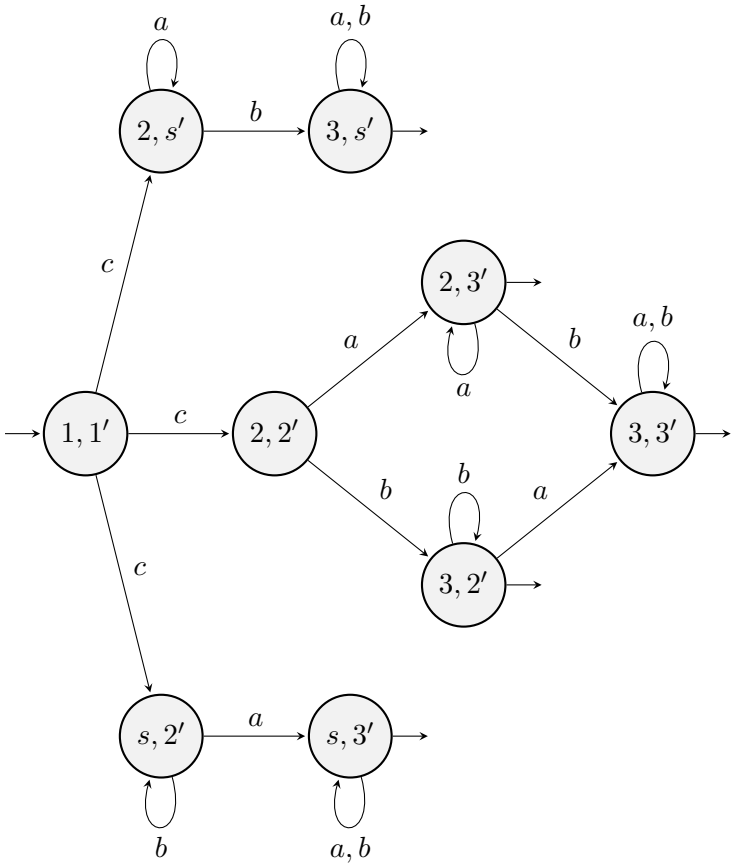
2. We then select an arbitrary state $q_1 \cdot q_2$ in S. If $q_1 \cdot s_2$ has not been instantiated yet (s_2 being the historic sink state of $\mathcal{A}_{\mathcal{H}_2}$), we add the empty word transition $(q_1 \cdot q_2) \xrightarrow{\varepsilon} (q_1 \cdot s_2)$ and we put $q_1 \cdot s_2$ in Q_{explore}. Similarly, if $s_1 \cdot q_2$ has not been instantiated yet, then we add the transition $q_1 \cdot q_2 \xrightarrow{\varepsilon} s_1 \cdot q_2$ and $s_1 \cdot q_2$ is added to Q_{explore}.

At the end of the union algorithm, we thus get an automaton with ε-transitions. To eliminate them, we could use the approach presented in Chapter 2, but it would not be efficient. In fact, it can be easily shown that in our case, there does not exist two consecutive ε-transitions. Moreover, two states linked by an ε-transition cannot be in the same strongly connected component (this follows from the fact that the only state reachable from an historic sink state is itself). Hence, it is actually possible to remove the ε-transitions very efficiently.

Figure 57 shows the effect of this modification. When examining the strongly connected component $\{(2, 2')\}$, we instantiated $(2, s')$ and $(s, 2')$ and added the ε-transitions $(2, 2') \xrightarrow{\varepsilon} (2, s')$ and $(2, 2') \xrightarrow{\varepsilon} (s, 2')$. The next expansion phase then discovered $(3, s')$ and $(s, 3')$, and the last limit set phase eventually found the historic limit transitions $\big(\{2\}, \{s'\}\big) \to (3, s')$ and $\big(\{s\}, \{2'\}\big) \to (s, 3')$. Finally, the ε-transitions were eliminated and replaced by the successor transitions $(1, 1') \xrightarrow{c} (2, s')$ and $(1, 1') \xrightarrow{c} (s, 2')$. It can be checked that the finite history automaton in Figure 57 indeed accepts the union of the languages accepted by the operands represented in Figure 55a and 55b.

These four modifications suffice to implement the union of finite history automata. All the elements that were not explicitly mentioned above can be left unchanged (e.g., the conversion to a plain automaton on linear orderings, the interleaving of expansion and limit set phases, the manipulation of the red arcs, etc.). The historic sink state can be considered as a regular state, and the historic sink limit set as a regular historic limit set.

Our union algorithm could probably be improved. First, there could exist better solutions that are not based on sink states and sink limit sets. Secondly, it seems possible to update the union algorithm to make it instantiate less states. For instance, in Figure 57, States $(2, 2'), (2, 3'), (3, 2')$ and $(3, 3')$ could be removed, since any word that can be accepted from $(2, 2')$ can also be accepted from $(2, s')$ or $(s, 2')$. A solution would be to explicitly erase these states. Alternatively, one could be find another way of modifying the limit set phase, to avoid instantiating $(2, s')$ and $(s, 2')$. We let these potential improvements for some future work.



$\big(\{2\},\{2'\}\big) \to (3,3') \qquad \big(\{2\},\{s'\}\big) \to (3,s') \qquad \big(\{s\},\{2'\}\big) \to (s,3')$

Figure 57: Correct result.

Chapter 6

Implementation

This section is dedicated to the practical implementation of the algorithms described in Chapter 5. It is divided into two parts: Section 6.1 covers some relevant details of the implementation, and Section 6.2 presents a few promising experimental results.

6.1 Implementation details

The principles of Chapter 4 and the algorithms of Chapter 5 have been turned into a practical implementation in the C language. This implementation is integrated to the LASH toolset, which implements, amongst other things, finite-word automata and their manipulation (see [13] for more details). It extends the functionalities already provided by LASH to automata on linear orderings. More precisely, our implementation consists of two distinct parts:

- The first part, which is common with Thomas Braipson, is an implementation of plain automata on linear orderings (Definition 1). Appendix A briefly explains how we extended the finite-word automaton data structure (that was already implemented in LASH) to implement automata on linear orderings. This extension represents approximately 1000 lines of code. The main part is contained in `automaton-lo.c`.
- The second part is an implementation of the product of automata on linear orderings. It also contains an implementation of the intersection algorithm, that can be considered as a particular case of the product operation. The union algorithm outlined in Section 5.4 is not yet implemented (but adapting the intersection implementation to the union operation would not be difficult). This part of the implementation represents approximately 7000 lines of code, across 14 files. We did not use any external code (but the implementation relies extensively on functionalities that were already provided by the LASH toolset, e.g., the product of finite-word automata). One should mention, however, that a custom implementation of Tarjan's algorithm (see [18]) was used to compute strongly connected components.

We do not want to enter all the implementation details here (we refer to the code itself for all these details). However, Sections 6.1.1 and 6.1.2 address two interesting practical problems that are worth being briefly discussed.

6.1.1 Deleting states

The function `filter_states` introduced in Chapter 5 deletes some states that are not reachable or not co-reachable. However, in practice, deleting a state is not a trivial operation: it requires to delete successor transitions and limit transitions in which this state is involved, and the current strongly connected components hierarchy (see Chapter 5) must be recomputed. Even though it is conceptually

simple to solve these problems, in practice it cannot be done efficiently enough. The state deletion operation is in fact a highly recurring operation (it will be illustrated in Section 6.2) and must therefore be extremely efficient.

There exists, however, a simple workaround that consists in deleting the states *lazily*, i.e., to equip each state with a Boolean flag `deleted` and to set it to `True` for all states that must be deleted. The states that are lazily deleted are then ignored in the future intersection operations.

Formally, let $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$ be two finite history automata whose useless states have been lazily deleted via the function `filter_states`. Then, during the computation of the intersection between $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$, the expansion function does not instantiate any state $q_1 \cdot q_2$ such that q_1 or q_2 is lazily deleted. In some sense, deleting lazily a state in an automaton does not impact it at all, but impacts the future intersection operations in which it is involved.

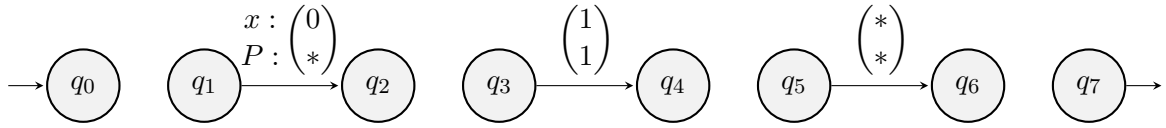
Of course, deleting lazily some states of an automaton does not make it possible to decrease its size. But this does not really matter in practice. This method allows us to avoid accumulating too many useless states through multiple intersection operations. When computing the intersection of a large number of automata (which is the most frequent use case in the context of the decision procedure), this is what really matters.

6.1.2 Handling an exponential number of labels

Recall that in the context of the decision procedure (Section 3.1), an ubiquitous task is to compute the intersection or the union of n atoms, where n can be large. An example of atom is recalled in Figure 58. The wildcard symbol $*$ is used when it does not matter to read the symbol 0 or the symbol 1 on a tape. Therefore, the successor transition $q_5 \xrightarrow{\binom{*}{*}} q_6$ represents the 2^2 transitions

$$q_5 \xrightarrow{\binom{0}{0}} q_6, \quad q_5 \xrightarrow{\binom{1}{0}} q_6, \quad q_5 \xrightarrow{\binom{0}{1}} q_6, \quad \text{and} \quad q_5 \xrightarrow{\binom{1}{1}} q_6.$$

However, in practice it is infeasible to store explicitly all such transitions, since the number of variables



$$q_0, q_2 \to \{q_1, q_2\} \to q_1, q_3$$

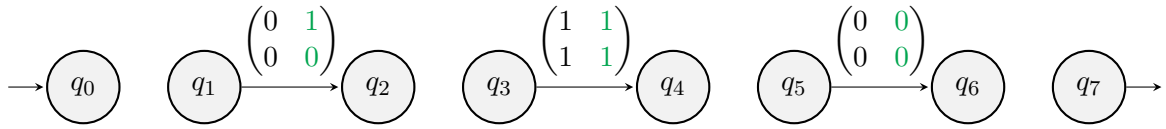
$$q_4, q_6 \to \{q_5, q_6\} \to q_5, q_7$$

Figure 58: Atom recognizing $P(x)$ on $\mathbb{R}$.

in a formula (i.e., the number of tapes in the atoms) may be large. For instance, in Section 6.2 we run a test with 1000 variables. In this case, the successor transition $q_5 \to q_6$ from Figure 58 must in fact be labelled by a vector of 1000 wildcard symbols. Translating this "wildcard label" into all the 2^{1000} labels that it represents is not a conceivable solution.

There exist several solutions to this problem. A simple and effective one is to equip each label with a Boolean *mask*, represented in green in Figure 59. The symbol 0 in a mask indicates that the corresponding symbol in the label is not important, i.e., it can indifferently be a 0 or a 1. In other

words, the symbols 0 in the mask indicate the locations of the wildcards. For instance, the label $\binom{0}{*}$ in Figure 58 becomes $\binom{0\quad 1}{0\quad 0}$ in Figure 59. It would also be correct to translate $\binom{0}{*}$ into $\binom{0\quad 1}{1\quad 0}$, since the second value of the label does not matter.



$$q_0, q_2 \to \{q_1, q_2\} \to q_1, q_3$$

$$q_4, q_6 \to \{q_5, q_6\} \to q_5, q_7$$

Figure 59: Atom recognizing $P(x)$ on $\mathbb{R}$, where the labels are equipped with masks.

The next step is to rewrite the label comparison test performed during the expansion step. Let us denote by & the bitwise AND operator and by | the bitwise OR operator. Let q_1 and q_2 be two states of finite history automata $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$, respectively. Assume that the state $q_1 \cdot q_2$ must be explored during the intersection of $\mathcal{A}_{\mathcal{H}_1}$ and $\mathcal{A}_{\mathcal{H}_2}$, and that q_1 (resp. q_2) admits an outgoing successor transition with label ℓ_1 and mask m_1 (resp. with label ℓ_2 and mask m_2). These two successor transitions can be followed simultaneously in each operand iff

$$\ell_1 \,\&\, m_1 \,\&\, m_2 = \ell_2 \,\&\, m_1 \,\&\, m_2,$$

i.e., iff the symbols that are important in both ℓ_1 and ℓ_2 match. If it is the case, the state $q_1 \cdot q_2$ can be left by a successor transition with label ℓ and mask m defined as

$$m = m_1 \mid m_2$$

and

$$\ell = (\ell_1 \,\&\, m_1) \mid (\ell_2 \,\&\, m_2).$$

For instance, if $\ell_1 = (1\,0\,0\,1)^T$, $m_1 = (1\,1\,0\,0)^T$, $\ell_2 = (1\,1\,1\,0)^T$ and $m_2 = (1\,0\,1\,0)^T$, then a successor transition leaving $q_1 \cdot q_2$ must be added to $\mathcal{A}_{\mathcal{H}}$ and this transition is labelled by $\ell = (1\,0\,1\,0)^T$, with a mask $m = (1\,1\,1\,0)^T$.

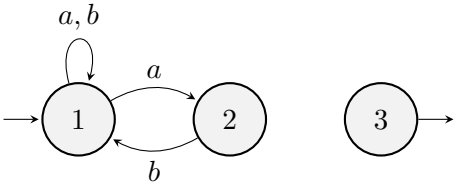
6.2 Experimental results

This section illustrates the strengths and limitations of our implementation by running several experiments.[1] First of all, one should mention that this implementation is not optimal. A lot of practical details could be fine-tuned, e.g., the implementation of the SCC hierarchy described in Section 5.1 and its supported operations. However, we try to show in this section that the ideas from Chapter 4 can indeed be turned into a reasonably efficient program, and that automata on linear orderings can indeed be used as practical data structures in the context of a decision procedure. We also try to highlight the benefits brought by all the strategies developed in this work (e.g., representing the limit sets concisely, deleting useless limit sets and states, etc.).

[1] All the experiments have been run on a computer that has 4 GB of memory and a processor with a clock frequency of 3.7 GHz.

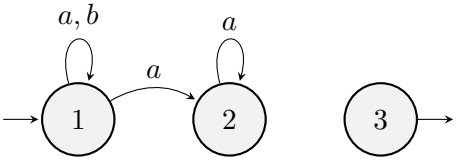
6.2.1 Handling a large number of limit sets concisely

Let us start by running a test that illustrates the need of a concise representation of the limit sets. Consider the two plain automata on linear orderings represented in Figure 60. The first one accepts all the words of length $\mathbb{N}$ that contain infinitely many symbols b. The second one, on the other hand, accepts all the words of this length that contain a finite number of symbols b. Hence, computing their intersection leads to an automaton accepting the empty language.



$\{1, 2\} \to 3$

(a) First operand $\mathcal{A}_1$.

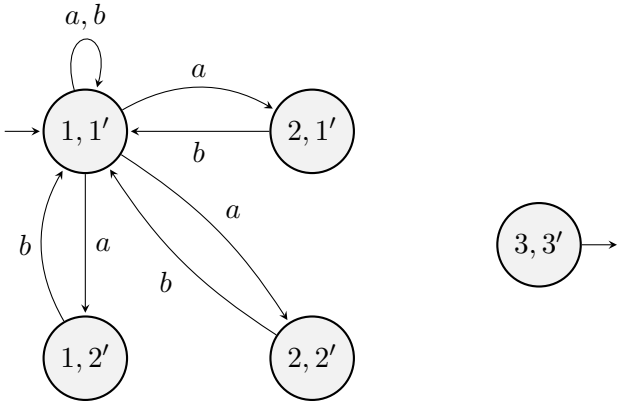


$\{2\} \to 3$

(b) Second operand $\mathcal{A}_2$.

Figure 60: Operands used in our experiment.

The experiment that we ran consists in computing the intersection of n copies of $\mathcal{A}_1$, and then in eventually computing the intersection between the obtained automaton and $\mathcal{A}_2$. This should yield an automaton accepting the empty language. The hardness of this task comes from the fact that computing the intersection between n copies of $\mathcal{A}_1$ leads to an automaton with many states and, more importantly, many limit transitions. For instance, the intersection between $\mathcal{A}_1$ and itself is represented in Figure 61. In fact, computing the intersection between n copies of $\mathcal{A}_1$ leads to an automaton with $O(2^n)$ states. Table 1 shows the results observed for different values of n, in two main cases. In the



$\{(1,1'),(2,1'),(1,2'),(2,2')\} \to (3,3')$
$\{(1,1'),(1,2'),(2,2')\} \to (3,3')$
$\{(1,1'),(2,1'),(2,2')\} \to (3,3')$

$\{(1,1'),(2,1'),(1,2')\} \to (3,3')$
$\{(1,1'),(2,2')\} \to (3,3')$

Figure 61: Automaton obtained after computing the intersection between $\mathcal{A}_1$ and itself.

first case, finite history automata are not used as internal formalism during the intersection operations. This corresponds to the left column of Table 1. It can be seen that this approach is not scalable. For $n \geq 4$, it leads to memory exhaustion. Even though the final automaton (which accepts the empty language) can be easily stored in memory (it does not have any limit transition and all its states are

lazily deleted), the peak memory usage is prohibitive. This is due to the prohibitively large number of limit sets in the automaton resulting from the intersection of n copies of $\mathcal{A}_1$ (this number is in fact the maximum number of limit sets observed during the computation). For $n = 1$, this number of limit sets is 5, as Figure 61 shows. It then grows dramatically fast: for $n = 3$ it exceeds $3 \cdot 10^5$.

Note that in this case, deleting the useless limit sets would not help us. All the numbers of limit sets recorded in Table 1 refer to useful limit sets. The solution is of course to use the concise representation of limit transitions provided by finite history automata. With this formalism, the maximal number of (historic) limit sets observed during the computation is 1, for any n (see the second column of Table 1). This follows from the fact that $\mathcal{A}_1$ and $\mathcal{A}_2$ have only one limit set. Actually, when using finite history automata as internal representation during the computation, the bottleneck becomes the exponential number of states. Solving the problem for $n = 15$ requires to handle automata with up to 2^{16} states, which explains why 44 seconds are necessary.

	Plain automata			Finite history automata		
n	Runtime	Max. # lim. sets	Peak memory	Runtime	Max. # lim. sets	Peak memory
1	≈ 0	5	≈ 1.5 MB	≈ 0	1	≈ 1.5 MB
2	150 μs	109	≈ 1.5 MB	≈ 0	1	≈ 1.5 MB
3	0.2 s	32 297	25 MB	≈ 0	1	≈ 1.5 MB
4	/	/	Failure	150 μs	1	≈ 1.5 MB
5	/	/	Failure	0.5 ms	1	≈ 1.5 MB
10	/	/	Failure	40 ms	1	3.5 MB
15	/	/	Failure	44 s	1	82 MB

Table 1: Results obtained when computing the intersection between $\mathcal{A}_1$ replicated n times and $\mathcal{A}_2$.

Importantly, the results reported in the second column of Table 1 crucially rely on the final intersection with $\mathcal{A}_2$, which yields an automaton without limit transition. Without this final step, it would be necessary to convert a finite history automaton into a plain automaton with a prohibitively large number of limit sets, as in the first column of the table. In other words, finite history automata make it possible to dramatically decrease the peak memory usage of the computation, but they do not have any impact on the final result. In our example, the peak memory usage is much higher than the amount of memory necessary to store the final result of the computation, hence finite history automata are invaluable.

Arguably, this example is pathological, since it makes no sense to compute the intersection between n copies of the same automaton. However, it is often the case that the peak memory usage is the bottleneck of the computation. For instance, in practice one may expect that computing the intersection of n automata on linear orderings often yields an automaton accepting the empty language, especially if n is large. For these frequent cases, using finite history automata as internal representation is highly beneficial.

6.2.2 A decision procedure example

We now run a test related to the decision procedure for the monadic first-order theory of order introduced in Section 3.1. This test was developed together with Thomas Braipson. [6] Our goal is to build an automaton recognizing the formula Φ_n (in which there are n free first-order variables $x_1, x_2, \ldots, x_n$ and

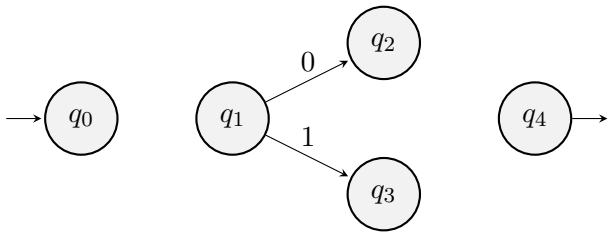
one uninterpreted unary predicate P) defined as

$$\begin{aligned}\Phi_n : \quad & \forall y \forall z \exists u \exists v.\ y < z \implies \Big(y < u < z \wedge y < v < z \wedge P(u) \wedge \neg P(v)\Big) \\ & \wedge \Big(x_1 < x_2 < x_3 < \cdots < x_n\Big) \\ & \wedge \Big(P(x_1) \wedge \neg P(x_2) \wedge P(x_3) \wedge \ldots \neg P(x_n)\Big).\end{aligned}$$

On $\mathbb{R}$, this formula is satisfiable. It states that:

- The points in which the predicate P is True and the points in which it is False are densely mixed on $\mathbb{R}$ (this property is called *chaoticity*). Note that this formula was already introduced in Example 17.
- The first-order variables $x_1, x_2, \ldots, x_n$ are sorted in increasing order.
- The predicate P is True in x_i iff $i \in \{1, 2, \ldots, n\}$ is odd (and we assume that n is always even).

Although the formula Φ_n is quantified, one can easily build a formula that recognizes it, without general algorithm for universal quantification. Thanks to Example 18, we know that the formula $\forall y \forall z \exists u \exists v.\ y < z \implies \big(y < u < z \wedge y < v < z \wedge P(u) \wedge \neg P(v)\big)$ is recognized by the automaton in Figure 62. Note that compared to Example 18, two states have been merged in order to get a *compact shuffle* construct. This construct was introduced by Thomas Braipson to accelerate his non-emptiness test. [6] It turns out that using this form of automata makes the intersection operation faster as well.



$$q_0, q_2, q_4 \to \{q_1, q_2, q_3\} \to q_1, q_4$$

Figure 62: Automaton recognizing the density of a predicate.

We can now apply the intersection algorithm to compute the intersection between

- the automaton in Figure 62 (to which n tapes must be added, of course),
- $n - 1$ atoms encoding the sub-formulas $x_i < x_{i+1}$ for $i \in \{1, 2, \ldots, n-1\}$,
- $\frac{n}{2}$ atoms encoding $P(x_i)$ for $i \in \{1, 3, 5, \ldots, n-1\}$, and
- $\frac{n}{2}$ atoms encoding $\neg P(x_i)$ for $i \in \{2, 4, 6, \ldots, n\}$.

When $n = 2$, the result of this intersection is depicted in Figure 63. This automaton can be interpreted as follows:

- There are three adjacent open intervals I_1, I_2 and I_3 of $\mathbb{R}$ on which P is chaotic. They correspond to the three shuffle constructs.
- I_1 and I_2 are separated by x_1, and P is True in this point.
- I_2 and I_3 are separated by x_2 and P is False in this point.

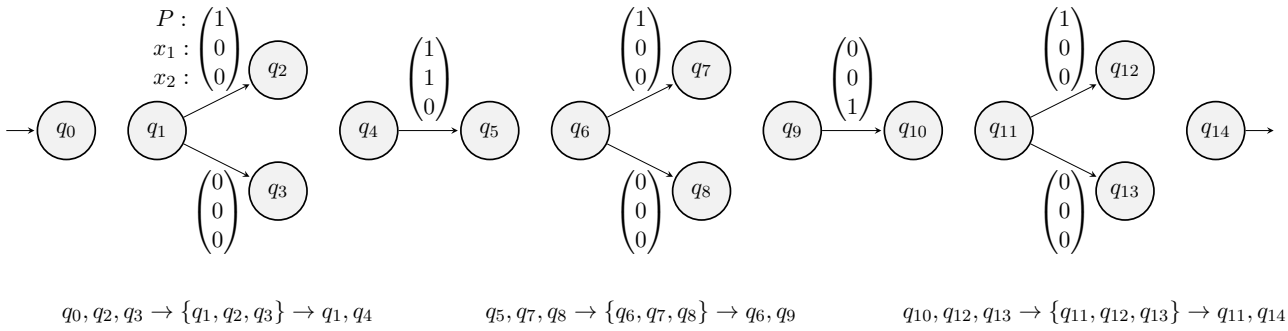


Figure 63: Automaton recognizing Φ_2.

The experimental runtimes observed for increasing values of the number n of first-order variables are shown in Figure 64. This figure also shows the runtimes of the non-emptiness test of Thomas Braipson (this test finds an accepted word on $\mathbb{R}$ for any n, of course). As previously hinted in Chapter 3, the

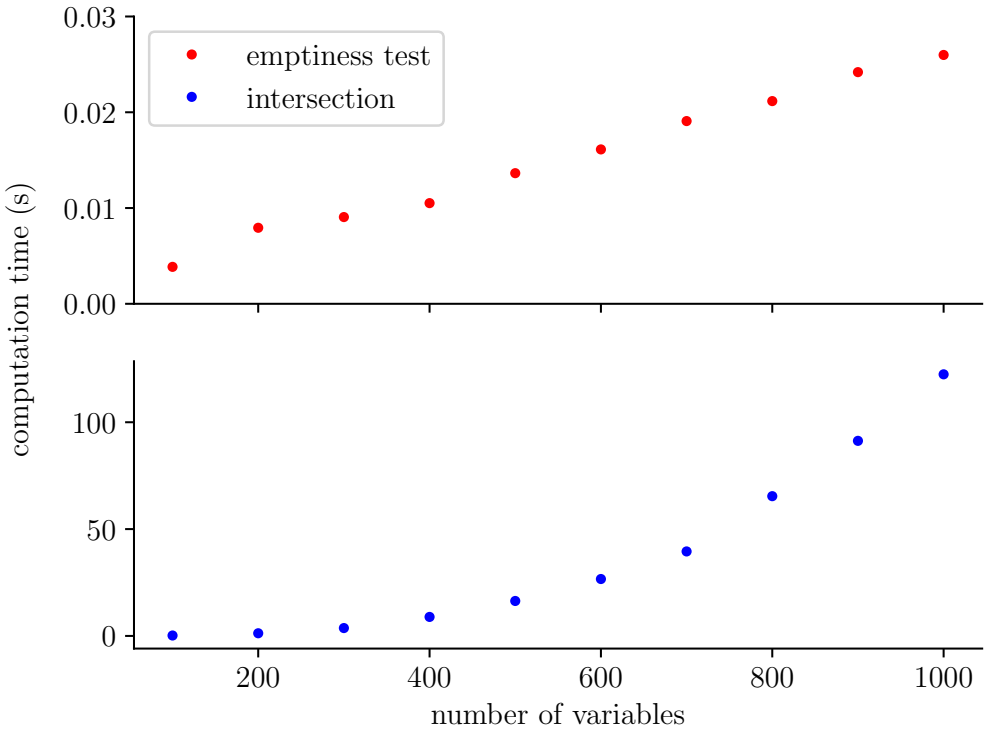


Figure 64: Runtime of the decision procedure (i.e., multiple intersection operations followed by a non-emptiness test) as a function of the number n of first-order variables.

time complexity of the decision procedure crucially depends on the implementation of the intersection algorithm (or, more generally, of the product algorithm). The non-emptiness test represents a negligible fraction of the total runtime.

Let us now analyse the behaviour of the intersection algorithm on this example. First, as in Figure 63, the final automaton is in fact a sequence of $n+1$ compact shuffle constructs separated by n isolated points, that each refer to one first order variable. Therefore, all the states in the final automaton are reachable and co-reachable, and all its limit sets are useful. In some sense, our algorithm thus

outputs an "optimal" result, since it computes the smallest possible automaton recognizing Φ_n. This is a consequence of two important strategies discussed in Chapter 5:

- Our intersection algorithm instantiates states sparingly, as the intersection algorithm for finite-word automata.
- Thanks to reachability graphs, some states that are not reachable or not co-reachable are (lazily) deleted. This is extremely important in practice: after each intermediate intersection operation, approximately 50% of the states are lazily deleted. Without this mechanism, computing the automaton recognizing Φ_9 requires more than 10 minutes (in fact, the algorithm becomes exponential).

In fact, the intersection algorithm that we developed in this work seems to be well-suited for the tasks related to the decision procedure. After analysing the results shown in Figure 64 (and collecting additional runtime measurements), we found that the time needed to compute the automaton recognizing Φ_n grows in $O(n^3)$ (i.e., a polynomial of degree 3 almost perfectly fits the raw measurements). This can be explained by the following observations:

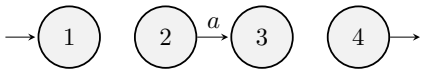
- The number of states and successor transitions in the automaton recognizing Φ_k is a linear function of k, for all $1 \leq k \leq n$.
- The size of the labels (i.e., the number of tapes) is $n + 1$, i.e., a linear function of n. A label comparison test has thus a linear complexity.
- The number of intermediate intersections to perform before getting the final result is $2n$, i.e., another linear function of n.

The limit sets are treated efficiently in this example, mainly because all the strongly connected components are small in the operands (they never include more than 3 states) and remain small throughout the intermediate intersection operations. Converting the last finite state automaton into a plain automaton on linear orderings can thus be done efficiently.

6.2.3 Nested limit sets example

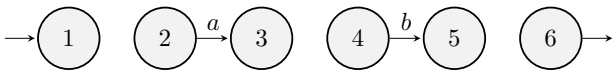
The two previous tests do not show the behaviour of the intersection algorithm when many limit sets are nested. In other words, we did not yet tested the ability of the SCC hierarchy to handle several levels of strongly connected components. This is the purpose of this section.

Consider the two automata on linear orderings $\mathcal{A}_1$ and $\mathcal{A}_2$ represented in Figure 65. They accept the languages $sh(a)$ and $\big(sh(a) \cdot b\big)^\omega$, respectively. Observe that it is easy to build $\mathcal{A}_2$ from $\mathcal{A}_1$, by adding a successor transition reading the symbol b and a limit transition $\{2, 3, 4, 5\} \to 6$ of level 2. This operation



$1, 3 \to \{2, 3\} \to 2, 4$

(a) Automaton $\mathcal{A}_1$, accepting $sh(a)$.



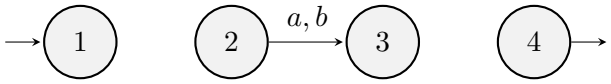
$1, 3, 5 \to \{2, 3\} \to 2, 4 \qquad \{2, 3, 4, 5\} \to 6$

(b) Automaton $\mathcal{A}_2$, accepting $(sh(a) \cdot b)^\omega$.

Figure 65

can in fact be repeated an arbitrary number of times. We define $\mathcal{A}_n$ as the automaton obtained by applying this operation $n - 1$ times, starting from $\mathcal{A}_1$. The automaton $\mathcal{A}_n$ accepts the language L_n defined inductively as $\big(L_{n-1} \cdot b\big)^\omega$, where $L_1 = sh(a)$. It has n nested limit sets, and therefore its SCC hierarchy has n levels.

We now compute, for different values of n, the intersection between $\mathcal{A}_n$ and the automaton depicted in Figure 66, that accepts the language $sh(a+b)$. The result of this intersection is exactly $\mathcal{A}_n$, which



$1, 3 \to \{2, 3\} \to 2, 4$

Figure 66: Second operand, accepting the language $sh(a+b)$.

is expected since $L_n \subseteq sh(a+b)$ for all n. The measured runtimes are recorded in Table 2.

n	2	5	10	50	100	250	500	1000
Runtime	≤ 1 ms	≤ 1 ms	8 ms	93 ms	419 ms	2.75 s	39 s	10 min

Table 2: Runtimes observed for the computation of the intersection between $\mathcal{A}_n$ and the automaton in Figure 66.

It can be checked (with additional measures) that the time complexity for this task is $O\big(P(n)\big)$ for some polynomial P. However, the degree of P is probably too large to tackle large problems. A solution is to implement carefully the SCC hierarchy (which is used extensively in this task). We do not enter the details here (they are available in the code), but for now its implementation is too naive. Enhancing it could potentially improve significantly the runtimes reported in Table 2.

However, notice that in the context of the decision procedure, the intersection operations that need to be run are not similar to the one studied here. Most of the time, they involve a large number of operands that have only a few levels in their SCC hierarchy (or even only one for the atoms). We saw in Section 6.2.2 that our algorithm is reasonably efficient in this case.

The intersection between $\mathcal{A}_n$ and the second operand yields an automaton identical to $\mathcal{A}_n$. Interestingly, the intersection algorithm is thus able, in some cases, to deal with automata that have large strongly connected components. In fact, the largest strongly connected component of $\mathcal{A}_n$, i.e., the strongly connected component of level n, includes all the states except the initial and the final ones. Nevertheless, the conversion algorithm from finite state automata to plain automata did run successfully. Even though converting an historic limit set into a plain limit set involves an enumeration of the subsets of a strongly connected component, the conversion is extremely efficient in our case, since only the maximal subset (i.e., the strongly connected component itself) covers the historic limit set. This explains why Algorithm 5, even in its simple version, runs in polynomial time in our example. This is also often the case in the context of the decision procedure.

Finally, it is important to mention that in many cases, our intersection algorithm is able to quickly detect when the intersection of the languages of the operands is empty. This is due to the fact that states are instantiated progressively, starting from the initial ones. For instance, if the second operand in Figure 66 is replaced by an automaton accepting the language $sh(a, b)$ (similar to the automaton in Figure 62), then the intersection algorithm outputs in less than 0.1 ms an automaton whose states are all lazily deleted (which means that it accepts the empty language). This is the case for all the values of n. It comes from the fact that our algorithm quickly realizes that it is impossible to instantiate additional states, and thus terminates almost immediately.

Chapter 7

Conclusion and perspectives

As a conclusion, let us briefly recall the context in which our work took place, its contributions and the future questions, problems and challenges that it raises.

7.1 Context

Automata on linear orderings are highly expressive mathematical objects that generalize the concepts of finite-word, infinite-word and transfinite-word automata. Up to now, they have been considered as purely mathematical objects. The research team in which this thesis was carried out aims at deciding the monadic first-order theory of order on $\mathbb{R}$ or $\mathbb{Q}$ by means of these automata. It amounts to solving three distinct sub-problems: representing and manipulating these automata on linear orderings efficiently, applying universal quantification on them, and checking the non-emptiness of their accepted language on $\mathbb{R}$ or $\mathbb{Q}$. This master's thesis tackles the first sub-problem.

7.2 Contributions

In Chapter 2, we introduced (together with Thomas Braipson and Bernard Boigelot) *epsilon automata on linear orderings*, that generalize plain automata by allowing successor transitions to be labelled by the empty word. We proved that, with an appropriate choice of semantics, the presence of epsilon transitions does not augment the expressivity of automata on linear orderings. We then illustrated the power of our epsilon automata by correcting a minor mistake in the literature.

In Chapter 3, we explained why the product of automata on linear ordering is in fact the most critical basic operation (mainly because it serves as a basis for the intersection and the union operations), and is an intrinsically hard problem for which a naive solution performs poorly. For some problem instances, we managed to implement it efficiently (both in terms of time and memory usage), by investigating several strategies both at the theoretical and at the practical level. Most of these strategies aim at mitigating the explosion of the number of limit sets during one or several product operations, primarily in the context of the decision procedure (i.e., when dealing with automata that admit small strongly connected components).

In Chapter 4, we first introduced theoretically the notion of *useless* limit set and we explained how computing the strongly connected components of the *reachability graph* of an automaton makes it possible to correctly detect some of these limit sets, even in presence of *auto-referencing* limit sets. Then, we defined a new form of automata called *finite history automata*. This formalism makes it possible to store implicitly a large number of limit transitions. Using finite history automata as internal representation during one or several product operations is another way of making them both faster and more memory-efficient. More precisely, in many practical cases, it decreases the peak memory usage

observed during the computation, which is valuable since decision procedures are often bottlenecked by this peak.

In Chapter 5, we translated these theoretical findings into practical data structures and algorithms. We developed a method (inspired from the intersection algorithm for finite-word automata) that computes the intersection of finite history automata efficiently, by means of a data structure called *strongly connected components hierarchy*. This data structure can be considered as a joint implementation of the limit-set deletion algorithm and finite history automata. We then addressed the problem of converting a finite state automaton into a plain automaton on linear orderings, which can be done efficiently in many practical cases. Next, we explained how the union of automata on linear orderings could be implemented by slightly modifying the intersection algorithm.

Finally, we discussed some implementation details in Chapter 6: the deletion of some states that are not reachable or not co-reachable (this can be seen as a by-product of reachability graphs), and the manipulation of a large number of labels. These implementation tricks are less theoretically grounded, but are of crucial importance to get an implementation that works in practice. We then showed experimentally that our implementation of the intersection algorithm (that relies itself on the implementation of automata on linear orderings made together with Thomas Braipson) is reasonably efficient, at least for the typical tasks that we expect to arise in the context of the decision procedure. We observed that it has interesting properties in practice, even in other settings: in some cases, it is able to quickly detect when the automaton resulting from the intersection operation does not accept any word, it deletes a lot of states during each intermediate intersection, it can handle nested limit sets rather efficiently, and it often decreases the peak of memory usage observed during the computation.

7.3 Perspectives

The theoretical insights, algorithms and implementations provided in this work can all be extended and improved in many different ways.

First, epsilon automata could be turned into practical data structures. For now, they only serve as a powerful tool that simplifies theoretical developments and proofs. But epsilon transitions could in fact be useful in practice (for instance, in the universal quantification algorithm). However, allowing such transitions to appear in practical data structures requires to design a more efficient ε-transitions deletion algorithm. The algorithm that we presented in Chapter 2 is only a theoretical way of showing the equivalence between epsilon automata and plain automata on linear orderings. Building a more sophisticated algorithm that is efficient in most practical cases is an interesting challenge.

Next, regarding the product of automata on linear orderings, we could investigate other research directions. The solution elaborated in this work is only one solution amongst others, based on the idea of deleting some limit sets and representing them concisely. There might exist, for instance, other formalisms that are more sophisticated than finite history automata and admit completely different properties. An ideal intersection algorithm should probably combine different possible strategies and find the one that best fits the instance of the problem given as input.

Throughout this master's thesis (especially in Chapters 4 and 5), we tried to prove the key parts of the product algorithm, or at least to motivate its correctness. However, before using it in the context of the decision procedure, we should probably strengthen some of these proofs and write additional ones. Automata on linear orderings are objects that sometimes behave non intuitively. Formal proofs would make sure that the algorithm that we designed is entirely correct.

In addition, we could wonder whether all the expressive power of automata on linear orderings is necessary to decide the monadic first-order theory of order on $\mathbb{R}$ and $\mathbb{Q}$. In other words, it might be possible to restrict the formalism of automata on linear orderings in the context of the decision procedure. In this case, we could maybe represent the "restricted" automata more efficiently, and design algorithms that are more tailored to them.

Regarding the algorithms, we already mentioned throughout this work several possible improvements. For instance, the conversion algorithm from finite history automata to plain automata on linear orderings should be refined: ideally, it must take into account the fact that a useful limit set must be a (non-necessarily maximal) strongly connected component (but we saw that it raises several issues, e.g., the need of red arcs labelled by limit sets). The intersection of finite history automata could also be improved. For instance, the number of candidate historic limit sets that are examined in each strongly connected component could maybe be decreased. This probably requires, however, to design a new data structure that is more sophisticated than a strongly connected component hierarchy. Fine-tuning the implementation of the latter is another way of improving the performances of our algorithms.

Finally, this master's thesis addresses the problem of representing and manipulating automata on linear orderings efficiently, but it completely ignores the two other sub-problems related to the decision procedure. Implementing the latter requires to think about the problem as a whole, which brings both new improvement strategies and new interesting questions. For instance, the non-emptiness test developed by Thomas Braipson could probably benefit from a hierarchical organisation of the limit sets [6]. Our strongly connected components hierarchy can thus maybe serve in other contexts. The question of using finite history automata outside the scope of the product operation is another research direction. Plain automata might not be the best common formalism to use in the decision procedure. Using finite history automata (or a variant of them) instead of plain automata could dramatically improve the efficiency of the product operation, since the final conversion from a finite history automaton to a plain automaton would become unnecessary.

Appendix A

Implementation of automata on linear orderings

In this appendix (common with Thomas Braipson), we explain and motivate our implementation of automata on linear orderings as a data structure. This implementation is done as a part of the LASH toolset [13]. This toolset is a library written in the C language able to manipulate finite-state automata, on finite or infinite words. Our contribution consists in extending these automata to automata on linear orderings.

A.1 Original automata

First, we describe the original implementation of automata in LASH. An automaton is represented as a data structure composed of

- an array of *states* Q,
- miscellaneous information, and
- a pointer towards a future extension.

Each state consists of a data structure containing information about initial and final status and an array of *transitions*. A transition consists of:

- Its destination state, represented as the corresponding index in the state array.
- Its label, consisting in a sequence of alphabet symbols. Each alphabet symbol is represented by an array of bytes, whose length is constant across the automaton.

A LASH automaton is thus an annotated adjacency list.

A.2 Automata on linear orderings

An automaton on linear orderings consists of a LASH automaton whose extension stores information about limit transitions. The implementation of our extension consists of a data structure composed of

- an array of *limit sets* P, and
- an array of *right-limit transitions* R.

A limit set is represented by a hash set (provided as a utility by the original toolset) and an array of right-limit transitions. The hash set contains the indices of the states composing it. Right-limit transitions are encoded as follows. An array whose length is the number of states in the automaton contains pointers to limit set index arrays. Each index refers to the limit set involved in the right-limit transition of the automaton.

The reason why right-limit transitions are not represented in the same fashion as left-limit transitions is that many operations require to know which successor and limit transitions can be followed from a given state. Symmetrically, many operations require to know to which states left-limit transitions can be followed. This justifies why left-limit transitions are stored near limit sets.

Remark

Limit sets are stored in the array P in the order in which they are created by the user (this paradigm is borrowed from how states are added in the core of the toolset). This implies in particular that several limit sets can have the same content.

Bibliography

[1] Barrett, C., Kroening, D., Melham, T.: Problem solving for the 21st century: Efficient solver for satisfiability modulo theories. Knowledge Transfer Report, Technical Report 3, London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering (Jun 2014)

[2] Bès, A., Carton, O.: A Kleene theorem for languages of words indexed by linear orderings. International Journal of Foundations of Computer Science **17**(03), 519–541 (2006)

[3] Boigelot, B., Braipson, T., Clara, T.: Epsilon automata on linear orderings, submitted for publication

[4] Boigelot, B., Fontaine, P., Vergain, B.: Deciding satisfiability for fragments with unary predicates and difference arithmetic (short paper). In: Bright, C., Davenport, J.H. (eds.) Proceedings of the 6th SC-Square Workshop co-located with the SIAM Conference on Applied Algebraic Geometry. CEUR Workshop Proceedings, vol. 3273, pp. 18–26 (2021)

[5] Boigelot, B., Fontaine, P., Vergain, B.: Universal first-order quantification over automata. International Journal of Foundations of Computer Science (2023)

[6] Braipson, T.: Non Emptiness Test for Automata on Linear Orderings: an Efficient Implementation. Master's thesis, University of Liège (2024-2025)

[7] Bruyère, V., Carton, O.: Automata on linear orderings. Journal of Computer and System Sciences **73**(1), 1–24 (2007)

[8] Bruyère, V., Hansel, G., Michaux, C., Villemaire, R.: Logic and p-recognizable sets of integers. Bulletin of the Belgian Mathematical Society - Simon Stevin **1** (01 1994)

[9] Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proc. International Congress on Logic, Methodology and Philosophy of Science. pp. 1–12. Stanford University Press, Stanford (1962)

[10] Cristau, J.: Automata and temporal logic over arbitrary linear time. In: Proc. IARCS 29th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LIPIcs, vol. 4, pp. 133–144 (2009)

[11] Déharbe, D.: Satisfiability solving for software verification. International Journal on Software Tools for Technology Transfer **11**, 255–260 (07 2009)

[12] Grinberg, D.: An introduction to graph theory (2025), `https://arxiv.org/abs/2308.04512`

[13] The Liège Automata-based Symbolic Handler (LASH), available at : `https://people.montefiore.uliege.be/boigelot/research/lash/`

[14] Muller, D.E.: Infinite sequences and finite machines. In: 2013 IEEE 54th Annual Symposium on Foundations of Computer Science. pp. 3–16. IEEE Computer Society, Los Alamitos, CA, USA (Oct 1963)

[15] Rabinovich, A.: Temporal logics over linear time domains are in PSPACE. Information and Computation **210**, 40–67 (2012)

[16] Rispal, C., Carton, O.: Complementation of rational sets on countable scattered linear orderings. International Journal of Foundations of Computer Science **16**(04), 767–786 (2005)

[17] Rosenstein, J.G.: Linear orderings. Academic press (1982)

[18] Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing **1**(2), 146–160 (1972)

[19] Vardi, M.: An automata-theoretic approach to linear temporal logic. Lecture Notes in Computer Science (08 1996)

[20] Vardi, M.Y.: The Büchi complementation saga. In: Proceedings of the 24th Annual Conference on Theoretical Aspects of Computer Science. p. 12–22. STACS'07, Springer-Verlag, Berlin, Heidelberg (2007)