Coinduction based algorithm to decide Büchi automata equivalence

Laureline Pinault
M2 internship supervised by Denis Kuperberg and Damien Pous
ENS de Lyon

February 2017 - June 2017

Contents

1 Introduction

Automata are abstract machines which are rather simple compared to Turing machines. Yet they are studied a lot in Computer Science. Indeed, not only are they able to model a lot of programs, but some classes of automata enable also a simplification of a lot of indecidable problems for Turing machines. In this internship, we were interested in Büchi automata. These automata deal with infinite words. They can represent both reactive systems and their specifications in LTL logic. Therefore, it is particularly useful to have an efficient way to test equivalence or inclusion of such automata. [GO01]

The usual method to check whether a reactive system P follows a given LTL formula F or not is to compute the Büchi automaton of P and the Büchi automaton of ¬F and checks if the intersection of the two Büchi automata is empty [CVWY91]. However, an efficient way to check if the Büchi automaton of P is included in the Büchi automaton of F could give a better algorithm.

In this work we are interested in adapting algorithms running on finite words automata to Büchi automata. Usually equivalence of finite words automata is checked either via minimisation or through Hopcroft and Karp's algorithm. The advantage of Hopcroft and Karp's algorithm is that it can be executed on the fly on an automaton whose transitions are computed on demand. Moreover minimisation is not possible for Büchi automata (a unique minimum Büchi automaton does not necessarily exists for rational languages of infinite words). Thus we wanted to extend Hopcroft and Karp's algorithm and more particularly an improvement developped in [BP13].

However those algorithms run on a deterministic automaton or at least a determinisable one and it is a well-known result that non-deterministic Büchi automata are strictly more expressive than deterministic ones. One possibility is to the Safra construction to obtain deterministic Müller automata, but this will mean dealing with rather complicated accepting conditions of infinite runs. Besides, the article [CNP93] offers a construction that allows to work with automata over finite words instead of Büchi automaton to solve problems of equivalence or inclusion of languages. However the construction makes the number of states grows exponentially. So the stake of this work is to take advantage of the particular structure of the constructed automaton to develop optimisations in order to have an hopefully efficient algorithm.

The rest of this report is organized as follow. First I will present Hopcroft and Karp's algorithm and its improvement for the case of non-determinitic automata: `HKC`. I will then present the construction allowing to use the algorithms on finite words. And last but not least, I will present the algorithm on Büchi automata along with two improvements: some pre-processing and a way to represent the states more compactly.

2 Algorithms for equivalence of automata over finite words

Hopcroft and Karp's algorithm has first been designed to run on determinitic finite words automata. However, as any non-determinitic finite words automaton can be transformed into an equivalent deterministic one, the algorithm can be run on it. Besides, the determinized automaton have a specific shape, [BP13] takes advantage of it and improve Hopcroft and Karp's algorithm for this case. In this section I will first present the algorithm for deterministic finite automata (DFA) and then explain the improvement for non-deterministic finite automata (NFA).

2.1 Hopcroft and Karp's algorithm for DFA

Deterministic finite automata. The deterministic finite automata are finite state machines that take as input a finite string and accept or reject it accordingly to a uniquely determined computation.

Formally, a *deterministic finite automaton* (DFA) over the alphabet Σ is a triple (X, t, o) where X is a finite set of states, $t : X \to X^\Sigma$ is the transition function which returns for each state x and each letter a the next state $x' = t_a(x)$ and $o : X \to \{0, 1\}$ is the output function which determines whether a state x is final ($o(x) = 1$) or not ($o(x) = 0$).

In this work we deal with a slightly more general notion of automaton: we allow the output function to take value in any semi-lattice $(V, +, 0)$ instead of only the boolean one[1]. A semi-lattice $(V, +, 0)$ consists of a set V, a binary operation $+ : V^2 \to V$ which is associative, commutative, idempotent and has 0 as a neutral element. For instance the set of booleans $(\{0, 1\}, \texttt{or}, 0)$ and the set of languages $(2^{\Sigma^*}, \cup, \varnothing)$ are semi-lattices. More generally for any set X its power set $(\mathcal{P}(X), \cup, \varnothing)$ is a semi-lattice. The set of possible output value is required to be a semi-lattice so that the construction used to determinized automata works.

Language equivalence. A word is accepted by an automaton with outputs in $\{0, 1\}$ if the reading of the word in the automaton end up in a state whose value is 1. The set of words accepting by the automaton when starting from a given state is called the language of this state. A language that can be recognized by a finite automaton is called rational.

More generally we can define the language of a state for the more general automaton we defined above. For (X, t, o) a DFA with value in V, we define a function $[\![\cdot]\!] : X \to V^{\Sigma^*}$ that maps states to V-valued languages. For any x in X, it is defined as follow:

$$\begin{cases} [\![x]\!](\varepsilon) & = o(x) \\ [\![x]\!](a \cdot u) & = [\![t_a(x)]\!](u) \text{ for } a \in \Sigma \text{ and } u \in \Sigma^* \end{cases}$$

When $V = \{0, 1\}$, the function $[\![x]\!](\cdot)$ can be assimilated with a subset of Σ^* and is often noted $\mathcal{L}(x)$.

[1] This definition of automata is linked to Moore and Mealy machines.

Definition 1. *We say that two states x and y are language equivalent and write $x \sim y$ when $[\![x]\!] = [\![y]\!]$.*

Without loss of generality, we consider that the states we want to decide the equivalence of belongs to the same automaton. Indeed, if we have two automaton we can assume we only have one by doing the union. Besides if we have two disjoint it might be useful to first merge them partially thanks to similarity before launching the algorithm to test the language equivalence.

In order to be able to determine the language equivalence relation, we will introduce a function operating over relations. Let b be a function over the set of relations over X defined as $b : R \mapsto \{(x, y) \mid o(x) = o(y) \wedge \forall a \in \Sigma,\ t_a(x)\ R\ t_a(y)\}$. Language equivalence coincide with the greatest fixpoint of b, which means that any post-fixpoint of b is included in the language equivalence relation. It will found the principle of the Hopcroft and Karp's algorithm: the algorithm tries to construct a relation R that is a post-fixpoint of b and contains the pair given as input. With the coinduction vocabulary, we say that the algorithm tries to construct a b-bisimulation. For some details on those notion, we will refer the lector to [BP13]. In fact we will present here a more direct way that does not need the introduction of other notions.

Proposition 1. *$x \sim y$ if and only if there exists R such that xRy and $R \subseteq b(R)$.*

Proof. $\underline{\Rightarrow:}$ $\sim$ suits.

$\underline{\Leftarrow:}$ We show by induction on $|u|$ that $\forall u \in \Sigma^*, \forall Y', Z', Y'RZ' \Rightarrow [\![Y']\!](u) = [\![Z']\!](u)$.

□

Hopcroft and Karp's algorithm. Hopcroft and Karp's algorithm (see algorithm 1) takes as input two states x and y of a given automaton (X, t, o) and decide whether they are language equivalent or not. It works by assuming that the two states are language equivalent and exploring the automaton to see whether it is consistent or not.

The algorithm keeps track of a list of pairs it has to check. At each step it inspect one of them. If the outputs are not equal then it means that we can read two words that have not the same value, thus the algorithm reject. If it sees again a pair it already checked, the algorithm goes on and if it is the first time it sees it, it adds its voisins to be list of pairs to be checked.

As the language equivalence is a reflexive, transitive and symmetric relation, the algorithm assume that the one it constructs will also be, then instead of testing if a given pair is in the set of pairs it already checked, it tests if it is on its equivalence closure noted $e(\cdot)$. It allows to potentially not exploring the whole automaton.

Algorithm 1: HK(x, y)

```
R := ∅;
Todo := {(x, y)};
while Todo ≠ ∅ do
    Extract (x', y') from Todo;
    if (x', y') ∈ e(R ∪ Todo) then skip;
    if o(x') ≠ o(y') then return False;
    forall a ∈ Σ do
        Insert (t_a(x'), t_a(y')) in Todo
    Insert (x', y') in R;
Return True;
```

Theorem 1. *The algorithm HK is correct: it terminates and $x \sim y$ if and only if* HK(x, y).

Proof.

- It terminates because each pair can be inserted only once in $Todo$ and we extract one of them at each iteration.
- If the algorithm returns `True` on (x, y), then $x \sim y$ because the algorithm build a relation which is a post-fixpoint of the function b previously defined, namely the relation $e(R)$. Indeed one can easily checked that we have the following loop invariant: $(x, y) \in (R \cup Todo) \ \wedge \ R \subseteq b(e(R \cup Todo))$. Since $Todo$ is empty at the end of the loop we eventually have $(x, y) \in R \ \wedge \ R \subseteq b(e(R))$. By applying e which is monotone[2] and contains the identity (*i.e.* for any R, $R \subseteq e(R)$) we obtain that $(x, y) \in e(R) \ \wedge \ e(R) \subseteq e(b(e(R)))$. Moreover e has a very interesting property[3] with respect to b: $e(b(\cdot)) \leq b(e(\cdot))$ that can easily be proved. Thus we obtain: $e(R) \subseteq b(e(e(R))) = b(e(R))$. Then by proposition 1, $x \sim y$.
- If the algorithm returns `False` on (x, y), then $x \not\sim y$. Indeed, for all (x', y') inserted in $Todo$ during the algorithm, there exists $u \in \Sigma^*$ such that $x \xrightarrow{u} x'$ and $y \xrightarrow{u} y'$, meaning that by reading u we can go from x to x' and from y to y'. Since the algorithm returns `False`, there exists (x', y') such that $o(x') \neq o(y')$ *i.e.* $[\![x']\!](\varepsilon) \neq [\![y']\!](\varepsilon)$. Thus it exists $u \in \Sigma^*$ such that $[\![x]\!](u) \neq [\![y]\!](u)$ so $x \not\sim y$.

□

2.2 HKC algorithm for NFA

Non deterministic finite automata Non deterministic finite automata are defined almost as deterministic one but several transitions outgoing from a single state are allowed.

Formally, a *non deterministic finite automaton* (NFA) over the input alphabet Σ is a triple (X, t, o) where X is a finite set of states, $t : X \to \mathcal{P}(X)^\Sigma$ is the transition function

[2]Here monotone means non decreasing.

[3]In coinduction theory, the function e is said to be compatible and we say we do a b-bisimulation up-to e.

and $o : X -> V$ is the output function which assigns to each state a value in a semi-lattice $(V, +, 0)$.

As said previously, non deterministic finite automata can be transformed into equivalent deterministic one. As the idea is to follow transition from set of states to set of states it is called the powerset construction:

The *powerset construction* of a NFA (X, t, o) is the equivalent deterministic finite automaton (DFA) $(\mathcal{P}(X), o^\sharp, t^\sharp)$ where $t^\sharp : \mathcal{P}(X) \to \mathcal{P}(X)^\Sigma$ and $o^\sharp : \mathcal{P}(X) \to (V, +, 0)$ are defined for all $Y \subseteq X$ and $a \in \Sigma$ as follows:

$$t_a^\sharp(Y) = \begin{cases} t_a(y) & \text{if } Y = \{y\} \text{ with } y \in X \\ 0 & \text{if } Y = 0 \\ t_a^\sharp(Y_1) + t_a^\sharp(Y_2) & \text{if } Y = Y_1 \cup Y_2 \end{cases}$$

$$o^\sharp(Y) = \begin{cases} o(y) & \text{if } Y = \{y\} \text{ with } y \in X \\ 0 & \text{if } Y = 0 \\ o^\sharp(Y_1) + o^\sharp(Y_2) & \text{if } Y = Y_1 \cup Y_2 \end{cases}$$

Note that for all $a \in \Sigma$, $t_a^\sharp$ and $o^\sharp$ are semi-lattices homorphisms from $(\mathcal{P}(X), \cup, \varnothing)$ to $(V, +, 0)$. These properties are fundamental for the improvement of Hopcroft and Karp's algorithm we are going to present.

The more convenient way to define the languages of a NFA is to do it through the powerset construction: the language of $x \in X$ in (X, t, o) is defined as the language of $\{x\}$ in $(\mathcal{P}(X), t^\sharp, o^\sharp)$, namely $[\![\{x\}]\!]$. Thus we focus on language equivalence of sets of states.

Then to determine if $Y \sim Z$ for $Y, Z \subseteq X$, we can simply launch Hopcroft and Karp's algorithm on the powerset construction. However we can use the fact that the states on the powerset are set of states to improve the algorithm, as described in [BP13].

HKC Intuitively, if we know that all the parts of the sets are in relation, then we know that the sets are in relation. Formally we define a function u over the set of relation of $\mathcal{P}(X)$ as $u : R \mapsto \{(Y_1 \cup Y_2, Z_1 \cup Z_2) | Y_1 R Z_1 \wedge Y_2 R Z_2\}$, and we add the closure with respect to u to the equivalence one. The closure we obtain is called the congruence closure and is noted c.

HKC (see algorithm 2) takes as input two sets of states Y and Z of a given NFA (X, t, o) and decide whether they are language equivalent or not. It is the same algorithm as Hopcroft and Karp's one but it works specifically on a powerset construction and use c instead of e.

Algorithm 2: HKC(Y, Z)

```
R := ∅;
Todo := {(Y, Z)};
while Todo ≠ ∅ do
    Extract (Y', Z') from Todo;
    if (Y', Z') ∈ c(R ∪ Todo) then skip;
    if o♯(Y') ≠ o♯(Z') then return False;
    forall a ∈ Σ do
        Insert (t_a♯(Y'), t_a♯(Z')) in Todo
    Insert (Y', Z') in R;
Return True;
```

Theorem 2. *The algorithm* HKC *is correct: it terminates and* $Y \sim Z$ *if and only if* HKC(Y, Z).

Proof. As c has the same properties as e (namely it is monotone, it contains the identity and $c(b(\cdot)) \leq b(c(\cdot))$), the same proof works. □

Example 1. *The Figure 1 shows an example (taken from [BP13]) of execution of the algorithms on a NFA. The relation R constructed with Hopcroft and Karp's algorithm consists of the edges 1,2,3 and 4 while the one constructed with* HKC *consists only of the edges 1 and 2, meaning that the algorithm does not explore the whole automaton. Indeed:* $\{x\}R\{u\} \Rightarrow \{x, y\}c(R)\{u, y\} \Rightarrow \{x, y\}c(R)\{y, y, z\} \Rightarrow \{x, y\}c(R)\{y, z\} \Rightarrow \{x, y\}c(R)\{u\}$.

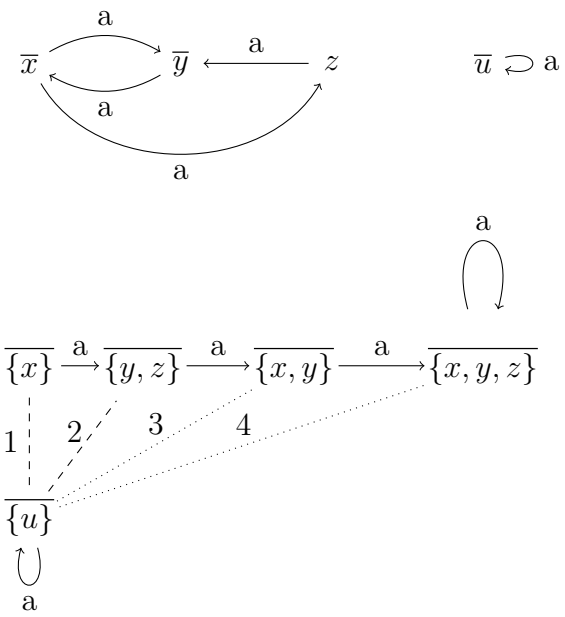

Figure 1: Example of the running of Hopcroft and Karp's and HKC algorithms on a NFA.

Computation of the congruence closure. For the `HKC` algorithm to be effective, we need a way to check whether a pair belongs to the congruence closure of a given relation, which as shown by the example 1 is not trivial.

The idea is that instead of working with the states of the automaton we manipulate their normal form in a rewriting system. For a relation R, we define $\rightsquigarrow_R$ as the smallest irreflexive relation satisfying the following rules:

$$\frac{Y\ R\ Z}{Y \rightsquigarrow_R Y \cup Z} \qquad \frac{Y\ R\ Z}{Z \rightsquigarrow_R Y \cup Z} \qquad \frac{Z \rightsquigarrow_R Z'}{Y \cup Z \rightsquigarrow_R Y \cup Z'}$$

Proposition 2. *For all relations R, the relation $\rightsquigarrow_R$ is convergent.*

Proof. **$\rightsquigarrow_R$ is terminating.** If $Z \rightsquigarrow_R Z'$ then $|Z| < |Z'|$ (by induction on the derivation tree) and $|Z'|$ is bounded by $|X|$w.

$\rightsquigarrow_R$ is confluent. By induction on the derivation tree, $\rightsquigarrow_R$ is locally confluent. Since $\rightsquigarrow_R$ is terminating and locally confluent, Newman's lemma shows that $\rightsquigarrow_R$ is confluent.

□

We denote by $Y{\downarrow_R}$ the normal form of a set Y w.r.t $\rightsquigarrow_R$. Intuitively, the normal form of a set is the largest set of its equivalence class.

Theorem 3. *For all relations R, and for all $Y, Z \in \mathcal{P}(X)$, we have $Y{\downarrow_R} = Z{\downarrow_R}$ iff $(Y, Z) \in c(R)$.*

Proof. $\Rightarrow$ Let's assume that $Y{\downarrow_R} = Z{\downarrow_R}$. We show by induction that for all $Z, Z' \in X$, if $Z \rightsquigarrow_R Z'$ then $Z\ c(R)\ Z'$. Thus, $Y\ c(R)\ Y{\downarrow_R}$ and $Z\ c(R)\ Z{\downarrow_R}$, so $Y\ c(R)\ Z$.

$\Leftarrow$ By induction on the derivation of $Y\ c(R)\ Z$.

□

Thus, in order to check if $(Y, Z) \in c(R \cup todo)$, we only have to compute the normal form of Y and Z with respect to $\rightsquigarrow_{R \cup todo}$. Since each pair of $R \cup todo$ may be used only once as a rewriting rule, the time for checking whether $(Y, Z) \in c(R \cup todo)$ is bounded by $r^2 n$ with $r = |R \cup todo|$ and $n = |X|$. This algorithm is used for its simplicity and because it behaves well in practice.

3 From Büchi automata to finite words automata

A *Büchi automaton* over the alphabet Σ has the same formal definition as a finite automaton, *i.e.* a triple (X, t, o) where X is a finite set of states, t is the transition function going from X to either X^Σ if the automaton is deterministic or $\mathcal{P}(X)^\Sigma$ if it is not, $o : X \to \{0, 1\}$ is the outpout function which determines whether a state x is final ($o(x) = 1$) or not ($o(x) = 0$). However, a Büchi automaton will read infinite words. Then an infinite word will be accepted

by the automaton if and only if we see infinitly often a final state. More formally, given $u = u_1u_2...$ an infinite word over Σ, we say that $\chi = x_1x_2...$ is a calculus of (X, t, o) on u if for all $i \in \mathbb{N}$, $x_{i+1} \in t_{u_i}(x_i)$ and u is accepted by the state x if there exists a calculus $\chi = xx_2...$ such that $\{i \in \mathbb{N} | o(x_i) = 1\}$ is infinite. We note $\mathcal{L}^\omega(x)$ the set of all infinite words that are accepted by x.

The set of languages that can be recognized by a Büchi automaton are called the *rational ω-laguages*.

The subject of this section is to explain the results of [CNP93]. Namely that a set of infinite words recognizing by a Büchi automaton is entirely characterized by its subset of ultimately periodic words (the words of the form $u \cdot v^\omega$). Moreover thoses sets can be represented by rational languages of finite words. This fact allows to manipulate DFAs instead of Büchi automata.

In this section we first define the set of ultimately periodic words of a Rational ω-Language and then show that it is represented by a rational language by constructing an automaton recognizing it.

3.1 Ultimately Periodic Words of a Rational ω-Language

Ultimately periodic words are infinite words of the form $u \cdot v^\omega$ with $(u, v) \in \Sigma^* \times \Sigma^+$. The set of ultimately periodic words of an ω-language $\mathcal{L}$ is composed of all ultimately periodic words of $\mathcal{L}$. More formally, $UP(\Sigma^\omega) = \{u \cdot v^\omega | (u, v) \in \Sigma^* \times \Sigma^+\}$ is the set of all ultimately periodic words and for $\mathcal{L}$ an ω-language over Σ we note $UP(\mathcal{L}) = \mathcal{L} \cap UP(\Sigma^\omega)$ its subset of ultimately periodic words.

Proposition 3. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two ω-rational languages such that $UP(\mathcal{L}_1) = UP(\mathcal{L}_2)$, then $\mathcal{L}_1 = \mathcal{L}_2$.*

Proof. $(\mathcal{L}_1 \cup \mathcal{L}_2) \backslash (\mathcal{L}_1 \cap \mathcal{L}_2)$ is an ω-rational language containing no ultimately periodic word. Yet any non-empty rational ω-language contains at least one ultimately periodic word, so $(\mathcal{L}_1 \cup \mathcal{L}_2) \backslash (\mathcal{L}_1 \cap \mathcal{L}_2)$ is empty and $\mathcal{L}_1 = \mathcal{L}_2$. □

The set of ultimately periodic words of a rational ω-language is thus characteristic of this language. To compare two rational ω-languages it suffices to compare their set of ultimately periodic words. The ultimately periodic word $u \cdot v^\omega \in \Sigma^\omega$ can be represented by the finite word $u \cdot \$ \cdot v \in (\Sigma \cup \$)^*$. Then we can have a language of finite words characterizing a rational ω-language $\mathcal{L}$, namely $\mathcal{L}_\$ = \{u \cdot \$ \cdot v | u \cdot v^\omega \in UP(\mathcal{L})\}$. We will show in the following subsection that this language of finite words happens to be rational, which allows to work easily with it.

3.2 A Finite Automaton Recognizing Ultimately Periodic Words of a Rational ω-Language

There are many ways to prove the rationality of the language $\mathcal{L}_\$$. Here we will present the direct construction of a finite automaton recognizing it which is presented in [CNP93], because we will use this automaton to decide equivalence of Büchi automata. For this we divide the language between prefixes and periods of ultimately periodic words.

Let $\mathcal{L}$ be a rational ω-language and (X, t, o) a non-deterministic Büchi automaton recognizing $\mathcal{L}$. For $x, y \in X$ we define $\mathcal{M}_{x,y} = \{u \in \Sigma^* | x \xrightarrow{u} y\}$ and $\mathcal{N}_y = \{v \in \Sigma^+ | v^\omega \in \mathcal{L}^\omega(y)\}$. Then $(\mathcal{L}^\omega(x))_\$ = \bigcup_{y \in X} \mathcal{M}_{x,y} \cdot \$ \cdot \mathcal{N}_y$. Indeed:

$$\begin{aligned}
w \in (\mathcal{L}^\omega(x))_\$ &\Leftrightarrow \exists u, v \in \Sigma^* \times \Sigma^+ \text{ such that } w = u \cdot \$ \cdot v \wedge u \cdot v^\omega \in \mathcal{L}^\omega(x) \\
&\Leftrightarrow \exists u, v \in \Sigma^* \times \Sigma^+, \exists y \in X \text{ such that } w = u \cdot \$ \cdot v \wedge x \xrightarrow{u} y \wedge v^\omega \in \mathcal{L}^\omega(y) \\
&\Leftrightarrow \exists u, v \in \Sigma^* \times \Sigma^+, \exists y \in X \text{ such that } w = u \cdot \$ \cdot v \wedge u \in \mathcal{M}_{x,y} \wedge v \in \mathcal{N}_y \\
&\Leftrightarrow w \in \bigcup_{y \in X} \mathcal{M}_{x,y} \cdot \$ \cdot \mathcal{N}_y
\end{aligned}$$

The languages $\mathcal{M}_{x,y}$ are rational beacause they are recognized by the finite automata $(X, t, \{y\})$ with initial state x. We want to construct automata recognizing the languages $\mathcal{N}_y$. The problem is that for v^ω to be in $\mathcal{L}^\omega(y)$ there needs to be a cycling run of the word v^k going through a final state for some $k \in \mathbb{N}$ but not necessarly for v itself. For instance, on the automaton of the Figure 3.2, we need to read $(ab)^3$ before finding an accepting cycle. Moreover it could happen that even if the cycle has a small period, you have to read $v^{k'}$ as a prefix before acceding it. The idea of the construction is then to simulate the automaton on each state. We obtain a vector state showing all the calculi of v on the automaton and then we can compute the run of the v^k until we loop and check whether the loop contains a final state or not.

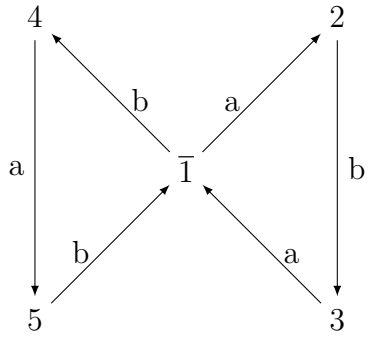

Figure 2: A Büchi automaton with final state 1.

Example 2. *(taken in [CNP93]) Below are the first states of the automaton recognizing $\mathcal{N}_y$ for the automaton of the Figure* **??**.

$$\begin{pmatrix}1,0\\2,0\\3,0\\4,0\\5,0\end{pmatrix} \xrightarrow{a} \begin{pmatrix}2,0\\\bot\\1,1\\5,0\\\bot\end{pmatrix} \xrightarrow{b} \begin{pmatrix}3,0\\\bot\\4,1\\1,1\\\bot\end{pmatrix}$$

From this calculus, we can build the following calculi of the automaton on the word ab, starting from state 1:

$$1 \overset{ab}{\rightsquigarrow} 3 \ , \ 3 \overset{ab}{\rightsquigarrow} 4 \ , \ 4 \overset{ab}{\rightsquigarrow} 1$$

which allow us to find a loop containing a repeated final state showing that ab belongs to $\mathcal{N}_1$, $\mathcal{N}_3$, and $\mathcal{N}_4$.

Formally let (X, t, o) be a Büchi automaton such that $X = \{x_1, x_2, ..., x_m\}$ and let $y \in X$. Without loss of generality we suppose that the automaton is complete. We construct $\mathcal{A}_{\mathcal{N}_y} = (Q, \tau, \omega_y)$ the finite automaton recognizing $\mathcal{N}_y$. The states are vectors of length m containing on each line an element of X and a boolean telling whether we saw a final state during the calculus or not: $Q = (X \times \{0, 1\})^m$. The transition and output functions τ and ω_y are defined as follow:

- There is a transition labelled by a letter a between $q = ((y_1, b_1), ..., (y_m, b_m))$ and $q' = ((y'_1, b'_1), ..., (y'_m, b'_m))$ if and only if $\forall i \in [|1, m|]$, $y'_i \in t_a(y_i) \wedge b'_i = max(b_i, o(y'_i))$. It means that the transition function of the Büchi automaton is independently applied on each line and the knowledge of whether or not a final state has been seen has been propagated.

- A state $q = ((y_1, b_1), ..., (y_m, b_m))$ is final if, from the y-th line, a loop containing a boolean at 1 can be found. Formally, let σ mapping the lines to the index of state on this line, *i.e.* $\sigma(k)$ is the i such that $y_k = x_i$ and let $(j_k)_{k \geq 0}$ be the infinite sequence defined recursively by $j_0 = y$ and $j_{k+1} = \sigma(j_k)$. This sequence ranges only over a finite set of values. Let thus s be the smallest integer satisfying $j_s \in \{j_k | 0 \leq k < s\}$ and s' the only integer such that $s' < s$ and $j_{s'} = j_s$. Then the state q is final if and only if $1 \in \{b_{j_k} | s' \leq k < s\}$.

Proposition 4. *Let $q_0 = ((x_1, 0), ...(x_m, 0))$, then $\mathcal{L}(q_0) = \mathcal{N}_x$.*

The proof can be found in [CNP93].

Then we can obtain a finite automaton recognizing $\mathcal{L}_\$$ as the disjoint union of the automaton $(X, t, \varnothing)$ and the automata $\mathcal{A}_{\mathcal{N}_y}$ for each y to which we're adding the edges $(y, \$, q_{0,y})$ where $q_{0,y}$ is the initial state of $\mathcal{A}_{\mathcal{N}_y}$. We call the obtained automaton $\mathcal{A}_\$$.

The automata $\mathcal{A}_{\mathcal{N}_x}$ have all the same structure, only the acceptance conditions change. So to recognize $\mathcal{N}_Y = \bigcup_{y\in Y}\mathcal{N}_y$ for $Y \subseteq X$, an acceptance condition wich is the union of all acceptance conditions is defined: $\Omega_Y = max_{x\in Y}\omega_x$. Then the automaton (Q, τ, Ω_Y) recognizes $\mathcal{N}_Y$.

4 HKC on Büchi automata

At the end of the previous section, a NFA was built of from a Büchi automaton such that checking language equivalence on this NFA is equivalent to checking the language equivalence on the Büchi automaton. Then the algorithm HKC described in section 2 can be directly applied to the constructed NFA to resolve the language equivalence problem of Büchi automata.

A feature that differs from the finite words case is that if we have two disjoint automata $\mathcal{A}$ and $\mathcal{B}$, we do not want to join them before doing the construction described in the section **??**. We'd rather do the union afterhand and launch HKC on $\mathcal{A}_\$ \cup \mathcal{B}_\$$.

Due to the specific structure of the constructed NFA, there is room for improvements of the algorithm. Those improvements are dealt with in this section.

We begin by exposing separatly the two steps of the algoritm which intuitively correspond repectively to checking equivalence of the prefixes and periods of ultimately periodic words.

In the previous section we transformed a language of infinite words $\mathcal{L} \in 2^{\Sigma^\omega}$ into a language of finite words $\mathcal{L}_\$ \in 2^{\Sigma^*\times\$\times\Sigma^+} = 2^{\Sigma^*\times\Sigma^+} = (2^{\Sigma^+})^{\Sigma^*}$. As $(2^{\Sigma^+}, \cup, \varnothing)$ is a semi lattice we can view the NFA that recognize $\mathcal{L}_\$$ as a NFA with value in $(2^{\Sigma^+}, \cup, \varnothing)$: when reading a word in $\mathcal{M}_{x,y}$ from the state x it will outuput the language $\mathcal{N}_y$ (the languages $\mathcal{M}_{x,y}$ and $\mathcal{N}_y$ have been defined in section 3.2).

Formally let (X, t, o) be a non deterministic Büchi automaton and $Y, Z \subseteq X$. $\mathcal{L}^\omega(Y) = \mathcal{L}^\omega(Z)$ in the Büchi automaton if and only if $\mathcal{L}(Y) = \mathcal{L}(Z)$ in the NFA (X, t, o') where $o' : \begin{cases} X & \to (2^{\Sigma^+}, \cup, \varnothing) \\ x & \mapsto \mathcal{N}_x \end{cases}$.

As the language $\mathcal{N}_x$ is entirely determined by x, we can also express o' as

$o' : \begin{cases} X & \to (\mathcal{P}(X)/\equiv, \cup, \varnothing) \\ x & \mapsto \{x\} \end{cases}$ with $Y \equiv Z$ iff $\mathcal{N}_Y = \mathcal{N}_Z$

So if we know a way to test if ${o'}^\sharp(Y') \equiv {o'}^\sharp(Z')$ for any Y', $Z' \subseteq X$, we can apply HKC on $\mathcal{A}'$ to test if $\mathcal{L}(Y) = \mathcal{L}(Z) \Leftrightarrow \mathcal{L}^\omega(Y) = \mathcal{L}^\omega(Z)$.

As testing ${o'}^\sharp(Y') \equiv {o'}^\sharp(Z')$ is testing an equality between languages, it seems natural to then use again HKC to test it. In fact, we can construct the automata $\mathcal{A}_{\mathcal{N}_{Y'}} \cup \mathcal{A}_{\mathcal{N}_{Z'}}$ and run HKC on it.

We'd like to improve the efficiency of the test "$\mathcal{N}_{Y'} = \mathcal{N}_{Z'}$" thanks to two remarks:

- As the structure of each $\mathcal{A}_{\mathcal{N}_Y}$ is the same for any $Y \subseteq X$, it is redundant to lauch HKC for every Y', Z' we encounter during the main algorithm. Instead we would like to

preprocess all the ordered pairs HKC would compare and then only check if the output condition is respected for all the ordered pairs.

- As [CNP93] notices, the structure of the automaton $\mathcal{A}_{\mathcal{N}_x}$ described in section 3 allows for a more efficient determinization than the direct powerset construction described in section 2.

4.1 Pre-processing

The idea is to do HKC in two steps: first the list of all ordered pairs which would be in R if the acceptance condition were always met is established, then the fact that the acceptance condition is indeed always met is checked.

The algorithm 3 describes the preprocessing of the list of ordered pairs. It is the same as HKC but the acceptance condition is not checked.

Algorithm 3: `Preprocessing`

```
1 Pairs := ∅;
2 Todo := {(q_0, q'_0)};
3 while Todo ≠ ∅ do
4 |   Extract (q_1, q_2) from Todo;
5 |   if (q_1, q_2) ∈ c(Pairs ∪ Todo) then skip;
6 |   forall a ∈ Σ do
7 |   |   Insert (τ_a^♯(q_1), τ_a^♯(q_2)) in Todo
8 |   Insert (q_1, q_2) in Pairs;
9 Return Pairs;
```

The algorithm 4 checks the acceptance condition for all the ordered pairs returned by the preprocessing.

Algorithm 4: `Verification`(Y, Z)

```
1 forall (q_1, q_2) ∈ Pairs do
2 |   if Ω_Y^♯(q_1) ≠ Ω_Z^♯(q_2) then return False;
3 Return True;
```

Theorem 4. *Both algorithms terminate and* `Verification`(Y, Z) *returns* `True` *if and only if* $\mathcal{N}_Y = \mathcal{N}_Z$.

Proof.

- `Preprocessing` terminates for the same arguments as HK and `Verification` terminates because the set of ordered pairs produced by `Preprocessing` is finite.

- Let $Y, Z \subseteq X$ such that $\mathcal{N}_Y = \mathcal{N}_Z$. HKC$(q_0, q'_0)$ on the automaton $\mathcal{A}_{\mathcal{N}_Y} \cup \mathcal{A}_{\mathcal{N}_Z}$ returns `True`. Then the list $Pairs$ returned by `Preprocessing` is the same as the list R computed by HKC(q_0, q'_0). Indeed the test in the 6th line of HKC(Y, Z) never returns `False` and thus can be ignored, which is exactly the algorithm `Preprocessing`. Then

`Verification`(q_0, q_0') run the test for all ordered pairs of $Pairs$, *i.e.* for all ordered pairs of R, and this test never fails because `HKC`(q_0, q_0') does not return `False`, so `Verification`(q_0, q_0') returns `True`.

- Let $Y, Z \subseteq X$ such that $\mathcal{N}_Y \neq \mathcal{N}_Z$. `HKC`$(q_0, q_0')$ on the automaton $\mathcal{A}_{\mathcal{N}_Y} \cup \mathcal{A}_{\mathcal{N}_Z}$ returns `False`: it encounters (q_1, q_2) such that $\Omega_Y^\sharp(q_1) \neq \Omega_Z^\sharp(q_2)$. Until this moment, the test has always been successful and so `HKC`(q_0, q_0') and `Preprocessing` had the same behaviour, which means that `Preprocessing` will also encounter this ordered pair and it will add it to $Pairs$. Then `Verification`(Y, Z) will test whether or not $\Omega_Y^\sharp(q_1) = \Omega_Z^\sharp(q_2)$ and will thus returns `False`.

□

Remark 1. *In the case where we have non disjoint automata, we have to run* `Preprocessing` *on $\mathcal{A}_{\mathcal{N}_y}$ and a copy of itself (otherwise the algorithm stop at the first step when seeing the pair (q_0, q_0) because of the reflexive closure). This come from the fact that even if the automata have the same structure, they potentially have different output functions. But in this case the execution of* `Preprocessing` *is degenerated and in this particular case we can express more easily the congruence function as a problem of covering.*

4.2 Compression of the states

As soon as the studied Büchi automaton is not deterministic, the automaton $\mathcal{A}_\$$ is not deterministic either: it contains the automaton itself and the $\mathcal{A}_{\mathcal{N}_x}$ are not deterministic either. However as it is a automaton over finite words, it can be determinized by the powerset construction exposed in section 2. But then it yields a number of states equals to two power the number of state of $\mathcal{A}_\$$ which is $m + m \cdot (|Q|) = m \cdot ((2m)^m + 1)$, where m is the number of states of the Büchi automaton. Yet accessible states of the $\mathcal{A}_{\mathcal{N}_x}$ automata have a particular shape which allows for a compressed representation of the states and provides a bound on their number.

Indeed as we simulate the Büchi automaton on each line of the vector states independently, for a given transition and a given line the set of accessible states does not change with the other lines. Therefore for a given line the value taken does not depend on the value of the other lines. Thus a state in the determinized $\mathcal{A}_{\mathcal{N}_x}$ can be described entirely described by the possible sets for each line.

Formally, let $\mathcal{A}_{\mathcal{N}_x} = (Q, \tau, \omega_x)$ recognizing $\mathcal{N}_x$ as in section 3. In the DFA $(\mathcal{P}(Q), \tau^\sharp, \omega_x^\sharp)$ corresponding to the powerset construction described in section 2, if P is an accessible state and if $p = ((p_1, b_1), ..., (p_m, b_m))$ and $p' = ((p_1', b_1'), ...(p_m', b_m'))$ are in P then any $p'' = ((p_1'', f_1''), ...(p_m'', f_m''))$ with $(p_k'', f_k'') \in \{(p_k, f_k), (p_k', f_k')\}$ for all $k \in [\![1, m]\!]$ is also in P. The state P is thus entirely defined by the sets P_k describing the possibilities for the k-th line of P.

Then the number of state of the determinized automaton recognizing $\mathcal{N}_x$ is $(2^{2m})^m = 2^{2m^2}$. In total we obtain $2^m + 2^m \cdot 2^{2m^2} = 2^m + 2^{2m^2+m}$ states for the whole determinization of $\mathcal{A}_\$$.

In this subsection we first define the determinized automaton we obtain by representing the states as described previously and then explain the problem this represention present for computing the congruence and a way to solve it.

Intelligent determinized automaton recognizing $\mathcal{N}_x$. Following the ideas previouly described, we define (Q', τ', ω'_x) a DFA recognizing $\mathcal{N}_x$ with:

- The set of states is $Q' = (\mathcal{P}(X \times \{0,1\}))^m$ (instead of $\mathcal{P}(Q) = \mathcal{P}\left((X \times \{0,1\})^m\right)$).
- The transition function is the natural extension of the one described in section 3. There is a transition labelled by a letter a between $P = (P_1, ...P_m)$ and $P' = (P'_1, ...P'_m)$ if and only if $\forall i \in [|1, m|]$, $P'_i = \{(y', b') | \exists (y, b) \in P_i, y' \in t_a(y) \wedge b' = max(b, o(y'))\}$.
- The output function is also the natural extension of the one described in section 3: there must exists a vector included in the state that is final for x. The problem lays in computing this output function without having to expand exponentially all the paths. A solution is to see the state as a directed graph represented by its adjacency lists, the edges being labelled either by 0 or 1. To know if the state is final for x, we have to know if there exists a cycle containing a edge labelled by 1 accessible from x. Thus by computing the strongly connected components we can compute all the output functions (*i.e.* the ω'_x for all $x \in X$). The full algorithm, taking as input a given state P, is the following:

  1. Computing all the strongly connected components, for instance with Tarjan's algorithm.
  2. Keep the strongly connected components having a final edge.
  3. Reverse the edges and compute the connected components of the final strongly connected components. We get the set of all states $x \in X$ such that P is final for x.

Once again the automata have all the same structures, only the acceptance conditions change. So to recognize $\mathcal{N}_Y = \bigcup_{y \in Y} \mathcal{N}_y$ for $Y \subseteq X$, we define an acceptance condition which is the union of all acceptance conditions: $\Omega'_Y = max_{x \in Y} \omega'_x$. Then the DFA $\mathcal{A}'_{\mathcal{N}_Y} = (Q', \tau', \Omega'_Y)$ recognizes $\mathcal{N}_Y$.

As for a classic powerset determinization, this construction can be done on the fly, when exploring the automaton. So we can run `HKC` or the combinaison of `Preprocessing` and `Verification` on it.

In fact τ' and ω'_x behave as $\tau^\sharp$ and $\omega^\sharp_x$ but instead of being lattices homomorphisms from $(\mathcal{P}(Q), \subseteq, \cup)$ to $(\mathcal{P}(Q), \subseteq, \cup)$ and $(\{0,1\}, \leq, \texttt{or})$ they are from $(Q', \leq, \texttt{max})$ to $(Q', \leq, \texttt{max})$ and $(\{0,1\}, \leq, \texttt{or})$, where Q' is seen as a set of graphs represented by their adjacency matrices (matrices over $\{\bot, 0, 1\}$), $\leq$ and `max` being the pointwise operations on the matrices. Yet they are homomorphisms only when we look at the states of the automaton (Q', τ', ω'_x),

meaning that if P is a state of Q' such that it is the compression of a state $S = S_1 \cup S_2$ then then $\tau'(P) = \tau'(P_1)$ `max` $\tau'(P_2)$ and $\omega'_x(P) = \omega'_x(P_1)$ `or` $\omega'_x(P_2)$ where P_1 (resp. P_2) is the compression of S_1 (resp. S_2).

Indeed if we take $P_1 = \begin{pmatrix} 0 & \bot \\ 1 & \bot \end{pmatrix}$ and $P_2 = \begin{pmatrix} \bot & 1 \\ \bot & 0 \end{pmatrix}$, we have P_1 `max` $P_2 = \begin{pmatrix} 0 & 1 \\ 1 & \bot \end{pmatrix} = P$ and $\omega'_1(P) = 1 \neq \omega'_1(P_1)$ `or` $\omega'_1(P_2) = 0$ `or` $0 = 0$. It comes from the fact that P does not correspond to the compression of $S_1 \cup S_2$ where S_i stands for the decompressed version of P_i (in fact the state $S_1 \cup S_2$ is not a state of the automaton).

This problem arises when computing the congruence closure as described in section 2. In fact when applying the rewriting rules, the union of states that is performed may not give a real state of the automaton. The typical case is when we run `HKC` on two disjoint automata, then the rewriting rules unite states coming from different automata. If we take the automata consisting of the only state P_i looping on itself, then $P_1 R P_2$ because neither of them are final, and as we previously said we can not unite them with `max`. Thus, in order to compute the congruence with the rewriting system presented in section 2, we have to decompressed the states and work with the lattice $(\mathcal{P}(Q), \subseteq, \cup)$, which is costful.

In the next paragraph we explain how to adapt the rewriting system to allow to work with the compressed states.

Adapting the rewriting system. Let R be a relation over Q'. Let Π be a 'list' of compressed states. Let P, P' be two compressed states such that PRP'. If $P \leq \Pi$ then we 'add' P' to Π, where 'list', $\leq$ and 'add' are to be defined. In the rewriting system presented in section 2, the list corresponds to a single subset of X, the $\leq$ to the inclusion and the 'add' to the union.

The natural way to define 'add' is by the list contructor. $\leq$ is then defined as: $P \leq \Pi$ if and only if for any vector $v \in P$ there exists $P'' \in \Pi$ such that $v \in P''$. Two questions arise: is there no way to have a more efficient way to 'add' an compressed state to a list of compressed state and how to test $\leq$.

A possibility to improve the constructor is to try to compress the new state with other when possible, the idea being that if we can compress then we can do it step by step. The algorithm becomes: to insert P' in Π, we browse Π until we find P'' such that P' and P'' only differ on one line, which means we merge them and then insert P' `max` P'' in Π. If we can not find any P'' satisfying this requirement then we simply add P' to Π.

At first we thought that this operation might allow for a simple test of $\leq$: we hoped that to check wether $P \leq \Pi$ we only had to check wether there existed $P'' \in \Pi$ such that $P \leq P''$. But we found an example where the contructor does not even allow to reach a normal form: if we take $\Pi = \left\{ \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \begin{pmatrix} 1 \\ 4 \end{pmatrix} \right\}$ and try to insert $\begin{pmatrix} 1 \\ 3 \end{pmatrix}$ we can either obtain $\Pi = \left\{ \begin{pmatrix} 1,2 \\ 3 \end{pmatrix}, \begin{pmatrix} 1 \\ 4 \end{pmatrix} \right\}$ or $\Pi = \left\{ \begin{pmatrix} 1 \\ 3,4 \end{pmatrix}, \begin{pmatrix} 2 \\ 3 \end{pmatrix} \right\}$. And in the first case if we try to know whether $P = \begin{pmatrix} 1 \\ 3,4 \end{pmatrix} \in \Pi$

or not, we have to decompressed P, which could lead to exponential complexity.

After not succeeding in finding a polynomial algorithm to solve the problem of deciding wether a compressed state is lesser or equal than, we tried to look up its theoritical complexity. In fact we showed that this problem is CoNP-Complete.

Theorem 5. *Let $\Pi = P_1 :: P_2 :: ... :: P_k$ be a list of compressed states and P be a compressed state. Then the problem of deciding wether or not $P \leq \Pi$ is CoNP-Complete.*

Proof.

- The problem is in CoNP. In fact to show that $P \not\leq \Pi$ we can take as witness v a vector of P which does not belong to any of the P_i.
- To show that the problem is CoNP-Hard we will show that its complementary problem is NP-Hard via a reduction from 3-SAT. Let $\mathcal{I} = C_1 \wedge ...C_k$ be an instance of 3-SAT. We note $x_1, ..., x_n$ the boolean variables of $\mathcal{I}$. We construct an instance of our problem as $P = (\{1, 2\}, ..., \{1, 2\})$ of length n and $\Pi = P_1 :: ...P_k$ where P_i is a vector of length n such that the j-th line contains $\{1\}$ if $\texttt{not}(x_j) \in C_i$, $\{2\}$ if $(x_j) \in C_i$, and $\{1, 2\}$ if neither x_j nor $\texttt{not}(x_j) \in C_i$. Let's now show that $\mathcal{I}$ is satisfiable if and only if $P \not\leq \Pi$.

  $\Rightarrow$ If $\mathcal{I}$ is not satisfiable then for all instantiation ϕ, there exists C_i such that ϕ does not satisfy C_i. Moreover the set of vectors of P is in bijection with the set of all instanciation via the function $f : \phi \mapsto v_\phi$ such that its j-th line is set to 1 if ϕ assigns x_j to `true` and to 2 otherwise. Then $v_\phi \leq P_i$ if and only if ϕ does not satisfy C_i. Thus:

  $$\begin{aligned} \mathcal{I} \text{ is not satisfiable } &\Rightarrow \forall \phi, \exists C_i \text{ such that } \phi \text{ does not satisfy } C_i \\ &\Rightarrow \forall v \in P, \exists P_i \text{ such that } v \leq P_i \\ &\Rightarrow P \leq \Pi \end{aligned}$$

  We can conclude that if $P \not\leq \Pi$ then $\mathcal{I}$ is satifiable.

  $\Leftarrow$ If $P \leq \Pi$, for all ϕ there is a natural number i such that $v_\phi \in P_i$, meaning that ϕ does not satisfy C_i. So $\mathcal{I}$ can not be satisfied. We can conclude that if $\mathcal{I}$ is satifiable then $P \not\leq \Pi$.

  The problem is indeed CoNP-Complete.

□

As the problem is CoNP-Complete, we wanted to transform it into a SAT-problem so that we can use a SAT-solver to solve it. Indeed SAT-solver are often efficient in practice to solve problems that are in NP and considered a difficult.

Let $\Pi = P_1 :: P_2 :: ... :: P_k$ be a list of compressed states and P be a compressed state. We take as variables $x_1, ..., x_n$ a set of bounded natural numbers (it can be encoded in binary).

The SAT formula equivalent to $P \leq \Pi$ is then:

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in P \wedge \left(\bigwedge_{i=1}^{n} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \notin P_i \right)$$

$$\text{Where: } \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in P_i = \bigwedge_{j=1}^{n} x_j \in (P_i)_j = \bigwedge_{j=1}^{n} \bigvee_{y \in (P_i)_j} (x_j = y)$$

We can then solve the SAT-formula using a SAT-solver.

Remark 2. *In the case of non-disjoint automata from the start, the covering problem for the non-optimized determinization of the automaton become exactly an instance of the problem of deciding if $P \leq \Pi$ for Π a list of compressed states and P a compressed state. Thus we can also solve it using a SAT-solver.*

5 Conclusion

To conclude, we showed during this internship that the algorithm `HKC` [BP13] can be adapted to run on the structure constructed in [CNP93] to study Büchi automata with finite words automata. Moreover we provide solid theoritical foundations towards optimisations. Even if a lot of other improvements can be thought of (see the following paragraph), we can hope that the algorithm presented in this report run well in practice.

This works is intended to be continued during a phd starting in september 2017. It will explore the following possibilities:

- Implementation: As the goal of the algorithm is rather to be efficient in practice than to provide theoritical bounds, it is necessary to implement and test it. A priori the implementation will be based on the pre-existing one for `HKC`. It is the next step of the project.
- Formalize: Try to formalize all the content of this report as coinduction up-to techniques, if possible.
- Improvement of the congruence: On the same idea that the congruence takes advantage of the structure of the states in the powerset construction, the specific structure of the vector states in $\mathcal{A}_{\mathcal{N}_x}$ could be exploited.
- Other way to compute the congruence: In the case of finite words automata, the congruence can also be computed by other ways. Perhaps one of them could be more easily adapted than the rewriting system. In particular, one way is to express the congruence as a logic formula. If we can adapt it we could directly launch a SAT-solver on it.

- LTL: As it seems to be one of the most frequent application, it could be interesting if there is something particular in the Büchi automaton generated by the LTL formula that we could exploit.
- Other automata: We could try to extend this algorithm to even more classes of automata. The first automata to look at would be the parity automata, because the Büchi automata are a particular case of parity automata.

References

[BP13] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *Principle of Programming Languages (POPL)*, pages 457–468, Roma, Italy, January 2013. ACM. 16p.

[CNP93] Hugues Calbrix, Maurice Nivat, and Andreas Podelski. Ultimately periodic words of rational ω-languages. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 554–566. Springer, 1993.

[CVWY91] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. *Memory efficient algorithms for the verification of temporal properties*, pages 233–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[GO01] Paul Gastin and Denis Oddoux. *Fast LTL to Büchi Automata Translation*, pages 53–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.