Checking NFA Equivalence with Bisimulations up to Congruence

Filippo Bonchi Damien Pous *

CNRS, ENS Lyon, Université de Lyon, LIP (UMR 5668)

{filippo.bonchi,damien.pous}@ens-lyon.fr

Abstract

We introduce *bisimulation up to congruence* as a technique for proving language equivalence of non-deterministic finite automata. Exploiting this technique, we devise an optimisation of the classical algorithm by Hopcroft and Karp [18]. We compare our approach to the recently introduced antichain algorithms, by analysing and relating the two underlying coinductive proof methods. We give concrete examples where we exponentially improve over antichains; experimental results moreover show non negligible improvements.

Categories and Subject Descriptors F.4.3 [*Mathematical Logic*]: Decision Problems; F.1.1 [*Models of computation*]: Automata; D.2.4 [*Program Verification*]: Model Checking

Keywords Language Equivalence, Automata, Bisimulation, Coinduction, Up-to techniques, Congruence, Antichains.

1. Introduction

Checking language equivalence of finite automata is a classical problem in computer science, which finds applications in many fields ranging from compiler construction to model checking.

Equivalence of deterministic finite automata (DFA) can be checked either via minimisation [17] or through Hopcroft and Karp's algorithm [2, 18], which exploits an instance of what is nowadays called a *coinduction proof principle* [25, 27, 29]: two states recognise the same language if and only if there exists a *bisimulation* relating them. In order to check the equivalence of two given states, Hopcroft and Karp's algorithm creates a relation containing them and tries to build a bisimulation by adding pairs of states to this relation: if it succeeds then the two states are equivalent, otherwise they are different.

On the one hand, minimisation algorithms have the advantage of checking the equivalence of all the states at once (while Hopcroft and Karp's algorithm only check a given pair of states). On the other hand, they have the disadvantage of needing the whole automata from the beginning, while Hopcroft and Karp's algorithm can be executed "on-the-fly" [13], on a lazy DFA whose transitions are computed on demand.

* Work partially funded by the PiCoq project, ANR-10-BLAN-0305 and by the COGIP (PEPS-CNRS) project.

POPL'13, January 23–25, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1832-7/13/01...$10.00

This difference is fundamental for our work and for other recently introduced algorithms based on *antichains* [1, 31]. Indeed, when starting from non-deterministic finite automata (NFA), the powerset construction used to get deterministic automata induces an exponential factor. In contrast, the algorithm we introduce in this work for checking equivalence of NFA (as well as those in [1, 31]) usually does not build the whole deterministic automaton, but just a small part of it. We write "usually" because in few bad cases, the algorithm still needs exponentially many states of the DFA.

Our algorithm is grounded on a simple observation on determinised NFA: for all sets X and Y of states of the original NFA, the union (written $+$) of the language recognised by X (written $[\![X]\!]$) and the language recognised by Y ($[\![Y]\!]$) is equal to the language recognised by the union of X and Y ($[\![X+Y]\!]$). In symbols:

$$[\![X+Y]\!] = [\![X]\!] + [\![Y]\!] \tag{1}$$

This fact leads us to introduce a sound and complete proof technique for language equivalence, namely *bisimulation up to context*, that exploits both *induction* (on the operator $+$) and *coinduction*: if a bisimulation R equates both the (sets of) states X_1, Y_1 and X_2, Y_2, then $[\![X_1]\!] = [\![Y_1]\!]$ and $[\![X_2]\!] = [\![Y_2]\!]$ and, by (1), we can immediately conclude that also $X_1 + X_2$ and $Y_1 + Y_2$ are language equivalent. Intuitively, bisimulations up to context are bisimulations which *do not need to relate* $X_1 + X_2$ and $Y_1 + Y_2$ when X_1 (resp. X_2) and Y_1 (resp. Y_2) are already related.

To illustrate this idea, let us check the equivalence of states x and u in the following NFA. (Final states are overlined, labelled edges represent transitions.)

$$\begin{tikzcd}
x \arrow[rr, bend left=50, "a"] & z \arrow[l, "a"] \arrow[r, bend left, "a"] & \overline{y} \arrow[l, bend left, "a"] & & u \arrow[rr, bend left=50, "a"] \arrow[r, bend left, "a"] & w \arrow[l, bend left, "a"] & \overline{v} \arrow[l, "a"]
\end{tikzcd}$$

The determinised automaton is depicted below.

$$\begin{tikzcd}
\{x\} \arrow[r, "a"] \arrow[d, dash, dashed, "1"'] & \overline{\{y\}} \arrow[r, "a"] \arrow[d, dash, dashed, "2"'] & \{z\} \arrow[r, "a"] \arrow[d, dash, dashed, "3"'] & \overline{\{x,y\}} \arrow[r, "a"] \arrow[d, dash, dotted, "4"'] & \overline{\{y,z\}} \arrow[r, "a"] \arrow[dl, dash, dotted, "5"'] & \overline{\{x,y,z\}} \arrow[loop below, "a"] \arrow[dll, dash, dotted, "6"'] \\
\{u\} \arrow[r, "a"'] & \overline{\{v,w\}} \arrow[r, "a"'] & \{u,w\} \arrow[r, "a"'] & \overline{\{u,v,w\}} \arrow[loop right, "a"]
\end{tikzcd}$$

Each state is a set of states of the NFA, final states are overlined: they contain at least one final state of the NFA. The numbered lines show a relation which is a bisimulation containing x and u. Actually, this is the relation that is built by Hopcroft and Karp's algorithm (the numbers express the order in which pairs are added).

The dashed lines (numbered by 1, 2, 3) form a smaller relation which is not a bisimulation, but a bisimulation up to context: the equivalence of states $\{x, y\}$ and $\{u, v, w\}$ could be immediately deduced from the fact that $\{x\}$ is related to $\{u\}$ and $\{y\}$ to $\{v, w\}$, without the need of further exploring the determinised automaton.

Bisimulations up-to, and in particular bisimulations up to context, have been introduced in the setting of concurrency theory [25, 28] as a proof technique for bisimilarity of CCS or π-calculus processes. As far as we know, they have never been used for proving language equivalence of NFA.

Among these techniques one should also mention *bisimulation up to equivalence*, which, as we show in this paper, is implicitly used in the original Hopcroft and Karp's algorithm. This technique can be briefly explained by noting that not all bisimulations are equivalence relations: it might be the case that a bisimulation relates (for instance) X and Y, Y and Z but not X and Z. However, since $[\![X]\!] = [\![Y]\!]$ and $[\![Y]\!] = [\![Z]\!]$, we can immediately conclude that X and Z recognise the same language. Analogously to bisimulations up to context, a bisimulation up to equivalence *does not need to relate* X and Z when they are both related to some Y.

The techniques of up-to equivalence and up-to context can be combined resulting in a powerful proof technique which we call *bisimulation up to congruence*. Our algorithm is in fact just an extension of Hopcroft and Karp's algorithm that attempts to build a bisimulation up to congruence instead of a bisimulation up to equivalence. An important consequence when using up to congruence is that we do not need to build the whole deterministic automata, but just those states that are needed for the bisimulation up-to. For instance, in the above NFA, the algorithm stops after equating z and $u + v$ and does not build the remaining four states. Despite their use of the up to equivalence technique, this is not the case with Hopcroft and Karp's algorithm, where all accessible subsets of the deterministic automata have to be visited at least once.

The ability of visiting only a small portion of the determinised automaton is also the key feature of the antichain algorithm [31] and its optimisation exploiting similarity [1]. The two algorithms are designed to check *language inclusion* rather than equivalence, but we can relate these approaches by observing that the two problems are equivalent ($[\![X]\!] = [\![Y]\!]$ iff $[\![X]\!] \subseteq [\![Y]\!]$ and $[\![Y]\!] \subseteq [\![X]\!]$; and $[\![X]\!] \subseteq [\![Y]\!]$ iff $[\![X]\!] + [\![Y]\!] = [\![Y]\!]$ iff $[\![X + Y]\!] = [\![Y]\!]$).

In order to compare with these algorithms, we make explicit the coinductive up-to technique underlying the antichain algorithm [31]. We prove that this technique can be seen as a restriction of up to congruence, for which *symmetry* and *transitivity* are not allowed. As a consequence, the antichain algorithm usually needs to explore more states than our algorithm. Moreover, we show how to integrate the optimisation proposed in [1] in our setting, resulting in an even more efficient algorithm.

In summary, the contributions of this work are: (1) the observation that Hopcroft and Karp implicitly use bisimulations up to equivalence, (2) an efficient algorithm for checking language equivalence (and inclusion), based on a powerful up to technique, and (3) a comparison with antichain algorithms, by recasting them into our coinductive framework.

Outline

Section 2 recalls Hopcroft and Karp's algorithm for DFA, showing that it implicitly exploits bisimulation up to equivalence. Section 3 describes the novel algorithm, based on bisimulations up to congruence. We compare this algorithm with the antichain one in Section 4, and we show how to exploit similarity in Section 5. Section 6 is devoted to benchmarks. Sections 7 and 8 discuss related and future works. Omitted proofs can be found in [6].

Notation

We denote sets by capital letters $X, Y, S, T \dots$ and functions by lower case letters $f, g, \dots$ Given sets X and Y, $X \times Y$ is their Cartesian product, $X \uplus Y$ is the disjoint union and X^Y is the set of functions $f\colon Y \to X$. Finite iterations of a function $f\colon X \to X$ are denoted by f^n (formally, $f^0(x) = x$, $f^{n+1}(x) = f(f^n(x))$). The collection of subsets of X is denoted by $\mathcal{P}(X)$. The (omega) iteration of a function $f\colon \mathcal{P}(X) \to \mathcal{P}(X)$ is denoted by f^ω (formally, $f^\omega(Y) = \bigcup_{n\geq 0} f^n(Y)$). For a set of letters A, $A^\star$ denotes the set of all finite words over A; ϵ the empty word; and w_1w_2 the concatenation of words $w_1, w_2 \in A^\star$. We use 2 for the set $\{0, 1\}$ and $2^{A^\star}$ for the set of all languages over A.

2. Hopcroft and Karp's algorithm for DFA

A deterministic finite automaton (DFA) over the alphabet A is a triple (S, o, t), where S is a finite set of states, $o\colon S \to 2$ is the output function, which determines if a state $x \in S$ is final ($o(x) = 1$) or not ($o(x) = 0$), and $t\colon S \to S^A$ is the transition function which returns, for each state x and for each letter $a \in A$, the next state $t_a(x)$. For $a \in A$, we write $x \xrightarrow{a} x'$ to mean that $t_a(x) = x'$. For $w \in A^\star$, we write $x \xrightarrow{w} x'$ for the least relation such that (1) $x \xrightarrow{\epsilon} x$ and (2) $x \xrightarrow{aw'} x'$ iff $x \xrightarrow{a} x''$ and $x'' \xrightarrow{w'} x'$.

For any DFA, there exists a function $[\![-]\!]\colon S \to 2^{A^\star}$ mapping states to languages, defined for all $x \in S$ as follows:

$$[\![x]\!](\epsilon) = o(x) \ , \qquad [\![x]\!](aw) = [\![t_a(x)]\!](w) \ .$$

The language $[\![x]\!]$ is called the language accepted by x. Given two automata (S_1, o_1, t_1) and (S_2, o_2, t_2), the states $x_1 \in S_1$ and $x_2 \in S_2$ are said to be *language equivalent* (written $x_1 \sim x_2$) iff they accept the same language.

Remark 1. *In the following, we will always consider the problem of checking the equivalence of states of one single and fixed automaton (S, o, t). We do not loose generality since for any two automata (S_1, o_1, t_1) and (S_2, o_2, t_2) it is always possible to build an automaton $(S_1 \uplus S_2, o_1 \uplus o_2, t_1 \uplus t_2)$ such that the language accepted by every state $x \in S_1 \uplus S_2$ is the same as the language accepted by x in the original automaton (S_i, o_i, t_i). For this reason, we also work with automata without explicit initial states: we focus on the equivalence of two arbitrary states of a fixed DFA.*

2.1 Proving language equivalence via coinduction

We first define bisimulation. We make explicit the underlying notion of progression which we need in the sequel.

Definition 1 (Progression, Bisimulation)**.** *Given two relations $R, R' \subseteq S \times S$ on states, R* progresses to *R', denoted $R \rightarrowtail R'$, if whenever $x\ R\ y$ then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in A$, $t_a(x)\ R'\ t_a(y)$.*

A bisimulation *is a relation R such that $R \rightarrowtail R$.*

As expected, bisimulation is a sound and complete proof technique for checking language equivalence of DFA:

Proposition 1 (Coinduction)**.** *Two states are language equivalent iff there exists a bisimulation that relates them.*

2.2 Naive algorithm

Figure 1 shows a naive version of Hopcroft and Karp's algorithm for checking language equivalence of the states x and y of a deterministic finite automaton (S, o, t). Starting from x and y, the algorithm builds a relation R that, in case of success, is a bisimulation. In order to do that, it employs the set (of pairs of states) *todo* which, intuitively, at any step of the execution, contains the pairs (x', y') that must be checked: if (x', y') already belongs to R, then it has already been checked and nothing else should be done. Otherwise, the algorithm checks if x' and y' have the same outputs (i.e., if both are final or not). If $o(x') \neq o(y')$, then x and y are different. If $o(x') = o(y')$, then the algorithm inserts (x', y') in R and, for all $a \in A$, the pairs $(t_a(x'), t_a(y'))$ in *todo*.

```
Naive(x, y)

(1) $R$ is empty; $todo$ is empty;
(2) insert $(x,y)$ in $todo$;
(3) while $todo$ is not empty, do
  (3.1)  extract $(x',y')$ from $todo$;
  (3.2)  if $(x',y') \in R$ then continue;
  (3.3)  if $o(x') \neq o(y')$ then return $false$;
  (3.4)  for all $a \in A$,
            insert $(t_a(x'), t_a(y'))$ in $todo$;
  (3.5)  insert $(x',y')$ in $R$;
(4) return $true$;
```

Figure 1. Naive algorithm for checking the equivalence of states x and y of a DFA (S, o, t); R and *todo* are sets of pairs of states. The code of $\texttt{HK}(x, y)$ is obtained by replacing step 3.2 with `if` $(x', y') \in e(R)$ `then continue`.

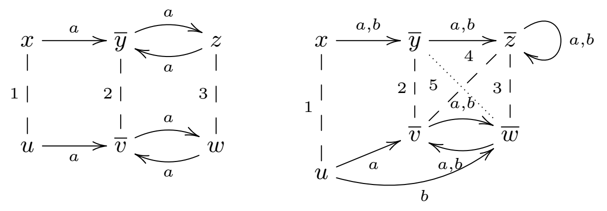

Figure 2. Checking for DFA equivalence.

Proposition 2. *For all $x, y \in S$, $x \sim y$ iff* $\texttt{Naive}(x, y)$.

Proof. We first observe that if $\texttt{Naive}(x, y)$ returns true then the relation R that is built before arriving to step 4 is a bisimulation. Indeed, the following proposition is an invariant for the loop corresponding to step 3:

$$R \rightarrowtail R \cup todo$$

This invariant is preserved since at any iteration of the algorithm, a pair (x', y') is removed from *todo* and inserted in R after checking that $o(x') = o(y')$ and adding $(t_a(x'), t_a(y'))$ for all $a \in A$ in *todo*. Since *todo* is empty at the end of the loop, we eventually have $R \rightarrowtail R$, i.e., R is a bisimulation. By Proposition 1, $x \sim y$.

We now prove that if $\texttt{Naive}(x, y)$ returns false, then $x \not\sim y$. Note that for all (x', y') inserted in *todo*, there exists a word $w \in A^\star$ such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$. Since $o(x') \neq o(y')$, then $[\![x']\!](\epsilon) \neq [\![y']\!](\epsilon)$ and thus $[\![x]\!](w) = [\![x']\!](\epsilon) \neq [\![y']\!](\epsilon) = [\![y]\!](w)$, that is $x \not\sim y$. □

Since both Hopcroft and Karp's algorithm and the one we introduce in Section 3 are simple variations of this naive one, it is important to illustrate its execution with an example. Consider the DFA with input alphabet $A = \{a\}$ in the left-hand side of Figure 2, and suppose we want to check that x and u are language equivalent.

During the initialisation, (x, u) is inserted in *todo*. At the first iteration, since $o(x) = 0 = o(u)$, (x, u) is inserted in R and (y, v) in *todo*. At the second iteration, since $o(y) = 1 = o(v)$, (y, v) is inserted in R and (z, w) in *todo*. At the third iteration, since $o(z) = 0 = o(w)$, (z, w) is inserted in R and (y, v) in *todo*. At the fourth iteration, since (y, v) is already in R, the algorithm does nothing. Since there are no more pairs to check in *todo*, the relation R is a bisimulation and the algorithm terminates returning true.

These iterations are concisely described by the numbered dashed lines in Figure 2. The line i means that the connected pair is inserted in R at iteration i. (In the sequel, when enumerating iterations, we ignore those where a pair from *todo* is already in R so that there is nothing to do.)

Remark 2. *Unless it finds a counter-example,* $\texttt{Naive}$ *constructs the* smallest *bisimulation that relates the two starting states (see Proposition 8 in [6]). On the contrary, minimisation algorithms [17] are designed to compute the* largest *bisimulation relation for a given automaton. For instance, taking automaton on the left of Figure 2, they would equate the states x and w which are language equivalent, while* $\texttt{Naive}(x, u)$ *does not relate them.*

2.3 Hopcroft and Karp's algorithm

The naive algorithm is quadratic: a new pair is added to R at each non-trivial iteration, and there are only n^2 such pairs, where $n = |S|$ is the number of states of the DFA. To make this algorithm (almost) linear, Hopcroft and Karp actually record a set of *equivalence classes* rather than a set of visited pairs. As a consequence, their algorithm may stop earlier when an encountered pair of states is not already in R but in its reflexive, symmetric, and transitive closure. For instance, in the right-hand side example from Figure 2, we can stop when we encounter the dotted pair (y, w) since these two states already belong to the same equivalence class according to the four previous pairs.

With this optimisation, the produced relation R contains at most n pairs (two equivalence classes are merged each time a pair is added). Formally, and ignoring the concrete data structure to store equivalence classes, Hopcroft and Karp's algorithm consists in simply replacing step 3.2 in Figure 1 with

```
(3.2)  if $(x',y') \in e(R)$ then continue;
```

where $e\colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ is the function mapping each relation $R \subseteq S \times S$ into its symmetric, reflexive, and transitive closure. We hereafter refer to this algorithm as $\texttt{HK}$.

2.4 Bisimulations up-to

We now show that the optimisation used by Hopcroft and Karp corresponds to exploiting an "up-to technique".

Definition 2 (Bisimulation up-to)**.** *Let $f\colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ be a function on relations on S. A relation R is a* bisimulation up to f *if $R \rightarrowtail f(R)$, i.e., whenever $x \mathrel{R} y$ then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in A$, $t_a(x) \mathrel{f(R)} t_a(y)$.*

With this definition, Hopcroft and Karp's algorithm just consists in trying to build a bisimulation up to e. To prove the correctness of the algorithm, it suffices to show that any bisimulation up to e is contained in a bisimulation. We use for that the notion of compatible function [26, 28]:

Definition 3 (Compatible function)**.** *A function $f\colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ is* compatible *if it is monotone and it preserves progressions: for all $R, R' \subseteq S \times S$,*

$$R \rightarrowtail R' \text{ entails } f(R) \rightarrowtail f(R')\,.$$

Proposition 3. *Let f be a compatible function. Any bisimulation up to f is contained in a bisimulation.*

Proof. Suppose that R is a bisimulation up to f, i.e., that $R \rightarrowtail f(R)$. Using compatibility of f and by a simple induction on n, we get $\forall n, f^n(R) \rightarrowtail f^{n+1}(R)$. Therefore, we have

$$\bigcup_n f^n(R) \rightarrowtail \bigcup_n f^n(R)\,,$$

in other words, $f^\omega(R) = \bigcup_n f^n(R)$ is a bisimulation. This latter relation trivially contains R, by taking $n = 0$. □

We could prove directly that e is a compatible function; we however take a detour to ease our correctness proof for the algorithm we propose in Section 3.

Lemma 1. *The following functions are compatible:*

- id: *the identity function;*
- $f \circ g$: *the composition of compatible functions f and g;*
- $\bigcup F$: *the pointwise union of an arbitrary family F of compatible functions: $\bigcup F(R) = \bigcup_{f \in F} f(R)$;*
- f^ω: *the (omega) iteration of a compatible function f.*

Lemma 2. *The following functions are compatible:*

- *the constant reflexive function: $r(R) = \{(x,x) \mid x \in S\}$;*
- *the converse function: $s(R) = \{(y,x) \mid x\ R\ y\}$;*
- *the squaring function: $t(R) = \{(x,z) \mid \exists y, x\ R\ y\ R\ z\}$.*

Intuitively, given a relation R, $(s \cup \mathrm{id})(R)$ is the symmetric closure of R, $(r \cup s \cup \mathrm{id})(R)$ is its reflexive and symmetric closure, and $(r \cup s \cup t \cup \mathrm{id})^\omega(R)$ is its symmetric, reflexive and transitive closure: $e = (r \cup s \cup t \cup \mathrm{id})^\omega$. Another way to understand this decomposition of e is to recall that for a given R, $e(R)$ can be defined inductively by the following rules:

$$\frac{}{x\ e(R)\ x}\ r \qquad \frac{x\ e(R)\ y}{y\ e(R)\ x}\ s \qquad \frac{x\ e(R)\ y\ \ y\ e(R)\ z}{x\ e(R)\ z}\ t \qquad \frac{x\ R\ y}{x\ e(R)\ y}\ \mathrm{id}$$

Theorem 1. *Any bisimulation up to e is contained in a bisimulation.*

Proof. By Proposition 3, it suffices to show that e is compatible, which follows from Lemma 1 and Lemma 2. ☐

Corollary 1. *For all $x, y \in S$, $x \sim y$ iff $\mathtt{HK}(x,y)$.*

Proof. Same proof as for Proposition 2, by using the invariant $R \rightarrowtail e(R) \cup todo$. We deduce that R is a bisimulation up to e after the loop. We conclude with Theorem 1 and Proposition 1. ☐

Returning to the right-hand side example from Figure 2, Hopcroft and Karp's algorithm constructs the relation

$$R_{\mathtt{HK}} = \{(x,u), (y,v), (z,w), (z,v)\}$$

which is not a bisimulation, but a bisimulation up to e: it contains the pair (x,u), whose b-transitions lead to (y,w), which is not in $R_{\mathtt{HK}}$ but in its equivalence closure, $e(R_{\mathtt{HK}})$.

3. Optimised algorithm for NFA

We now move from DFA to non-deterministic automata (NFA). We start with standard definitions about semi-lattices, determinisation, and language equivalence for NFA.

A *semi-lattice* $(X, +, 0)$ consists of a set X and a binary operation $+\colon X \times X \to X$ which is associative, commutative, idempotent (ACI), and has $0 \in X$ as identity. Given two semi-lattices $(X_1, +_1, 0_1)$ and $(X_2, +_2, 0_2)$, an *homomorphism* of semi-lattices is a function $f\colon X_1 \to X_2$ such that for all $x, y \in X_1$, $f(x +_1 y) = f(x) +_2 f(y)$ and $f(0_1) = 0_2$. The set $2 = \{0,1\}$ is a semi-lattice when taking $+$ to be the ordinary Boolean or. Also the set of all languages $2^{A^\star}$ carries a semi-lattice where $+$ is the union of languages and 0 is the empty language. More generally, for any set X, $\mathcal{P}(X)$ is a semi-lattice where $+$ is the union of sets and 0 is the empty set. In the sequel, we indiscriminately use 0 to denote the element $0 \in 2$, the empty language in $2^{A^\star}$, and the empty set in $\mathcal{P}(X)$. Similarly, we use $+$ to denote the Boolean or in 2, the union of languages in $2^{A^\star}$, and the union of sets in $\mathcal{P}(X)$.

A non-deterministic finite automaton (NFA) over the input alphabet A is a triple (S, o, t), where S is a finite set of states, $o\colon S \to 2$ is the output function (as for DFA), and $t\colon S \to \mathcal{P}(S)^A$ is the transition relation, which assigns to each state $x \in S$ and input letter $a \in A$ a set of possible successor states.

The *powerset construction* transforms any NFA (S, o, t) in the DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ where $o^\sharp\colon \mathcal{P}(S) \to 2$ and $t^\sharp\colon \mathcal{P}(S) \to \mathcal{P}(S)^A$ are defined for all $X \in \mathcal{P}(S)$ and $a \in A$ as follows:

$$o^\sharp(X) = \begin{cases} o(x) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ o^\sharp(X_1) + o^\sharp(X_2) & \text{if } X = X_1 + X_2 \end{cases}$$

$$t_a^\sharp(X) = \begin{cases} t_a(x) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ t_a^\sharp(X_1) + t_a^\sharp(X_2) & \text{if } X = X_1 + X_2 \end{cases}$$

Observe that in $(\mathcal{P}(S), o^\sharp, t^\sharp)$, the states form a semi-lattice $(\mathcal{P}(S), +, 0)$, and $o^\sharp$ and $t^\sharp$ are, by definition, semi-lattices homomorphisms. These properties are fundamental for the up-to technique we are going to introduce; in order to highlight the difference with generic DFA (which usually do not carry this structure), we introduce the following definition.

Definition 4. *A* determinised NFA *is a DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ obtained via the powerset construction of some NFA (S, o, t).*

Hereafter, we use a new notation for representing states of determinised NFA: in place of the singleton $\{x\}$, we just write x and, in place of $\{x_1, \dots, x_n\}$, we write $x_1 + \cdots + x_n$.

For an example, consider the NFA (S, o, t) depicted below (left) and part of the determinised NFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ (right).

$$\begin{tikzcd} \overline{x} \arrow[r, bend left, "a"] \arrow[rr, bend right=50, "a"'] & \overline{y} \arrow[l, bend left, "a"] & z \arrow[l, "a"'] \end{tikzcd} \qquad \begin{tikzcd} \overline{x} \arrow[r, "a"] & \overline{y+z} \arrow[r, "a"] & \overline{x+y} \arrow[r, "a"] & \overline{x+y+z} \arrow[loop below, "a"] \end{tikzcd}$$

In the determinised NFA, x makes one single a-transition going into $y + z$. This state is final: $o^\sharp(y+z) = o^\sharp(y) + o^\sharp(z) = o(y) + o(z) = 1 + 0 = 1$; it makes an a-transition into $t_a^\sharp(y+z) = t_a^\sharp(y) + t_a^\sharp(z) = t_a(y) + t_a(z) = x + y$.

The language accepted by the states of an NFA (S, o, t) can be conveniently defined via the powerset construction: the language accepted by $x \in S$ is the language accepted by the singleton $\{x\}$ in the DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$, in symbols $[\![\{x\}]\!]$. Therefore, in the following, instead of considering the problem of language equivalence of states of the NFA, we focus on language equivalence of *sets* of states of the NFA: given two sets of states X and Y in $\mathcal{P}(S)$, we say that X and Y are language equivalent ($X \sim Y$) iff $[\![X]\!] = [\![Y]\!]$. This is exactly what happens in standard automata theory, where NFA are equipped with sets of initial states.

3.1 Extending coinduction to NFA

In order to check whether two sets of states X and Y of an NFA (S, o, t) are language equivalent, we can simply employ the bisimulation proof method on $(\mathcal{P}(S), o^\sharp, t^\sharp)$. More explicitly, a bisimulation for an NFA (S, o, t) is a relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, such that whenever $X\ R\ Y$ then (1) $o^\sharp(X) = o^\sharp(Y)$, and (2) for all $a \in A$, $t_a^\sharp(X)\ R\ t_a^\sharp(Y)$. Since this is just the old definition of bisimulation (Definition 1) applied to $(\mathcal{P}(S), o^\sharp, t^\sharp)$, we get that $X \sim Y$ iff there exists a bisimulation relating them.

Remark 3 (Linear time v.s. branching time)**.** *It is important not to confuse these bisimulation relations with the standard Milner-and-Park bisimulations [25] (which* strictly *imply language equivalence): in a standard bisimulation R, if the following states x and* *y of an NFA are in R,*

$$\begin{tikzcd}[row sep=small] & x_1 & & & y_1 \\ x \arrow[ru, "a"] \arrow[rd, "a"'] & \vdots & & y \arrow[ru, "a"] \arrow[rd, "a"'] & \vdots \\ & x_n & & & y_m \end{tikzcd}$$

then each x_i should be in R with some y_j (and vice-versa). Here, instead, we first transform the transition relation into

$$x \xrightarrow{a} x_1 + \cdots + x_n \qquad y \xrightarrow{a} y_1 + \cdots + y_m \ ,$$

using the powerset construction, and then we require that the sets $x_1 + \cdots + x_n$ and $y_1 + \cdots + y_m$ are related by R.

3.2 Bisimulation up to congruence

The semi-lattice structure $(\mathcal{P}(S), +, 0)$ carried by determinised NFA makes it possible to introduce a new up-to technique, which is not available with plain DFA: *up to congruence*. This technique relies on the following function.

Definition 5 (Congruence closure). *Let $u\colon \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S)) \to \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S))$ be the function on relations on sets of states defined for all $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ as:*

$$u(R) = \{(X_1 + X_2,\ Y_1 + Y_2) \mid X_1\ R\ Y_1 \text{ and } X_2\ R\ Y_2\}\,.$$

The function $c = (r \cup s \cup t \cup u \cup \mathrm{id})^\omega$ is called the congruence closure *function.*

Intuitively, $c(R)$ is the smallest equivalence relation which is closed with respect to $+$ and which includes R. It could alternatively be defined inductively using the rules r, s, t, and id from the previous section, and the following one:

$$\frac{X_1\ c(R)\ Y_1 \qquad X_2\ c(R)\ Y_2}{X_1 + X_2\ c(R)\ Y_1 + Y_2}\ u$$

We call bisimulations up to congruence the bisimulations up to c. We report the explicit definition for the sake of clarity:

Definition 6 (Bisimulation up to congruence). *A bisimulation up to congruence for an NFA (S, o, t) is a relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, such that whenever $X\ R\ Y$ then*

1. *$o^\sharp(X) = o^\sharp(Y)$ and*
2. *for all $a \in A$, $t^\sharp_a(X)\ c(R)\ t^\sharp_a(Y)$.*

We then show that bisimulations up to congruence are sound, using the notion of compatibility:

Lemma 3. *The function u is compatible.*

Proof. We assume that $R \rightarrowtail R'$, and we prove that $u(R) \rightarrowtail u(R')$. If $X\ \ u(R)\ \ Y$, then $X = X_1 + X_2$ and $Y = Y_1 + Y_2$ for some X_1, X_2, Y_1, Y_2 such that $X_1\ R\ Y_1$ and $X_2\ R\ Y_2$. By assumption, we have $o^\sharp(X_1) = o^\sharp(Y_1)$, $o^\sharp(X_2) = o^\sharp(Y_2)$, and for all $a \in A$, $t^\sharp_a(X_1)\ R'\ t^\sharp_a(Y_1)$ and $t^\sharp_a(X_2)\ R'\ t^\sharp_a(Y_2)$. Since $o^\sharp$ and $t^\sharp$ are homomorphisms, we deduce $o^\sharp(X_1 + X_2) = o^\sharp(Y_1 + Y_2)$, and for all $a \in A$, $t^\sharp_a(X_1 + X_2)\ u(R')\ t^\sharp_a(Y_1 + Y_2)$. □

Theorem 2. *Any bisimulation up to congruence is contained in a bisimulation.*

Proof. By Proposition 3, it suffices to show that c is compatible, which follows from Lemmas 1, 2 and 3. □

In the Introduction, we already gave an example of bisimulation up to context, which is a particular case of bisimulation up to congruence (up to context corresponds using just the function $(r \cup u \cup \mathrm{id})^\omega$, without closing under s and t).

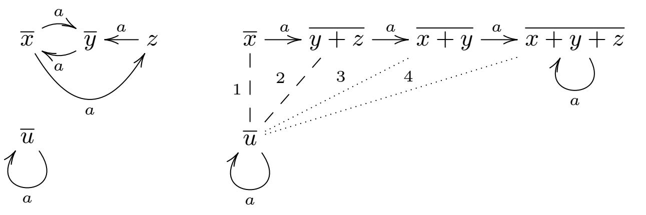

Figure 3. Bisimulations up to congruence, on a single letter NFA.

```
                         Naive(X, Y)

(1) R is empty; todo is empty;
(2) insert (X, Y) in todo;
(3) while todo is not empty, do
 (3.1)  extract (X', Y') from todo;
 (3.2)  if $(X',Y') \in R$ then continue;
 (3.3)  if $o^\sharp(X') \neq o^\sharp(Y')$ then return false;
 (3.4)  for all $a \in A$,
          insert $(t^\sharp_a(X'), t^\sharp_a(Y'))$ in todo;
 (3.5)  insert (X', Y') in R;
(4) return true;
```

Figure 4. On-the-fly naive algorithm, for checking the equivalence of sets of states X and Y of an NFA (S, o, t). The code for on-the-fly $\mathtt{HK}(X, Y)$ is obtained by replacing the test in step `3.2` with $(X', Y') \in e(R)$; the code for $\mathtt{HKC}(X, Y)$ is obtained by replacing this test with $(X', Y') \in c(R \cup todo)$.

A more involved example illustrating the use of all ingredients of the congruence closure function (c) is given in Figure 3. The relation R expressed by the dashed numbered lines (formally $R = \{(x,\ u), (y + z,\ u)\}$) is neither a bisimulation nor a bisimulation up to equivalence since $y{+}z \xrightarrow{a} x{+}y$ and $u \xrightarrow{a} u$, but $(x{+}y,\ u) \notin e(R)$. However, R is a bisimulation up to congruence. Indeed, we have $(x + y,\ u) \in c(R)$:

$$\begin{aligned} x + y\ \ & c(R)\ u + y && ((x, u) \in R) \\ & c(R)\ y + z + y && ((y + z, u) \in R) \\ & =\ \ y + z && \\ & c(R)\ u && ((y + z, u) \in R) \end{aligned}$$

In contrast, we need four pairs to get a bisimulation up to e containing (x, u): this is the relation depicted with both dashed and dotted lines in Figure 3.

Note that we can deduce many other equations from R; in fact, $c(R)$ defines the following partition of sets of states:

$$\{0\}, \{y\}, \{z\}, \{x, u, x{+}y, x{+}z, \text{ and the 9 remaining subsets}\}.$$

3.3 Optimised algorithm for NFA

Algorithms for NFA can be obtained by computing the determinised NFA on-the-fly [13]: starting from the algorithms for DFA (Figure 1), it suffices to work with sets of states, and to inline the powerset construction. The corresponding code is given in Figure 4. The naive algorithm (`Naive`) does not use any up to technique, Hopcroft and Karp's algorithm (`HK`) reasons up to equivalence in step `3.2`, and the optimised algorithm, referred as `HKC` in the sequel, relies on up to congruence: step `3.2` becomes

```
 (3.2)  if $(X',Y') \in c(R \cup todo)$ then continue;
```

Observe that we use $c(R \cup todo)$ rather than $c(R)$: this allows us to skip more pairs, and this is safe since all pairs in $todo$ will eventually be processed.

Corollary 2. *For all $X, Y \in \mathcal{P}(S)$, $X \sim Y$ iff* $\mathtt{HKC}(X, Y)$.

Proof. Same proof as for Proposition 2, by using the invariant $R \rightarrowtail c(R \cup todo)$ for the loop. We deduce that R is a bisimulation up to congruence after the loop. We conclude with Theorem 2 and Proposition 1. □

The most important point about these three algorithms is that they compute the states of the determinised NFA lazily. This means that only *accessible* states need to be computed, which is of practical importance since the determinised NFA can be exponentially large. In case of a negative answer, the three algorithms stop even before all accessible states have been explored; otherwise, if a bisimulation (possibly up-to) is found, it depends on the algorithm:

- With `Naive`, all accessible states need to be visited, by definition of bisimulation.
- With `HK`, the only case where some accessible states can be avoided is when a pair (X, X) is encountered: the algorithm skips this pair so that the successors of X are not necessarily computed (this situation rarely happens in practice—it actually never happens when starting with disjoint automata). In the other cases where a pair (X, Y) is skipped, then X and Y are necessarily already related to some other states in R, so that their successors will eventually be explored.
- With `HKC`, only a small portion of the accessible states is built (check the experiments in Section 6). To see a concrete example, let us execute `HKC` on the NFA from Figure 3. After two iterations, $R = \{(x, u), (y + z, u)\}$. Since $x + y\ c(R)\ u$, the algorithm stops without building the states $x + y$ and $x + y + z$. Similarly, in the example from the Introduction, `HKC` does not construct the four states corresponding to pairs 4, 5, and 6.

This ability of `HKC` to ignore parts of the determinised NFA comes from the up to congruence technique, which allows one to infer properties about states that were not necessarily encountered before. As we shall see in Section 4, the efficiency of antichain-based algorithms [1, 31] also comes from their ability to skip large parts of the determinised NFA.

3.4 Computing the congruence closure

For the optimised algorithm to be effective, we need a way to check whether some pairs belong to the congruence closure of some relation (step `3.2`). We present here a simple solution based on set rewriting; the key idea is to look at each pair (X, Y) in a relation R as a pair of rewriting rules:

$$X \to X + Y \qquad\qquad Y \to X + Y \ ,$$

which can be used to compute normal forms for sets of states. Indeed, by idempotence, $X\ R\ Y$ entails $X\ c(R)\ X + Y$.

Definition 7. *Let $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ be a relation on sets of states. We define $\rightsquigarrow_R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ as the smallest irreflexive relation that satisfies the following rules:*

$$\frac{X\ R\ Y}{X \rightsquigarrow_R X + Y} \qquad \frac{X\ R\ Y}{Y \rightsquigarrow_R X + Y} \qquad \frac{Z \rightsquigarrow_R Z'}{U + Z \rightsquigarrow_R U + Z'}$$

Lemma 4. *For all relations R, the relation $\rightsquigarrow_R$ is convergent.*

In the sequel, we denote by $X{\downarrow_R}$ the normal form of a set X w.r.t. $\rightsquigarrow_R$. Intuitively, the normal form of a set is the largest set of its equivalence class. Recalling the example from Figure 3, the common normal form of $x + y$ and u can be computed as follows (R is the relation $\{(x, u), (y + z, u)\}$):

$$\begin{tikzcd}
x+y \arrow[dr, rightsquigarrow] & & & & u \arrow[dl, rightsquigarrow] \\
& x+y+u \arrow[dr, rightsquigarrow] & & x+u \arrow[dl, rightsquigarrow] & \\
& & x+y+z+u & &
\end{tikzcd}$$

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

Thus, for checking $(X, Y) \in c(R \cup todo)$ we only have to compute the normal form of X and Y with respect to $\rightsquigarrow_{R\cup todo}$. Note that each pair of $R\cup todo$ may be used only once as a rewriting rule, but we do not know in advance in which order to apply these rules. Therefore, the time required to find one rule that applies is in the worst case rn where $r = |R \cup todo|$ is the size of the relation $R\cup todo$, and $n = |S|$ is the number of states of the NFA (assuming linear time complexity for set-theoretic union and containment of sets of states). Since we cannot apply more than r rules, the time for checking whether $(X, Y) \in c(R \cup todo)$ is bounded by $r^2 n$.

We tried other solutions, notably by using binary decision diagrams [9]. We have chosen to keep the presented rewriting algorithm for its simplicity and because it behaves well in practice.

3.5 Complexity hints

The complexity of `Naive`, `HK` and `HKC` is closely related to the size of the relation that they build. Hereafter, we use $v = |A|$ to denote the number of letters in A.

Lemma 5. *The three algorithms require at most $1 + v{\cdot}|R|$ iterations, where $|R|$ is the size of the produced relation; moreover, this bound is reached whenever they return true.*

Therefore, we can conveniently reason about $|R|$.

Lemma 6. *Let $R_{\mathtt{Naive}}$, $R_{\mathtt{HK}}$, and $R_{\mathtt{HKC}}$ denote the relations produced by the three algorithms. We have*

$$|R_{\mathtt{HKC}}|, |R_{\mathtt{HK}}| \le m \qquad\qquad |R_{\mathtt{Naive}}| \le m^2 \ , \tag{2}$$

where $m \le 2^n$ is the number of accessible states in the determinised NFA and n is the number of states of the NFA. If the algorithms return true, we moreover have

$$|R_{\mathtt{HKC}}| \le |R_{\mathtt{HK}}| \le |R_{\mathtt{Naive}}| \ . \tag{3}$$

As shown below in Section 4.2.4, $R_{\mathtt{HKC}}$ can be exponentially smaller than $R_{\mathtt{HK}}$. Notice however that the problem of deciding NFA language equivalence is PSPACE-complete [24], and that none of the algorithms presented here is in PSPACE: all of them store a set of visited pairs, and in the worst case, this set can become exponentially large with all of them. (This also holds for the antichain algorithms [1, 31] which we describe in Section 4.) Instead, the standard PSPACE algorithm does not store any set of visited pairs: it checks all words of length smaller than 2^n. While this can be done in polynomial space, this systematically requires exponential time.

3.6 Using `HKC` for checking language inclusion

For NFA, language inclusion can be reduced to language equivalence in a rather simple way. Since the function $[\![-]\!]\colon \mathcal{P}(S) \to 2^{A^\star}$ is a semi-lattice homomorphism (see Theorem 7 in [6]), for any given sets of states X and Y, $[\![X{+}Y]\!] = [\![Y]\!]$ iff $[\![X]\!]{+}[\![Y]\!] = [\![Y]\!]$ iff $[\![X]\!] \subseteq [\![Y]\!]$. Therefore, it suffices to run $\mathtt{HKC}(X{+}Y, Y)$ to check the inclusion $[\![X]\!] \subseteq [\![Y]\!]$.

In such a situation, all pairs that are eventually manipulated by `HKC` have the shape $(X'{+}Y', Y')$ for some sets X', Y'. Step `3.2` of `HKC`, where it checks whether the current pair belongs to the congruence closure of the relation, can thus be simplified. First, the pairs in the current relation can only be used to rewrite from right to left. Second, the following lemma allows one to avoid unnecessary normal form computations:

Lemma 7. *For all sets X, Y and for all relations R, we have $X{+}Y\ c(R)\ Y$ iff $X \subseteq Y{\downarrow_R}$.*

Proof. We first prove that for all $X, Y,\ X{\downarrow_R} = Y{\downarrow_R}$ iff $X \subseteq Y{\downarrow_R}$ and $Y \subseteq X{\downarrow_R}$, using the fact that the normalisation function ${\downarrow_R}\colon\ X \mapsto X{\downarrow_R}$ is monotone and idempotent. The announced result follows by Theorem 3 since $Y \subseteq (X+Y){\downarrow_R}$ is always true and $X{+}Y \subseteq Y{\downarrow_R}$ iff $X \subseteq Y{\downarrow_R}$. □

At this point, the reader might wonder whether checking the two inclusions separetly is more convenient than checking the equivalence directly. Hereafter, we show that this is not the case.

Lemma 8. *Let X, Y be two sets of states; let $R_\subseteq$ and $R_\supseteq$ be the relations computed by* `HKC`*$(X{+}Y, Y)$ and* `HKC`*$(X{+}Y, X)$, respectively. If $R_\subseteq$ and $R_\supseteq$ are bisimulations up to congruence, then the following relation is a bisimulation up to congruence:*

$$R_= = \{(X', Y') \mid (X'{+}Y', Y') \in R_\subseteq \text{ or } (X'{+}Y', X') \in R_\supseteq\}.$$

On the contrary, checking the equivalence directly actually allows one to skip some pairs that cannot be skipped when reasoning by double inclusion. As an example, consider the DFA on the right of Figure 2. The relation computed by `HKC`(x, u) contains only four pairs (because the fifth one follows from transitivity). Instead, the relations built by `HKC`$(x, x{+}u)$ and `HKC`$(u{+}x, u)$ would both contain five pairs: transitivity cannot be used since our relations are now oriented (from $y \le v$, $z \le v$ and $z \le w$, we cannot deduce $y \le w$). Another example, where we get an exponential factor by checking the equivalence directly rather than through the two inclusions, can be found in Section 4.2.4.

In a sense, the behaviour of the coinduction proof method here is similar to that of standard proofs by induction, where one often has to strengthen the induction predicate to get a (nicer) proof.

4. Antichain algorithm

In [31], De Wulf et al. have proposed the *antichain* approach for checking language inclusion of NFA. We show that this approach can be explained in terms of *simulations up to upward-closure* that, in turn, can be seen as a special case of bisimulations up to congruence. Before doing so, we recall the standard notion of antichain and we describe the antichain algorithm (`AC`).

Given a partial order $(X, \sqsubseteq)$, an *antichain* is a subset $Y \subseteq X$ containing only incomparable elements (that is, for all $y_1, y_2 \in Y$, $y_1 \not\sqsubseteq y_2$ and $y_2 \not\sqsubseteq y_1$). `AC` exploits antichains over the set $S \times \mathcal{P}(S)$, where the ordering is given by $(x_1, Y_1) \sqsubseteq (x_2, Y_2)$ iff $x_1 = x_2$ and $Y_1 \subseteq Y_2$.

In order to check $[\![X]\!] \subseteq [\![Y]\!]$ for two sets of states X, Y of an NFA (S, o, t), `AC` maintains an antichain of pairs (x', Y'), where x' is a state of the NFA and Y' is a state of the determinised automaton. More precisely, the automaton is explored non-deterministically (via t) for obtaining the first component of the pair and deterministically (via $t^\sharp$) for the second one. If a pair such that x' is accepting ($o(x') = 1$) and Y' is not ($o^\sharp(Y') = 0$) is encountered, then a counter-example has been found. Otherwise all derivatives of the pair along the automata transitions have to be inserted into the antichain, so that they will be explored. If one of these pairs p is larger than a previously encountered pair p' ($p' \sqsubseteq p$) then the language inclusion corresponding to p is subsumed by p' so that p can be skipped; otherwise, if $p \sqsubseteq p_1, \ldots, p_n$ for some pairs $p_1, \ldots, p_n$ that are already in the antichain, then one can safely remove these pairs: they are subsumed by p and, by doing so, the set of visited pairs remains an antichain.

Remark 4. *An important difference between* `HKC` *and* `AC` *consists in the fact that the former inserts pairs in* todo *without checking whether they are redundant (this check is performed when the pair is processed), while the latter removes all redundant pairs whenever a new one is inserted. Therefore, the cost of an iteration with* `HKC` *is merely the cost of the corresponding congruence check, while the cost of an iteration with* `AC` *is merely that of inserting all successors of the corresponding pair and simplifying the antichain.*

Note that the above description corresponds to the “forward” antichain algorithm, as described in [1]. Instead, the original antichain algorithm, as first described in [31], is “backward” in the sense that the automata are traversed in the reversed way, from accepting states to initial states. The two versions are dual [31] and we could similarly define the backward counterpart of `HKC` and `HK`. We however stick to the forward versions for the sake of clarity.

4.1 Coinductive presentation

Leaving apart the concrete data structures used to manipulate antichains, we can rephrase this algorithm using a coinductive framework, like we did for Hopcroft and Karp’s algorithm.

First define a notion of *simulation*, where the left-hand side automaton is executed non-deterministically:

Definition 8 (Simulation)**.** *Given two relations $T, T' \subseteq S \times \mathcal{P}(S)$, T* s-progresses *to T', denoted $T \rightarrowtail_s T'$, if whenever $x\ T\ Y$ then*

1. *$o(x) \le o^\sharp(Y)$ and*
2. *for all $a \in A$, $x' \in t_a(x)$, $x'\ T'\ t^\sharp_a(Y)$.*

A simulation *is a relation T such that $T \rightarrowtail_s T$.*

As expected, we obtain the following coinductive proof principle:

Proposition 4 (Coinduction)**.** *For all sets X, Y, we have $[\![X]\!] \subseteq [\![Y]\!]$ iff there exists a simulation T such that for all $x \in X$, $x\ T\ Y$.*

(Note that like for our notion of bisimulation, the above notion of simulation is weaker than the standard one from concurrency theory [25], which *strictly* entails language inclusion—Remark 3.)

To account for the antichain algorithm, where we can discard pairs using the preorder $\sqsubseteq$, it suffices to define the *upward closure* function $\uparrow\colon \mathcal{P}(S \times \mathcal{P}(S)) \to \mathcal{P}(S \times \mathcal{P}(S))$ as

$$\uparrow\! T = \{(x, Y) \mid \exists (x', Y') \in T \text{ s.t. } (x', Y') \sqsubseteq (x, Y)\} \ .$$

A pair belongs to the upward closure $\uparrow\! T$ of a relation $T \subseteq S \times \mathcal{P}(S)$, if and only if this pair is subsumed by some pair in T. In fact, rather than trying to construct a simulation, `AC` attempts to construct a simulation up to upward closure.

Like for `HK` and `HKC`, this method can be justified by defining the appropriate notion of s-compatible function, showing that any simulation up to an s-compatible function is contained in a simulation, and showing that the upward closure function ($\uparrow$) is s-compatible.

Theorem 4. *Any simulation up to $\uparrow$ is contained in a simulation.*

Corollary 3. *For all $X, Y \in \mathcal{P}(S)$, $[\![X]\!] \subseteq [\![Y]\!]$ iff* `AC`(X, Y).

4.2 Comparing `HKC` and `AC`

The efficiency of the two algorithms strongly depends on the number of pairs that they need to explore. In the following (Sections 4.2.3 and 4.2.4), we show that `HKC` can explore far fewer pairs than `AC`: when checking language inclusion of automata that share some states, or when checking language equivalence. We would also like to formally prove that (a) `HKC` never explores more than `AC`, and (b) when checking inclusion of disjoint automata, `AC` never explores more than `HKC`. Unfortunately, the validity of these statements highly depends on numerous assumptions about the two algorithms (e.g., on the exploration strategy) and their potential proofs seem complicated and not really informative. For these reasons, we preferred to investigate the formal correspondence at the level of the coinductive proof techniques, where it is much cleaner.

4.2.1 Language inclusion: HKC can mimic AC

As explained in Section 3.6, we can check the language inclusion of two sets X, Y by executing $\texttt{HKC}(X{+}Y, Y)$. We now show that for any simulation up to upward closure that proves the inclusion $[\![X]\!] \subseteq [\![Y]\!]$, there exists a bisimulation up to congruence of the same size which proves the same inclusion. For $T \subseteq S \times \mathcal{P}(S)$, let $\widehat{T} \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ denote the relation $\{(x + Y, Y) \mid x\ T\ Y\}$.

Lemma 9. *We have* $\widehat{\uparrow T} \subseteq c(\widehat{T})$.

Proof. If $(x + Y, Y) \in \widehat{\uparrow T}$, then there exists $Y' \subseteq Y$ such that $(x, Y') \in T$. By definition, $(x + Y', Y') \in \widehat{T}$ and $(Y, Y) \in c(\widehat{T})$. By the rule (u), $(x + Y' + Y, Y' + Y) \in c(\widehat{T})$ and since $Y' \subseteq Y$, $(x + Y, Y) \in c(\widehat{T})$. □

Proposition 5. *If T is a simulation up to $\uparrow$, then $\widehat{T}$ is a bisimulation up to c.*

Proof. First observe that if $T \rightarrowtail_s T'$, then $\widehat{T} \rightarrowtail u^\omega(\widehat{T'})$. Therefore, if $T \rightarrowtail_s \uparrow T$, then $\widehat{T} \rightarrowtail u^\omega(\widehat{\uparrow T})$. By Lemma 9, $\widehat{T} \rightarrowtail u^\omega(c(\widehat{T})) = c(\widehat{T})$. □

(Note that transitivity and symmetry are not used in the above proofs: the constructed bisimulation up to congruence is actually a bisimulation up to context $(r \cup u \cup id)^\omega$.)

The relation $\widehat{T}$ is not the one computed by HKCsince the former contains pairs of the shape $(x + Y, Y)$, while the latter has pairs of the shape $(X + Y, Y)$ with X possibly not a singleton. However, note that manipulating pairs of the two kinds does not change anything since by Lemma 7, $(X + Y, Y) \in c(R)$ iff for all $x \in X$, $(x + Y, Y) \in c(R)$.

4.2.2 Inclusion: AC can mimic HKC on disjoint automata

As shown in Section 4.2.3 below, HKC can be faster than AC, thanks to the up to transitivity technique. However, in the special case where the two automata are disjoint, transitivity cannot help, and the two algorithms actually match each other.

Suppose that the automaton (S, o, t) is built from two disjoint automata (S_1, o_1, t_1) and (S_2, o_2, t_2) as described in Remark 1. Let R be the relation obtained by running $\texttt{HKC}(X_0{+}Y_0, Y_0)$ with $X_0 \subseteq S_1$ and $Y_0 \subseteq S_2$. All pairs in R are necessarily of the shape $(X{+}Y,\ Y)$ with $X \subseteq S_1$ and $Y \subseteq S_2$. Let $\overline{R} \subseteq S \times \mathcal{P}(S)$ denote the relation $\{(x, Y) \mid \exists X,\ x \in X \text{ and } X{+}Y\ R\ Y\}$.

Lemma 10. *If S_1 and S_2 are disjoint, then $\overline{c(R)} \subseteq \uparrow(\overline{R})$.*

Proof. Suppose that $x\ \overline{c(R)}\ Y$, i.e., $x \in X$ with $X + Y\ c(R)\ Y$. By Lemma 7, we have $X \subseteq Y\!\downarrow_R$, and hence, $x \in Y\!\downarrow_R$. By definition of R the pairs it contains can only be used to rewrite from right to left; moreover, since S_1 and S_2 are disjoint, such rewriting steps cannot enable new rewriting rules, so that all steps can be performed in parallel: we have $Y\!\downarrow_R = \sum_{X'+Y' R Y' \subseteq Y} X'$. Therefore, there exist some X', Y' with $x \in X'$, $X'{+}Y'\ R\ Y'$, and $Y' \subseteq Y$. It follows that $(x, Y') \in \overline{R}$, hence $(x, Y) \in \uparrow(\overline{R})$. □

Proposition 6. *If S_1 and S_2 are disjoint, and if R is a bisimulation up to congruence, then $\overline{R}$ is a simulation up to upward closure.*

Proof. First observe that for all relations R, R', if $R \rightarrowtail R'$, then $\overline{R} \rightarrowtail_s \overline{R'}$. Therefore, if $R \rightarrowtail c(R)$, then $\overline{R} \rightarrowtail_s \overline{c(R)}$. We deduce $\overline{R} \rightarrowtail_s \uparrow(\overline{R})$ by Lemma 10. □

$$\begin{tikzcd}
x \arrow[loop left, "{a,b}"] \arrow[r, "a"] & x_1 \arrow[r, "{a,b}"] & \cdots \arrow[r, "{a,b}"] & \overline{x_n} \\
y \arrow[loop left, "{a,b}"] \arrow[r, "b"] & y_1 \arrow[r, "{a,b}"] & \cdots \arrow[r, "{a,b}"] & \overline{y_n} \\
z \arrow[loop left, "{a,b}"] \arrow[r, "{a,b}"] & z_1 \arrow[r, "{a,b}"] & \cdots \arrow[r, "{a,b}"] & \overline{z_n}
\end{tikzcd}$$

Figure 5. Family of examples where HKC exponentially improves over AC and HK; we have $x + y \sim z$.

4.2.3 Inclusion: AC cannot mimic HKC on merged automata

The containment of Lemma 10 does not hold when S_1 and S_2 are not disjoint since c can exploit transitivity while $\uparrow$ cannot. For a concrete grasp, take $R = \{(x + y, y), (y + z, z)\}$ and observe that $(x, z) \in \overline{c(R)}$ but $(x, z) \notin \uparrow(\overline{R})$. This difference makes it possible to find bisimulations up to c that are much smaller than the corresponding simulations up to $\uparrow$, and for HKC to be more efficient than AC. An example, where HKC is exponentially better than AC for checking language inclusion of automata sharing some states, is given in [7].

4.2.4 Language equivalence: AC cannot mimic HKC

AC can be used to check language equivalence, by checking the two underlying inclusions. However, checking equivalence directly can be better, even in the disjoint case. To see this on a simple example, consider the DFA on the right-hand side of Figure 2. If we use AC twice to prove $x \sim u$, we get the following antichains

$$T_1 = \{(x, u), (y, v), (y, w), (z, v), (z, w)\}$$
$$T_2 = \{(u, x), (v, y), (w, y), (v, z), (w, z)\}$$

containing five pairs each. Instead, four pairs are sufficient with HK or HKC, thanks to up to symmetry and up to transitivity.

For a more interesting example, consider the family of NFA given in Figure 5, where n is an arbitrary natural number. Taken together, the states x and y are equivalent to the state z: they recognise the language $(a{+}b)^\star(a{+}b)^{n+1}$. Alone, the state x (resp. y) recognises the language $(a{+}b)^\star a(a{+}b)^n$ (resp. $(a{+}b)^\star b(a{+}b)^n$).

For $i \leq n$, let $X_i = x{+}x_1{+}\ldots{+}x_i$, $Y_i = y{+}y_1{+}\ldots{+}y_i$, and $Z_i = z{+}z_1{+}\ldots{+}z_i$; for $N \subseteq [1..i]$, furthermore set

$$X_i^N = x + \sum_{j \in N} x_j\ , \qquad \overline{Y}_i^N = y + \sum_{j \in [1..n] \setminus N} y_j\ .$$

In the determinised NFA, $x + y$ can reach all the states of the shape $X_i^N{+}\overline{Y}_i^N$, for $i \leq n$ and $N \subseteq [1..i]$. For instance, for $n{=}i{=}2$, we have $x{+}y \overset{aa}{\to} x{+}y{+}x_1{+}x_2$, $x{+}y \overset{ab}{\to} x{+}y{+}y_1{+}x_2$, $x{+}y \overset{ba}{\to} x{+}y{+}x_1{+}y_2$, and $x{+}y \overset{bb}{\to} x{+}y{+}y_1{+}y_2$. Instead, z reaches only $n{+}1$ distinct states, those of the form Z_i.

The smallest bisimulation relating $x + y$ and z is

$$R = \{(X_i^N + \overline{Y}_i^N,\ Z_i) \mid i \leq n, N \subseteq [1..i]\},$$

which contains $2^{n+1}{-}1$ pairs. This is the relation computed by $\texttt{Naive}(x, y)$ and $\texttt{HK}(x, y)$—the up to equivalence technique (alone) does not help in HK. With AC, we obtain the antichains $T_x + T_y$ (for $[\![x + y]\!] \subseteq [\![z]\!]$) and T_z (for $[\![x + y]\!] \supseteq [\![z]\!]$), where:

$$T_x = \{(x_i,\ Z_i) \mid i \leq n\},$$
$$T_y = \{(y_i,\ Z_i) \mid i \leq n\},$$
$$T_z = \{(z_i,\ X_i^N + \overline{Y}_i^N) \mid i \leq n, N \subseteq [1..i]\}.$$

Note that T_x and T_y have size $n + 1$, and T_z has size $2^{n+1}{-}1$.

The language recognised by x or y are known for having a minimal DFA with 2^n states [19]. So, checking $x + y \sim z$ via minimisation (e.g., [17]) would also require exponential time.

This is not the case with HKC, which requires only polynomial time in this case. Indeed, $\mathtt{HKC}(x+y, z)$ builds the relation

$$\begin{aligned} R' = &\{(x + y,\ z)\} \\ &\cup \{(x + Y_i + y_{i+1},\ Z_{i+1}) \mid i < n\} \\ &\cup \{(x + Y_i + x_{i+1},\ Z_{i+1}) \mid i < n\} \end{aligned}$$

which is a bisimulation up to congruence and which only contains $2n + 1$ pairs. To see that this is a bisimulation up to congruence, consider the pair $(x{+}y{+}x_1{+}y_2,\ Z_2)$ obtained from $(x{+}y,\ z)$ after reading the word ba. This pair does not belong to R' but to its congruence closure. Indeed, we have

$$\begin{aligned} x{+}y{+}x_1{+}y_2 \ c(R')\ & Z_1{+}y_2 & (x{+}y{+}x_1\ R'\ Z_1) \\ c(R')\ & x{+}y{+}y_1{+}y_2 & (x{+}y{+}y_1\ R'\ Z_1) \\ c(R')\ & Z_2 & (x{+}y{+}y_1{+}y_2\ R'\ Z_2) \end{aligned}$$

(Check Lemma 18 in [6] for a complete proof.)

5. Exploiting Similarity

Looking at the example in Figure 5, a natural idea would be to first quotient the automaton by graph isomorphism. By doing so, we would merge the states x_i, y_i, z_i, and we would obtain the following automaton, for which checking $x{+}y \sim z$ is much easier.

$$\begin{tikzcd}
x \arrow[loop left, "{a,b}"] \arrow[dr, "a"] & & & \\
y \arrow[loop left, "{a,b}"] \arrow[r, "b"'] & m_1 \arrow[r, "{a,b}"] & \cdots \arrow[r, "{a,b}"] & \overline{m_n} \\
z \arrow[loop left, "{a,b}"] \arrow[ur, "{a,b}"'] & & &
\end{tikzcd}$$

As shown in [1, 12], one can actually do better with the antichain algorithm, by exploiting any preorder contained in language inclusion (e.g., similarity [25]). In this section, we rephrase this technique for antichains in our coinductive framework, and we show how this idea can be embedded in HKC, resulting in an even stronger algorithm.

5.1 AC with similarity: AC'

For the sake of clarity, we fix the preorder to be *similarity*, which can be computed in quadratic time [14] (or even less [16]):

Definition 9 (Similarity). Similarity *is the largest relation on states* $\preceq \subseteq S \times S$ *such that* $x \preceq y$ *entails:*

1. $o(x) \le o(y)$ *and*
2. *for all* $a \in A, x' \in S$ *such that* $x \xrightarrow{a} x'$*, there exists some* y' *such that* $y \xrightarrow{a} y'$ *and* $x' \preceq y'$*.*

One extends similarity to a preorder $\preceq^{\forall\exists} \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, and to a preorder $\sqsubseteq^{\preceq} \subseteq (S \times \mathcal{P}(S)) \times (S \times \mathcal{P}(S))$ on antichain pairs, as:

$$\begin{aligned} X \preceq^{\forall\exists} Y \quad &\text{if} \quad \forall x \in X, \exists y \in Y, x \preceq y \ , \\ (x', Y') \sqsubseteq^{\preceq} (x, Y) \quad &\text{if} \quad x \preceq x' \text{ and } Y' \preceq^{\forall\exists} Y \ . \end{aligned}$$

The new antichain algorithm [1], which we call AC', is similar to AC, but the antichain is now taken w.r.t. the new preorder $\sqsubseteq^{\preceq}$. Formally, let $\curlywedgeuparrow\colon \mathcal{P}(S \times \mathcal{P}(S)) \to \mathcal{P}(S \times \mathcal{P}(S))$ be the function defined for all relations $T \subseteq S \times \mathcal{P}(S)$ by $x \curlywedgeuparrow T\ Y$ if

$$x \preceq^{\forall\exists} Y \text{ or } (x', Y') \sqsubseteq^{\preceq} (x, Y) \text{ for some } (x', Y') \in T \ .$$

While AC consists in trying to build a simulation up to $\uparrow$, AC' tries to build a simulation up to $\curlywedgeuparrow$, i.e., it skips a pair (x, Y) if either (a) it is subsumed by another pair of the antichain or (b) $x \preceq^{\forall\exists} Y$.

Theorem 5. *Any simulation up to* $\curlywedgeuparrow$ *is contained in a simulation.*

Corollary 4. *The antichain algorithm proposed in [1] is sound and complete: for all sets* X, Y*,* $[\![X]\!] \subseteq [\![Y]\!]$ *iff* AC'(X, Y)*.*

Optimisation 1(a) and optimisation 1(b) in [1] are simply (a) and (b), as discussed above. Another optimisation, called Optimisation 2, is presented in [1]: if $y_1 \preceq y_2$ and $y_1, y_2 \in Y$ for some pair (x, Y), then y_1 can be safely removed from Y. Note that while this is useful to store smaller sets, it does not allow one to explore less since the pairs encountered with or without Optimisation 2 are always equivalent w.r.t. the ordering $\sqsubseteq^{\preceq}$: $Y \preceq^{\forall\exists} Y \setminus y_1$ and, for all $a \in A$, $t^\sharp_a(Y) \preceq^{\forall\exists} t^\sharp_a(Y \setminus y_1)$.

5.2 HKC with similarity: HKC'

Although HKC is primarily designed to check language equivalence, we can also extend it to exploit the similarity preorder. It suffices to notice that for any similarity pair $x \preceq y$, we have $x{+}y \sim y$.

Let $\overline{\preceq}$ denote the relation $\{(x{+}y,\ y) \mid x \preceq y\}$, let r' denote the constant function to $\overline{\preceq}$, and let $c' = (r' {\cup} s {\cup} t {\cup} u {\cup} id)^\omega$. Accordingly, we call HKC' the algorithm obtained from HKC (Figure 4) by replacing $(X, Y) \in c(R \cup \mathit{todo})$ with $(X, Y) \in c'(R \cup \mathit{todo})$ in step 3.2. Notice that the latter test can be reduced to rewriting thanks to Theorem 3 and the following lemma.

Lemma 11. *For all relations R,* $c'(R) = c(R \cup \overline{\preceq})$*.*

In other words to check whether $(X, Y) \in c'(R \cup \mathit{todo})$, it suffices to compute the normal forms of X and Y w.r.t. the rules from $R \cup \mathit{todo}$ plus the rules $x + y \leftarrow y$ for all $x \preceq y$.

Theorem 6. *Any bisimulation up to* c' *is contained in a bisimulation.*

Proof. Consider the constant function $r''\colon \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S)) \to \mathcal{P}(\mathcal{P}(S) {\times} \mathcal{P}(S))$ mapping all relations to $\sim$. Since language equivalence ($\sim$) is a bisimulation, we immediately obtain that this function is compatible. Thus so is the function $c'' = (r'' {\cup} s {\cup} t {\cup} u {\cup} id)^\omega$. We have that $\overline{\preceq}$ is contained in $\sim$, so that any bisimulation up to c' is a bisimulation up to c''. Since c'' is compatible, such a relation is contained in a bisimulation, by Proposition 3. □

Note that in the above proof, we can replace $\overline{\preceq}$ by any other relation contained in $\sim$. Intuitively, bisimulations up to c'' correspond to classical *bisimulations up to bisimilarity* [25] from concurrency.

Corollary 5. *For all sets* X, Y*, we have* $X \sim Y$ *iff* HKC'(X, Y)*.*

5.3 Relationship between HKC' and AC'

Like in Section 4.2.1, we can show that for any simulation up to $\curlywedgeuparrow$ there exists a corresponding bisimulation up to c', of the same size.

Lemma 12. *For all relations* $T \subseteq S \times \mathcal{P}(S)$*,* $\widehat{\curlywedgeuparrow T} \subseteq c'(\widehat{T})$*.*

Proposition 7. *If* T *is a simulation up to* $\curlywedgeuparrow$*, then* $\widehat{T}$ *is a bisimulation up to* c'*.*

However, even for checking inclusion of disjoint automata, AC' cannot mimic HKC', because now the similarity relation allows one to exploit transitivity. To see this, consider the example given in Figure 6, where we want to check that $[\![z]\!] \subseteq [\![x + y]\!]$, and for which the similarity relation is shown on the right-hand side.

Since this is an inclusion of disjoint automata, HKC and AC, which do not exploit similarity, behave the same (cf. Sections 4.2.1 and 4.2.2). Actually, they also behave like HK and they require $2^{n+1}{-}1$ pairs. On the contrary, the use of similarity allows HKC' to prove the inclusion with only $2n + 1$ pairs, by computing the

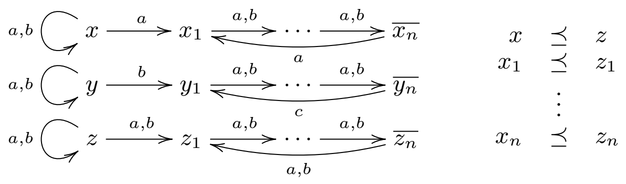

Figure 6. Family of examples where `HKC'` exponentially improves over `AC'`, for inclusion of disjoint automata: we have $[\![z]\!] \subseteq [\![x+y]\!]$.

following bisimulation up to c' (Lemma 19 in [6]):

$$\begin{aligned} R'' = & \{(z+x+y,\ x+y)\} \\ & \cup \{(Z_{i+1}+X_i+y+y_{i+1},\ X_i+y+y_{i+1}) \mid i < n\} \\ & \cup \{(Z_{i+1}+X_{i+1}+y,\ X_{i+1}+y) \mid i < n\}\ , \end{aligned}$$

where $X_i = x+x_1+\ldots+x_i$ and $Z_i = z+z_1+\ldots+z_i$.

Like in Section 4.2.4, to see that this is a bisimulation up to c' (where we do exploit similarity), consider the pair obtained after reading the word ab: $(Z_2+x+y+x_2+y_1,\ x+y+x_2+y_1)$. This pair does not belong to R'' or $c(R'')$, but it does belong to $c'(R'')$. Indeed, by Lemmas 7 and 11, this pair belong to $c'(R'')$ iff $Z_2 \subseteq (x+y+x_2+y_1){\downarrow}_{R''\cup\preceq}$, and we have

$$\begin{array}{lll} & x+y+x_2+y_1 & \\ \rightsquigarrow_{R''\cup\preceq} & Z_1+x+y+y_1+x_2 & (Z_1+x+y+y_1\ R''\ x+y+y_1) \\ \rightsquigarrow_{R''\cup\preceq} & Z_1+X_1+y+y_1+x_2 = Z_1+X_2+y+y_1 & (x_1 \preceq z_1) \\ \rightsquigarrow_{R''\cup\preceq} & Z_2+X_2+y+y_1+x_2 & (Z_2+X_2+y\ R''\ X_2+y) \end{array}$$

On the contrary, `AC'` is not able to exploit similarity in this case, and it behaves like `AC`: both of them compute the same antichain T_z as in the example from Section 4.2.4, which has $2^{n+1}-1$ elements.

In fact, even when considering inclusion of disjoint automata, the use of similarity tends to virtually merge states, so that `HKC'` can use the up to transitivity technique which `AC` and `AC'` lack.

5.4 A short recap

Figure 7 summarises the relationship amongst the presented algorithms, in the general case and in the special case of language inclusion of disjoint automata. In this diagram, an arrow `X`$\rightarrow$`Y` (from an algorithm `X` to `Y`) means that (a) `Y` can explore less states than `X`, and (b) `Y` can mimic `X`, i.e., the proof technique of `Y` is at least as powerful as the one of `X`. (The labels on the arrows point to the sections showing these relations; unlabelled arrows are not illustrated in this paper, they are easily inferred from what we have shown.)

6. Experimental assessment

To get an intuition of the average behaviour of `HKC` on various NFA, and to compare it with `HK` and `AC`, we provide some benchmarks on random automata and on automata obtained from model-checking problems. In both cases, we conduct the experiments on a MacBook pro 2.4GHz Intel Core i7, with 4GB of memory, running OS X Lion (10.7.4). We use our OCaml implementation [7] for `HK`, `HKC`, and `HKC'`, while we use the `libvata` C++ library [21] for `AC` and `AC'`.[1] (To our knowledge, `libvata` is the most efficient implementation currently available for these algorithms, even though this library was designed for tree automata rather than plain NFA.)

6.1 Random automata

For a given size n, we generate a thousand random NFA with n states and two letters. According to [30], we use a linear transi-

[1] More precisely, the upward algorithms provided by default in `libvata`.

General case

$$\begin{tikzcd} & & \texttt{HKC'} & \\ & \texttt{HKC} \arrow[ur, "(5)"] & & \texttt{AC'} \arrow[ul, "(5.3)"'] \\ \texttt{HK} \arrow[ur, "(3)"] & & \texttt{AC} \arrow[ul, "(4.2)"'] \arrow[ur, "(5)"] & \\ & \texttt{Naive} \arrow[ul, "(2.4)"] \arrow[ur] & & \end{tikzcd}$$

Disjoint inclusion case

$$\begin{tikzcd} \texttt{HKC'} \\ \texttt{AC'} \arrow[u, "(5.3)"'] \\ \texttt{HKC} \leftrightarrow \texttt{AC}\ (4.2.2) \arrow[u] \\ \texttt{HK} \leftrightarrow \texttt{Naive} \arrow[u] \end{tikzcd}$$

Figure 7. Relationship between the various algorithms.

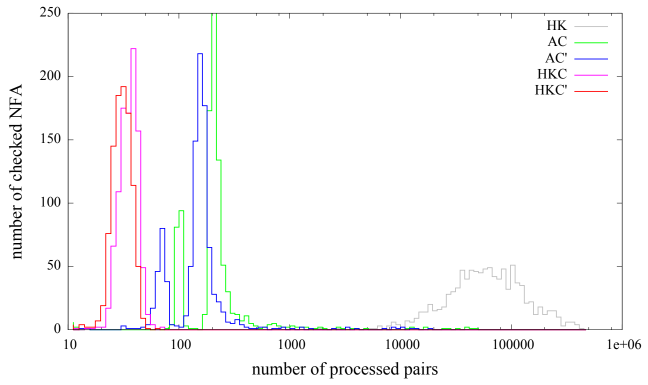

Figure 8. Distributions of the number of processed pairs, for the 1000 NFA with 100 states and 2 letters from Table 1.

tion density of 1.25 (which means that the expected out-degree of each state and with respect to each letter is 1.25): Tabakov and Vardi empirically showed that one statistically gets more challenging NFA with this particular value. We generate NFA without accepting states: by doing so, we make sure that the algorithms never encounter a counter-example, so that they always continue until they find a (bi)simulation up to: these runs correspond to their worst cases for all possible choices of accepting states for the given NFA.[2]

We run all algorithms on these NFA, starting from two distinct singleton sets, to measure the required time and the number of processed pairs: for `HK`, `HKC`, and `HKC'`, this is the number of pairs put into the bisimulation up to (R); for `AC` and `AC'`, this is the number of pairs inserted into the antichain. The timings for `HKC'` and `AC'` do not include the time required to compute similarity.

We report the median values (50%), the last deciles (90%), the last percentiles (99%), and the maximum values (100%) in Table 1. For instance, for $n = 70$, 90% of the examples require less than 155ms with `HK`; equivalently, 10% of the examples require more than 155ms. (For a few tests, `libvata` ran out of memory, whence the ∞ symbols in the table.) We also plotted on Figure 8 the distribution of the number of processed pairs when $n = 100$.

`HKC` and `AC` are several orders of magnitude better than `HK`, and `HKC` is usually two to ten times faster than `AC`. Moreover, for the first four lines, `HKC` is much more predictable than `AC`, i.e., the last percentiles and maximal values are of the same order as the median value. (`AC` seems to become more predictable for larger values of n.) The same relative behaviour can be observed between `HKC'` and `AC'`; moreover, `HKC` alone is apparently faster than `AC'`.

Also recall that the size of the relations generated by `HK` is a lower bound for the number of accessible states of the determinised

[2] To get this behaviour for `AC` and `AC'`, we actually had to trick `libvata`, which otherwise starts by removing non-coaccessible states, and thus reduces any of these NFA to the empty one.

$n = \|S\|$	algo.	required time (seconds)				number of processed pairs				mDFA size
		50%	90%	99%	100%	50%	90%	99%	100%	50%
50	HK	0.007	0.022	0.050	0.119	2511	6299	12506	25272	$\sim$1000
	AC	0.002	0.003	0.142	1.083	112	245	2130	5208	
	HKC	0.000	0.000	0.000	0.000	21	26	32	63	
	AC’	0.002	0.002	0.038	0.211	79	131	1098	1926	
	HKC’	0.000	0.000	0.000	0.000	18	23	28	58	
70	HK	0.047	0.155	0.413	0.740	10479	28186	58782	87055	$\sim$6000
	AC	0.002	0.003	1.492	4.163	150	285	8383	15575	
	HKC	0.000	0.000	0.000	0.000	27	34	40	49	
	AC’	0.002	0.003	0.320	0.884	110	172	3017	6096	
	HKC’	0.000	0.000	0.000	0.000	23	29	36	44	
100	HK	0.373	1.207	3.435	5.660	58454	164857	361227	471727	$\sim$30000
	AC	0.003	0.004	3.214	36.990	204	298	13801	48059	
	HKC	0.000	0.000	0.000	0.001	36	44	54	70	
	AC’	0.003	0.004	0.738	6.966	152	211	4087	18455	
	HKC’	0.000	0.000	0.000	0.001	31	39	46	64	
300	AC	0.009	0.010	0.028	0.750	562	622	2232	14655	–
	HKC	0.001	0.002	0.003	0.009	86	104	118	132	
	AC’	0.012	0.013	0.022	0.970	433	484	920	14160	
	HKC’	0.001	0.001	0.002	0.006	76	91	104	116	
500	AC	0.014	0.015	0.039	∞	918	986	2571	∞	–
	HKC	0.002	0.005	0.008	0.018	130	154	176	193	
	AC’	0.025	0.028	0.042	∞	710	772	1182	∞	
	HKC’	0.002	0.004	0.007	0.013	115	136	154	169	
1000	AC	0.029	0.031	0.038	∞	1808	1878	2282	∞	–
	HKC	0.007	0.022	0.055	0.093	228	271	304	337	
	AC’	0.074	0.080	0.092	∞	1409	1488	1647	∞	
	HKC’	0.008	0.019	0.041	0.077	202	238	265	299	

Table 1. Running the five presented algorithms to check language equivalence on random NFA with two letters.

NFA (Lemma 6 (2)); one can thus see in Table 1 that HKC usually explores an extremely small portion of these DFA (e.g., less than one per thousand for $n = 100$). The last column reports the median size of the minimal DFA for the corresponding parameters, as given in [30]. HK usually explores much more states than what would be necessary with a minimal DFA, while HKC and AC need much less.

6.2 Automata from regular model-checking

Checking language inclusion of NFA can be useful for model-checking, where one sometimes has to compute a sequence of NFA by iteratively applying a transducer, until a fixpoint is reached [8]. To know that the fixpoint is reached, one typically has to check whether an NFA is contained in another one.

Abdulla et al. [1] use such benchmarks to test their algorithm (AC’) against the plain antichain algorithm (AC [31]). We reuse them to test HKC’ against AC’ in a concrete scenario. We take the sequences of automata kindly provided by L. Holik, which come from those used in [1] and arise from the model checking of various programs (the bakery algorithm, bubble sort, and a producer-consumer system). For all these sequences, we check the inclusions of consecutive pairs, in both directions. We separate the results into those for which a counter-example is found, and those for which the inclusion holds.

The results are given in Table 2. As expected, HKC and AC roughly behave the same: we have inclusions of disjoint automata. HKC’ is however quite faster than AC’: up to transitivity can be exploited thanks to similarity pairs. Like in Table 1, the timings do not include the time required to compute similarity, which is given separately (computed using libvata). Also note that over the 546 positive answers, 368 are obtained immediately by similarity.

7. Related work

A similar notion of bisimulation up to congruence has already been used to obtain decidability and complexity results about context-free processes, under the name of *self-bisimulations*. Caucal [10] introduced this concept to give a shorter and nicer proof of the result by Baeten et al. [4]: bisimilarity is decidable for normed context-free processes. Christensen et al [11] then generalised the result to all context-free processes, also by using self-bisimulations. Hirshfeld et al. [15] used a refinement of this notion to get a polynomial algorithm for bisimilarity in the normed case.

There are two main differences with the ideas we presented here. First, the above papers focus on bisimilarity rather than language equivalence (recall that although we use bisimulation relations, we check language equivalence since we work on the determinised NFA—Remark 3). Second, we consider a notion of bisimulation up to congruence where the congruence is taken with respect to non-determinism (union of sets of states). Self-bisimulations are also bisimulations up to congruence, but the congruence is taken with respect to word concatenation. We cannot consider this operation in our setting since we do not have the corresponding monoid structure in plain NFA.

Other approaches, that are independent from the algebraic structure (e.g., monoids or semi-lattices) and the behavioural equivalence (e.g., bisimilarity or language equivalence) are shown in [5, 22, 23, 26]. These propose very general frameworks into which our up to congruence technique fits as a very special case. To our knowledge, bisimulation up to congruence has never been proposed as a technique for proving language equivalence of NFA.

8. Conclusions and future work

We showed that the standard algorithm by Hopcroft and Karp for checking language equivalence of DFA relies on a bisimulation up to equivalence proof technique; this allowed us to design a new algorithm (HKC) for the non-deterministic case, where we exploit a novel technique called up to congruence.

We then compared HKC to the recently introduced antichain algorithms [31] (AC): when checking the inclusion of disjoint automata, the two algorithms are equivalent, in all the other cases

algorithm	inclusions (546 pairs)				counter-examples (518 pairs)			
	50%	90%	99%	100%	50%	90%	99%	100%
AC	0.036	0.860	4.981	5.084	0.009	0.094	1.412	2.887
HKC	0.049	0.798	6.494	6.762	0.000	0.014	0.916	2.685
AC'	0.013	0.167	1.326	1.480	0.012	0.107	1.047	1.134
HKC'	0.000	0.034	0.224	0.345	0.001	0.005	0.025	0.383
sim_time	0.039	0.185	0.574	0.618	0.038	0.193	0.577	0.593

Table 2. Timings, in seconds, for language inclusion of disjoint NFA generated from model-checking.

HKC is more efficient since it can use transitivity to prune a larger portion of the state-space.

The difference between these two approaches becomes even more striking when considering some optimisation exploiting similarity. Indeed, as nicely shown with AC' [1], the antichains approach can widely benefit from the knowledge one gets by first computing similarity. Inspired by this work, we showed that both our proof technique (bisimulation up to congruence) and our algorithm (HKC) can be easily modified to exploit similarity. The resulting algorithm (HKC') is now more efficient than AC' even for checking language inclusion of disjoint automata.

We provided concrete examples where HKC and HKC' are exponentially faster than AC and AC' (Sections 4.2.4 and 5.3) and we proved that the coinductive techniques underlying the formers are at least as powerful as those exploited by the latters (Propositions 5 and 7). We finally compared the algorithms experimentally, by running them on both randomly generated automata, and automata resulting from model checking problems. It appears that for these examples, HKC and HKC' perform better than AC and AC'.

Our implementation of the presented algorithms is available online [7], together with Coq proof scripts, and with an applet making it possible to run the algorithms on user-provided examples.

As future work, we plan to extend our approach to tree automata. In particular, it seems promising to investigate if further up-to techniques can be defined for regular tree expressions. For instance, the algorithms proposed in [3, 20] exploit some optimisation which suggest us coinductive up-to techniques.

Acknowledgments

We are grateful to M. Bonsangue, J. Rutten, and A. Silva for the stimulating discussions that eventually led to this work; and to L. Doyen, P. Habermehl, L. Holík and B. Koenig for their help with antichain algorithms.

References

[1] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *Proc. TACAS*, vol. 6015 of *LNCS*, pages 158–174. Springer, 2010.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In *FPCA*, vol. 523 of *LNCS*, pages 427–447. Springer, 1991.

[4] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. In *Proc. PARLE (II)*, vol. 259 of *LNCS*, pages 94–111. Springer, 1987.

[5] F. Bartels. *On generalized coinduction and probabilistic specification formats*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

[6] F. Bonchi and D. Pous. Extended version of this abstract, with omitted proofs. http://hal.inria.fr/hal-00639716/, 2012.

[7] F. Bonchi and D. Pous. Web appendix for this paper. http://perso.ens-lyon.fr/damien.pous/hknt, 2012.

[8] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. CAV*, vol. 3114 of *LNCS*. Springer, 2004.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[10] D. Caucal. Graphes canoniques de graphes algébriques. *ITA*, 24:339–352, 1990.

[11] S. Christensen, H. Hüttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Information and Computation*, 121(2):143–148, 1995.

[12] L. Doyen and J.-F. Raskin. Antichain Algorithms for Finite Automata. In *Proc. TACAS*, vol. 6015 of *LNCS*. Springer, 2010.

[13] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1 (2/3):251–273, 1992.

[14] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. FOCS*, pages 453–462. IEEE Computer Society, 1995.

[15] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science*, 158(1&2):143–159, 1996.

[16] L. Holík and J. Šimáček. Optimizing an LTS-Simulation Algorithm. *Computing and Informatics*, 2010(7):1337–1348, 2010.

[17] J. E. Hopcroft. An n log n algorithm for minimizing in a finite automaton. In *Proc. International Symposium of Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[18] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. TR 114, Cornell Univ., December 1971.

[19] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[20] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[21] O. Lengál, J. Simácek, and T. Vojnar. Vata: A library for efficient manipulation of non-deterministic tree automata. In *TACAS*, vol. 7214 of *LNCS*, pages 79–94. Springer, 2012.

[22] M. Lenisa. From set-theoretic coinduction to coalgebraic coinduction: some results, some problems. *ENTCS*, 19:2–22, 1999.

[23] D. Lucanu and G. Rosu. Circular coinduction with special contexts. In *Proc. ICFEM*, vol. 5885 of *LNCS*, pages 639–659. Springer, 2009.

[24] A. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proc. STOC*, pages 1–9. ACM, 1973.

[25] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[26] D. Pous. Complete lattices and up-to techniques. In *Proc. APLAS*, vol. 4807 of *LNCS*, pages 351–366. Springer, 2007.

[27] J. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proc. CONCUR*, vol. 1466 of *LNCS*, pages 194–218. Springer, 1998.

[28] D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8:447–479, 1998.

[29] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[30] D. Tabakov and M. Vardi. Experimental evaluation of classical automata constructions. In *Proc. LPAR*, vol. 3835 of *LNCS*, pages 396–411. Springer, 2005.

[31] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. CAV*, vol. 4144 of *LNCS*, pages 17–30. Springer, 2006.