Checking NFA equivalence with bisimulations up to congruence

Filippo Bonchi, Damien Pous

Checking NFA equivalence with bisimulations up to congruence

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

Damien Pous
CNRS, Université de Grenoble, LIG (UMR 5217)

***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 [13] that, instead of computing the whole determinised automata, explores only a small portion of it. Although the optimised algorithm remains exponential in worst case (the problem is PSPACE-complete), experimental results show improvements of several orders of magnitude over the standard algorithm.**

I. Introduction

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

Equivalence of deterministic finite automata (DFA) can be checked either via minimisation [12], [7] or through Hopcroft and Karp's algorithm [13], [2], which exploits an instance of what is nowadays called a *coinduction proof principle* [19], [25], [23]: 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[1], while Hopcroft and Karp's algorithm can be executed "on-the-fly" [10], on a lazy DFA whose transitions are computed on demand.

This difference is fundamental for our work and for other recently introduced algorithms based on *antichains* [30], [1]. 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 [30], [1]) 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 (otherwise we would have solved in polynomial time the problem of language equivalence, which is PSPACE-complete [18]).

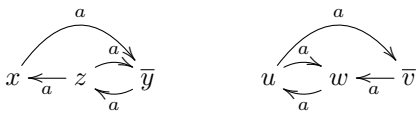
[1]There are few exceptions, like [15] which minimises labelled transition systems w.r.t. bisimilarity rather than trace equivalence.

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:

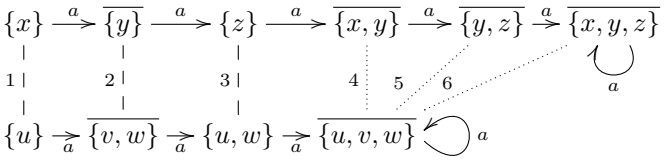
$$[\![X + Y]\!] = [\![X]\!] + [\![Y]\!] \tag{1}$$

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 better illustrate this idea, consider the following example, where we check the equivalence of the states x and u from the NFA depicted below. (Final states are overlined, labelled edges represent transitions.)



The determinised automaton is depicted below.



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 the standard Hopcroft and Karp's algorithm (the numbers express the order in which pairs are added).

The dashed lines (numbered by 1, 2 and 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 [19], [20], [24] 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 the up to congruence technique, 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 of the DFA. 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.

Summarising, the contributions of this work are

(1) the observation that Hopcroft and Karp implicitly use bisimulations up to equivalence for DFA (Section II),
(2) a sound and complete proof technique for proving language equivalence of NFA (Section III-B), and
(3) an efficient algorithm for checking language equivalence of NFA (Sections III-C and III-D).

Outline

The rest of the paper is structured as follows. Section II recalls Hopcroft and Karp's algorithm for DFA, showing that it implicitly exploits bisimulation up to equivalence. Section III describes the novel algorithm, based on bisimulations up to congruence. Section IV discusses its complexity as well as experimental data showing that the algorithm usually performs much better than the other known algorithms. Sections V and VI provide related and future work. Omitted proofs can be found in the appendix.

Notation

We denote sets by capital letters $X, Y, S, T \ldots$ and functions by lower case letters $f, g, \ldots$ Given sets X and Y, $X \times Y$ is the Cartesian product of X and Y, $X \uplus Y$ 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 $f \colon \mathcal{P}(X) \to \mathcal{P}(X)$ on a powerset 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 $w_1 \cdot w_2$ (and $w_1 w_2$) the concatenation of words $w_1, w_2 \in A^\star$. We use 2 to denote the set $\{0, 1\}$ and $2^{A^\star}$ to denote the set of all formal languages over A.

II. Hopcroft and Karp's algorithm for DFA

A deterministic finite automaton (DFA) 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, 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 input letter $a \in A$, the next state $t_a(x)$. For $a \in A$, we will write $x \xrightarrow{a} x'$ to mean that $t_a(x) = x'$. For $w \in A^\star$, we will 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:

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

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 they 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.*

A. 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.*

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') ∈ R then skip;
 (3.3)  if o(x') ≠ o(y') then return false;
 (3.4)  for all a ∈ 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 HK(x, y) is obtained by replacing step 3.2 with `if` $(x', y') \in e(R)$ `then skip`.

B. 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*.

For the time being, we avoid discussing which data structures are convenient for implementing R and *todo* (as well as any complexity issue), and we focus on correctness.

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

Proof: We first observe that if 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, we deduce $x \sim y$.

We now prove that if 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 III are simple variations of this naive

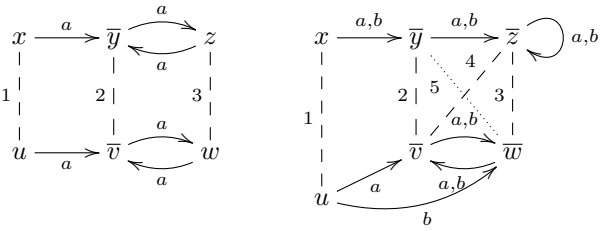


Figure 2. Checking for DFA equivalence.

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,* Naive *constructs the* smallest *bisimulation that relates the two starting states (see Proposition 4 in Appendix A). On the contrary, minimisation algorithms [12], [7] are designed to compute the* largest *bisimulation relation for a given automaton. For instance, taking the left-hand side automaton from Figure 2, they would equate the states x and w which are language equivalent, while* Naive(x, u) *does not relate them.*

C. 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') ∈ e(R) then skip;
```

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 `HK`.

D. 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 $x\ R\ y$ then*

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

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 [24], [22]:

Definition 3 (Compatible function)**.** *A function $f\colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ is* compatible *if 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 III.

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: $r(R) = \{(x, x) \mid \forall x \in S\}$*
- *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:

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

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* `HK`(x, y).

Proof: Same proof as for Proposition 2, by using the invariant $R \rightarrowtail e(R) \cup todo$ for the loop. 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

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

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_{HK} but in its equivalence closure, $e(R_{HK})$.

III. 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:

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

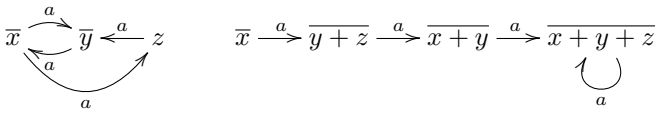
$$t^\sharp_a(X) = \begin{cases} t_a(x) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ t^\sharp_a(X_1) + t^\sharp_a(X_2) & \text{if } X = X_1 + X_2 \end{cases}$$

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 $x_1 + \dots + x_n$. As usual, 0 is the empty set.

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).



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 $t^\sharp_a(y + z) = t^\sharp_a(y) + t^\sharp_a(z) = t_a(y) + t_a(z) = x + y$.

The language accepted by the states of a 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.

A. Extending coinduction to NFA

In order to check if 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 a 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 $a \in A$, $t^\sharp_a(X) \; R \; t^\sharp_a(Y)$.

Since this is just the old definition of bisimulation (Definition 1) applied to $(\mathcal{P}(S), o^\sharp, t^\sharp)$, it is immediate to see 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 [19] (which* strictly *imply language equivalence): in a standard bisimulation R, if the following states x and y of an NFA are in R,*

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

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

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

B. 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 a 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

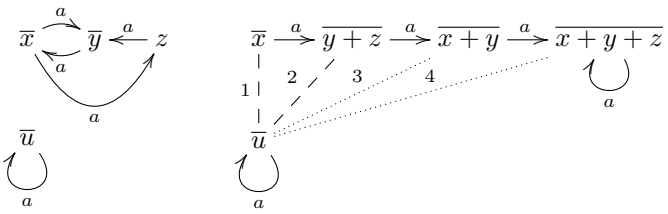


Figure 3. Bisimulations up to congruence, on a single letter NFA.

deduce $o^\sharp(X_1 + X_2) = o^\sharp(Y_1 + Y_2)$, and for all $a \in A$, $t_a^\sharp(X_1 + X_2)\ u(R')\ t_a^\sharp(Y_1 + Y_2)$. ∎

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 to use just the function $(u \cup \mathrm{id})^\omega$, without closing under r, s and t).

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 $(x+y,u) \in c(R)$:

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

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:

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

C. Optimised algorithm for NFA

Algorithms for NFA can be obtained by computing the determinised NFA on-the-fly [10]: starting from the algorithms for DFA (Figure 1), it suffices to work with sets of states, and to inline the powerset construction. Their code is given in Figure 4. The naive algorithm (`Naive`) does not use any up to technique, while Hopcroft and Karp's algorithm (`HK`) reasons up to equivalence in step `3.2`.

The optimised algorithm, referred as `HKC` in the sequel, relies on up to congruence: step `3.2` becomes

`(3.2)` `if` $(X',Y') \in c(R \cup \mathit{todo})$ `then skip;`

`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') ∈ R then skip;
 (3.3)  if o♯(X') ≠ o♯(Y') then return false;
 (3.4)  for all a ∈ A,
          insert (t_a♯(X'), t_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 a NFA (S,o,t). The code for on-the-fly `HK`(X,Y) is obtained by replacing the test in step `3.2` with $(X',Y') \in e(R)$; the code for `HKC`(X,Y) is obtained by replacing this test with $(X',Y') \in c(R \cup \mathit{todo})$.

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

Proof: Same proof as for Proposition 2, by using the invariant $R \rightarrowtail c(R \cup \mathit{todo})$ 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, the situation 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 IV). 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.

D. 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 rewriting modulo ACI; the key idea is to look at each pair (X, Y) in a relation R as a pair of rewriting rules:

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

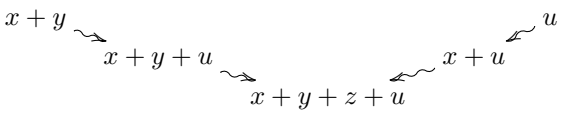
which can be used to compute normal forms for sets of states. Indeed, by idempotence, $X\ R\ Y$ entails $X\ c(R)\ X + Y$.

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 relation 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 $\{(x, u), (y + z, u)\}$):



Theorem 3. *For all relation 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, in order to check if $(X, Y) \in c(R \cup todo)$ 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 [6]. We have chosen to keep the presented rewriting algorithm for its simplicity and because it behaves pretty well in practice; we leave for future work the study of more elaborated solutions.

E. Universality and language inclusion

As a special case, any algorithm for testing language equality of NFA can be used to test NFA for *universality* (i.e., whether the automaton accepts all words) or language inclusion. Indeed, for universality, it suffices to compare the automaton with the trivial NFA with one accepting state (see, e.g., the example in Figure 3); for language inclusion, we can use the fact that the function $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^\star}$ is a semi-lattice homomorphism (see Appendix A), so that given two starting sets of states X and Y, we have $[\![X]\!] \subseteq [\![Y]\!]$ iff $[\![X]\!] + [\![Y]\!] = [\![Y]\!]$ iff $[\![X + Y]\!] = [\![Y]\!]$.

In the special case of universality, `HKC` actually coincides with the antichain algorithm proposed in [30] (except for the concrete data structures). This algorithm proceeds as follows. It explores the determinised NFA to check that all reachable sets contain at least one accepting state, and the key idea is to stop whenever it encounters a set which contains one of the previously visited sets: universality of the larger is subsumed by universality of the smaller set, and like with `HKC`, this allows one to skip rather large portions of the determinised NFA. To see that it coincides with `HKC`, consider a run of $\texttt{HKC}(X_0, u)$, where u is the unique state of the full and trivial automaton, and X_0 is the set of initial states of the NFA to check for universality. At each iteration, $R \cup todo$ is of the form $\{(X_i, u) \mid i \in I\}$, so that $u{\downarrow} = u + \sum_{i \in I} X_i$, and for a given set X (disjoint from u),

$$X{\downarrow} = \begin{cases} u{\downarrow} & \text{if there is a set } X_i \subseteq X \\ X & \text{otherwise} \end{cases}$$

In other words, a pair (X, u) is skipped iff X contains one of the previously encountered sets.

The authors of [30] also propose a variation of their universality algorithm to check language inclusion of NFA, without exploring the whole determinised NFA. In contrast with the case of universality, this algorithm does not coincide with the one we propose here.

IV. 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_{Naive}, R_{HK}, and R_{HKC} denote the relations produced by the three algorithms. We have*

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

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

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

As shown below, R_{HKC} can actually be exponentially smaller than R_{HK}.

A. Exponential speed-up

Consider the family of NFA given in Figure 5, where n is an arbitrary natural number. The states x and y are equivalent: they both recognise the language $(a + b)^\star \cdot b \cdot (a + b)^n$.

In the determinised NFA, x can reach all the states of the shape $x + \sum_{i \in N} x_i$ where $N \subseteq [1..n]$. For instance, for $n = 2$,

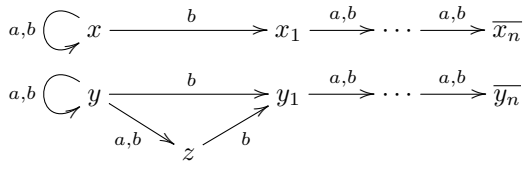


Figure 5. Family of examples where HKC exponentially improves over HK.

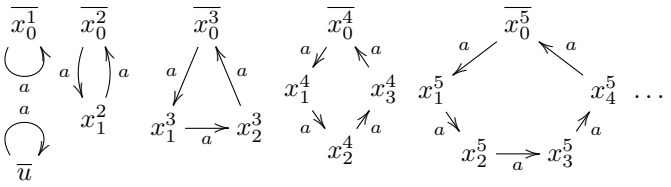


Figure 6. Single letter example for which HKC requires exponential time.

$x \xrightarrow{aa} x$, $x \xrightarrow{ab} x + x_1$, $x \xrightarrow{ba} x + x_2$ and $x \xrightarrow{bb} x + x_1 + x_2$; analogously y can reach all the states of the shape $y + z + \sum_{i\in N} y_i$. The smallest bisimulation relating x and y is

$$R = \{(x, y)\} \cup \{(x + \sum_{i\in N} x_i,\ y + z + \sum_{i\in N} y_i) \mid N \subseteq [1..n]\},$$

which contains $2^n + 1$ pairs. This is the relation computed by $\texttt{Naive}(x, y)$ and $\texttt{HK}(x, y)$ (here, the up to equivalence technique does not help). Instead, $\texttt{HKC}(x, y)$ builds the relation

$$R' = \{(x, y), (x, y+z)\} \cup \{(x + x_i,\ y + z + y_i) \mid i \in [1..n]\},$$

which is a bisimulation up to congruence (check Lemma 10 in Appendix D for a formal proof) and which only contains $n+2$ pairs. It is worth to note that R' is like a "basis" of R: all the pairs $(x', y') \in R$ can be generated from R' by iteratively applying the function u (Definition 5).

Also notice that the language recognised by x and y is known for having a minimal DFA with 2^n states [14]. Therefore, checking their equivalence by using a minimisation algorithm (e.g., [12], [7]) would also require exponential time.

The antichain algorithm from [30] also solves this example in polynomial time and its improved version from [1] which first computes *similarity* (in the branching-time sense) would answer immediately: x simulates y and y simulates x. However, it suffices to modify the branching structure of the x_i and y_i to disable this behaviour.

B. Case requiring exponential time

Even though the previous example shows that the optimised algorithm can be polynomial where HK is exponential, the problem of checking language equivalence of NFA is PSPACE-complete and NP-complete when restricted to the one-letter case [18]. We now show an example on the single-letter alphabet $A = \{a\}$ where HKC requires exponential time.

The example is shown in Figure 6; given a natural number n, we compare u with $x_0^1 + \cdots + x_0^n$ (hereafter denoted by X_0). It is immediate to see that u accepts the universal language $a^\star$, while each x_0^i accepts $(a^i)^\star$. Checking that $X_0 \sim u$ thus amounts to the following (trivial) equality of regular expressions: $\sum_{i=1}^{n} (a^i)^\star = a^\star$.

For any natural number k, let X_k be the set of states

$$X_k = \sum_{i=1}^{n} x_{k \bmod i}^i \ .$$

For all k, we have $X_k \xrightarrow{a} X_{k+1}$, and this sequence is periodic, of period $p = \mathrm{lcm}[1..n]$, the least common multiplier of the first n natural numbers (that is greater than 2^n for $n > 7$ [21]).

By running HK between X_0 and u, we get a bisimulation up to equivalence in p steps (this relation is actually a bisimulation, the up-to equivalence technique is not used). We can show that the optimised algorithm behaves exactly the same: the up-to congruence technique does not help. Intuitively, at the j-th iteration, we have

$$R = \{(X_k,\ u) \mid k < j\} \ ;$$

therefore if $j < p$ then (X_j, u) does not belong to R. This pair does not belong to $c(R)$ either: $u{\downarrow_R} = u + \sum_{k<j} X_k$ while X_j is in normal form ($X_j{\downarrow_R} = X_j$) and does not contain u.

Since this example actually corresponds to a universality problem, the antichain algorithm from [30] coincides with HKC (see Section III-E) and also requires exponential time.

On the contrary, if we check the equivalence of X_0 and x_0^1 (rather than u), HK still requires $p = \mathrm{lcm}[1..n]$ steps, but HKC now only requires n iterations: at step $n - 1$, we have $R = \{(X_k,\ x_0^1) \mid k < n\}$, and the pair (X_n, x_0^1) belongs to $c(R)$: we have $x_0^1{\downarrow_R} \subseteq X_n{\downarrow_R}$ since x_0^1 belongs to X_n, and $x_0^1{\downarrow_R}$ is the full state (i.e., $\{x_i^j \mid i < j\}$) since we went through all states of each cycle. Therefore, $X_n{\downarrow_R} = x_0^1{\downarrow_R}$, and thus $X_n \ c(R)\ x_0^1$ by Theorem 3. The antichain algorithm from [30] requires $o(n^2)$ iterations in this case, and its improved version [1] stops just after the computation of the largest similation: all states are simulated by x_0^1. This is another case where HKC and antichain algorithms bring an exponential speed-up over HK.

C. Experimental assessment

To get an intuition of the average behaviour of HKC on larger NFA, we performed tests on random automata. The results are given in Table I and Figure 7; we proceeded as follows.

For a given size n, we generate a thousand random NFA with n states and two letters. According to what is done in [27], we use a linear transition 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 most challenging NFA with this particular value. We generate NFA without accepting states: by doing so, we make sure that HK and HKC never encounter a pair of distinguishable sets, so that they always continue until they find a bisimulation up to; this corresponds to the worst case for all possible choices of accepting states.

We then run the algorithms on these NFA, starting each time with two distinct singleton sets, and we measure the required time. We report in Table I the median values, the last deciles,

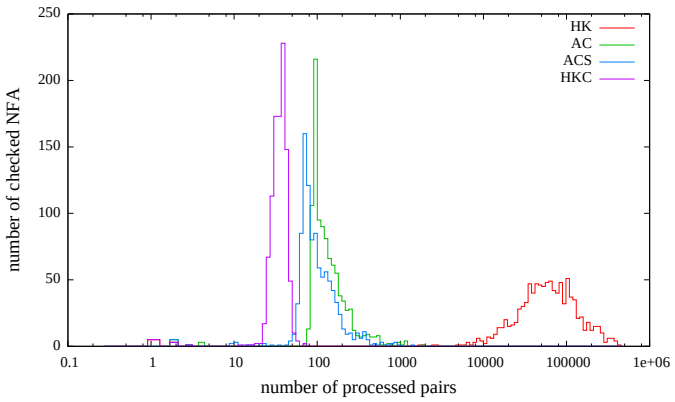


Figure 7. Distribution of the number of pairs processed with HK, AC, and HKC, for the 1000 NFA with 100 states and 2 letters used in Table I.

the last percentiles, and the maximal values. For instance, for $n = 70$, 90% of the examples require less than 0.649s with HK; equivalently, 10% of the examples require more than 0.649s.

The algorithms we tested are HK and HKC, the (forward) antichain algorithm from [30], which we call AC, and ACS, an optimisation of AC proposed in [1], which relies on a preliminary computation of the largest simulation. All algorithms were implemented in OCaml, with the same degree of optimisation. We do not use BDDs; while this is fair to compare HK, HKC, and AC, the timings for ACS are biased: our implementation of the relation $\preceq^{\forall\exists}$, which is heavily used in ACS, is certainly not as efficient as with BDDs. (Note however that we do not include the time needed to compute the largest simulations in Table I.)

In order to have a measure of the effectiveness of these approaches that is independent from the actual implementation, we also report the number of *processed pairs*. For HK and HKC, this number is just the size of the produced relation, or equivalently, the number of non trivial iterations (cf. Lemma 5). For AC and ACS, this is the number of outer iterations, i.e., the loop at line 4 in [1, Algorithm 2].

One can notice that our algorithm is several orders of magnitude better than HK, and two to ten times better than the antichain ones. More importantly, it is more predictable (the last percentile is of the same order as the median value, which is not the case with the other algorithms). The second point can better be seen on Figure 7, where we plotted the distribution of the number of processed pairs in the special case where $n = 100$: these numbers are much more dispersed with HK and AC than with HKC.

Also notice that the size of the relations generated by HK is a lower bound for the number of accessible states of the determinised NFA (Lemma 6 (2)); one can thus see in Table I 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 [27]. One can see that HK usually explores much more states than what would be necessary with a minimal DFA, while HKC needs much less.

We performed experiments with NFA for which the algorithms most always return a counter-example (i.e., by randomly adding a few accepting states). We basically observed the same behaviour: the up to congruence technique allows to cut very large branches in the state-space exploration.

V. 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 [8] introduced this concept to give a shorter and nicer proof of the result by Baeten, Bergstra, and Klop [3]: bisimilarity is decidable for normed context-free processes. Christensen, Hüttel, and Stirling [9] then generalised the result to all context-free processes, also by using self-bisimulations. Hirshfeld, Jerrum, and Moller [11] 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 [16], [4], [22], [17]. These propose very general frameworks into which "our" up to congruence technique fits as a very special case. To best of our knowledge, bisimulation up to congruence has never been proposed as a technique for proving language equivalence of NFA.

VI. 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. Thanks to this optimisation, HKC is several orders of magnitude faster than HK, and usually much faster than more recent antichain based algorithms [1], [30].

Our algorithm cannot be used for minimising automata (see Remark 2: it does not build the largest bisimulation). However, in case one only wants to check the equivalence of two states, HKC is more efficient than standard minimisation algorithms (independently from the minimisation procedure, the average size of the minimal DFA is much larger than the average number of states explored with HKC, as shown in Table I.)

We put an implementation of the presented algorithms online [5], together with Coq certified proofs and with an applet allowing to test the algorithms on user-provided examples.

		required time (seconds)				number of processed pairs				mDFA size
$n = \|S\|$	algo.	50%	90%	99%	100%	50%	90%	99%	100%	50%
50	HK	0.018	0.048	0.092	0.200	2511	6299	12506	25272	∼1000
	AC	0.000	0.001	0.004	0.034	58	115	331	1396	
	ACS	0.001	0.002	0.011	0.125	44	89	250	1117	
	HKC	0.000	0.000	0.000	0.001	21	26	32	63	
70	HK	0.215	0.649	1.377	2.803	10479	28186	58782	87055	∼6000
	AC	0.001	0.002	0.013	0.081	84	171	546	1935	
	ACS	0.002	0.006	0.051	0.411	64	135	409	1447	
	HKC	0.001	0.001	0.001	0.002	27	33	40	49	
100	HK	1.528	4.493	10.957	18.248	58454	164857	361227	471727	∼30000
	AC	0.002	0.005	0.026	0.112	117	245	785	1901	
	ACS	0.003	0.014	0.147	0.734	89	188	620	1451	
	HKC	0.001	0.002	0.003	0.006	35	44	54	70	
200	AC	0.006	0.016	0.040	0.126	227	441	846	1861	–
	ACS	0.009	0.042	0.155	0.458	175	341	651	1030	
	HKC	0.003	0.005	0.007	0.010	61	74	87	104	
300	AC	0.012	0.034	0.104	0.552	336	645	1365	3947	–
	ACS	0.018	0.092	0.416	3.426	256	500	1064	2879	
	HKC	0.007	0.011	0.015	0.035	86	103	118	129	
500	AC	0.032	0.109	0.353	1.384	556	1153	2586	5960	–
	ACS	0.051	0.325	1.649	12.030	428	915	1927	4990	
	HKC	0.018	0.029	0.043	0.064	129	154	175	192	
1000	AC	0.133	0.388	1.910	34.327	1088	2010	5735	26760	–
	ACS	0.211	1.096	9.538	193.245	835	1558	4390	19466	
	HKC	0.068	0.110	0.177	0.278	228	269	303	337	
3000	AC	1.556	4.587	15.125	142.319	3217	5612	11740	41800	–
	HKC	0.735	1.150	1.885	3.917	566	666	745	811	
5000	AC	4.888	16.971	80.361	561.909	5249	9957	25535	72893	–
	HKC	2.309	3.682	7.775	11.190	869	1009	1103	1214	

Table I
RUNNING HK, HKC, AND THE ALGORITHMS FROM [30] AND [1] TO CHECK LANGUAGE EQUIVALENCE ON RANDOM NFA WITH TWO LETTERS.

As future work, we would like to understand the average complexity of HKC. An inherent problem comes from the difficulty to characterise the average shape of determinised NFA [29], [27]. To avoid this problem, with HKC, we could try to focus on the properties of congruence relations. For instance, given a number of states, how long can be a sequence of (incrementally independent) pairs of sets of states whose congruence closure collapses into the full relation? (This number is an upper-bound for the size of the relations produced by HKC.) We found ad-hoc examples where this number is exponential, but we suspect it to be really small in average.

REFERENCES

[1] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *Proc. TACAS*, volume 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] 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)*, volume 259 of *LNCS*, pages 94–111. Springer, 1987.

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

[5] F. Bonchi and D. Pous. Web appendix for this paper. http://sardes.inrialpes.fr/~pous/hknt, 2012.

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

[7] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical Theory of Automata*, volume 12(6), pages 529–561. Polytechnic Press, NY, 1962.

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

[15] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *Proc. STOC*, pages 264–274. ACM, 1992.

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

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

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

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

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I/II. *Information and Computation*, 100(1):1–77, 1992.

[21] M. Nair. On Chebyshev-type inequalities for primes. *Amer. Math. Monthly*, 89:126–129, 1982.

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

[23] J. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proc. CONCUR*, volume 1466 of *LNCS*, pages 194–218. Springer, 1998.
[24] D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8:447–479, 1998.
[25] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.
[26] A. Silva, F. Bonchi, M. Bonsangue, and J. Rutten. Generalizing the powerset construction, coalgebraically. In *Proc. FSTTCS*, volume 8 of *LIPIcs*, pages 272–283. Leibniz-Zentrum fuer Informatik, 2010.
[27] D. Tabakov and M. Vardi. Experimental evaluation of classical automata constructions. In *Proc. LPAR*, volume 3835 of *LNCS*, pages 396–411. Springer, 2005.
[28] D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *LICS*, pages 280–291, 1997.
[29] B. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, 1995.
[30] M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. CAV: Computer-Aided Verification*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.

APPENDIX

A. Smallest bisimulation and compositionality

In this appendix, we show some (unrelated) properties that have been discussed through the paper. They are interesting but not strictly necessary for the formal development of our theory.

The first property concerns the relation computed by $\texttt{Naive}(x, y)$. The following proposition shows that it is the *smallest bisimulation* relating x and y.

Proposition 4. *Let x and y be two states of a DFA. Let R_{Naive} be the relation built by $\texttt{Naive}(x, y)$. If $\texttt{Naive}(x, y) = true$, then R_{Naive} is the smallest bisimulation relating x and y, i.e., $R_{Naive} \subseteq R$, for all bisimulations R such that $(x, y) \in R$.*

Proof: We have already shown in Proposition 2 that R_{Naive} is a bisimulation. We need to prove that it is the smallest. Let R be a bisimulation such that $(x, y) \in R$. For all words $w \in A^*$ and pair of states (x', y') such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$, it must hold that $(x', y') \in R$ (by definition of bisimulation).

By construction, for all $(x', y') \in R_{Naive}$ there exists a word $w \in A^*$, such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$. Therefore all the pairs in R_{Naive} must be also in R, that is $R_{Naive} \subseteq R$. ∎

The second property is

$$[\![X + Y]\!] = [\![X]\!] + [\![Y]\!] \ ,$$

which we have used in the Introduction to give an intuition of bisimulation up to context. Since Theorem 2 does not directly rely on this property, we have avoided to prove it in the main text. Even if it is not needed, we believe that this property is interesting, since it follows from the categorical observation made in [26] that determinized NFA are bialgebras [28]. For this reason, we prove here that $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ is a semi-lattice homomorphism.

Theorem 4. *Let (S, o, δ) be a non-deterministic automaton and $(\mathcal{P}(S), o^\sharp, \delta^\sharp)$ be the corresponding deterministic automaton obtained through the powerset construction. The function $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ is a semi-lattice homomorphism, that is, for all $X_1, X_2 \in \mathcal{P}(S)$,*

$$[\![X_1 + X_2]\!] = [\![X_1]\!] + [\![X_2]\!] \quad \textit{and} \quad [\![0]\!] = 0 \, .$$

Proof: We prove that for all words $w \in A^*$, $[\![X_1 + X_2]\!](w) = [\![X_1]\!](w) + [\![X_2]\!](w)$, by induction on w.

- for ϵ, we have:

$$\begin{aligned} [\![X_1 + X_2]\!](\epsilon) &= o^\sharp(X_1 + X_2) \\ &= o^\sharp(X_1) + o^\sharp(X_2) = [\![X_1]\!](\epsilon) + [\![X_2]\!](\epsilon) \, . \end{aligned}$$

- for $a \cdot w$, we have:

$$\begin{aligned} & [\![X_1 + X_2]\!](a \cdot w) \\ &= [\![\delta^\sharp(X_1 + X_2)(a)]\!](w) && \text{(by definition)} \\ &= [\![\delta^\sharp(X_1)(a) + \delta^\sharp(X_2)(a)]\!](w) && \text{(by definition)} \\ &= [\![\delta^\sharp(X_1)(a)]\!](w) + [\![\delta^\sharp(X_2)(a)]\!](w) \\ & && \text{(by induction hypothesis)} \\ &= [\![X_1]\!](a \cdot w) + [\![X_2]\!](a \cdot w) \, . && \text{(by definition)} \end{aligned}$$

For the second part, we prove that for all words $w \in A^*$, $[\![0]\!](w) = 0$, again by induction on w. *Base case:* $[\![0]\!](\epsilon) = o^\sharp(0) = 0$. *Inductive case:* $[\![0]\!](a \cdot w) = [\![\delta^\sharp(0)(a)]\!](w) = [\![0]\!](w)$ that by induction hypothesis is 0. ∎

B. Proofs of Section II

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

Proof: Let $R_{[\![-]\!]}$ be the relation $\{(x, y) \mid [\![x]\!] = [\![y]\!]\}$. We prove that $R_{[\![-]\!]}$ is a bisimulation. If $x \ R_{[\![-]\!]} \ y$, then $o(x) = [\![x]\!](\epsilon) = [\![y]\!](\epsilon) = o(y)$. Moreover, for all $a \in A$ and $w \in A^*$, $[\![t(x)(a)]\!](w) = [\![x]\!](a \cdot w) = [\![y]\!](a \cdot w) = [\![t(y)(a)]\!](w)$ that means $[\![t(x)(a)]\!] = [\![t(y)(a)]\!]$, that is $t(x)(a) \ R_{[\![-]\!]} \ t(y)(a)$.

We now prove the other direction. Let R be a bisimulation. We want to prove that $x \ R \ y$ entails $[\![x]\!] = [\![y]\!]$, i.e., for all $w \in A^*$, $[\![x]\!](w) = [\![y]\!](w)$. We proceed by induction on w. For $w = \epsilon$, we have $[\![x]\!](\epsilon) = o(x) = o(y) = [\![y]\!](\epsilon)$. For $w = a \cdot w'$, since R is a bisimulation, we have $t(x)(a) \ R \ t(y)(a)$ and thus $[\![t(x)(a)]\!](w') = [\![t(y)(a)]\!](w')$ by induction. This allows us to conclude since $[\![x]\!](a \cdot w') = [\![t(x)(a)]\!](w')$ and $[\![y]\!](a \cdot w') = [\![t(y)(a)]\!](w')$. ∎

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.

Proof: The first two points are straightforward;

For the third one, assume that F is a family of compatible functions. Suppose that $R \rightarrowtail R'$; for all $f \in F$, we have $f(R) \rightarrowtail f(R')$ so that $\bigcup_{f \in F} f(R) \rightarrowtail \bigcup_{f \in F} f(R')$.

For the last one, assume that f is compatible; for all n, f^n is compatible because (a) $f^0 = id$ is compatible (by the first point) and (b) $f^{n+1} = f \circ f^n$ is compatible (by the second point and induction hypothesis). By definition $f^\omega = \bigcup_n f^n$ and thus, by the third point, f^ω is compatible. ∎

Lemma 2. The following functions are compatible:

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

Proof:

r: observe that the identity relation $Id = \{(x,x) \mid \forall x \in S\}$ is always a bisimulation, i.e., $Id \rightarrowtail Id$. Thus for all R, R' $r(R) = Id \rightarrowtail Id = r(R')$.

s: observe that the definition of progression is completely symmetric. Therefore, if $R \rightarrowtail R'$, then $s(R) \rightarrowtail s(R')$.

t: assume that $R \rightarrowtail R'$. For each $(x,z) \in t(R)$, there exists y such that $(x,y) \in R$ and $(y,z) \in R$. By assumption, (1) $o'(x) = o'(y) = o'(z)$ and (2) for all $a \in A$, $t'(x)(a)\, R'\, t'(y)(a)\, R'\, t'(z)(a)$, that is $t'(x)(a)\, t(R')\, t'(z)(a)$.

■

C. Proofs of Section III

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

Proof: We have that $Z \rightsquigarrow_R Z'$ implies $|Z'| > |Z|$, where $|X|$ denotes the cardinality of the set X (note that $\rightsquigarrow_R$ is irreflexive). Since $|Z'|$ is bounded by $|S|$, the number of states of the NFA, the relation $\rightsquigarrow_R$ is strongly normalising. We can also check that whenever $Z \rightsquigarrow_R Z_1$ and $Z \rightsquigarrow_R Z_2$, either $Z_1 = Z_2$ or there is some Z' such that $Z_1 \rightsquigarrow_R Z'$ and $Z_2 \rightsquigarrow_R Z'$. Therefore, $\rightsquigarrow_R$ is convergent. ■

Lemma 7. *The relation $\rightsquigarrow_R$ is contained in $c(R)$.*

Proof: If $Z \rightsquigarrow_R Z'$ then there exists $(X,Y) \in (s \cup \mathrm{id})(R)$ such that $Z = Z + X$ and $Z' = Z + Y$. Therefore $Z\ c(R)\ Z'$ and, thus, $\rightsquigarrow_R$ is contained in $c(R)$. ■

Lemma 8. *Let $X, Y \in \mathcal{P}(S)$, we have $(X+Y)\downarrow_R = (X\downarrow_R + Y\downarrow_R)\downarrow_R$.*

Proof: Follows from confluence (Lemma 4) and from the fact that for all Z, Z', U, $Z \rightsquigarrow_R Z'$ entails $U + Z \rightsquigarrow_R U + Z'$. ■

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

Proof: From right to left. We proceed by induction on the derivation of $(X,Y) \in c(R)$. The cases for rules r, s, and t are straightforward. For rule id, suppose that $X\ R\ Y$, we have to show $X\downarrow_R = Y\downarrow_R$:

- if $X = Y$, we are done;
- if $X \subsetneq Y$, then $X \rightsquigarrow_R X + Y = Y$;
- if $Y \subsetneq X$, then $Y \rightsquigarrow_R X + Y = X$;
- if neither $Y \subseteq X$ nor $X \subseteq Y$, then $X, Y \rightsquigarrow_R X + Y$.

(In the last three cases, we conclude by confluence—Lemma 4.)
For rule u, suppose by induction that $X_i\downarrow_R = Y_i\downarrow_R$ for $i \in 1, 2$; we have to show that $(X_1 + Y_1)\downarrow_R = (X_2 + Y_2)\downarrow_R$. This follows by Lemma 8.

From left to right. By Lemma 7, we have $X\ c(R)\ X\downarrow_R$ for any set X, so that $X\ c(R)\ X\downarrow_R = Y\downarrow_R\ c(R)\ Y$. ■

D. Proofs of Section IV

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.

Proof: At each iteration, one pair is extracted from $todo$. The latter contains one pair before entering the loop and v pairs are added to it every time that a pair is added to R. ■

Lemma 9. *Let x and y be two states of a DFA. Let R_{Naive} and R_{HK} be relations computed by $\mathtt{Naive}(x,y)$ and $\mathtt{HK}(x,y)$, respectively. If $\mathtt{Naive}(x,y) = \mathtt{HK}(x,y) = true$, then $e(R_{Naive}) = e(R_{HK})$.*

Proof: By the proof of Proposition 3, $e^\omega(R_{HK})$ is a bisimulation. Since e is idempotent, we have $e^\omega = e$ and thus $e(R_{HK})$ is a bisimulation; we can thus deduce by Proposition 4 that $R_{Naive} \subseteq e(R_{HK})$. Moreover, by definition of the algorithms, we have $R_{HK} \subseteq R_{Naive}$. Summarising,

$$R_{HK} \subseteq R_{Naive} \subseteq e(R_{HK})$$

It follows that $e(R_{HK}) = e(R_{Naive})$, e being monotonic and idempotent. ■

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

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

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

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

Proof: For the first point, let PS denote the set of (determinised NFA) states accessible from the two starting states, so that $m = |PS| \leq 2^n$. Since $R_{Naive} \subseteq PS \times PS$, we deduce $|R_{Naive}| \leq m^2$. Since each pair added to R_{HK} merges two distinct equivalence classes in $e(R_{HK})$, we necessarily have $|R_{HK}| \leq m$ (the largest partition of PS has exactly m singletons). Similarly, each pair added to R_{HKC} merges *at least* two distinct equivalence classes in $c(R_{HK})$, so that we also have $|R_{HKC}| \leq m$.

For the second point, $|R_{HK}| \leq |R_{Naive}|$ follows from the fact that $R_{HK} \subseteq R_{Naive}$, by definition of the algorithms. However, the other inequality is less obvious, since $R_{HKC} \subseteq R_{HK}$ does *not* hold in general (see Remark 4 below).

By construction, $R_{HKC} \subseteq R_{Naive}$ and, since e is monotonic, $e(R_{HKC}) \subseteq e(R_{Naive}) = e(R_{HK})$ (the latter equality is given by Proposition 9). In particular, there are more equivalence classes in $e(R_{HKC})$ than in $e(R_{HK})$; using the same argument as above, we deduce that $|R_{HKC}| \leq |R_{HK}|$. ■

Remark 4. *Even though Lemma 6 ensures that $|R_{HKC}| \leq |R_{HK}|$, it is possible that $R_{HKC} \not\subseteq R_{HK}$. For an example,*

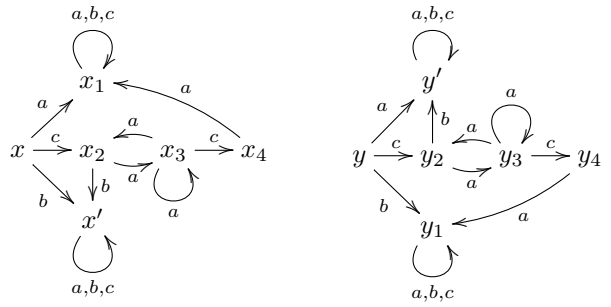


Figure 8. The relation built by $\mathtt{HKC}(x, y)$ is not included in $\mathtt{HK}(x, y)$.

consider Figure 8 and execute $\mathtt{HK}(x, y)$ *and* $\mathtt{HKC}(x, y)$ *with a depth-first strategy. For the first five iterations, the two algorithms behave the same: they build the relation*

$$R = \{(x, y), (x_1, y'), (x', y_1), (x_2, y_2), (x_3, y_3)\}.$$

Then, they encounter $(x_2 + x_3, y_2 + y_3)$ *which is not in* $e(R)$*, but in* $c(R)$*. Therefore* $\mathtt{HK}$ *inserts it into the relation and then visits* (x', y') *(which is reached by* $(x_2 + x_3, y_2 + y_3)$ *with a* b*-transition). Thus, the relation eventually produced by* $\mathtt{HK}$ *is*

$$R_{HK} = R \cup \{(x_2 + x_3, y_2 + y_3), (x', y'), (x_4, y_4)\}.$$

Instead $\mathtt{HKC}$ *skips* $(x_2 + x_3, y_2 + y_3)$ *and visits* (x_4, y_4) *(which is reached by* (x_3, y_3) *with a* c*-transition) and then* (x_1, y_1)*. Therefore, the relation produced by* $\mathtt{HKC}$ *is*

$$R_{HKC} = R \cup \{(x_4, y_4), (x_1, y_1)\}.$$

Note that $|R_{HKC}|$ *is strictly smaller than* $|R_{HK}|$*, but* $R_{HKC} \not\subseteq R_{HK}$*, since* $(x_1, y_1) \notin R_{HK}$*.*

Lemma 10. *The relation*

$$R' = \{(x, y), (x, y+z)\} \cup \{(x+x_i, y+y_i+z_i) \mid i \in [1 \dots n]\}$$

is a bisimulation up to congruence for the NFA in Fig. 5.

Proof: We consider each kind of pair of R' separately:

- (x, y): we have $o^\sharp(x) = 0 = o^\sharp(y)$ and $t_a^\sharp(x) = x \; R' \; y + z = t_a^\sharp(y)$ and, similarly, $t_b^\sharp(x) = x + x_1 \; R' \; y + z + y_1 = t_b^\sharp(y)$.
- $(x, y + z)$: as the previous point.
- $(x + x_i, y + z + y_i)$ for $i < n$:
  we have $o^\sharp(x + x_i) = 0 = o^\sharp(y + z + y_i)$ and

$$\begin{aligned} t_a^\sharp(x + x_i) &= x + x_{i+1} \\ &\quad R' \; y + z + y_{i+1} \\ &= t_a^\sharp(y + z + y_i)\,, \text{ and} \\ t_b^\sharp(x + x_i) &= x + x_1 + x_{i+1} \\ &\quad c(R') \; y + z + y_1 + y_{i+1} \\ &= t_b^\sharp(y + z + y_i)\,; \end{aligned}$$

- $(x + x_n, y + z + y_n)$:
  we have $o^\sharp(x + x_n) = 1 = o^\sharp(y + z + y_n)$ and

$$\begin{aligned} t_a^\sharp(x + x_n) &= x \\ &\quad R' \; y + z = t_a^\sharp(y + z + y_n)\,, \text{ and} \\ t_b^\sharp(x + x_n) &= x + x_1 \\ &\quad R' \; y + z + y_1 \\ &= t_b^\sharp(y + z + y_n)\,. \end{aligned}$$

■