Hacking nondeterminism with induction and coinduction.

Filippo Bonchi, Damien Pous

Hacking Nondeterminism with Induction and Coinduction

Filippo Bonchi Damien Pous
CNRS, ENS Lyon, LIP, Université de Lyon, 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 optimization of the classic algorithm by Hopcroft and Karp [13]. We compare our approach to the recently introduced antichain algorithms and we give concrete examples where we exponentially improve over antichains. Experimental results show significant improvements.

1. INTRODUCTION

Checking language equivalence of finite automata is a classic problem in computer science, with many applications in areas ranging from compilers to model checking.

Equivalence of deterministic finite automata (DFA) can be checked either via minimization [12] or through Hopcroft and Karp's algorithm [13], which exploits an instance of what is nowadays called a *coinduction proof principle* [17, 22, 20]: two states are equivalent 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, minimization algorithms have the advantage of checking the equivalence of all the states at once, while Hopcroft and Karp's algorithm only checks 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" [8], on a lazy DFA whose transitions are computed on demand.

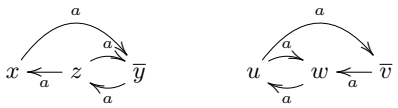
This difference is essential for our work and for other recently introduced algorithms based on *antichains* [25, 1, 7]. Indeed, when starting from non-deterministic finite automata (NFA), determinization induces an exponential factor. In contrast, the algorithm we introduce in this work for checking equivalence of NFA (as well as those using antichains) usually does not build the whole deterministic automaton, but just a small part of it. We write "usually" because in few cases, the algorithm can still explore an exponential number of states.

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

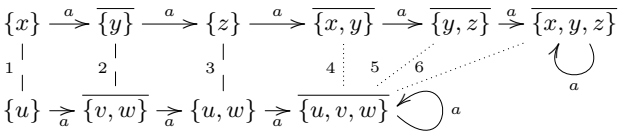
$$[\![X+Y]\!] = [\![X]\!] + [\![Y]\!] \tag{1}$$

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 relates the set of states X_1 with Y_1, and X_2 with Y_2, then $[\![X_1]\!] = [\![Y_1]\!]$ and $[\![X_2]\!] = [\![Y_2]\!]$ and, by (1), we can immediately conclude that $X_1 + X_2$ and $Y_1 + Y_2$ are language equivalent as well. Intuitively, bisimulations up to context are bisimulations which *do not need to relate* $X_1 + X_2$ with $Y_1 + Y_2$ when X_1 is already related with Y_1 and X_2 with Y_2.

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



The determinized 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 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 $\{x, y\}$ and $\{u, v, w\}$ is deduced from the fact that $\{x\}$ is related with $\{u\}$ and $\{y\}$ with $\{v, w\}$, without the need to further explore the automaton.

Extended Abstract, a full version of this paper is available in Proc. POPL, 2013, ACM. Work partially funded by the PiCoq (ANR-10-BLAN-0305) and PACE (ANR-12IS02001) projects.

http://doi.acm.org/10.1145/2713167.

Bisimulations up-to, and in particular bisimulations up to context, have been introduced in the setting of concurrency theory [17, 21] 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 Hopcroft and Karp's original algorithm. This technique can be explained by noting that not all bisimulations are equivalence relations: it might be the case that a bisimulation relates X with Y, and Y with Z, but not X with Z. However, since $[\![X]\!] = [\![Y]\!]$ and $[\![Y]\!] = [\![Z]\!]$, we can immediately conclude that X and Z recognize the same language. Analogously to bisimulations up to context, a bisimulation up to equivalence does *not* need to relate X with Z when they are both related with 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 property when using up to congruence is that we do not need to build the whole deterministic automata. For instance, in the above NFA, the algorithm stops after relating z with $u + w$ and does not build the remaining states. Despite their use of the up to equivalence, 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 determinized automaton is also the key feature of the antichain algorithm [25] and its optimization exploiting similarity [1, 7]. The two algorithms are designed to check *language inclusion* rather than equivalence and, for this reason, they do not exploit equational reasoning. As a consequence, the antichain algorithm usually needs to explore more states than ours. Moreover, we show how to integrate the optimization proposed in [1, 7] in our setting, resulting in an even more efficient algorithm.

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.

2. DETERMINISTIC AUTOMATA

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 $t_a(x)$. Any DFA induces a function $[\![\cdot]\!]$ mapping states to formal languages ($\mathcal{P}(A^\star)$), defined by $[\![x]\!](\epsilon) = o(x)$ for the empty word, and $[\![x]\!](aw) = [\![t_a(x)]\!](w)$ otherwise. For a state x, $[\![x]\!]$ is called the language accepted by x.

Throughout this paper, we consider a fixed automaton (S, o, t) and study the following problem: given two states x_1, x_2 in S, is it the case that they are language equivalent, that is, $[\![x_1]\!] = [\![x_2]\!]$?

$\texttt{Naive}(x, y)$

```
(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 continue;
 (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). The code of $\texttt{HK}(x, y)$ is obtained by replacing the test in step 3.2 with $(x', y') \in e(R)$.

This problem generalizes the familiar problem of checking whether two automata accept the same language: just take the union of the two automata as the automaton (S, o, t), and determine whether their respective starting states are language equivalent.

2.1 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 $R, R' \subseteq S^2$ 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.

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:

$$R \rightarrowtail R \cup todo$$

Since *todo* is empty at step 4, we have $R \rightarrowtail R$, i.e., R is a bisimulation. By Prop. 1, $x \sim y$. On the other hand, $\texttt{Naive}(x, y)$ returns false as soon as it finds a word which is accepted by one state and not the other. □

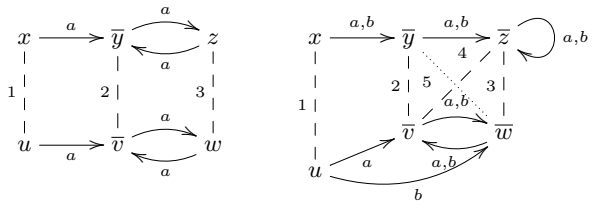


Figure 2: Checking for DFA equivalence.

For 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 initialization, (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.)

In the previous example, *todo* always contains at most one pair of states but, in general, it may contain several of them. We do not specify here how to choose the pair to extract in step `3.1`; we discuss this point in Section 3.2.

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 it encounters a pair of states that is not already in R but belongs to 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 optimization, the produced relation R contains at most n pairs. Formally, ignoring the concrete data structure used to store equivalence classes, Hopcroft and Karp's algorithm consists in replacing step `3.2` in Figure 1 with

```
(3.2)  if (x', y') ∈ e(R) then continue;
```

where $e\colon \mathcal{P}(S^2) \to \mathcal{P}(S^2)$ is the function mapping each relation $R \subseteq S^2$ into its symmetric, reflexive, and transitive closure. We refer to this algorithm as `HK`.

2.4 Bisimulations up-to

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

Definition 2 (Bisimulation up-to)**.** *Let $f\colon \mathcal{P}(S^2) \to \mathcal{P}(S^2)$ be a function on relations. A relation R is a* bisimulation up to f *if $R \rightarrowtail f(R)$, i.e., if $x\ R\ y$, 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. To this end, we the notion of compatible function [21, 19]:

Definition 3 (Compatible function)**.** *A function $f\colon \mathcal{P}(S^2) \to \mathcal{P}(S^2)$ is* compatible *if it is monotone and it preserves progressions: for all $R, R' \subseteq S^2$,*

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

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

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, defined by $f^\omega = \bigcup_i f^i$, with $f^0 = id$ and $f^{i+1} = f \circ f^i$;

r: the constant reflexive function: $r(\_) = \{(x, x) \mid x \in S\}$;

s: the converse function: $s(R) = \{(y, x) \mid x\ R\ y\}$;

t: 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 $e(R)$ 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.*

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

Proof. Same proof as for Prop. 2, by using the invariant $R \rightarrowtail e(R) \cup \mathit{todo}$. We deduce that R is a bisimulation up to e after the loop. We conclude with Thm. 1 and Prop. 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, $e(R_{\mathtt{HK}})$.

$\underline{\texttt{Naive}(X, Y)}$

```
(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 continue;
 (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 3: On-the-fly naive algorithm, for checking the equivalence of sets of states X and Y of an NFA (S, o, t). $\texttt{HK}(X, Y)$ is obtained by replacing the test in step 3.2 with $(X', Y') \in e(R)$, and $\texttt{HKC}(X, Y)$ is obtained by replacing it with $(X', Y') \in c(R \cup \mathit{todo})$.

3. NON-DETERMINISTIC AUTOMATA

We now move from DFA to non-deterministic automata (NFA). A NFA 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, and $t\colon S \to \mathcal{P}(S)^A$ is the transition relation: it assigns to each state $x \in S$ and letter $a \in A$ a set of possible successors.

The *powerset construction* transforms any NFA (S, o, t) into 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:

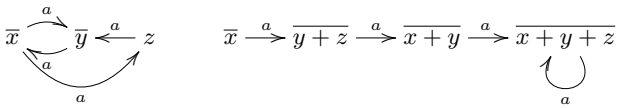
$$o^\sharp(X) = \begin{cases} o(x) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ o^\sharp(X_1) + o^\sharp(X_2) & \text{if } X = X_1 + X_2 \end{cases}$$

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

(Here we use the symbol $+$ to denote both set-theoretic union and Boolean or; similarly, we use 0 to denote both the empty set and the Boolean 'false'.) 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 stress the difference with generic DFA, which usually do not carry this structure, we use the following definition.

Definition 4. *A* determinized 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 determinized NFA: in place of the singleton $\{x\}$, we just write x and, in place of $\{x_1, \ldots, x_n\}$, we write $x_1 + \cdots + x_n$. Consider for instance the NFA (S, o, t) depicted below (left) and part of the determinized NFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ (right).



In the determinized NFA, x makes one single a-transition 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 $t_a^\sharp(y + z) = t_a^\sharp(y) + t_a^\sharp(z) = t_a(y) + t_a(z) = x + y$.

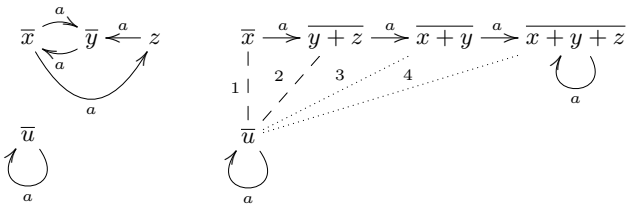


Figure 4: A bisimulation up to congruence.

Algorithms for NFA can be obtained by computing the determinized NFA on-the-fly [8]: 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 3. 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`.

3.1 Bisimulation up to congruence

The semi-lattice structure $(\mathcal{P}(S), +, 0)$ carried by determinized 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 $u\colon \mathcal{P}(\mathcal{P}(S)^2) \to \mathcal{P}(\mathcal{P}(S)^2)$ be the function on relations on sets of states defined for all $R \subseteq \mathcal{P}(S)^2$ 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:

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

Definition 6 (Bisimulation up to congruence)**.** *A* bisimulation up to congruence *for an NFA (S, o, t) is a relation $R \subseteq \mathcal{P}(S)^2$, such that whenever $X\ R\ Y$ then*

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

Lemma 2. *The function u is compatible.*

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

We already gave in Introduction an example of *bisimulation up to context*, which is a particular case of bisimulation up to congruence (up to context means up to $(r \cup u \cup \mathrm{id})^\omega$, without closing under s and t).

Figure 4 shows a more involved example illustrating the use of all ingredients of the congruence closure function (c). 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 e 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 $(x+y,\ u) \in c(R)$:

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

In contrast, we need four pairs to get a bisimulation up to equivalence containing (x,u): this is the relation depicted with both dashed and dotted lines in Figure 4.

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

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

3.2 Optimized algorithm for NFA

The optimized algorithm, called HKC in the sequel, relies on up to congruence: step 3.2 from Figure 3 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* 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 determinized NFA lazily. This means that only *accessible* states need to be computed, which is of practical importance since the determinized 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 never happens when starting with disjoint automata). In the other cases where a pair (X, Y) is skipped, X and Y are necessarily already related with some other states in R, so that their successors will eventually be explored.
- With HKC, accessible states are often skipped. For a simple example, let us execute HKC on the NFA from Figure 4. 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 determinized NFA can bring an exponential speed-up. For an example, consider the family of NFA in Figure 5, where n is an arbitrary natural number. Taken together, the states x and y are equivalent to z: they recognize the language $(a+b)^\star(a+b)^{n+1}$. Alone, x recognizes the language $(a+b)^\star a(a+b)^n$, which is known for having a minimal DFA with 2^n states.

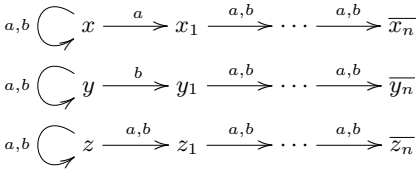


Figure 5: Family of examples where HKC exponentially improves over AC and HK; we have $x+y \sim z$.

Therefore, checking $x+y \sim z$ via minimization (as in [12]) requires exponential time, and the same holds for Naive and HK since all accessible states must be visited. This is not the case with HKC, which requires only polynomial time on this example. Indeed, HKC$(x+y, z)$ builds the relation

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

where $Y_i = y+y_1+\ldots+y_i$, and $Z_i = z+z_1+\ldots+z_i$. R' only contains $2n+1$ pairs and is a bisimulation up to congruence. To see this, consider the pair $(x+y+x_1+y_2,\ Z_2)$ obtained from $(x+y,\ z)$ after reading the word ba. Although this pair does not belong to R', it belongs to its congruence closure:

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

Remark 1. *In the above derivation, the use of transitivity is crucial: R' is a bisimulation up to congruence, but not a bisimulation up to context. In fact, there exists no bisimulation up to context of linear size proving $x+y \sim z$.*

We now discuss the exploration strategy, i.e., how to choose the pair to extract from the set *todo* in step 3.1. When looking for a counter-example, such a strategy has a large influence: a good heuristic can help in reaching it directly, while a bad one might lead to explore exponentially many pairs first. In contrast, the strategy does not impact much looking for an equivalence proof (when the algorithm eventually returns true). Actually, one can prove that the number of steps performed by Naive and HK in such a case does not depend on the strategy. This is not the case with HKC: the strategy can induce some differences. However, we experimentally observed that breadth-first and depth-first strategies usually behave similarly on random automata. This behaviour is due to the fact that we check congruence w.r.t. $R \cup todo$ rather than just R (step 3.2): with this optimization, the example above is handled in polynomial time whatever the chosen strategy. In contrast, without this small optimization, it requires exponential time with a depth-first strategy.

3.3 Computing the congruence closure

For the optimized algorithm to be effective, we need a way to check whether some pairs belong to the congruence

closure of a given relation (step `3.2`). We present 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:

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

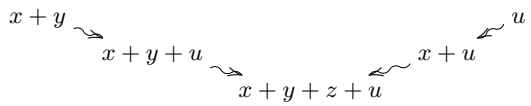
which can be used to compute normal forms for sets of states. Indeed, by idempotence, $X\ R\ Y$ entails $X\ c(R)\ X + Y$.

Definition 7. *Let $R \subseteq \mathcal{P}(S)^2$ be a relation on sets of states. We define $\rightsquigarrow_R \subseteq \mathcal{P}(S)^2$ 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 3. *For all relations R, $\rightsquigarrow_R$ is confluent and normalizing.*

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

We actually have $X{\downarrow_R} = Y{\downarrow_R}$ iff $X \subseteq Y{\downarrow_R}$ and $Y \subseteq X{\downarrow_R}$, so that the normal forms of X and Y do not necessarily need to be fully computed in practice. Still, the worst-case complexity of this sub-algorithm is quadratic in the size of the relation R (assuming we count the number of operations on sets: unions and inclusion tests).

Note that many algorithms were proposed in the literature to compute the congruence closure of a relation (see, e.g., [18, 23, 2]). However, they usually consider uninterpreted symbols or associative and commutative symbols, but not associative, commutative, and idempotent symbols, which is what we need here.

3.4 Using HKC for checking language inclusion

For NFA, language inclusion can be reduced to language equivalence: the semantics function $[\![-]\!]$ is a semi-lattice homomorphism, so that for all sets of states X, Y, $[\![X{+}Y]\!] = [\![Y]\!]$ iff $[\![X]\!] + [\![Y]\!] = [\![Y]\!]$ iff $[\![X]\!] \subseteq [\![Y]\!]$. Therefore, it suffices to run $\texttt{HKC}(X{+}Y, Y)$ 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` can thus be simplified. First, the pairs in the current relation only have to be used to rewrite from right to left. Second, the following lemma shows that we do not necessarily need to compute normal forms:

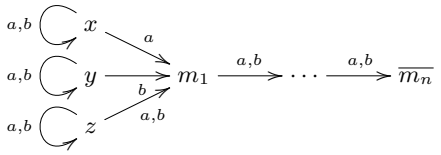
Lemma 4. *For all sets X, Y and for all relations R, we have $X{+}Y\ c(R)\ Y$ iff $X \subseteq Y{\downarrow_R}$.*

At this point, the reader might wonder whether checking the two inclusions separately is more convenient than checking the equivalence directly. This is not the case: 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 $\texttt{HKC}(x, u)$ contains only four pairs (because the fifth one follows from transitivity). Instead, the relations built by $\texttt{HKC}(x, x{+}u)$ and $\texttt{HKC}(u{+}x, u)$ would both contain five pairs: transitivity cannot be used since our relations are now oriented (from $y \leq v$, $z \leq v$ and $z \leq w$, we cannot deduce $y \leq w$). Figure 5 shows another example, where we get an exponential factor by checking the equivalence directly rather than through the two inclusions: transitivity, which is crucial to keep the relation computed by $\texttt{HKC}(x{+}y, z)$ small (see Remark 1), cannot be used when checking the two inclusions separately.

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.

3.5 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, one would merge the states x_i, y_i, z_i, and one would obtain the following automaton, for which checking $x{+}y \sim z$ is much easier.



As shown in [1, 7] for antichain algorithms, one can do better, by exploiting any preorder contained in language inclusion. Hereafter, we show how this idea can be embedded in `HKC`, resulting in an even stronger algorithm.

For the sake of clarity, we fix the preorder to be *similarity* [17], which can be computed in quadratic time [10].

Definition 8 (Similarity)**.** Similarity *is the largest relation on states $\preceq \subseteq S^2$ such that $x \preceq y$ entails:*

1. *$o(x) \leq o(y)$ 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 $x' \preceq y'$.*

To exploit similarity pairs in `HKC`, 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-to-$\overline{\preceq}$ function, and let $c' = (r' \cup s \cup t \cup u \cup id)^\omega$. Accordingly, we call `HKC'` the algorithm obtained from `HKC` (Figure 3) by replacing $(X, Y) \in c(R \cup \mathit{todo})$ with $(X, Y) \in c'(R \cup \mathit{todo})$ in step `3.2`. The latter test can be reduced to rewriting thanks to Theorem 3 and the following lemma.

Lemma 5. *For all relations R, $c'(R) = c(R \cup \overline{\preceq})$.*

Theorem 4. *Any bisimulation up to c' is contained in a bisimulation.*

Corollary 3. *For all sets X, Y, $X \sim Y$ iff $\texttt{HKC'}(X, Y)$.*

4. ANTICHAIN ALGORITHMS

Even though the problem of deciding NFA equivalence is PSPACE-complete [16], neither `HKC` nor `HKC'` are in PSPACE: both of them keep track of the states they explored in the determinized NFA, and there can be exponentially many such states. This also holds for `HK` and for the more recent

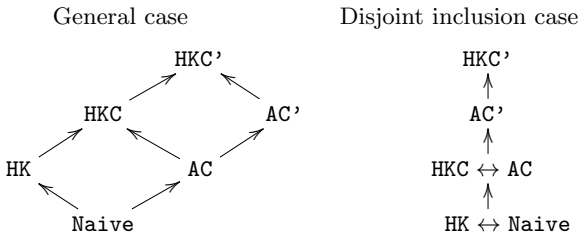


Figure 6: Relationships among the algorithms.

antichain algorithm [25] (called `AC` in the following) and its optimization (`AC'`) exploiting similarity [1, 7].

The latter algorithms can be explained in terms of coinductive proof techniques: we establish in [4] that they actually construct bisimulations up to context, i.e., bisimulations up to congruence for which one does not exploit *symmetry* and *transitivity*.

Theoretical comparison. We compared the various algorithms in details in [4]. Their relationship is summarised in Figure 6, where an arrow `X`→`Y` means that (a) `Y` can explore exponentially fewer states than `X`, and (b) `Y` can mimic `X`, i.e., the coinductive proof technique underlying `Y` is at least as powerful as the one of `X`.

In the general case, `AC` needs to explore much more states than `HKC`: the use of transitivity, which is missing in `AC`, allows `HKC` to drastically prune the exploration. For instance, to check $x+y \sim z$ in Figure 5, `HKC` only needs a linear number of states (see Remark 1), while `AC` needs exponentially many states. In contrast, in the special case where one checks for the inclusion of disjoint automata, `HKC` and `AC` exhibit the same behaviour. Indeed, `HKC` cannot make use of transitivity in such a situation, as explained in Section 3.4. Things change when comparing `HKC'` and `AC'`: even for checking inclusion of disjoint automata, `AC'` cannot always mimic `HKC'`: the use of similarity tends to virtually merge states, so that `HKC'` can use the up to transitivity technique which `AC'` lack.

Experimental comparison. The theoretical relationships drawn in Figure 6 are substantially confirmed by an empirical evaluation of the performance of the algorithms. Here, we only give a brief overview; see [4] for a complete description of those experiments.

We compared our OCaml implementation [4] for `HK`, `HKC` and `HKC'`, and the `libvata` C++ library [14] for `AC` and `AC'`. We use a breadth-first exploration strategy: we represent the set *todo* from Figure 3 as a FIFO queue. As mentioned at the end of Section 3.2, considering a depth-first strategy here does not alter the behaviour of `HKC` in a noticeable way.

We performed experiments using both random automata and a set of automata arising from model-checking problems.

• *Random automata.* We used Tabakov and Vardi's model [24] to generate 1000 random NFA with two letters and a given number of states. We executed all algorithms on these NFA, and we measured the number of processed pairs, i.e., the number of required iterations (like `HKC`, `AC` is a loop inside which pairs are processed). We observe that `HKC` improves over `AC` by one order of magnitude, and `AC` improves over `HK` by two orders of magnitude. Using up-to similarity (`HKC'` and `AC'`) does not improve much; in fact, similarity is almost the identity relation on such random automata. The corresponding distributions for `HK`, `HKC`, and `AC` are plotted on Figure 7, for automata with 100 states. Note that while `HKC` only improves by one order of magnitude over `AC` when considering the average case, it improves by several orders of magnitude when considering the worst cases.

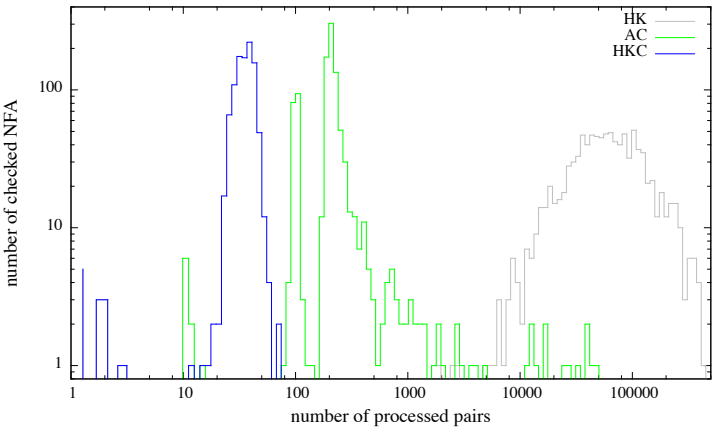


Figure 7: Distributions of the number of processed pairs, for 1000 experiments with random NFA

• *Model checking automata.* Abdulla et al. [1, 7] used automata sequences arising from regular model-checking experiments [5] to compare their algorithm (`AC'`) against `AC`. We reused these sequences to test `HKC'` against `AC'` in a concrete scenario. For all those sequences, we checked the inclusions of all consecutive pairs, in both directions. The timings are given in Table 1, where we report the median values (50%), the last deciles (90%), the last percentiles (99%), and the maximum values (100%). We distinguish between the experiments for which a counter-example was found, and those for which the inclusion did hold. For `HKC'` and `AC'`, we display the time required to compute similarity on a separate line: this preliminary step is shared by the two algorithms. As expected, `HKC` and `AC` roughly behave the same: we test inclusions of disjoint automata. `HKC'` is however quite faster than `AC'`: up to transitivity can be exploited thanks to similarity pairs. Also note that over the 546 positive answers, 368 are obtained immediately by similarity.

5. CONCLUSIONS

Our implementation of `HKC` is available online [4], together with proofs mechanized in the Coq proof assistant and an interactive applet making it possible to test the presented algorithms online, on user-provided examples.

Several notions analogous to bisimulations up to congruence can be found in the literature. For instance, *self-bisimulations* [6, 11] have been used to obtain decidability and complexity results about context-free processes. The main difference with bisimulation up to congruence is that self-bisimulations are proof techniques for bisimilarity rather than language equivalence. Other approaches, that are independent from the equivalence (like bisimilarity or language) are shown in [15, 3, 19]. These papers propose very general frameworks into which our up to congruence technique fits as a very special case. However, to our knowledge, bisimu-

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
sim_time	0.039	0.185	0.574	0.618	0.038	0.193	0.577	0.593
`AC'` - sim_time	0.013	0.167	1.326	1.480	0.012	0.107	1.047	1.134
`HKC'` - sim_time	0.000	0.034	0.224	0.345	0.001	0.005	0.025	0.383

Table 1: Timings, in seconds, for language inclusion of disjoint NFA generated from model-checking.

lation up to congruence has never been proposed before as a technique for proving language equivalence of NFA.

We conclude with directions for future work.

Complexity. The presented algorithms, as well as those based on antichains, have exponential complexity in the worst case while they behave rather well in practice. For instance, in Figure 7, one can notice that over a thousand random automata, very few require to explore a large amount of pairs. This suggests that an accurate analysis of the average complexity might be promising. An inherent problem comes from the difficulty to characterise the average shape of determinised NFA [24]. 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`.) One can find ad-hoc examples where this number is exponential, but we suspect it to be rather small in average.

Model checking. The experiments summarized in Table 1 show the efficiency of our approach for regular model-checking using automata on finite words. As in the case of antichains, our approach extends to automata on finite trees. We plan to implement such a generalisation and link it with tools performing regular tree model-checking.

In order to face other model-checking problems, it would be useful to extend up-to techniques to automata on infinite words, or trees. Unfortunately, the determinisation of these automata (the so called Safra's construction) does not seem suitable for exploiting neither antichains nor up to congruence. However, for some problems like LTL realizability [9] that can be solved without prior determinization (the so-called Safraless approaches), antichains have been crucial in obtaining efficient procedures. We leave as future work to explore whether up-to techniques could further improve such procedures.

6. REFERENCES

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

[2] L. Bachmair, I. V. Ramakrishnan, A. Tiwari, and L. Vigneron. Congruence closure modulo associativity and commutativity. In *FroCoS*, volume 1794 of *LNCS*, pages 245–259. Springer, 2000.

[3] F. Bartels. Generalised coinduction. *Math. Struct. in Comp. Sci.*, 13(2):321–348, 2003.

[4] F. Bonchi and D. Pous. Extended version of this abstract, with omitted proofs, and web appendix for this work. `http://hal.inria.fr/hal-00639716/` and `http://perso.ens-lyon.fr/damien.pous/hknt`, 2012.

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

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

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

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

[9] E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for ltl realizability. In *CAV*, volume 5643 of *LNCS*, pages 263–277. Springer, 2009.

[10] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*. IEEE Computer Society, 1995.

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

[12] J. E. Hopcroft. An n log n algorithm for minimizing in a finite automaton. In *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] O. Lengál, J. Simácek, and T. Vojnar. Vata: A library for efficient manipulation of non-deterministic tree automata. In *TACAS*, volume 7214 of *LNCS*, pages 79–94. Springer, 2012.

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

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

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

[18] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. of the ACM*, 27(2):356–364, 1980.

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

[20] J. Rutten. Automata and coinduction (an exercise in

coalgebra). In *CONCUR*, volume 1466 of *LNCS*, pages 194–218. Springer, 1998.

[21] D. Sangiorgi. On the bisimulation proof method. *Math. Struct. in Comp. Sci.*, 8:447–479, 1998.

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

[23] R. E. Shostak. Deciding combinations of theories. *J. of the ACM*, 31(1):1–12, 1984.

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

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