Checking NFA Equivalence with Bisimulations up to Congruence

Filippo Bonchi and Damien Pous

CNRS, LIP, ENS Lyon

POPL, Roma, 25.1.2o13

Language equivalence of finite automata

- Useful for model checking:
  - check that a program refines its specification
  - compute a sequence A_i of automata until $A_i \sim A_{i+1}$

(cf. abstract regular model checking)

- Useful in proof assistants:
  - decide the equational theory of Kleene algebra

$$(R \cup S)^\star = R^\star; (S; R^\star)^\star$$

(cf. the ATBR and RelationAlgebra Coq libraries)

- This work: a new algorithm

Outline

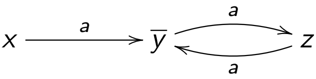
Deterministic Automata

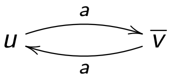
Non-Deterministic Automata

Comparison with other algorithms

Checking language equivalence

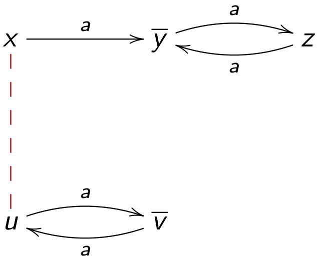
Deterministic case, first algorithm:





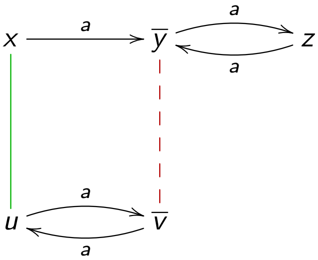
Checking language equivalence

Deterministic case, first algorithm:



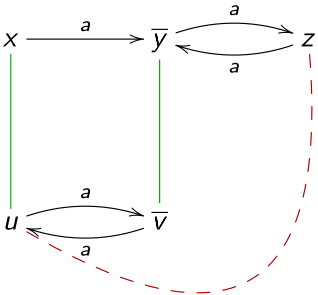
Checking language equivalence

Deterministic case, first algorithm:



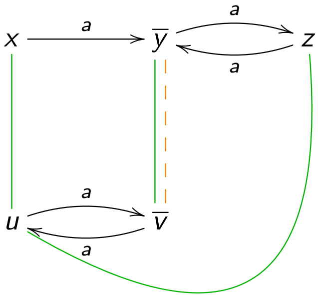
Checking language equivalence

Deterministic case, first algorithm:



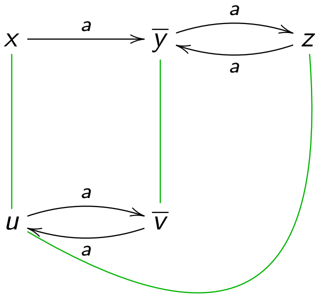
Checking language equivalence

Deterministic case, first algorithm:



Checking language equivalence

Deterministic case, first algorithm:



Checking language equivalence

Deterministic case, naive algorithm, correctness:

- A relation R is a bisimulation if $x \, R \, y$ entails
  - $o(x) = o(y)$;
  - for all a, $t_a(x) \, R \, t_a(y)$.

Checking language equivalence

Deterministic case, naive algorithm, correctness:

- A relation R is a bisimulation if $x \; R \; y$ entails
  - $o(x) = o(y)$;
  - for all a, $t_a(x) \; R \; t_a(y)$.
- *Theorem:* $L(x) = L(y)$ iff
  there exists a bisimulation R with $x \; R \; y$

Checking language equivalence

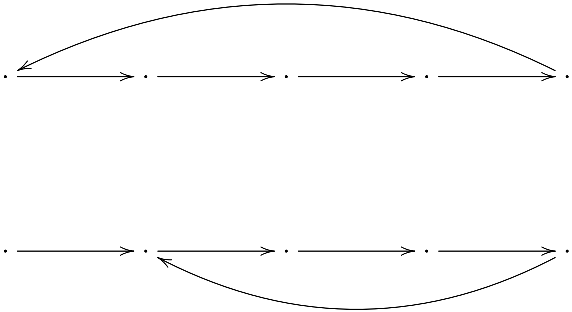
Deterministic case, naive algorithm, correctness:

- A relation R is a bisimulation if $x \; R \; y$ entails
  - $o(x) = o(y)$;
  - for all a, $t_a(x) \; R \; t_a(y)$.
- *Theorem:* $L(x) = L(y)$ iff
  there exists a bisimulation R with $x \; R \; y$

The previous algorithm attempts to construct a bisimulation

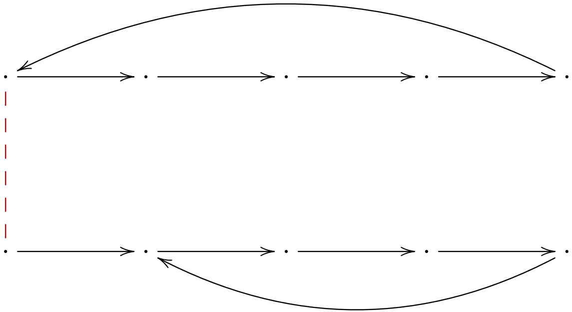
Checking language equivalence

Deterministic case, naive algorithm: quadratic complexity



Checking language equivalence

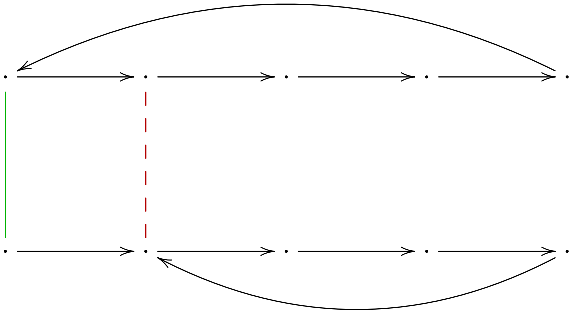
Deterministic case, naive algorithm: quadratic complexity



1 pairs

Checking language equivalence

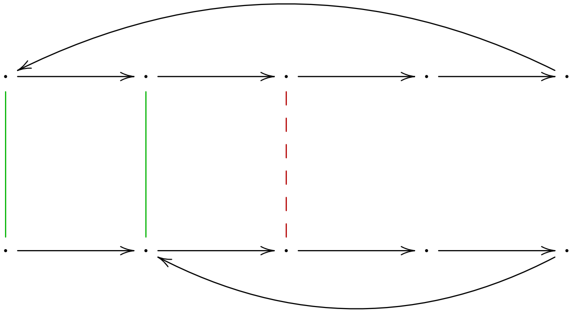
Deterministic case, naive algorithm: quadratic complexity



2 pairs

Checking language equivalence

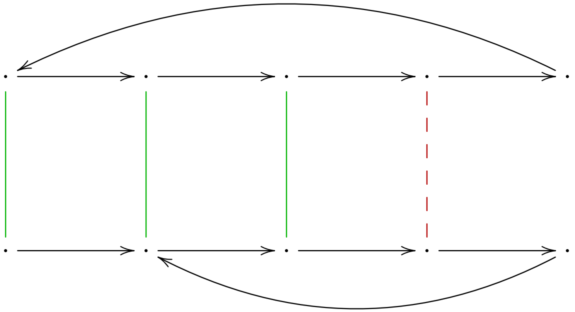
Deterministic case, naive algorithm: quadratic complexity



3 pairs

Checking language equivalence

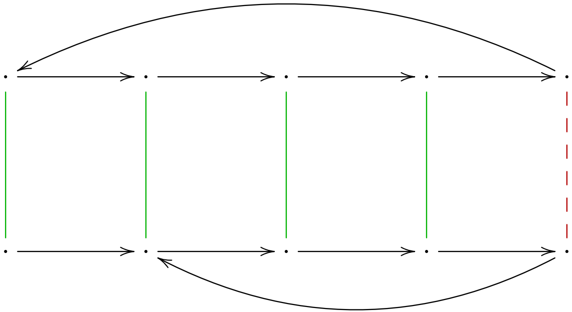
Deterministic case, naive algorithm: quadratic complexity



4 pairs

Checking language equivalence

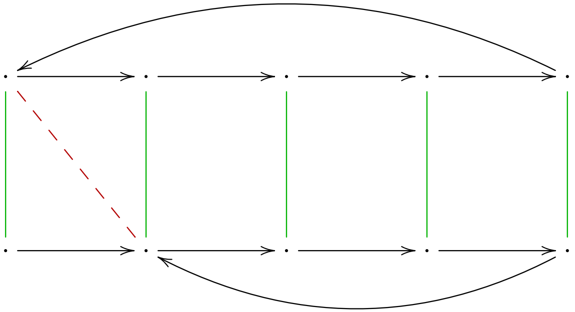
Deterministic case, naive algorithm: quadratic complexity



5 pairs

Checking language equivalence

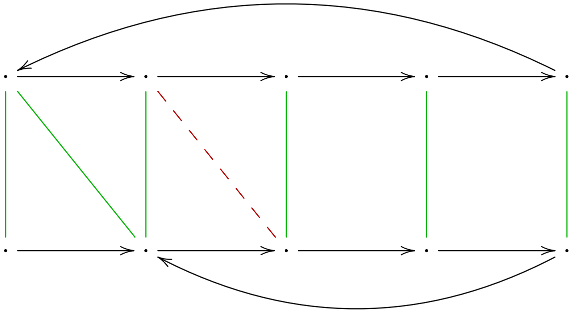
Deterministic case, naive algorithm: quadratic complexity



6 pairs

Checking language equivalence

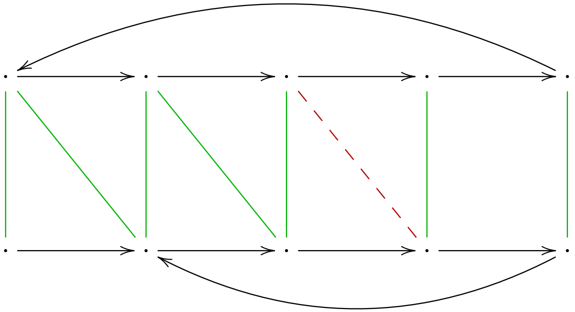
Deterministic case, naive algorithm: quadratic complexity



7 pairs

Checking language equivalence

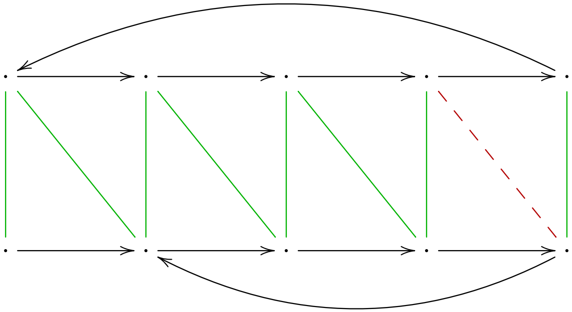
Deterministic case, naive algorithm: quadratic complexity



8 pairs

Checking language equivalence

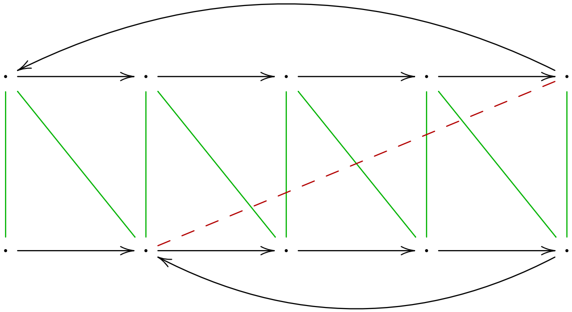
Deterministic case, naive algorithm: quadratic complexity



9 pairs

Checking language equivalence

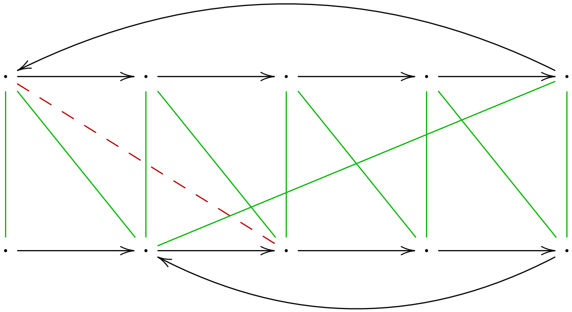
Deterministic case, naive algorithm: quadratic complexity



10 pairs

Checking language equivalence

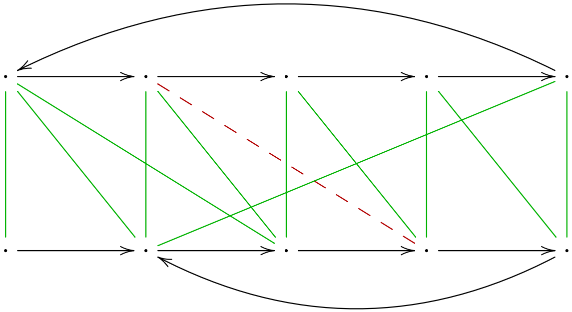
Deterministic case, naive algorithm: quadratic complexity



11 pairs

Checking language equivalence

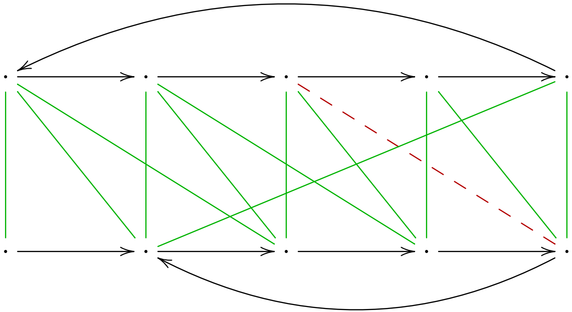
Deterministic case, naive algorithm: quadratic complexity



12 pairs

Checking language equivalence

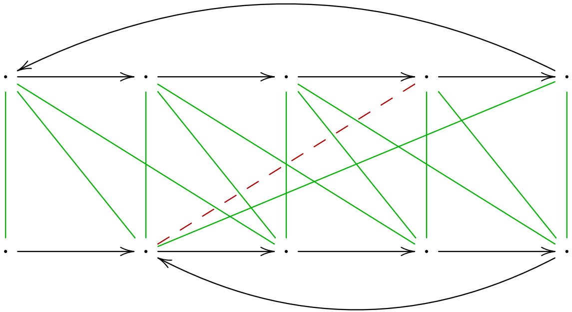
Deterministic case, naive algorithm: quadratic complexity



13 pairs

Checking language equivalence

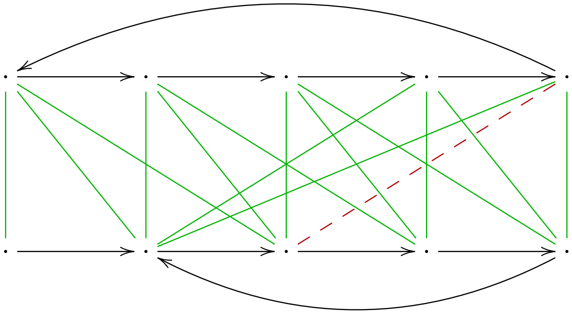
Deterministic case, naive algorithm: quadratic complexity



14 pairs

Checking language equivalence

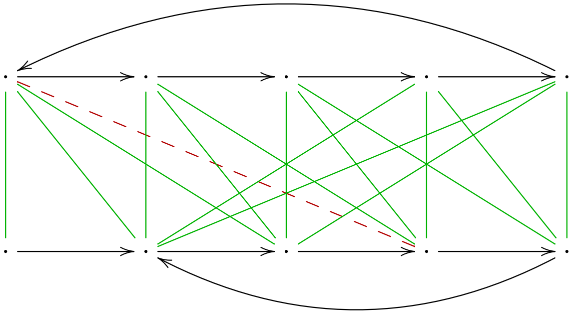
Deterministic case, naive algorithm: quadratic complexity



15 pairs

Checking language equivalence

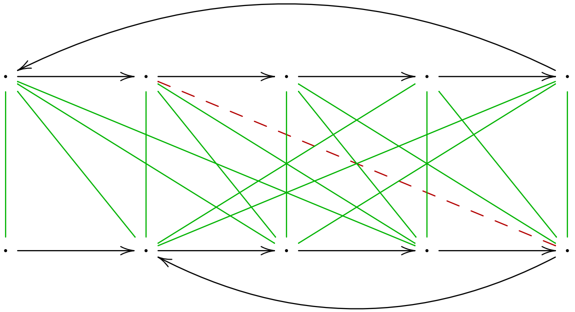
Deterministic case, naive algorithm: quadratic complexity



16 pairs

Checking language equivalence

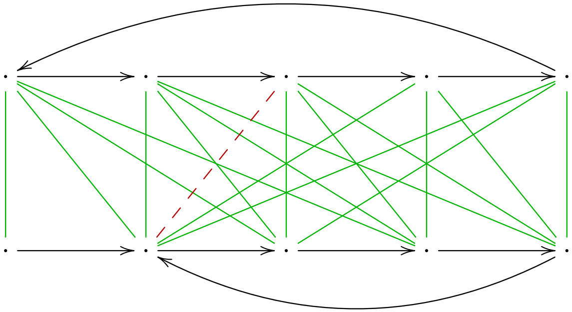
Deterministic case, naive algorithm: quadratic complexity



17 pairs

Checking language equivalence

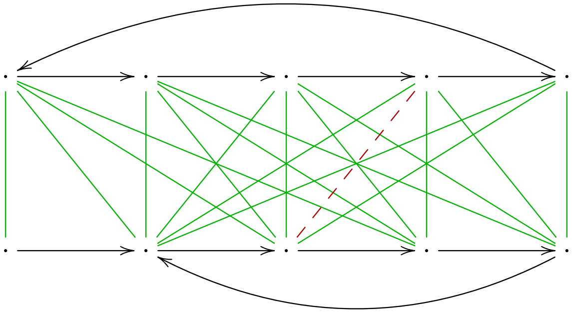
Deterministic case, naive algorithm: quadratic complexity



18 pairs

Checking language equivalence

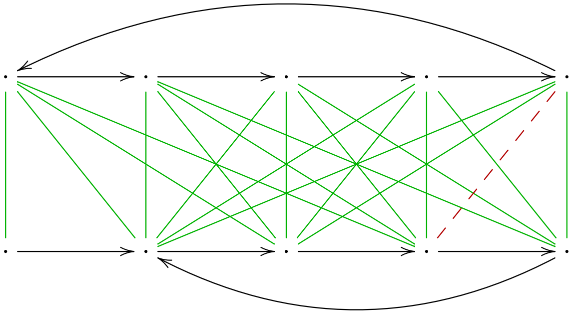
Deterministic case, naive algorithm: quadratic complexity



19 pairs

Checking language equivalence

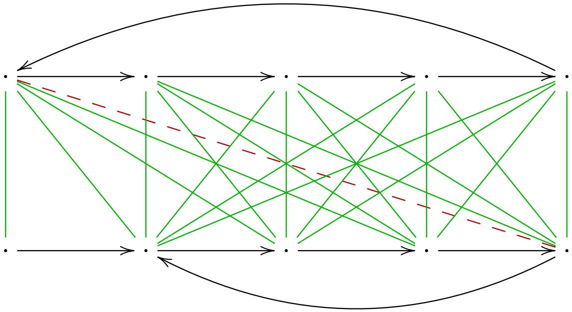
Deterministic case, naive algorithm: quadratic complexity



20 pairs

Checking language equivalence

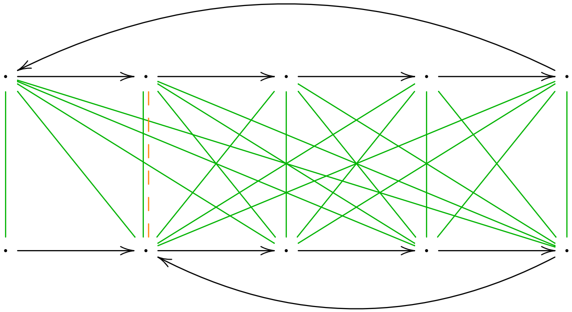
Deterministic case, naive algorithm: quadratic complexity



21 pairs

Checking language equivalence

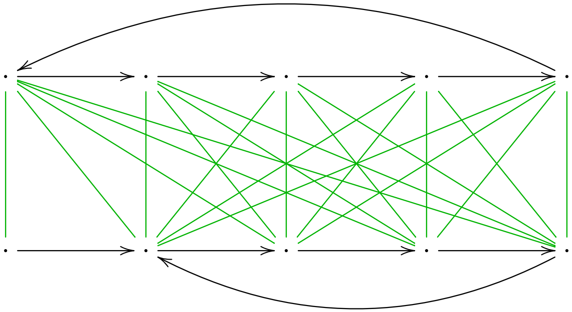
Deterministic case, naive algorithm: quadratic complexity



21 pairs

Checking language equivalence

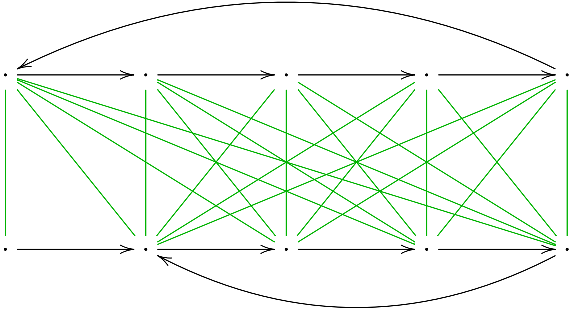
Deterministic case, naive algorithm: quadratic complexity



21 pairs

Checking language equivalence

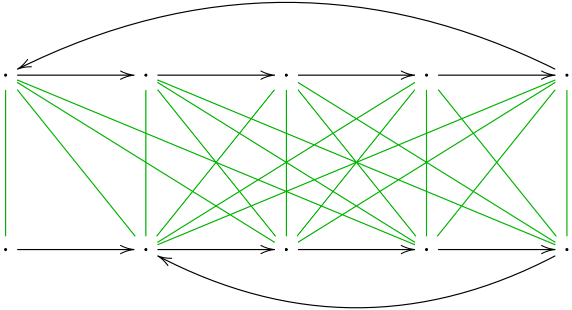
One can stop much earlier



21 pairs

Checking language equivalence

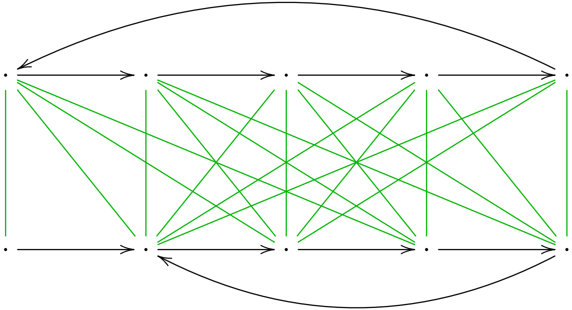
One can stop much earlier



~~21~~ 20 pairs

Checking language equivalence

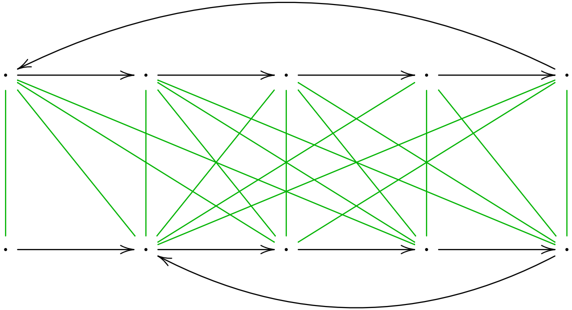
One can stop much earlier



~~21~~ 19 pairs

Checking language equivalence

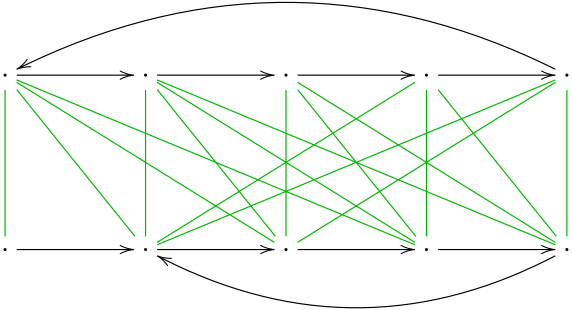
One can stop much earlier



~~21~~ 18 pairs

Checking language equivalence

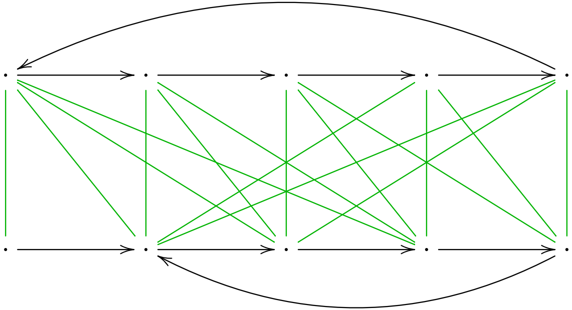
One can stop much earlier



~~21~~ 17 pairs

Checking language equivalence

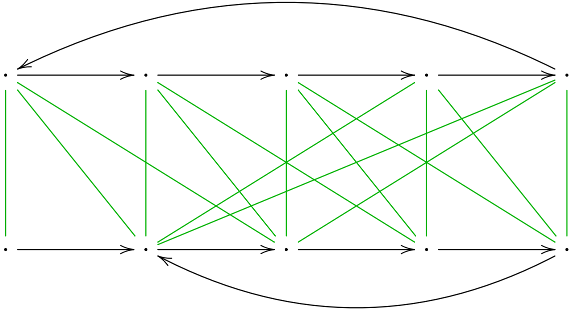
One can stop much earlier



~~21~~ 16 pairs

Checking language equivalence

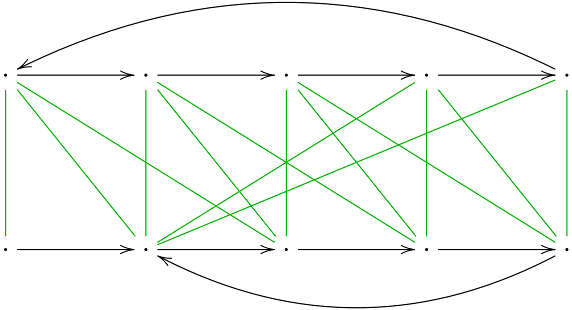
One can stop much earlier



~~21~~ 15 pairs

Checking language equivalence

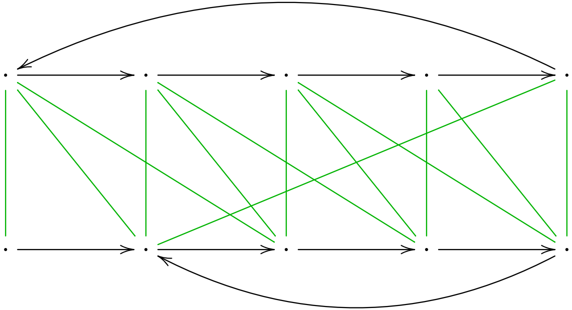
One can stop much earlier



~~21~~ 14 pairs

Checking language equivalence

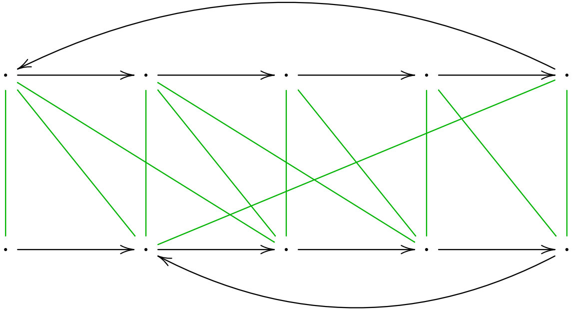
One can stop much earlier



~~21~~ 13 pairs

Checking language equivalence

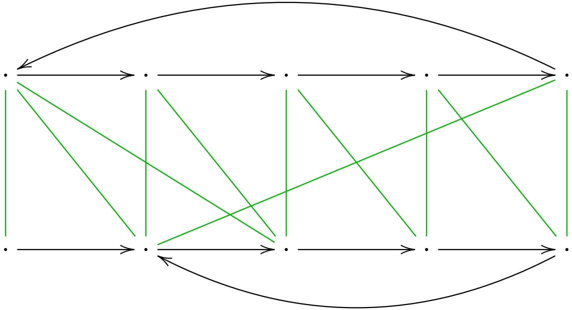
One can stop much earlier



~~21~~ 12 pairs

Checking language equivalence

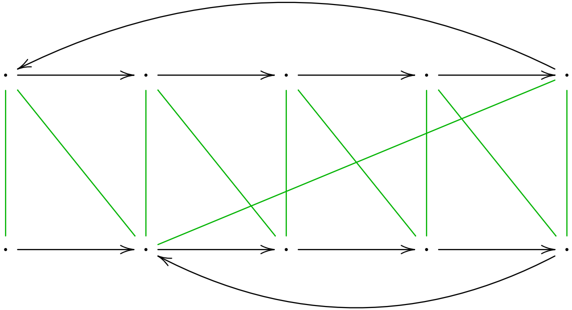
One can stop much earlier



~~21~~ 11 pairs

Checking language equivalence

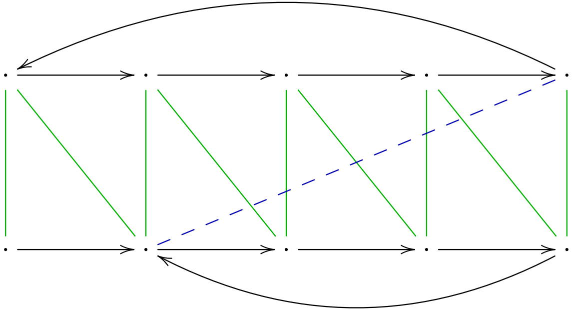
One can stop much earlier



~~21~~ 10 pairs

Checking language equivalence

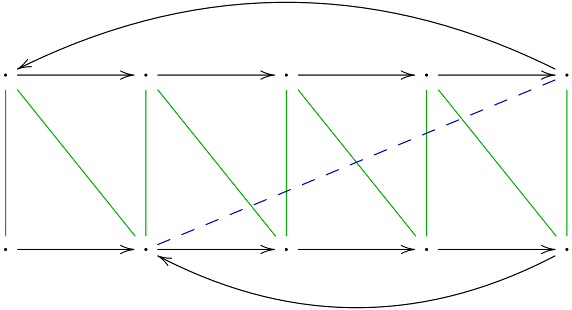
One can stop much earlier



~~21~~ 9 pairs

Checking language equivalence

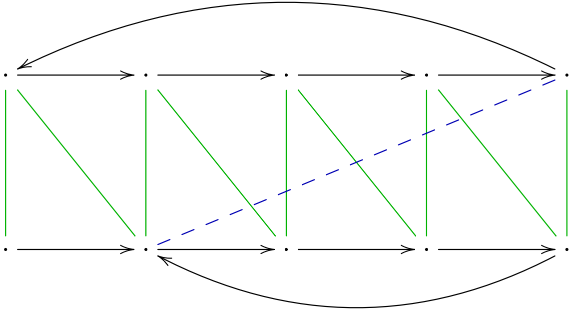
One can stop much earlier



[Hopcroft and Karp '71]

Checking language equivalence

One can stop much earlier



[Hopcroft and Karp '71]

Complexity: almost linear

[Tarjan '75]

Checking language equivalence

Correctness of HK algorithm, revisited:

- Denote by R^e the equivalence closure of R
- R is a bisimulation up to equivalence if $x \; R \; y$ entails
  - $o(x) = o(y)$;
  - for all a, $t_a(x) \; R^e \; t_a(y)$.

Checking language equivalence

Correctness of HK algorithm, revisited:

- Denote by R^e the equivalence closure of R
- R is a bisimulation up to equivalence if $x \mathrel{R} y$ entails
  - $o(x) = o(y)$;
  - for all a, $t_a(x) \mathrel{R^e} t_a(y)$.
- *Theorem:* $L(x) = L(y)$ iff
  there exists a bisimulation up to equivalence R, with $x \mathrel{R} y$

Checking language equivalence

Correctness of HK algorithm, revisited:

- Denote by R^e the equivalence closure of R
- R is a bisimulation up to equivalence if $x \; R \; y$ entails
  - $o(x) = o(y)$;
  - for all a, $t_a(x) \; R^e \; t_a(y)$.
- *Theorem:* $L(x) = L(y)$ iff
  there exists a bisimulation up to equivalence R, with $x \; R \; y$

Ten years before Milner and Park!

Outline

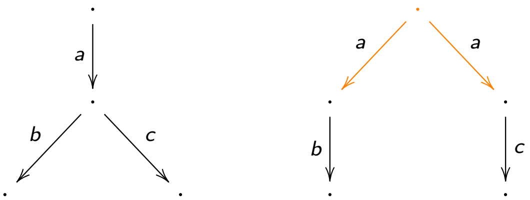
Deterministic Automata

Non-Deterministic Automata

Comparison with other algorithms

Non-Deterministic Automata

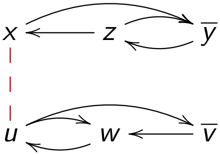
- Deterministic v.s. non-deterministic:



- Reduction to the deterministic case:
  - "powerset construction": $(S, t, o) \mapsto (\mathcal{P}(S), t^\#, o^\#)$
  - from states $(x, y, \dots)$ to sets of states $(X, Y, \dots)$

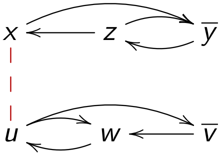
Checking language equivalence

Non-deterministic case: use Hopcroft and Karp on the fly:



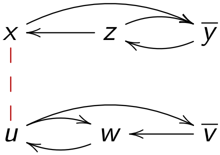
Checking language equivalence

Non-deterministic case: use Hopcroft and Karp on the fly:



Checking language equivalence

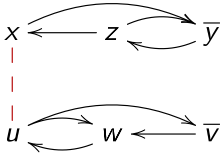
Non-deterministic case: use Hopcroft and Karp on the fly:

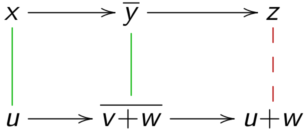


$$\begin{tikzcd}
x \arrow[r] \arrow[d, dash] & \overline{y} \arrow[d, dash, dashed] \\
u \arrow[r] & \overline{v+w}
\end{tikzcd}$$

Checking language equivalence

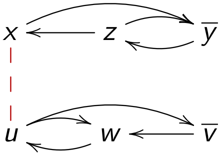
Non-deterministic case: use Hopcroft and Karp on the fly:





Checking language equivalence

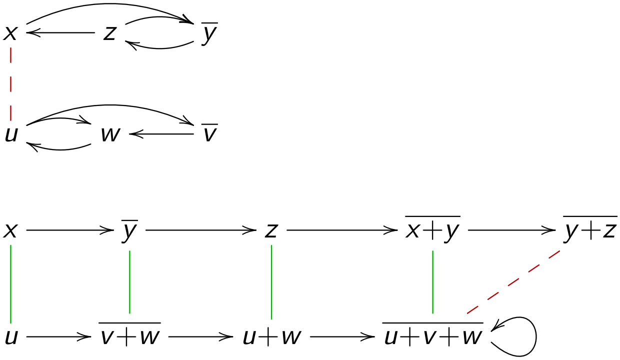
Non-deterministic case: use Hopcroft and Karp on the fly:



$$\begin{tikzcd}
x \arrow[r] \arrow[d, dash] & \overline{y} \arrow[r] \arrow[d, dash] & z \arrow[r] \arrow[d, dash] & \overline{x+y} \arrow[d, dash, dashed] \\
u \arrow[r] & \overline{v+w} \arrow[r] & u+w \arrow[r] & \overline{u+v+w}
\end{tikzcd}$$

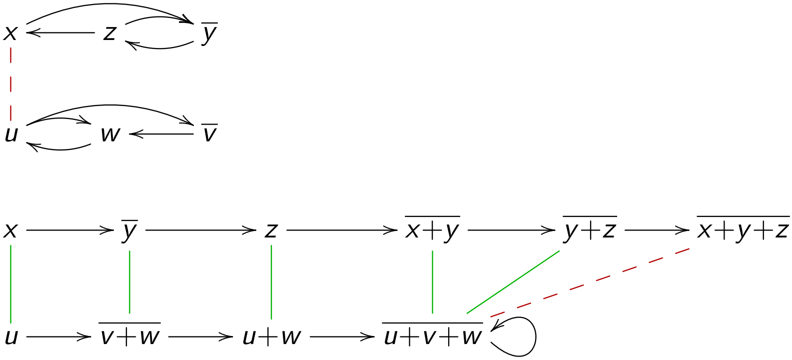
Checking language equivalence

Non-deterministic case: use Hopcroft and Karp on the fly:



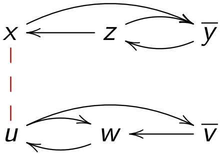
Checking language equivalence

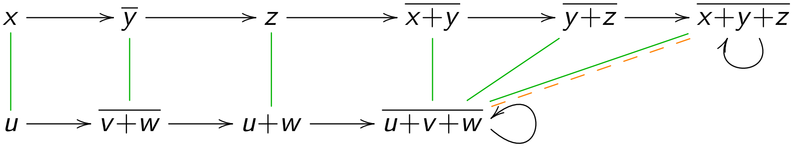
Non-deterministic case: use Hopcroft and Karp on the fly:



Checking language equivalence

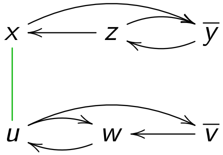
Non-deterministic case: use Hopcroft and Karp *on the fly*:

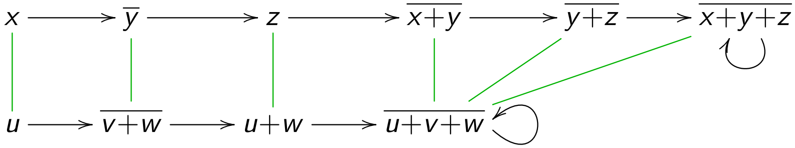




Checking language equivalence

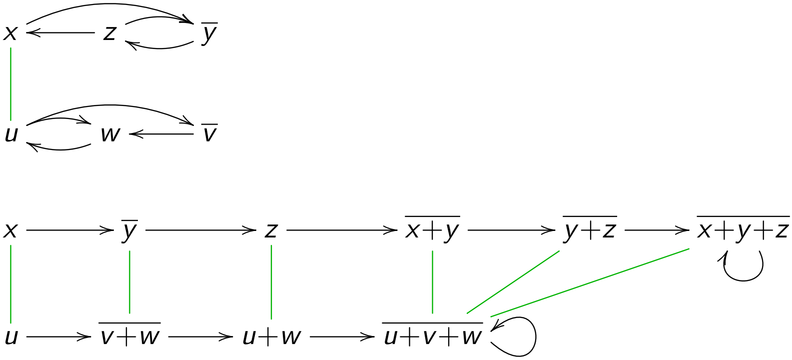
Non-deterministic case: use Hopcroft and Karp on the fly:





Checking language equivalence

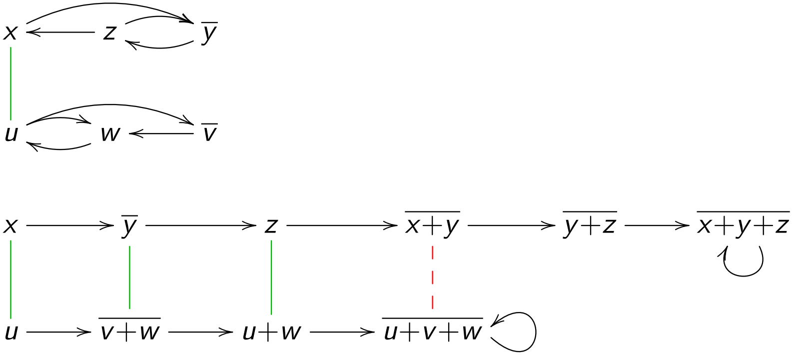
Non-deterministic case: use Hopcroft and Karp on the fly:



(correctness comes for free)

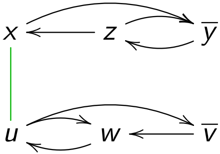
Checking language equivalence

One can do better:

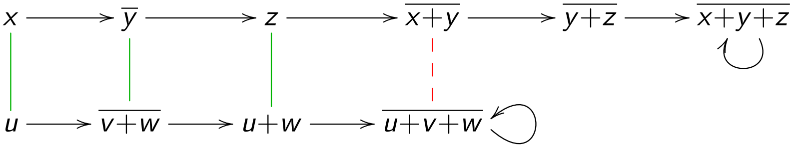


Checking language equivalence

One can do better:

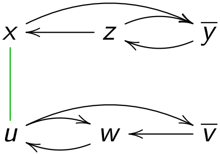


$$\begin{array}{rl} & (x,\ u)\\ + & (y,\ v+w)\\ \hline = & (x+y,\ u+v+w)\end{array}$$

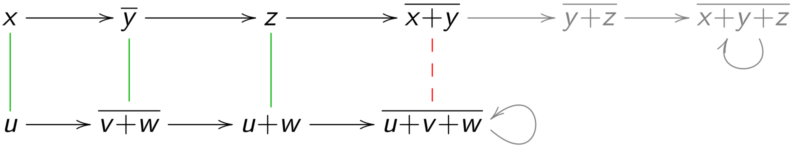


Checking language equivalence

One can do better:



$$\begin{array}{rl} & (x,\ u)\\ + & (y,\ v+w)\\ \hline = & (x+y,\ u+v+w)\end{array}$$



parts of the accessible subsets need not be explored

Correctness

- Denote by R^u the context closure of R:

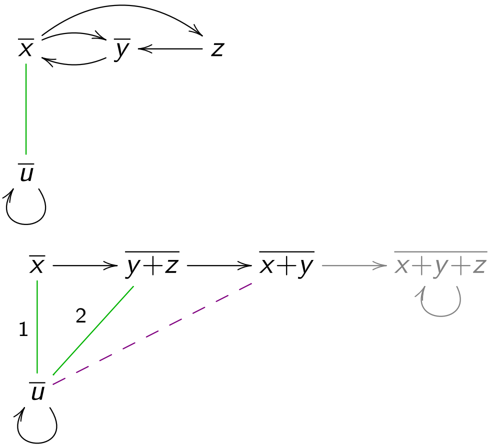
$$\frac{X \; R \; Y}{X \; R^u \; Y} \qquad\qquad \frac{X_1 \; R^u \; Y_1 \qquad X_2 \; R^u \; Y_2}{X_1 + X_2 \; R^u \; Y_1 + Y_2}$$

- R is a bisimulation up to context if $X \; R \; Y$ entails
  - $o^{\#}(X) = o^{\#}(Y)$;
  - for all a, $t_a^{\#}(X) \; R^u \; t_a^{\#}(Y)$.

- *Theorem:* $L(X) = L(Y)$ iff
  there exists a bisimulation up to context R, with $X \; R \; Y$

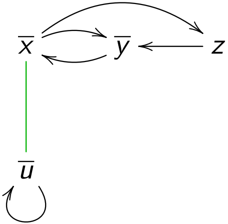
Checking language equivalence

One can do even better:

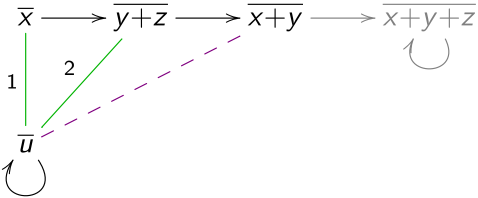


Checking language equivalence

One can do even better:



$$\begin{aligned} x+y &= u+y && (1)\\ &= y+z+y && (2)\\ &= y+z &&\\ &= u && (2) \end{aligned}$$



Correctness

- Let R^c denote the congruence closure of R

  (i.e., equivalence and context closure)

- R is a bisimulation up to congruence if $X\ R\ Y$ entails
  - $o^{\#}(X) = o^{\#}(Y)$;
  - for all a, $t_a^{\#}(X)\ R^c\ t_a^{\#}(Y)$.

- *Theorem:* $L(X) = L(Y)$ iff
  there exists a bisimulation up to congruence R, with $X\ R\ Y$

Congruence check

How to check whether $(X, Y) \in R^c$?

Congruence check

How to check whether $(X, Y) \in R^c$?

- R^c is an equivalence relation
- define a canonical element for each equivalence class

(take the largest set of the equivalence class)

Congruence check

How to check whether $(X, Y) \in R^c$?

- R^c is an equivalence relation
- define a canonical element for each equivalence class

  (take the largest set of the equivalence class)
- compute these canonical elements by set rewriting

  ($X, Y \to_R X+Y$ whenever $(X, Y) \in R$)

Congruence check

How to check whether $(X, Y) \in R^c$?

- R^c is an equivalence relation
- define a canonical element for each equivalence class

  (take the largest set of the equivalence class)
- compute these canonical elements by set rewriting

  ($X, Y \rightarrow_R X{+}Y$ whenever $(X, Y) \in R$)

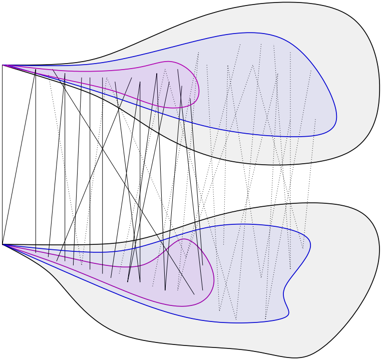
- *Theorem:* $(X, Y) \in R^c$ iff $X{\downarrow_R} = Y{\downarrow_R}$

Hopcroft and Karp with Contexts: HKC

- The resulting algorithm is called HKC, it combines
  - "up to equivalence" [HK'71, Milner'89]
  - "up to context" [MPW'92, Sangiorgi'95]

Hopcroft and Karp with Contexts: HKC

- The resulting algorithm is called HKC, it combines
  - "up to equivalence" [HK'71, Milner'89]
  - "up to context" [MPW'92, Sangiorgi'95]
- Good property: no need to explore all accessible states of the determinised automata



Outline

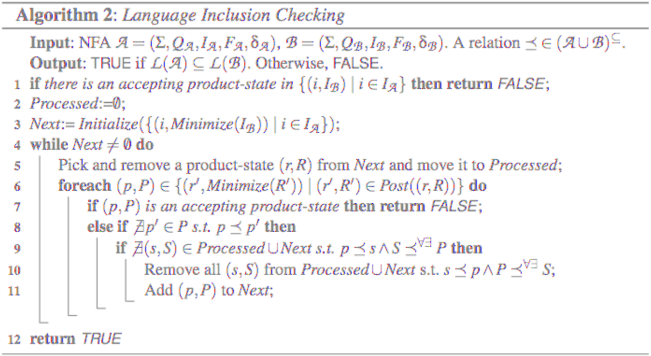
Deterministic Automata

Non-Deterministic Automata

Comparison with other algorithms

Antichain-based algorithms (AC)

- *“Antichains: a new algorithm for checking universality of finite automata”*
  De Wulf, Doyen, Henzinger, and Raskin, CAV ’06
  - Algorithms for language inclusion
  - Rough idea: iterate over an antichain to reach a fixpoint



Algorithm 2: *Language Inclusion Checking*

```
Input: NFA $\mathcal{A} = (\Sigma, Q_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}}, \delta_{\mathcal{A}})$, $\mathcal{B} = (\Sigma, Q_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}}, \delta_{\mathcal{B}})$. A relation $\preceq \in (\mathcal{A} \cup \mathcal{B})^{\subseteq}$.
Output: TRUE if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. Otherwise, FALSE.
if there is an accepting product-state in $\{(i, I_{\mathcal{B}}) \mid i \in I_{\mathcal{A}}\}$ then return FALSE;
Processed:=$\emptyset$;
Next:= Initialize($\{(i, Minimize(I_{\mathcal{B}})) \mid i \in I_{\mathcal{A}}\}$);
while Next $\neq \emptyset$ do
    Pick and remove a product-state $(r, R)$ from Next and move it to Processed;
    foreach $(p, P) \in \{(r', Minimize(R')) \mid (r', R') \in Post((r, R))\}$ do
        if $(p, P)$ is an accepting product-state then return FALSE;
        else if $\nexists p' \in P$ s.t. $p \preceq p'$ then
            if $\nexists (s, S) \in Processed \cup Next$ s.t. $p \preceq s \wedge S \preceq^{\forall\exists} P$ then
                Remove all $(s, S)$ from Processed $\cup$ Next s.t. $s \preceq p \wedge P \preceq^{\forall\exists} S$;
                Add $(p, P)$ to Next;
return TRUE
```

Antichain-based algorithms (AC)

- *"Antichains: a new algorithm for checking universality of finite automata"*
  De Wulf, Doyen, Henzinger, and Raskin, CAV '06
  - Algorithms for language inclusion
  - Rough idea: iterate over an antichain to reach a fixpoint

- *"Antichain Algorithms for Finite Automata"*
  Doyen and Raskin, TACAS '10
- *"When Simulation Meets Antichains"*
  Abdulla, Chen, Holík, Mayr, and Vojnar, TACAS '10

→ Exploit simulation preorders

(cf. Richard Mayr's talk)

Rephrasing antichains with coinduction

In the paper:

- Antichains (AC) rephrased as simulations up to upward closure
- One-to-one correspondence with bisimulations up to context
  (rather than bisimulations up to congruence for HKC)

Rephrasing antichains with coinduction

In the paper:

- Antichains (AC) rephrased as simulations up to upward closure
- One-to-one correspondence with bisimulations up to context
  (rather than bisimulations up to congruence for HKC)
- Exploiting simulation preorders in AC as an additional up-to technique
- Which can easily be adapted to HKC → HKC'

Comparing AC and HKC

1. Benchmarks

- Implementations
  - AC, AC': `libvata` (C++, for tree automata)
  - HK, HKC, HKC': homemade OCaml implementation

- Testcases
  - random automata (using [Tabakov, Vardi '05] model)
  - automata inclusions arising from model checking
    (the ones from [Abdulla, Chen, Holík, Mayr, and Vojnar '10])

Comparing AC and HKC

1. Benchmarks

- Implementations
  - AC, AC': `libvata` (C++, for tree automata)
  - HK, HKC, HKC': homemade OCaml implementation
- Testcases
  - random automata (using [Tabakov, Vardi '05] model)
  - automata inclusions arising from model checking
    (the ones from [Abdulla, Chen, Holík, Mayr, and Vojnar '10])

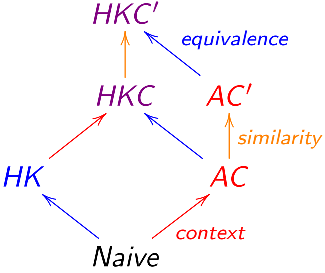
→ Up to two orders of magnitude faster than `libvata`

(lots of numbers in the paper)

Comparing AC and HKC

2. Formal analysis of the proof techniques

We established the following picture:

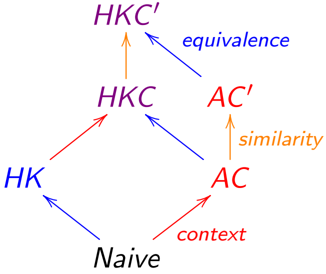


where an arrow means:

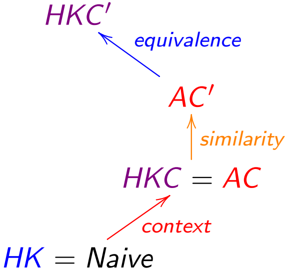
- the proof technique is at least as powerful
- there are examples yielding to an exponential improvement

Comparing AC and HKC

2. Formal analysis of the proof techniques

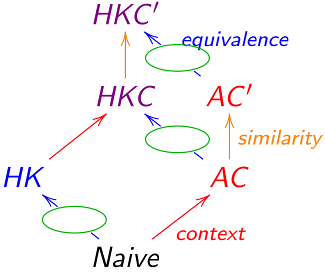


General case



Disjoint inclusion case

Intuition for HKC>AC in the equivalence case



HKC′

equivalence

HKC

AC′

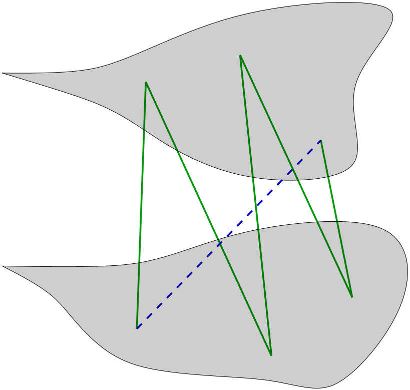
similarity

HK

AC

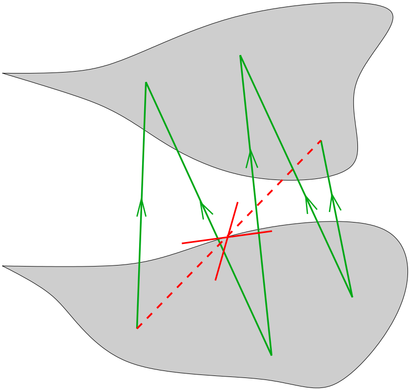
context

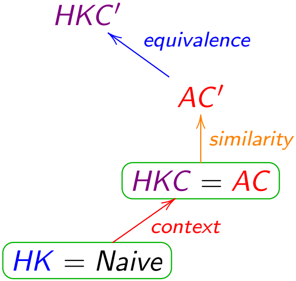
Naive



disjoint or non-disjoint equivalence check

Intuition for HKC=AC in the disjoint inclusion case





HKC'

equivalence

AC'

similarity

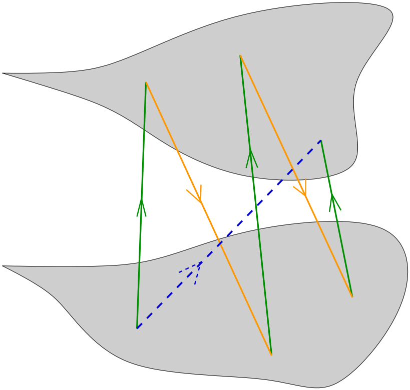
$HKC = AC$

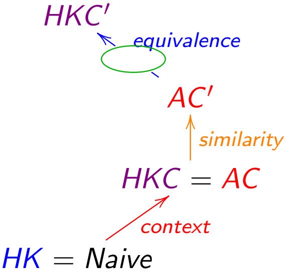
context

$HK = Naive$

disjoint inclusion check

Intuition for HKC'>AC' in the disjoint inclusion case





HKC'

equivalence

AC'

similarity

$HKC = AC$

context

$HK = Naive$

disjoint inclusion check, but with simulation preorder

Summary

- A new and efficient automata algorithm, exploiting ideas from concurrency theory: up-to techniques

  [Milner '89, Sangiorgi '95]

- A unified framework: coinduction, to rephrase and compare various algorithms from the literature
  - Hopcroft and Karp '71
  - Antichains '06
  - Antichains with similarity '10

- The algorithms can be tested online:

  `http://perso.ens-lyon.fr/damien.pous/hknt`