Checking NFA Equivalence with Bisimulations up to Congruence

Filippo Bonchi and Damien Pous

CNRS, LIP, ENS Lyon

POPL, Roma, 25.1.2013
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

\[
(R \cup S)^* = R^*; (S; R^*)^*
\]

(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:

\[ x \xrightarrow{a} y \xleftarrow{a} z \]

\[ u \xleftrightarrow{a} v \]
Checking language equivalence

Deterministic case, first algorithm:
Checking language equivalence

Deterministic case, first algorithm:

```
x a → y a → z
a ← u a ← v
```

\[\begin{array}{c}
x \xrightarrow{a} y \\
\downarrow \quad \quad \quad \downarrow \\
\quad u \xrightarrow{a} v
\end{array}\]
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
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
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

Complexity: almost linear

[Hopcroft and Karp '71]
[Tarjan '75]
Correctness of HK algorithm, revisited:

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

Ten years before Milner and Park!
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 $x R y$
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:

  ![Diagram](image)

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

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

\[
\begin{align*}
x & \longrightarrow z & \longrightarrow y \\
| &  & |
\end{align*}
\]

\[
\begin{align*}
u & \longrightarrow w & \longrightarrow v \\
| &  & |
\end{align*}
\]

(correctness comes for free)
Checking language equivalence

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

\[
\begin{align*}
  x &\rightarrow z \rightarrow y \\
  u &\rightarrow w \rightarrow v \\
  x &\leftarrow u
\end{align*}
\]
Checking language equivalence

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

```
x ← z ← y
|   |   |
|   |   |
u ← w ← v
```

```
x → y
|   |   |
|   |   |
u → v + w
```
Checking language equivalence

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

\[ x \rightarrow z \rightarrow \bar{y} \]
\[ u \rightarrow w \rightarrow \bar{v} \]

\[ x \rightarrow \bar{y} \rightarrow z \]
\[ u \rightarrow \bar{v} + w \rightarrow u + w \]
Checking language equivalence

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

\[
\begin{align*}
&\text{Non-deterministic case: use Hopcroft and Karp on the fly:} \\
&x \xrightarrow{} z \xleftarrow{} y \\
&\quad \uparrow \\
&\quad \uparrow \\
&u \xrightarrow{} w \xleftarrow{} v \\
&x \xrightarrow{} y \xrightarrow{} z \xrightarrow{} x+y \\
&\quad \uparrow \\
&u \xrightarrow{} v+w \xrightarrow{} u+w \xrightarrow{} u+v+w
\end{align*}
\]
Checking language equivalence

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

$x \leftarrow z \rightarrow \overline{y}$
$u \leftrightarrow w \rightarrow \overline{v}$

$x \rightarrow \overline{y} \rightarrow z \rightarrow \overline{x+y} \rightarrow \overline{y+z}$
$u \rightarrow \overline{v+w} \rightarrow u+w \rightarrow \overline{u+v+w}$
Checking language equivalence

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

\[\begin{align*}
x &\quad \xrightarrow{} \quad z \quad \xrightarrow{} \quad \overline{y} \\
u &\quad \xrightarrow{} \quad w \quad \xrightarrow{} \quad \overline{v}
\end{align*}\]

\[\begin{align*}
x &\quad \xrightarrow{} \quad \overline{y} \quad \xrightarrow{} \quad z \quad \xrightarrow{} \quad \overline{x+y} \quad \xrightarrow{} \quad \overline{y+z} \quad \xrightarrow{} \quad \overline{x+y+z} \\
u &\quad \xrightarrow{} \quad \overline{v+w} \quad \xrightarrow{} \quad u+w \quad \xrightarrow{} \quad \overline{u+v+w}
\end{align*}\]
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:

\[
x \rightarrow z \rightarrow \bar{y}\\
u \rightarrow w \rightarrow \bar{v}
\]

\[
x \rightarrow \bar{y} \rightarrow z \rightarrow \bar{x} + \bar{y} \rightarrow \bar{y} + \bar{z} \rightarrow \bar{x} + \bar{y} + \bar{z}\\
u \rightarrow \bar{v} + \bar{w} \rightarrow u + \bar{w} \rightarrow \bar{u} + \bar{v} + \bar{w}
\]

(correctness comes for free)
Checking language equivalence

One can do better:

\[
\begin{align*}
x & \rightarrow z & \rightarrow y \\
u & \rightarrow w & \rightarrow v
\end{align*}
\]

\[
\begin{align*}
x & \rightarrow \bar{y} & \rightarrow z & \rightarrow \bar{x+y} & \rightarrow \bar{y+z} & \rightarrow \bar{x+y+z} \\
u & \rightarrow \bar{v+w} & \rightarrow u+w & \rightarrow \bar{u+v+w}
\end{align*}
\]
Checking language equivalence

One can do better:

\[
\begin{align*}
(x, u) & \quad + \quad (y, v+w) \\
& = (x+y, u+v+w)
\end{align*}
\]
Checking language equivalence

One can do better:

\[(x, u) + (y, v+w) = (x+y, u+v+w)\]

parts of the accessible subsets need not be explored
Correctness

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

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

- $R$ 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:

\[
\begin{align*}
\bar{x} & \rightarrow \bar{y} \leftarrow \bar{z} \\
\bar{u} & \\
\rightarrow & \\
\downarrow & \\
\bar{x} & \rightarrow \bar{y} + \bar{z} \rightarrow \bar{x} + \bar{y} \rightarrow \bar{x} + \bar{y} + \bar{z}
\end{align*}
\]
Checking language equivalence

One can do even better:

\[
\begin{align*}
x + y &= u + y \quad (1) \\
&= y + z + y \quad (2) \\
&= y + z \\
&= u \quad (2)
\end{align*}
\]
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 $X R Y$. 
How to check whether \((X, Y) \in R^c\)?
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)
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 $(X, Y) \in R$)
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 $X \downarrow_R = Y \downarrow_R$
The resulting algorithm is called HKC, it combines
- “up to equivalence” [HK’71, Milner’89]
- “up to context” [MPW’92, Sangiorgi’95]
The resulting algorithm is called **HKC**, it combines

- “up to equivalence”
- “up to context”

Good property: no need to explore all accessible states of the determinised automata

[HK’71, Milner’89]
[MPW’92, Sangiorgi’95]
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 $A = (\Sigma, Q_A, I_A, F_A, \delta_A)$, $B = (\Sigma, Q_B, I_B, F_B, \delta_B)$. A relation $\preceq \in (A \cup B)^\subseteq$.
Output: TRUE if $L(A) \subseteq L(B)$. Otherwise, FALSE.
1 if there is an accepting product-state in $\{(i, I_B) \mid i \in I_A\}$ then return FALSE;
2 Processed:=\emptyset;
3 Next:= Initialize($\{(i, \text{Minimize}(I_B)) \mid i \in I_A\}$);
4 while Next $\neq \emptyset$ do
5 \hspace{1em} Pick and remove a product-state $(r, R)$ from Next and move it to Processed;
6 \hspace{1em} foreach $(p, P) \in \{(r', \text{Minimize}(R')) \mid (r', R') \in \text{Pos}(r, R)\}$ do
7 \hspace{2em} if $(p, P)$ is an accepting product-state then return FALSE;
8 \hspace{2em} else if $\not\exists p' \in P$ s.t. $p \preceq p'$ then
9 \hspace{3em} if $\not\exists (s, S) \in \text{Processed} \cup \text{Next}$ s.t. $p \preceq s$ \& $s \preceq \exists P$ then
10 \hspace{4em} Remove all $(s, S)$ from $\text{Processed} \cup \text{Next}$ s.t. $s \preceq p$ \& $p \preceq \exists S$;
11 \hspace{4em} Add $(p, P)$ to Next;
12 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:

\[ \text{HKC}' \xrightarrow{\text{equivalence}} \text{HKC} \xleftarrow{\text{context}} \text{HK} \xleftarrow{\text{similarity}} \text{AC} \xrightarrow{\text{context}} \text{Naive} \]

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

```
HKC'  equivalence  HKC  AC'  similarity
HKC   HKC         AC  HKC'  AC'     HKC = AC
HK     HK          AC  HKC'     AC'     HK = Naive
Naive  \\

General case
```

Disjoint inclusion case
Intuition for HKC $\triangleright$ AC in the equivalence case

disjoint or non-disjoint equivalence check
Intuition for $\text{HKC} = \text{AC}$ in the disjoint inclusion case

$\text{HK} = \text{Naive}$

$\text{HKC'}$ equivalence
$\text{AC'}$
similarity
$\text{HKC} = \text{AC}$

disjoint inclusion check
Intuition for HKC’ $\triangleright$ AC’ in the disjoint inclusion case

$\text{disjoint inclusion check, but with simulation preorder}$
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