Efficient Inclusion Checking on Explicit and Semi-Symbolic Tree Automata

Lukáš Holík[1,2] **Ondřej Lengál**[1] Jiří Šimáček[1,3] Tomáš Vojnar[1]

[1]Brno University of Technology, Czech Republic
[2]Uppsala University, Sweden
[3]VERIMAG, UJF/CNRS/INPG, Gières, France

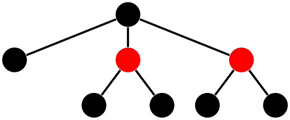
October 13, 2011

Outline

Trees

Very popular in computer science:

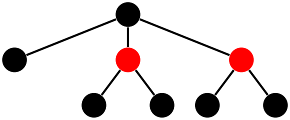
- data structures,
- computer network topologies,
- distributed protocols, . . .



Trees

Very popular in computer science:

- data structures,
- computer network topologies,
- distributed protocols, . . .



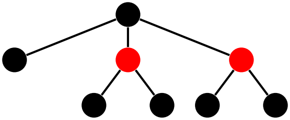
In formal verification:

- encoding of complex data structures

Trees

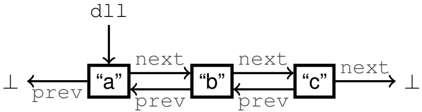
Very popular in computer science:

- data structures,
- computer network topologies,
- distributed protocols, . . .



In formal verification:

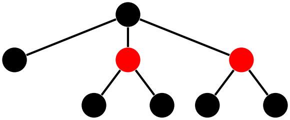
- encoding of complex data structures
  - e.g., doubly linked lists



Trees

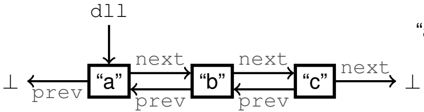
Very popular in computer science:

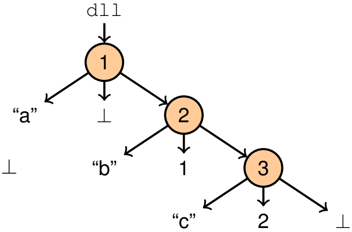
- data structures,
- computer network topologies,
- distributed protocols, . . .



In formal verification:

- encoding of complex data structures
  - e.g., doubly linked lists

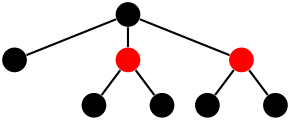




Trees

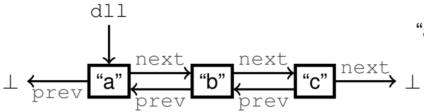
Very popular in computer science:

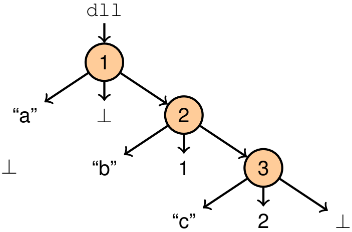
- data structures,
- computer network topologies,
- distributed protocols, . . .



In formal verification:

- encoding of complex data structures
  - e.g., doubly linked lists





  - . . .

Tree Automata

Finite Tree Automaton (TA): $\mathcal{A} = (Q, \Sigma, \Delta, F)$

- extension of finite automaton to trees:
  - Q ... set of states,
  - Σ ... finite alphabet of symbols with arity,
  - Δ ... set of transitions in the form of $p \xrightarrow{a} (q_1, \ldots, q_n)$,
  - F ... set of final states.

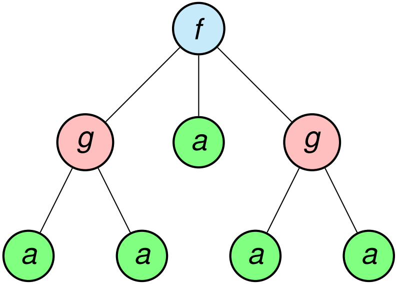
Tree Automata

Finite Tree Automaton (TA): $\mathcal{A} = (Q, \Sigma, \Delta, F)$

- extension of finite automaton to trees:
  - Q ... set of states,
  - Σ ... finite alphabet of symbols with arity,
  - Δ ... set of transitions in the form of $p \xrightarrow{a} (q_1, \ldots, q_n)$,
  - F ... set of final states.

Example:

$$\Delta = \{ \; \underline{s} \xrightarrow{f} (r, q, r), \; r \xrightarrow{g} (q, q), \; q \xrightarrow{a} \; \}$$

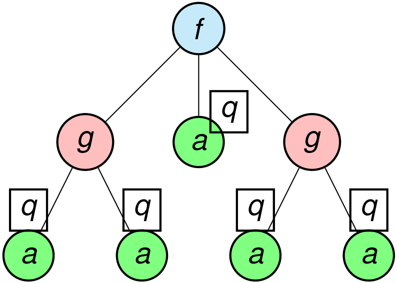


Finite Tree Automaton (TA): $\mathcal{A} = (Q, \Sigma, \Delta, F)$

- extension of finite automaton to trees:
  - Q ... set of states,
  - Σ ... finite alphabet of symbols with arity,
  - Δ ... set of transitions in the form of $p \xrightarrow{a} (q_1, \ldots, q_n)$,
  - F ... set of final states.

Example:

$$\Delta = \{ \quad \underline{s} \xrightarrow{f} (r, q, r), \quad r \xrightarrow{g} (q, q), \quad q \xrightarrow{a} \quad \}$$



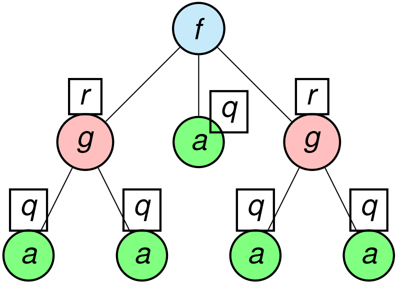
Tree Automata

Finite Tree Automaton (TA): $\mathcal{A} = (Q, \Sigma, \Delta, F)$

- extension of finite automaton to trees:
  - Q ... set of states,
  - Σ ... finite alphabet of symbols with arity,
  - Δ ... set of transitions in the form of $p \xrightarrow{a} (q_1, \ldots, q_n)$,
  - F ... set of final states.

Example:

$$\Delta = \{ \; \underline{s} \xrightarrow{f} (r, q, r), \quad r \xrightarrow{g} (q, q), \quad q \xrightarrow{a} \; \}$$



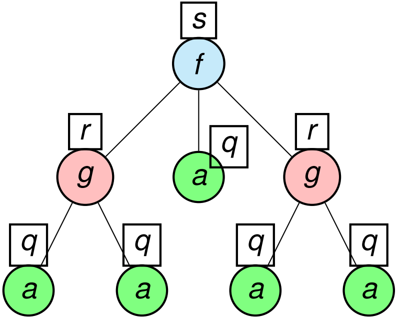
Tree Automata

Finite Tree Automaton (TA): $\mathcal{A} = (Q, \Sigma, \Delta, F)$

- extension of finite automaton to trees:
  - Q ... set of states,
  - Σ ... finite alphabet of symbols with arity,
  - Δ ... set of transitions in the form of $p \xrightarrow{a} (q_1, \ldots, q_n)$,
  - F ... set of final states.

Example:

$$\Delta = \{ \quad \underline{s} \xrightarrow{f} (r, q, r), \quad r \xrightarrow{g} (q, q), \quad q \xrightarrow{a} \quad \}$$



Tree Automata

Tree Automata

- can represent (infinite) sets of trees with regular structure,
- used in XML DBs, language processing, . . . ,
- . . . formal verification, decision procedures of some logics, . . .

Tree Automata

Tree Automata

- can represent (infinite) sets of trees with regular structure,
- used in XML DBs, language processing, . . . ,
- . . . formal verification, decision procedures of some logics, . . .

Tree automata in FV:

- often large due to determinisation

Tree Automata

Tree Automata

- can represent (infinite) sets of trees with regular structure,
- used in XML DBs, language processing, . . . ,
- . . . formal verification, decision procedures of some logics, . . .

Tree automata in FV:

- often large due to determinisation
  - often advantageous to use non-deterministic tree automata,

Tree Automata

Tree Automata

- can represent (infinite) sets of trees with regular structure,
- used in XML DBs, language processing, . . . ,
- . . . formal verification, decision procedures of some logics, . . .

Tree automata in FV:

- often large due to determinisation
  - often advantageous to use non-deterministic tree automata,
  - manipulate them without determinisation,

Tree Automata

Tree Automata

- can represent (infinite) sets of trees with regular structure,
- used in XML DBs, language processing, . . . ,
- . . . formal verification, decision procedures of some logics, . . .

Tree automata in FV:

- often large due to determinisation
  - often advantageous to use non-deterministic tree automata,
  - manipulate them without determinisation,
  - even for operations such as language inclusion (ARTMC, . . .),

Tree Automata

Tree Automata

- can represent (infinite) sets of trees with regular structure,
- used in XML DBs, language processing, . . . ,
- . . . formal verification, decision procedures of some logics, . . .

Tree automata in FV:

- often large due to determinisation
  - often advantageous to use non-deterministic tree automata,
  - manipulate them without determinisation,
  - even for operations such as language inclusion (ARTMC, . . .),
- handling large alphabets (MSO, WSkS).

Approaches to Checking Tree Automata Inclusion

- Approximate

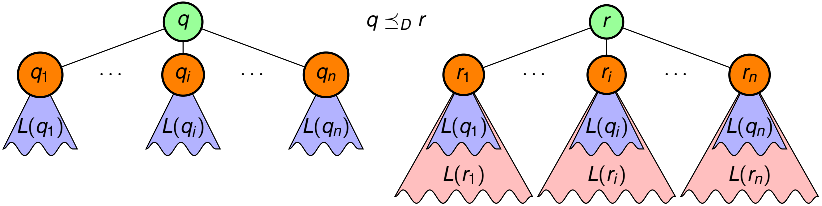
Approaches to Checking Tree Automata Inclusion

- Approximate
  - **downward simulation**: $q \preceq_D r \implies$
    - $\forall f \in \Sigma : q \xrightarrow{f} (q_1, \ldots, q_n) \implies r \xrightarrow{f} (r_1, \ldots, r_n), \forall 1 \leq i \leq n : q_i \preceq_D r_i$

Approaches to Checking Tree Automata Inclusion

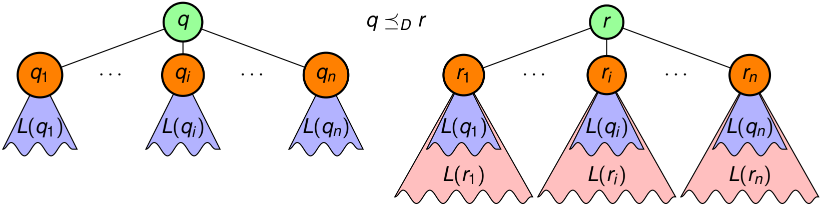
- Approximate
  - **downward simulation**: $q \preceq_D r \implies$

    ▸ $\forall f \in \Sigma : q \xrightarrow{f} (q_1, \ldots, q_n) \implies r \xrightarrow{f} (r_1, \ldots, r_n), \forall 1 \leq i \leq n : q_i \preceq_D r_i$



Approaches to Checking Tree Automata Inclusion

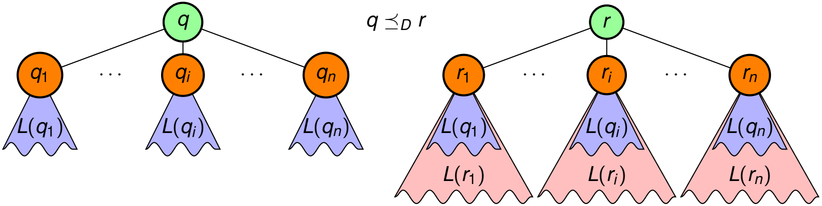
- Approximate
  - **downward simulation**: $q \preceq_D r \implies$
    - $\forall f \in \Sigma : q \xrightarrow{f} (q_1, \ldots, q_n) \implies r \xrightarrow{f} (r_1, \ldots, r_n), \forall 1 \leq i \leq n : q_i \preceq_D r_i$



(under-approximation: $q \preceq_D r \implies \mathcal{L}(q) \subseteq \mathcal{L}(r)$)

Approaches to Checking Tree Automata Inclusion

- Approximate
  - **downward simulation**: $q \preceq_D r \implies$
    - $\forall f \in \Sigma : q \xrightarrow{f} (q_1, \ldots, q_n) \implies r \xrightarrow{f} (r_1, \ldots, r_n), \forall 1 \leq i \leq n : q_i \preceq_D r_i$



(under-approximation: $q \preceq_D r \implies \mathcal{L}(q) \subseteq \mathcal{L}(r)$)

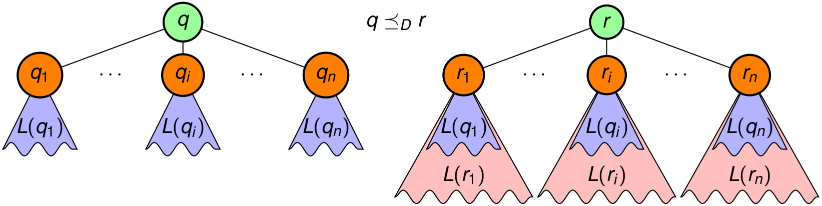
  - upward simulation

Approaches to Checking Tree Automata Inclusion

Approximate

- **downward simulation**: $q \preceq_D r \implies$

  ▸ $\forall f \in \Sigma : q \xrightarrow{f} (q_1, \ldots, q_n) \implies r \xrightarrow{f} (r_1, \ldots, r_n), \forall 1 \leq i \leq n : q_i \preceq_D r_i$



(under-approximation: $q \preceq_D r \implies \mathcal{L}(q) \subseteq \mathcal{L}(r)$)

- upward simulation
  - ▸ not compatible with language inclusion,
  - ▸ but can be used to speed up exact checking

Approaches to Checking Tree Automata Inclusion

Exact: **EXPTIME-complete . . .**

1

2

3

Approaches to Checking Tree Automata Inclusion

Exact: **EXPTIME-complete** . . .

- . . . but there are some highly efficient heuristics:

1

2

3

Approaches to Checking Tree Automata Inclusion

Exact: **EXPTIME-complete** …

- …but there are some highly efficient heuristics:
  - antichains[1]

[1] M. De Wulf, L. Doyen, T. Henzinger, J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of FA. CAV'06.
2

3

Approaches to Checking Tree Automata Inclusion

Exact: **EXPTIME-complete** ...

- ...but there are some highly efficient heuristics:
  - antichains[1]
  - antichains combined with simulation[2,3]

[1] M. De Wulf, L. Doyen, T. Henzinger, J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of FA. CAV'06.

[2] L. Doyen, J.-F. Raskin. Antichain Algorithms for Finite Automata. TACAS'10.

[3] P. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, T. Vojnar. When Simulation Meets Antichains. TACAS'10.

Inclusion Checking

Textbook algorithm for checking $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ on TA:

Inclusion Checking

Textbook algorithm for checking $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ on TA:

1. Bottom-up determinise $\mathcal{A}_B \to \mathcal{A}_B^D$.
   - Bottom-up DTA and NTA have the same power; not the same for top-down DTA.

Inclusion Checking

Textbook algorithm for checking $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ on TA:

1. Bottom-up determinise $\mathcal{A}_B \to \mathcal{A}_B^D$.
   - Bottom-up DTA and NTA have the same power; not the same for top-down DTA.
2. Complement $\mathcal{A}_B^D \to \overline{\mathcal{A}_B^D}$.

Inclusion Checking

Textbook algorithm for checking $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ on TA:

1. Bottom-up determinise $\mathcal{A}_B \to \mathcal{A}_B^D$.
   - Bottom-up DTA and NTA have the same power; not the same for top-down DTA.
2. Complement $\mathcal{A}_B^D \to \overline{\mathcal{A}_B^D}$.
3. Check $\mathcal{A}_S \cap \overline{\mathcal{A}_B^D} = \emptyset$.

Inclusion Checking

Textbook algorithm for checking $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ on TA:

1. Bottom-up determinise $\mathcal{A}_B \to \mathcal{A}_B^D$. (exponential explosion!)
   - Bottom-up DTA and NTA have the same power; not the same for top-down DTA.
2. Complement $\mathcal{A}_B^D \to \overline{\mathcal{A}_B^D}$.
3. Check $\mathcal{A}_S \cap \overline{\mathcal{A}_B^D} = \emptyset$.

Upward Inclusion Checking

On-the-fly approach:

Upward Inclusion Checking

On-the-fly approach:

1. Traverse $\mathcal{A}_S$ and $\mathcal{A}_B$ in parallel, bottom-up.

Upward Inclusion Checking

On-the-fly approach:

1. Traverse $\mathcal{A}_S$ and $\mathcal{A}_B$ in parallel, bottom-up.
2. Maintain a workset W of pairs (q, P), where $q \in Q_S, P \subseteq Q_B$.

Upward Inclusion Checking

On-the-fly approach:

1. Traverse $\mathcal{A}_S$ and $\mathcal{A}_B$ in parallel, bottom-up.
2. Maintain a workset W of pairs (q, P), where $q \in Q_S, P \subseteq Q_B$.
3. Generate tuples $(q_1, \ldots, q_n)$ and $(P_1, \ldots, P_n)$,
   - where $(q_1, P_1), \ldots, (q_n, P_n) \in W$.

Upward Inclusion Checking

On-the-fly approach:

1. Traverse $\mathcal{A}_S$ and $\mathcal{A}_B$ in parallel, bottom-up.
2. Maintain a workset W of pairs (q, P), where $q \in Q_S, P \subseteq Q_B$.
3. Generate tuples $(q_1, \ldots, q_n)$ and $(P_1, \ldots, P_n)$,
   - where $(q_1, P_1), \ldots, (q_n, P_n) \in W$.
4. $\forall f \in \Sigma$, generate (s, T), s.t. $(q_1, \ldots, q_n) \xrightarrow{f} s$, $(P_1, \ldots, P_n) \xrightarrow{f} T$.

Upward Inclusion Checking

On-the-fly approach:

1. Traverse $\mathcal{A}_S$ and $\mathcal{A}_B$ in parallel, bottom-up.
2. Maintain a workset W of pairs (q, P), where $q \in Q_S, P \subseteq Q_B$.
3. Generate tuples $(q_1, \ldots, q_n)$ and $(P_1, \ldots, P_n)$,
   - where $(q_1, P_1), \ldots, (q_n, P_n) \in W$.
4. $\forall f \in \Sigma$, generate (s, T), s.t. $(q_1, \ldots, q_n) \xrightarrow{f} s$, $(P_1, \ldots, P_n) \xrightarrow{f} T$.
5. If you encounter (f, R), where $f \in F_S, R \cap F_B = \emptyset$, return false.

Upward Inclusion Checking

On-the-fly approach:

1. Traverse $\mathcal{A}_S$ and $\mathcal{A}_B$ in parallel, bottom-up.
2. Maintain a workset W of pairs (q, P), where $q \in Q_S, P \subseteq Q_B$.
3. Generate tuples $(q_1, \ldots, q_n)$ and $(P_1, \ldots, P_n)$,
   - where $(q_1, P_1), \ldots, (q_n, P_n) \in W$.
4. $\forall f \in \Sigma$, generate (s, T), s.t. $(q_1, \ldots, q_n) \xrightarrow{f} s$, $(P_1, \ldots, P_n) \xrightarrow{f} T$.
5. If you encounter (f, R), where $f \in F_S, R \cap F_B = \emptyset$, return false.
6. If no new pairs are found, return true.

Upward Inclusion Checking

Optimisations:

Upward Inclusion Checking

Optimisations:

1. use antichains: maintain only such pairs which are sufficient to encounter a counterexample (if it exists):

Upward Inclusion Checking

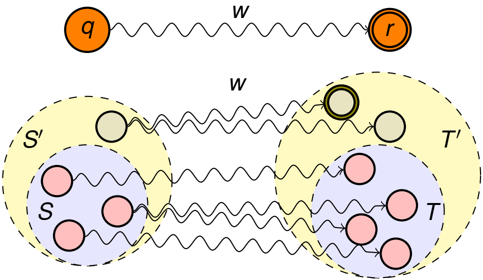
Optimisations:

1. use antichains: maintain only such pairs which are sufficient to encounter a counterexample (if it exists):
   - if $S \subseteq S'$ and both (q, S) and (q, S') are in workset W,
   - remove (q, S') from workset W.

Upward Inclusion Checking

Optimisations:

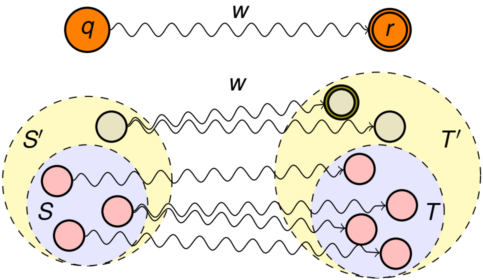
1. use antichains: maintain only such pairs which are sufficient to encounter a counterexample (if it exists):
   - if $S \subseteq S'$ and both (q, S) and (q, S') are in workset W,
   - remove (q, S') from workset W.



Upward Inclusion Checking

Optimisations:

1. use antichains: maintain only such pairs which are sufficient to encounter a counterexample (if it exists):
   - if $S \subseteq S'$ and both (q, S) and (q, S') are in workset W,
   - remove (q, S') from workset W.



2. use simulation to further prune the searched space.

Upward Inclusion Checking

Advantages:

Upward Inclusion Checking

Advantages:

- Straightforward extension of the antichain algorithm for FA. ☺

Upward Inclusion Checking

Advantages:

- Straightforward extension of the antichain algorithm for FA. ☺

Disadvantages:

Upward Inclusion Checking

Advantages:

- Straightforward extension of the antichain algorithm for FA. ☺

Disadvantages:

- Generating tuples is expensive. ☹

Upward Inclusion Checking

Advantages:

- Straightforward extension of the antichain algorithm for FA. ☺

Disadvantages:

- Generating tuples is expensive. ☹
- The counterexample may be at root . . . takes long to get there. ☹

Upward Inclusion Checking

Advantages:

- Straightforward extension of the antichain algorithm for FA. ☺

Disadvantages:

- Generating tuples is expensive. ☹
- The counterexample may be at root . . . takes long to get there. ☹
- Upward simulation $\rightarrow$ hard to compute and too strong. ☹

Upward Inclusion Checking

Advantages:

- Straightforward extension of the antichain algorithm for FA. ☺

Disadvantages:

- Generating tuples is expensive. ☹
- The counterexample may be at root . . . takes long to get there. ☹
- Upward simulation $\rightarrow$ hard to compute and too strong. ☹
- Not compatible with downward simulation (easy & rich). ☹

Downward Inclusion Checking

Downward Inclusion Checking

4

Downward Inclusion Checking

- inspired by XML Schema containment checking[4],

[4] H. Hosoya, J. Vouillon, B. C. Pierce. Regular Expression Types for XML. ACM Trans. Program. Lang. Sys., 27, 2005.

Downward Inclusion Checking

Downward Inclusion Checking

- inspired by XML Schema containment checking[4],
- does not follow the classic schema of inclusion algorithms,

[4] H. Hosoya, J. Vouillon, B. C. Pierce. Regular Expression Types for XML. ACM Trans. Program. Lang. Sys., 27, 2005.

Downward Inclusion Checking

Downward Inclusion Checking

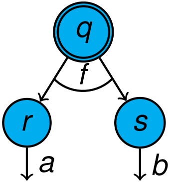
- inspired by XML Schema containment checking[4],
- does not follow the classic schema of inclusion algorithms,
- uses antichains and downward simulation.

[4] H. Hosoya, J. Vouillon, B. C. Pierce. Regular Expression Types for XML. ACM Trans. Program. Lang. Sys., 27, 2005.

Downward Inclusion Checking

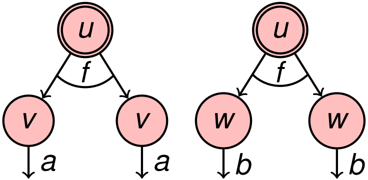
$\mathcal{A}_S$

$\underline{q} \xrightarrow{f} (r, s)$
$r \xrightarrow{a}$
$s \xrightarrow{b}$



$\mathcal{A}_B$

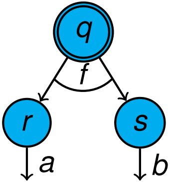
$\underline{u} \xrightarrow{f} (v, v)$
$\underline{u} \xrightarrow{f} (w, w)$
$v \xrightarrow{a}$
$w \xrightarrow{b}$



Downward Inclusion Checking

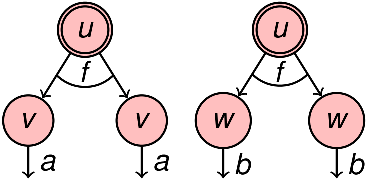
$\mathcal{A}_S$

$\underline{q} \xrightarrow{f} (r, s)$
$r \xrightarrow{a}$
$s \xrightarrow{b}$



$\mathcal{A}_B$

$\underline{u} \xrightarrow{f} (v, v)$
$\underline{u} \xrightarrow{f} (w, w)$
$v \xrightarrow{a}$
$w \xrightarrow{b}$



$\mathcal{L}(q) \subseteq \mathcal{L}(u)$ if and only if

$$\mathcal{L}(r) \times \mathcal{L}(s) \subseteq (\mathcal{L}(v) \times \mathcal{L}(v)) \cup (\mathcal{L}(w) \times \mathcal{L}(w))$$

(language inclusion of tuples!)

Checking language inclusion of tuples

Note that in general

$$(\mathcal{L}(v_1) \times \mathcal{L}(v_2)) \cup (\mathcal{L}(w_1) \times \mathcal{L}(w_2)) \neq (\mathcal{L}(v_1) \cup \mathcal{L}(w_1)) \times (\mathcal{L}(v_2) \cup \mathcal{L}(w_2))$$

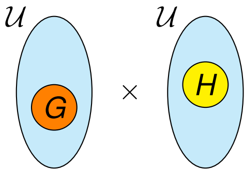
Checking language inclusion of tuples

Note that in general

$$(\mathcal{L}(v_1)\times\mathcal{L}(v_2))\cup(\mathcal{L}(w_1)\times\mathcal{L}(w_2)) \neq (\mathcal{L}(v_1)\cup\mathcal{L}(w_1))\times(\mathcal{L}(v_2)\cup\mathcal{L}(w_2))$$

However, for universe $\mathcal{U}$ and $G, H \subseteq \mathcal{U}$:

$$G \times H = (G \times \mathcal{U}) \cap (\mathcal{U} \times H)$$



Checking language inclusion of tuples

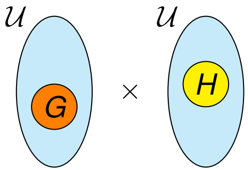
Note that in general

$$(\mathcal{L}(v_1)\times\mathcal{L}(v_2))\cup(\mathcal{L}(w_1)\times\mathcal{L}(w_2)) \neq (\mathcal{L}(v_1)\cup\mathcal{L}(w_1))\times(\mathcal{L}(v_2)\cup\mathcal{L}(w_2))$$

However, for universe $\mathcal{U}$ and $G, H \subseteq \mathcal{U}$:

$$G \times H = (G \times \mathcal{U}) \cap (\mathcal{U} \times H)$$

(let $\mathcal{U} = T_\Sigma$ ... all trees over Σ)



Checking language inclusion of tuples

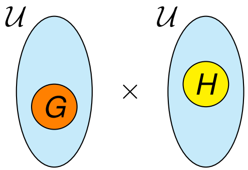
Note that in general

$$(\mathcal{L}(v_1)\times\mathcal{L}(v_2))\cup(\mathcal{L}(w_1)\times\mathcal{L}(w_2)) \neq (\mathcal{L}(v_1)\cup\mathcal{L}(w_1))\times(\mathcal{L}(v_2)\cup\mathcal{L}(w_2))$$

However, for universe $\mathcal{U}$ and $G, H \subseteq \mathcal{U}$:

$$G \times H = (G \times \mathcal{U}) \cap (\mathcal{U} \times H)$$

(let $\mathcal{U} = T_\Sigma$ . . . all trees over Σ)



$$\begin{array}{ccccccc} (\mathcal{L}(v_1) & \times & \mathcal{L}(v_2)) & \cup & (\mathcal{L}(w_1) & \times & \mathcal{L}(w_2)) = \\ ((\mathcal{L}(v_1)\times T_\Sigma) & \cap & (T_\Sigma\times\mathcal{L}(v_2))) & \cup & ((\mathcal{L}(w_1)\times T_\Sigma) & \cap & (T_\Sigma\times\mathcal{L}(w_2))) = \end{array}$$

Checking language inclusion of tuples

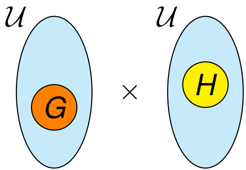
Note that in general

$$(\mathcal{L}(v_1)\times\mathcal{L}(v_2))\cup(\mathcal{L}(w_1)\times\mathcal{L}(w_2)) \neq (\mathcal{L}(v_1)\cup\mathcal{L}(w_1))\times(\mathcal{L}(v_2)\cup\mathcal{L}(w_2))$$

However, for universe $\mathcal{U}$ and $G, H \subseteq \mathcal{U}$:

$$G \times H = (G \times \mathcal{U}) \cap (\mathcal{U} \times H)$$

(let $\mathcal{U} = T_\Sigma$ ... all trees over Σ)



$$\begin{array}{ccccccc} (\mathcal{L}(v_1) & \times & \mathcal{L}(v_2)) & \cup & (\mathcal{L}(w_1) & \times & \mathcal{L}(w_2)) = \\ ((\mathcal{L}(v_1)\times T_\Sigma) & \cap & (T_\Sigma\times\mathcal{L}(v_2))) & \cup & ((\mathcal{L}(w_1)\times T_\Sigma) & \cap & (T_\Sigma\times\mathcal{L}(w_2))) = \end{array}$$

Using distributive laws, this becomes

$$\begin{array}{ccccccc} ((\mathcal{L}(v_1)\times T_\Sigma) & \cup & (\mathcal{L}(w_1)\times T_\Sigma)) & \cap & ((\mathcal{L}(v_1)\times T_\Sigma) & \cup & (T_\Sigma\times\mathcal{L}(w_2))) \cap \\ ((T_\Sigma\times\mathcal{L}(v_2)) & \cup & (\mathcal{L}(w_1)\times T_\Sigma)) & \cap & ((T_\Sigma\times\mathcal{L}(v_2)) & \cup & (T_\Sigma\times\mathcal{L}(w_2))) \end{array}$$

Checking language inclusion of tuples

$$\mathcal{L}(r) \times \mathcal{L}(s) \subseteq$$

$$((\mathcal{L}(v_1) \times T_\Sigma) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \cap ((\mathcal{L}(v_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(w_2))) \cap$$
$$((T_\Sigma \times \mathcal{L}(v_2)) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \cap ((T_\Sigma \times \mathcal{L}(v_2)) \cup (T_\Sigma \times \mathcal{L}(w_2)))$$

Checking language inclusion of tuples

$$\mathcal{L}(r) \times \mathcal{L}(s) \subseteq$$

$$((\mathcal{L}(v_1) \times T_\Sigma) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \cap ((\mathcal{L}(v_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(w_2))) \cap$$
$$((T_\Sigma \times \mathcal{L}(v_2)) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \cap ((T_\Sigma \times \mathcal{L}(v_2)) \cup (T_\Sigma \times \mathcal{L}(w_2)))$$

...is equal to checking

$$((\mathcal{L}(r) \times \mathcal{L}(s)) \subseteq (\mathcal{L}(v_1) \times T_\Sigma) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \wedge$$
$$((\mathcal{L}(r) \times \mathcal{L}(s)) \subseteq (\mathcal{L}(v_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(w_2))) \wedge \ldots$$

Checking language inclusion of tuples

$$\mathcal{L}(r) \times \mathcal{L}(s) \subseteq$$

$$((\mathcal{L}(v_1) \times T_\Sigma) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \cap ((\mathcal{L}(v_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(w_2))) \cap$$
$$((T_\Sigma \times \mathcal{L}(v_2)) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \cap ((T_\Sigma \times \mathcal{L}(v_2)) \cup (T_\Sigma \times \mathcal{L}(w_2)))$$

... is equal to checking

$$((\mathcal{L}(r) \times \mathcal{L}(s)) \subseteq (\mathcal{L}(v_1) \times T_\Sigma) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \wedge$$
$$((\mathcal{L}(r) \times \mathcal{L}(s)) \subseteq (\mathcal{L}(v_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(w_2))) \wedge \ldots$$

Each clause can be checked separately ...

Checking language inclusion of tuples

$$\mathcal{L}(r) \times \mathcal{L}(s) \subseteq$$

$$\begin{array}{ccccccc} ((\mathcal{L}(v_1)\times T_\Sigma) & \cup & (\mathcal{L}(w_1)\times T_\Sigma)) & \cap & ((\mathcal{L}(v_1)\times T_\Sigma) & \cup & (T_\Sigma\times\mathcal{L}(w_2))) \cap \\ ((T_\Sigma\times\mathcal{L}(v_2)) & \cup & (\mathcal{L}(w_1)\times T_\Sigma)) & \cap & ((T_\Sigma\times\mathcal{L}(v_2)) & \cup & (T_\Sigma\times\mathcal{L}(w_2))) \end{array}$$

…is equal to checking

$$\begin{array}{ccccc} ((\mathcal{L}(r)\times\mathcal{L}(s)) & \subseteq & (\mathcal{L}(v_1)\times T_\Sigma) & \cup & (\mathcal{L}(w_1)\times T_\Sigma)) \wedge \\ ((\mathcal{L}(r)\times\mathcal{L}(s)) & \subseteq & (\mathcal{L}(v_1)\times T_\Sigma) & \cup & (T_\Sigma\times\mathcal{L}(w_2))) \wedge \ldots \end{array}$$

Each clause can be checked separately …
…which is again checking inclusion of union of tuples, but now …

Checking language inclusion of tuples

$$\mathcal{L}(r) \times \mathcal{L}(s) \subseteq$$

$$\begin{array}{ccccccc} ((\mathcal{L}(v_1) \times T_\Sigma) & \cup & (\mathcal{L}(w_1) \times T_\Sigma)) & \cap & ((\mathcal{L}(v_1) \times T_\Sigma) & \cup & (T_\Sigma \times \mathcal{L}(w_2))) \cap \\ ((T_\Sigma \times \mathcal{L}(v_2)) & \cup & (\mathcal{L}(w_1) \times T_\Sigma)) & \cap & ((T_\Sigma \times \mathcal{L}(v_2)) & \cup & (T_\Sigma \times \mathcal{L}(w_2))) \end{array}$$

... is equal to checking

$$\begin{array}{ccccc} ((\mathcal{L}(r) \times \mathcal{L}(s)) & \subseteq & (\mathcal{L}(v_1) \times T_\Sigma) & \cup & (\mathcal{L}(w_1) \times T_\Sigma)) \land \\ ((\mathcal{L}(r) \times \mathcal{L}(s)) & \subseteq & (\mathcal{L}(v_1) \times T_\Sigma) & \cup & (T_\Sigma \times \mathcal{L}(w_2))) \land \ldots \end{array}$$

Each clause can be checked separately ...

... which is again checking inclusion of union of tuples, but now ...

... each tuple has a non-T_Σ language on a single position.

Checking language inclusion of tuples

$$\mathcal{L}(r) \times \mathcal{L}(s) \subseteq$$

$$((\mathcal{L}(v_1) \times T_\Sigma) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \cap ((\mathcal{L}(v_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(w_2))) \cap$$
$$((T_\Sigma \times \mathcal{L}(v_2)) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \cap ((T_\Sigma \times \mathcal{L}(v_2)) \cup (T_\Sigma \times \mathcal{L}(w_2)))$$

. . . is equal to checking

$$((\mathcal{L}(r) \times \mathcal{L}(s)) \subseteq (\mathcal{L}(v_1) \times T_\Sigma) \cup (\mathcal{L}(w_1) \times T_\Sigma)) \wedge$$
$$((\mathcal{L}(r) \times \mathcal{L}(s)) \subseteq (\mathcal{L}(v_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(w_2))) \wedge \ldots$$

Each clause can be checked separately . . .
. . . which is again checking inclusion of union of tuples, but now . . .
. . . each tuple has a non-T_Σ language on a single position.
⇒ **Checking language inclusion can be done component-wise.** ⇒

Checking language inclusion of tuples

$$\mathcal{L}(r) \times \mathcal{L}(s) \subseteq$$

$$\begin{array}{ccccccccc}((\mathcal{L}(v_1)\times T_\Sigma) & \cup & (\mathcal{L}(w_1)\times T_\Sigma)) & \cap & ((\mathcal{L}(v_1)\times T_\Sigma) & \cup & (T_\Sigma\times\mathcal{L}(w_2))) & \cap \\ ((T_\Sigma\times\mathcal{L}(v_2)) & \cup & (\mathcal{L}(w_1)\times T_\Sigma)) & \cap & ((T_\Sigma\times\mathcal{L}(v_2)) & \cup & (T_\Sigma\times\mathcal{L}(w_2)))\end{array}$$

. . . is equal to checking

$$\begin{array}{ccccc}((\mathcal{L}(r) \times \mathcal{L}(s)) & \subseteq & (\mathcal{L}(v_1)\times T_\Sigma) & \cup & (\mathcal{L}(w_1)\times T_\Sigma)) \wedge \\ ((\mathcal{L}(r) \times \mathcal{L}(s)) & \subseteq & (\mathcal{L}(v_1)\times T_\Sigma) & \cup & (T_\Sigma\times\mathcal{L}(w_2))) \wedge \ldots\end{array}$$

Each clause can be checked separately . . .
. . . which is again checking inclusion of union of tuples, but now . . .
. . . each tuple has a non-T_Σ language on a single position.
$\Rightarrow$ **Checking language inclusion can be done component-wise.** $\Rightarrow$

$$\iff \begin{array}{cccc}((\mathcal{L}(r) \subseteq \mathcal{L}(\{v_1, w_1\})) & \vee & (\mathcal{L}(s) \subseteq T_\Sigma)) & \wedge \\ ((\mathcal{L}(r) \subseteq \mathcal{L}(v_1)) & \vee & (\mathcal{L}(s) \subseteq \mathcal{L}(w_2))) & \wedge \ldots\end{array}$$

Basic Downward Inclusion Checking Algorithm

- DFS, maintain a workset W of product states (q_S, P_B).

Basic Downward Inclusion Checking Algorithm

- DFS, maintain a workset W of product states (q_S, P_B).
- Start the algorithm from (f, F_B) for each $f \in F_S$.

Basic Downward Inclusion Checking Algorithm

- DFS, maintain a workset W of product states (q_S, P_B).
- Start the algorithm from (f, F_B) for each $f \in F_S$.
- Alternating structure:

Basic Downward Inclusion Checking Algorithm

- DFS, maintain a workset W of product states (q_S, P_B).
- Start the algorithm from (f, F_B) for each $f \in F_S$.
- Alternating structure:
  - for all clauses ...

Basic Downward Inclusion Checking Algorithm

- DFS, maintain a workset W of product states (q_S, P_B).
- Start the algorithm from (f, F_B) for each $f \in F_S$.
- Alternating structure:
  - for all clauses ...
  - exists a position such that inclusion holds.

Basic Downward Inclusion Checking Algorithm

- DFS, maintain a workset W of product states (q_S, P_B).
- Start the algorithm from (f, F_B) for each $f \in F_S$.
- Alternating structure:
  - for all clauses …
  - exists a position such that inclusion holds.
- Sooner or later, the DFS either
  - reaches a leaf, or
  - reaches a pair (q_S, P_B) which is already in W.

Optimised Downward Inclusion Checking Algorithm

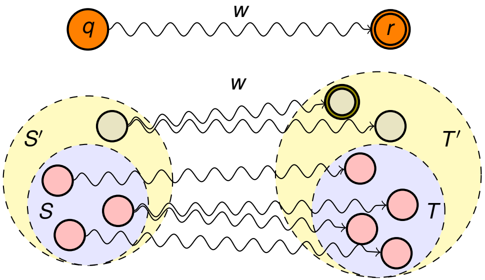
Optimisations:

1. It is possible to maintain a cache *NN* of pairs (q_S, P_B) for which $\mathcal{L}(q_S) \not\subseteq \mathcal{L}(P_B)$ has been shown and prune the search.

Optimised Downward Inclusion Checking Algorithm

Optimisations:

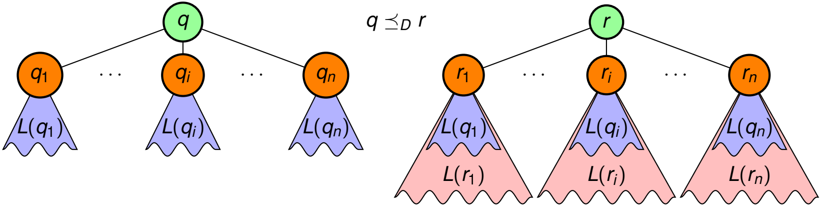
1. It is possible to maintain a cache *NN* of pairs (q_S, P_B) for which $\mathcal{L}(q_S) \not\subseteq \mathcal{L}(P_B)$ has been shown and prune the search.
2. Further, *NN* can be maintained as an antichain w.r.t. $\supseteq$
   - when $S \subseteq S'$, why store both (q, S) and (q, S')?
   - when $\mathcal{L}(q) \not\subseteq \mathcal{L}(S')$, then surely $\mathcal{L}(q) \not\subseteq \mathcal{L}(S)$.



Optimised Downward Inclusion Checking Algorithm

Optimisations:

1. It is possible to maintain a cache *NN* of pairs (q_S, P_B) for which $\mathcal{L}(q_S) \not\subseteq \mathcal{L}(P_B)$ has been shown and prune the search.
2. Further, *NN* can be maintained as an antichain w.r.t. $\supseteq$
   - when $S \subseteq S'$, why store both (q, S) and (q, S')?
   - when $\mathcal{L}(q) \not\subseteq \mathcal{L}(S')$, then surely $\mathcal{L}(q) \not\subseteq \mathcal{L}(S)$.
3. Moreover, *NN* can be maintained w.r.t. downward simulation $\preceq_D$.
   - $q \preceq_D r \implies \mathcal{L}(q) \subseteq \mathcal{L}(r)$



Optimised Downward Inclusion Checking Algorithm

Optimisations:

1. It is possible to maintain a cache *NN* of pairs (q_S, P_B) for which $\mathcal{L}(q_S) \not\subseteq \mathcal{L}(P_B)$ has been shown and prune the search.
2. Further, *NN* can be maintained as an antichain w.r.t. $\supseteq$
   - when $S \subseteq S'$, why store both (q, S) and (q, S')?
   - when $\mathcal{L}(q) \not\subseteq \mathcal{L}(S')$, then surely $\mathcal{L}(q) \not\subseteq \mathcal{L}(S)$.
3. Moreover, *NN* can be maintained w.r.t. downward simulation $\preceq_D$.
   - $q \preceq_D r \implies \mathcal{L}(q) \subseteq \mathcal{L}(r)$
4. Furthermore, workset can be also maintained w.r.t. $\preceq_D$.

Optimised Downward Inclusion Checking Algorithm

Optimisations:

1. It is possible to maintain a cache *NN* of pairs (q_S, P_B) for which $\mathcal{L}(q_S) \not\subseteq \mathcal{L}(P_B)$ has been shown and prune the search.
2. Further, *NN* can be maintained as an antichain w.r.t. $\supseteq$
   - when $S \subseteq S'$, why store both (q, S) and (q, S')?
   - when $\mathcal{L}(q) \not\subseteq \mathcal{L}(S')$, then surely $\mathcal{L}(q) \not\subseteq \mathcal{L}(S)$.
3. Moreover, *NN* can be maintained w.r.t. downward simulation $\preceq_D$.
   - $q \preceq_D r \implies \mathcal{L}(q) \subseteq \mathcal{L}(r)$
4. Furthermore, workset can be also maintained w.r.t. $\preceq_D$.
5. Even further, if $\exists s \in S : q \preceq_D s$, then surely $\mathcal{L}(q) \subseteq \mathcal{L}(S)$.

Experiments

Size	50–250	400–600
Pairs	323	64
Timeout	20 s	60 s
`Up`	31.21 %	9.38 %
`Up+s`	0.00 %	0.00 %
`Down`	53.50 %	39.06 %
`Down+s`	15.29 %	51.56 %
Avg up	1.71	0.34
Avg down	3.55	46.56

a)

Size	50–250	400–600
Pairs	323	64
Timeout	20 s	60 s
`Up+s`	81.82 %	20.31 %
`Down+s`	18.18 %	79.69 %
Avg up	1.33	9.92
Avg down	3.60	2116.29

b)

a) Comparison of methods (w/ simulation computation time).

b) Comparison of methods (w/o simulation computation time).

Semi-Symbolic TA

Several FV approaches yield automata with large alphabets:

- FV of programs with complex dynamic data structures,
- decision procedures of some logics: WSkS, MSO.

Semi-Symbolic TA

Several FV approaches yield automata with large alphabets:

- FV of programs with complex dynamic data structures,
- decision procedures of some logics: WSkS, MSO.

Current approach:

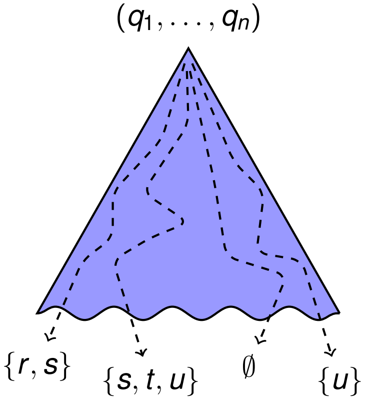
- use the MONA tree automata package (MTBDD-based)
- But only deterministic automata supported $\rightarrow$
  - often runs out of reasonable memory or time.

Dual representation

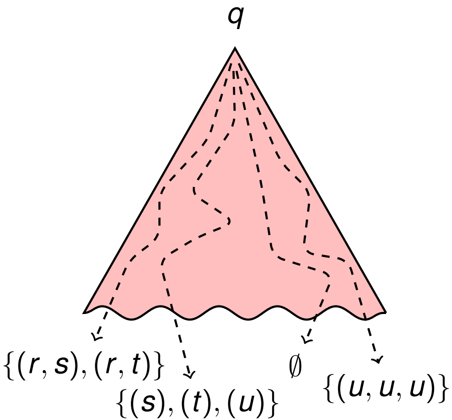
- Multi-terminal binary decision diagrams (MTBDDs)

Dual representation

- Multi-terminal binary decision diagrams (MTBDDs)
- Bottom-up:



- Top-down:



Bottom-up : inspired by MONA, but has sets of states in leaves.

Top-down : sets of state tuples in leaves.

Semi-Symbolic Encoding of Non-Deterministic TA

Algorithms for

- union,
- intersection,
- language inclusion checking (both upward and downward),
- downward simulation computation.
    - based on M. Henzinger, T. Henzinger, and P. Kopke's algorithm.

Semi-Symbolic Encoding of Non-Deterministic TA

Algorithms for

- union,
- intersection,
- language inclusion checking (both upward and downward),
- downward simulation computation.
  - based on M. Henzinger, T. Henzinger, and P. Kopke's algorithm.

Experiments:

Semi-Symbolic Encoding of Non-Deterministic TA

Algorithms for

- union,
- intersection,
- language inclusion checking (both upward and downward),
- downward simulation computation.
    - based on M. Henzinger, T. Henzinger, and P. Kopke's algorithm.

Experiments:

- Use of CUDD to implement MTBDDs.

Semi-Symbolic Encoding of Non-Deterministic TA

Algorithms for

- union,
- intersection,
- language inclusion checking (both upward and downward),
- downward simulation computation.
  - based on M. Henzinger, T. Henzinger, and P. Kopke's algorithm.

Experiments:

- Use of CUDD to implement MTBDDs.
- $\sim$ 8500 times faster downward inclusion checking than explicit representation for tested automata with large alphabets.

Conclusion

Conclusion

- An alternative downward approach to checking language inclusion of non-deterministic tree automata proposed, ...

Conclusion

- An alternative downward approach to checking language inclusion of non-deterministic tree automata proposed, . . .
- . . . that makes use of antichains and downward simulation.

Conclusion

- An alternative downward approach to checking language inclusion of non-deterministic tree automata proposed, ...
- ...that makes use of antichains and downward simulation.
- A new symbolic encoding of non-deterministic tree automata proposed.

Future work

- Optimise the downward inclusion to also cache pairs (q, S), such that $\mathcal{L}(q) \subseteq \mathcal{L}(S)$.

Future work

- Optimise the downward inclusion to also cache pairs (q, S), such that $\mathcal{L}(q) \subseteq \mathcal{L}(S)$.
- Replace CUDD with a more efficient MTBDD package.

Future work

- Optimise the downward inclusion to also cache pairs (q, S), such that $\mathcal{L}(q) \subseteq \mathcal{L}(S)$.
- Replace CUDD with a more efficient MTBDD package.
- Improve the symbolic downward simulation algorithm.

Future work

- Optimise the downward inclusion to also cache pairs (q, S), such that $\mathcal{L}(q) \subseteq \mathcal{L}(S)$.
- Replace CUDD with a more efficient MTBDD package.
- Improve the symbolic downward simulation algorithm.
- Create a tree automata package replacing MONA.

Thank you for your attention.

Questions?