Efficient Techniques for Manipulation of Non-deterministic 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

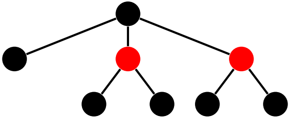
October 19, 2012

Outline

Trees

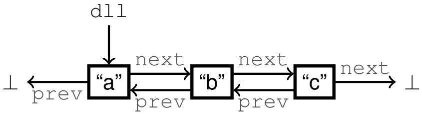
Very popular in computer science:

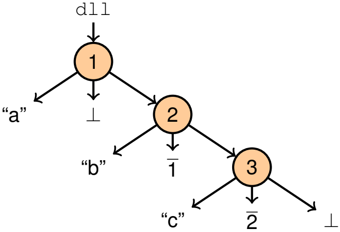
- data structures,
- computer network topologies,
- distributed protocols, . . .



In formal verification:

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





Tree Automata

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

- extension of finite automaton to trees:
  - Q ... finite 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 initial/final (root) states.
- two concepts: top-down vs. bottom-up

Tree Automata

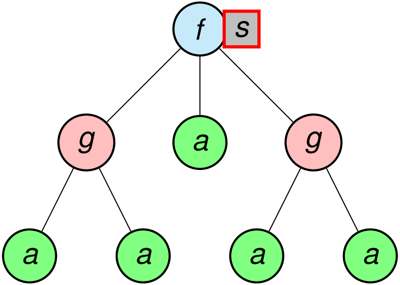
Finite Tree Automaton (TA): $\mathcal{A} = (Q, \Sigma, \Delta, F)$

- extension of finite automaton to trees:
  - Q ... finite 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 initial/final (root) states.

- two concepts: top-down vs. bottom-up

Example:

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



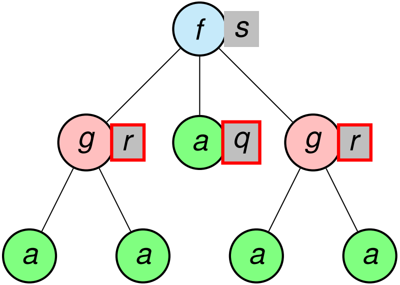
Tree Automata

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

- extension of finite automaton to trees:
  - Q ... finite 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 initial/final (root) states.
- two concepts: top-down vs. bottom-up

Example:

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



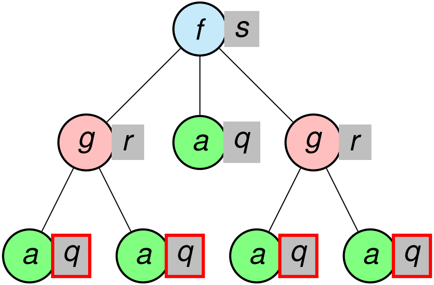
Tree Automata

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

- extension of finite automaton to trees:
  - Q ... finite 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 initial/final (root) states.
- two concepts: top-down vs. bottom-up

Example:

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



Tree Automata

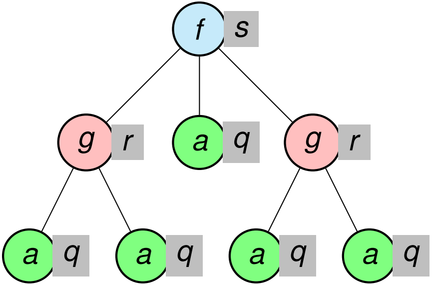
Finite Tree Automaton (TA): $\mathcal{A} = (Q, \Sigma, \Delta, F)$

- extension of finite automaton to trees:
  - Q ... finite 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 initial/final (root) states.

- two concepts: top-down vs. bottom-up

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 in FV:

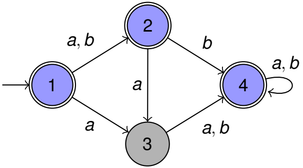
- often large due to determinisation
  - often advantageous to use non-deterministic tree automata,
  - manipulate them without determinisation,
  - even for operations such as language inclusion (ARTMC, . . .),
- handling large alphabets (MSO, WSkS).

- We focus on the problem of checking language inclusion.
- For simplicity, we demonstrate the ideas on:
  - finite automata,
  - and checking universality ($\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$).
- Their extension to tree automata is quite straightforward.

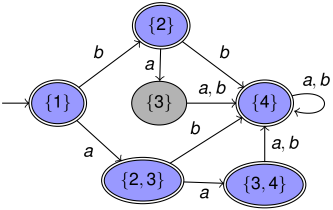
Finite Automata Universality Checking

- **PSPACE-complete**
- The **Textbook** algorithm for checking

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$$

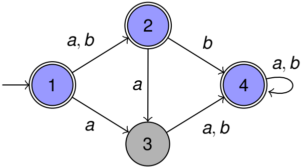


1. Determinise $\mathcal{A} \to \mathcal{A}^D$.
2. Complement $\mathcal{A}^D \to \overline{\mathcal{A}^D}$
   - by complementing the set of final states.
3. Check $\mathcal{L}(\overline{\mathcal{A}^D}) \stackrel{?}{=} \emptyset$,
   - search for a reachable final state.



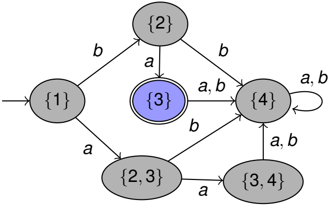
Finite Automata Universality Checking

- **PSPACE-complete**
- The **Textbook** algorithm for checking



$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$$

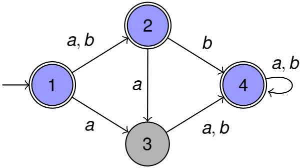
1. Determinise $\mathcal{A} \rightarrow \mathcal{A}^D$.
2. Complement $\mathcal{A}^D \rightarrow \overline{\mathcal{A}^D}$
   - by complementing the set of final states.
3. Check $\mathcal{L}(\overline{\mathcal{A}^D}) \stackrel{?}{=} \emptyset$,
   - search for a reachable final state.



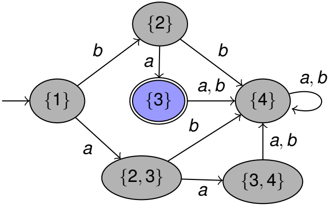
Finite Automata Universality Checking

- **PSPACE-complete**
- The **Textbook** algorithm for checking

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$$



1. Determinise $\mathcal{A} \rightarrow \mathcal{A}^D$.
   - **exponential explosion!**
2. Complement $\mathcal{A}^D \rightarrow \overline{\mathcal{A}^D}$
   - by complementing the set of final states.
3. Check $\mathcal{L}(\overline{\mathcal{A}^D}) \stackrel{?}{=} \emptyset$,
   - search for a reachable final state.



Finite Automata Universality Checking

- **PSPACE-complete**
- The **Textbook** algorithm for checking

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$$

1. Determinise $\mathcal{A} \to \mathcal{A}^D$.
   - **exponential explosion!**
2. Complement $\mathcal{A}^D \to \overline{\mathcal{A}^D}$
   - by complementing the set of final states.
3. Check $\mathcal{L}(\overline{\mathcal{A}^D}) \stackrel{?}{=} \emptyset$,
   - search for a reachable final state.

Inclusion checking

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{\supseteq} \mathcal{L}(\mathcal{B})$$

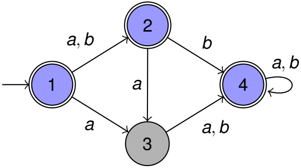
Inclusion checking

$$\mathcal{L}(\overline{\mathcal{A}^D}) \cap \mathcal{L}(\mathcal{B}) \stackrel{?}{=} \emptyset$$

Finite Automata Universality Checking

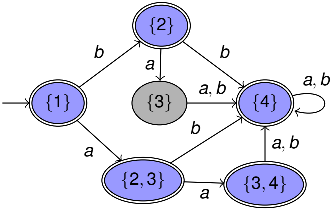
The **On-the-fly** algorithm for checking universality

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$$

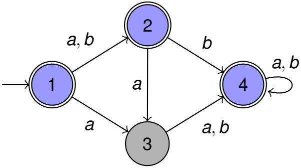


1. Traverse $\mathcal{A}$ from the initial states.
2. Perform on-the-fly determinisation, keep a *workset* of macrostates.
3. If encountered a macrostate P, such that $P \cap F = \emptyset$,
   - return **false**.
4. Otherwise, return **true**.

workset $= \{ \quad \}$



Finite Automata Universality Checking

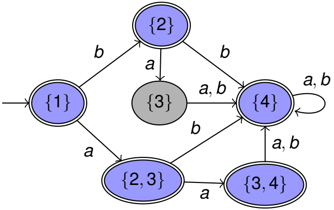


The **On-the-fly** algorithm for checking universality

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$$

1. Traverse $\mathcal{A}$ from the initial states.
2. Perform on-the-fly determinisation, keep a *workset* of macrostates.
3. If encountered a macrostate P, such that $P \cap F = \emptyset$,
   - return **false**.
4. Otherwise, return **true**.

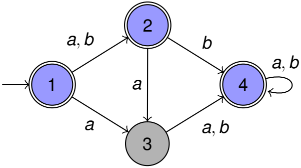
$$\textit{workset} = \{ \overbrace{\{\underline{1}\}}^{\textit{Init}} \}$$



Finite Automata Universality Checking

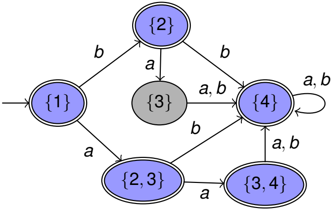
The **On-the-fly** algorithm for checking universality

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$$



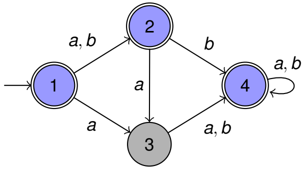
1. Traverse $\mathcal{A}$ from the initial states.
2. Perform on-the-fly determinisation, keep a *workset* of macrostates.
3. If encountered a macrostate P, such that $P \cap F = \emptyset$,
   - return **`false`**.
4. Otherwise, return **`true`**.

$$workset = \{ \overbrace{\{\underline{1}\}}^{Init}, \underbrace{\{\underline{2},3\}}_{\{1\}\xrightarrow{a}}, \overbrace{\{\underline{2}\}}^{\{1\}\xrightarrow{b}} \}$$



Finite Automata Universality Checking

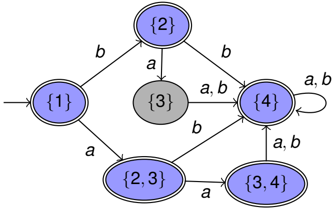
The **On-the-fly** algorithm for checking universality



$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$$

1. Traverse $\mathcal{A}$ from the initial states.
2. Perform on-the-fly determinisation, keep a *workset* of macrostates.
3. If encountered a macrostate P, such that $P \cap F = \emptyset$,
   - return **`false`**.
4. Otherwise, return **`true`**.

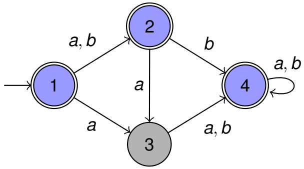
$$\mathit{workset} = \{ \overbrace{\{\underline{1}\}}^{\mathit{Init}}, \underbrace{\{\underline{2},3\}}_{\{1\}\xrightarrow{a}}, \overbrace{\{\underline{2}\}}^{\{1\}\xrightarrow{b}}, \underbrace{\{3,\underline{4}\}}_{\{2,3\}\xrightarrow{a}}, \overbrace{\{\underline{4}\}}^{\{2,3\}\xrightarrow{b}} \}$$



Finite Automata Universality Checking

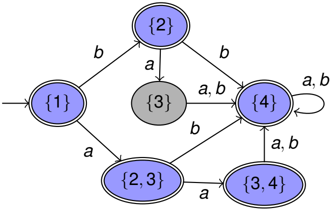
The **On-the-fly** algorithm for checking universality

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \Sigma^*$$



1. Traverse $\mathcal{A}$ from the initial states.
2. Perform on-the-fly determinisation, keep a *workset* of macrostates.
3. If encountered a macrostate P, such that $P \cap F = \emptyset$,
   - return `false`.
4. Otherwise, return `true`.

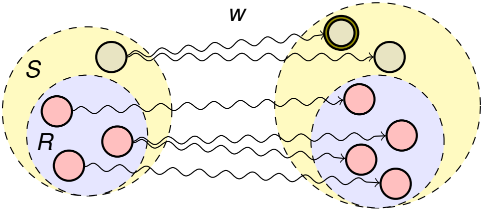
$$\mathit{workset} = \{\overbrace{\{\underline{1}\}}^{\mathit{Init}}, \underbrace{\{\underline{2}, 3\}}_{\{1\}\xrightarrow{a}}, \overbrace{\{\underline{2}\}}^{\{1\}\xrightarrow{b}}, \underbrace{\{3, \underline{4}\}}_{\{2,3\}\xrightarrow{a}}, \overbrace{\{\underline{4}\}}^{\{2,3\}\xrightarrow{b}}, \underbrace{\{3\}}_{\{2\}\xrightarrow{a}}\}$$



Finite Automata Universality Checking

Optimisations:

- The **Antichains** algorithm [De Wulf, Doyen, Henzinger, Raskin. CAV'06],
- keep only macrostates sufficient to encounter a non-final set:
  - if macrostates R and S, $R \subseteq S$, are both in *workset*,
    - remove S from *workset*.



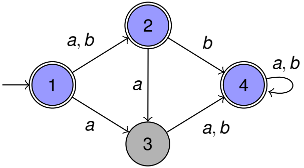
R has a bigger chance to encounter a non-final macrostate

Finite Automata Universality Checking

Optimisations:

- The **Antichains** algorithm [De Wulf, Doyen, Henzinger, Raskin. CAV'06],
- keep only macrostates sufficient to encounter a non-final set:
  - if macrostates R and S, $R \subseteq S$, are both in *workset*,
    - remove S from *workset*.

workset $= \{ \quad \}$

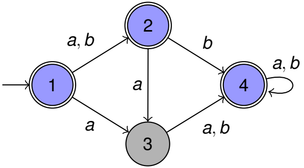


Finite Automata Universality Checking

Optimisations:

- The **Antichains** algorithm [De Wulf, Doyen, Henzinger, Raskin. CAV'06],
- keep only macrostates sufficient to encounter a non-final set:
  - if macrostates R and S, $R \subseteq S$, are both in *workset*,
    - remove S from *workset*.

$$\mathit{workset} = \{ \overbrace{\{\underline{1}\}}^{\mathit{Init}} \qquad\qquad \}$$

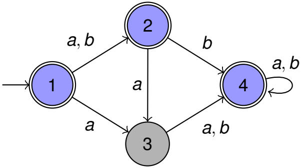


Finite Automata Universality Checking

Optimisations:

- The **Antichains** algorithm [De Wulf, Doyen, Henzinger, Raskin. CAV'06],
- keep only macrostates sufficient to encounter a non-final set:
  - if macrostates R and S, $R \subseteq S$, are both in *workset*,
    - remove S from *workset*.

$$\mathit{workset} = \{ \overbrace{\{\underline{1}\}}^{\mathit{Init}}, \underbrace{\{\underline{2}, 3\}}_{\{1\}\xrightarrow{a}}, \overbrace{\{\underline{2}\}}^{\{1\}\xrightarrow{b}} \qquad \}$$

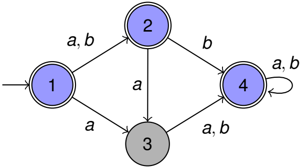


Finite Automata Universality Checking

Optimisations:

- The **Antichains** algorithm [De Wulf, Doyen, Henzinger, Raskin. CAV’06],
- keep only macrostates sufficient to encounter a non-final set:
  - if macrostates R and S, $R \subseteq S$, are both in *workset*,
    - remove S from *workset*.

$$\textit{workset} = \{ \overbrace{\{\underline{1}\}}^{\textit{Init}} \quad , \overbrace{\{\underline{2}\}}^{\{1\}\xrightarrow{b}} \qquad \}$$

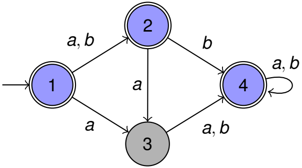


Finite Automata Universality Checking

Optimisations:

- The **Antichains** algorithm [De Wulf, Doyen, Henzinger, Raskin. CAV'06],
- keep only macrostates sufficient to encounter a non-final set:
  - if macrostates R and S, $R \subseteq S$, are both in *workset*,
    - remove S from *workset*.

$$\mathit{workset} = \{\overbrace{\{\underline{1}\}}^{\mathit{Init}}\}\quad ,\overbrace{\{\underline{2}\}}^{\{1\}\xrightarrow{b}}, \underbrace{\{3\}}_{\{2\}\xrightarrow{a}}, \overbrace{\{\underline{4}\}}^{\{2\}\xrightarrow{b}}\}$$



Optimisations:

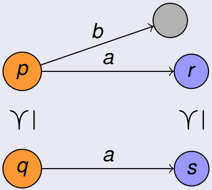
- The **Antichains + Simulation** algorithm [Abdulla, *et al.* TACAS'10],

Simulation

A preorder $\preceq$ such that

$$q \preceq p \implies$$

$$\left(\forall a \in \Sigma \,.\, q \xrightarrow{a} s \implies \exists r . p \xrightarrow{a} r \wedge s \preceq r\right)$$



Note that $q \preceq p \implies \mathcal{L}(q) \subseteq \mathcal{L}(p)$!

Finite Automata Universality Checking

Optimisations:

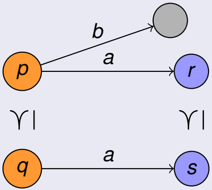
- The **Antichains + Simulation** algorithm [Abdulla, *et al.* TACAS'10],

Simulation

A preorder $\preceq$ such that

$$q \preceq p \implies$$

$$\left(\forall a \in \Sigma \,.\, q \xrightarrow{a} s \implies \exists r . p \xrightarrow{a} r \wedge s \preceq r\right)$$



Note that $q \preceq p \implies \mathcal{L}(q) \subseteq \mathcal{L}(p)$!

- refine *workset* using simulation
  - if macrostates R and S are both in *workset*, $\overbrace{R \preceq^{\forall\exists} S}^{\forall r \in R\, \exists s \in S \,.\, r \preceq s}$
    - remove S from *workset* (because $\mathcal{L}(R) \subseteq \mathcal{L}(S)$),
  - further, minimise macrostates w.r.t. $\preceq$: $\{p, q, x\} \Rightarrow \{p, \quad x\}$

Tree Automata Universality Checking

- **EXPTIME-complete**
- Checking whether $\mathcal{L}(\mathcal{A}) \stackrel{?}{=} T_\Sigma$.
- The (upward) **Textbook**, **On-the-fly**, and **Antichains** algorithms:
  - straightforward extension of the algorithms for FA,
  - perform upward (i.e. bottom-up) determinisation of the TA,
  - need to find tuples of macrostates to perform an upward transition.
- The (upward) **Antichains + Simulation** algorithm:
  - needs to use upward simulation (implies inclusion of "*open trees*")
    - usually not very rich.

TA Downward Universality Checking

- TA Downward Universality Checking: [Holík, *et al.* ATVA'11]

- inspired by XML Schema containment checking:
  - [Hosoya, Vouillon, Pierce. ACM Trans. Program. Lang. Sys., 2005],

- does not follow the classic schema of universality algorithms:
  - can't determinise: top-down DTA are strictly less powerful than TA,
  - however, there exists a complementation procedure.

TA Downward Universality Checking

$\mathcal{A}$

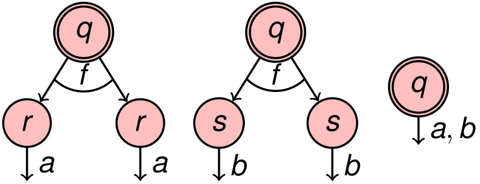
$\underline{q} \xrightarrow{f} (r, r)$ $\quad r \xrightarrow{a}$

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

$\underline{q} \xrightarrow{a}$

$\underline{q} \xrightarrow{b}$

$\Sigma = \{f_2, a_0, b_0\}$



$\mathcal{L}(q) = T_\Sigma$ if and only if

$$(\mathcal{L}(r) \times \mathcal{L}(r)) \cup (\mathcal{L}(s) \times \mathcal{L}(s)) = T_\Sigma \times T_\Sigma$$

(universality of tuples!)

TA Downward Universality Checking

Note that in general

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

TA Downward Universality Checking

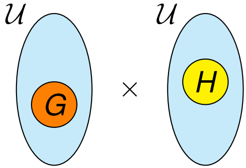
Note that in general

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

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)



TA Downward Universality Checking

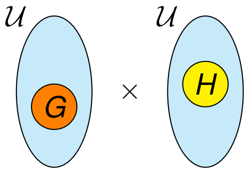
Note that in general

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

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)



$$(\mathcal{L}(v_1) \quad \times \quad \mathcal{L}(v_2)) \qquad \cup \qquad (\mathcal{L}(w_1) \quad \times \quad \mathcal{L}(w_2)) =$$

TA Downward Universality Checking

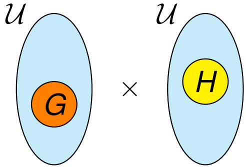
Note that in general

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

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)



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

TA Downward Universality Checking

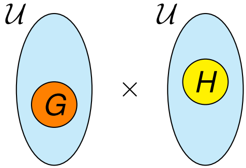
Note that in general

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

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

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

TA Downward Universality Checking

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

TA Downward Universality Checking

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

…is equal to …

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

TA Downward Universality Checking

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

TA Downward Universality Checking

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

Each clause can be checked separately . . .

TA Downward Universality Checking

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

Each clause can be checked separately . . .

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

TA Downward Universality Checking

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

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.

TA Downward Universality Checking

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

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 can be done component-wise.** ⇒

TA Downward Universality Checking

$$
\begin{aligned}
(\mathcal{L}(v_1)\times T_\Sigma) \quad &\cup \quad (\mathcal{L}(w_1)\times T_\Sigma)) = T_\Sigma \times T_\Sigma \quad \wedge \\
(\mathcal{L}(v_1)\times T_\Sigma) \quad &\cup \quad (T_\Sigma\times\mathcal{L}(w_2))) = T_\Sigma \times T_\Sigma \quad \wedge \\
(T_\Sigma\times\mathcal{L}(v_2)) \quad &\cup \quad (\mathcal{L}(w_1)\times T_\Sigma)) = T_\Sigma \times T_\Sigma \quad \wedge \\
(T_\Sigma\times\mathcal{L}(v_2)) \quad &\cup \quad (T_\Sigma\times\mathcal{L}(w_2))) = T_\Sigma \times T_\Sigma
\end{aligned}
$$

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.** $\Rightarrow$

$$
\iff \quad
\begin{aligned}
&(\mathcal{L}(\{v_1, w_1\}) = T_\Sigma) && && \wedge \\
&((\mathcal{L}(\{v_1\}) = T_\Sigma) && \vee \quad (\mathcal{L}(\{w_2\}) = T_\Sigma)) && \wedge \\
&((\mathcal{L}(\{w_1\}) = T_\Sigma) && \vee \quad (\mathcal{L}(\{v_2\}) = T_\Sigma)) && \wedge \\
& && \quad\; (\mathcal{L}(\underbrace{\{v_2, w_2\}}_{\text{macrostate}}) = T_\Sigma) &&
\end{aligned}
$$

Basic Downward Universality Algorithm

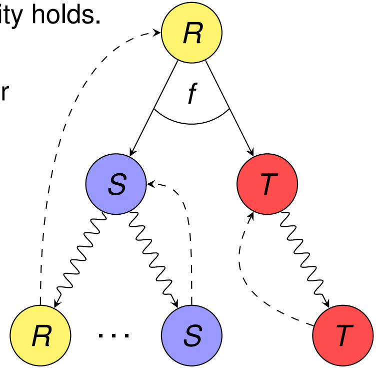
The **On-the-fly** algorithm:

- DFS, maintain *workset* of macrostates.
- Start the algorithm from macrostate F,
- Alternating structure:
  - for all clauses . . .
  - exists a position such that universality holds.

Basic Downward Universality Algorithm

The **On-the-fly** algorithm:

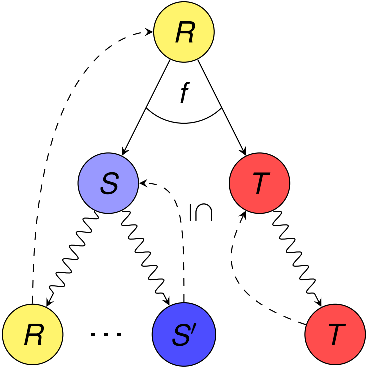
- DFS, maintain *workset* of macrostates.
- Start the algorithm from macrostate F,
- Alternating structure:
  - for all clauses ...
  - exists a position such that universality holds.
- Cut the DFS when
  - there is no transition for a symbol, or
  - macrostate is already in *workset*.



Optimisations of Downward TA Universality Algorithm

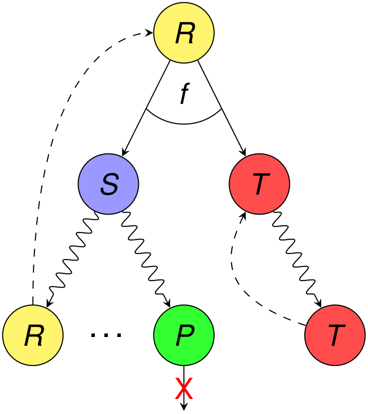
Optimisations: **Antichains**

1. Cut the DFS on macrostate S' when
   - a smaller macrostate S, $S \subseteq S'$, is already in *workset*,
     - if S is universal, S' will also be universal.



Optimisations: **Antichains**

2. If a macrostate P is found to be non-universal, cache it;
   - do not expand any new macrostate $P' \subseteq P$,
     - surely $\mathcal{L}(P') \neq T_\Sigma$.



Optimisations of Downward TA Universality Algorithm

Optimisations: **Antichains**

3. We wish to perform a similar optimisation as in **2.**

Optimisations of Downward TA Universality Algorithm

Optimisations: **Antichains**

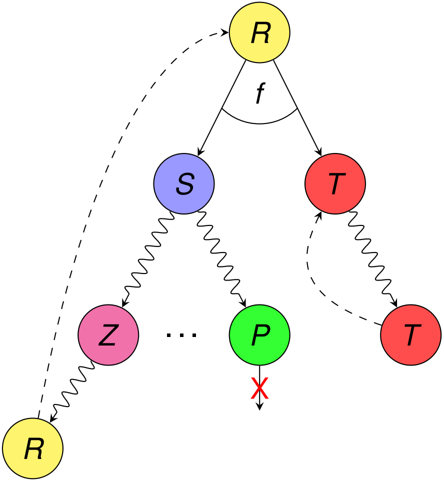
3. We wish to perform a similar optimisation as in **2.**
   However, we cannot simply cache macrostate R through which we return in the DFS!

- Why?

Optimisations: **Antichains**

3. We wish to perform a similar optimisation as in **2.**
   However, we cannot simply cache macrostate *R* through which we return in the DFS!

- Why?
- Universality of *R* might be falsified on other branches of the DFS!



Optimisations: **Antichains**

3. Solution: cache the set Z for which the universality condition holds, BUT together with the precondition why it holds:

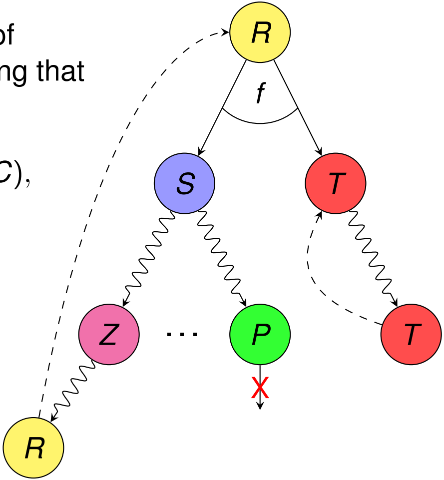
Optimisations of Downward TA Universality Algorithm

Optimisations: **Antichains**

3. Solution: cache the set Z for which the universality condition holds, BUT together with the precondition why it holds:
   - i.e. we maintain a pair of sets of macrostates (Ant, Con) meaning that $Ant \implies Con$, i.e.

$$\bigwedge_{A \in Ant} U(A) \implies \bigwedge_{C \in Con} U(C),$$

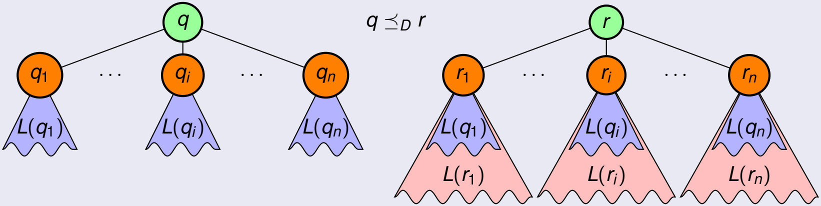
   - when the DFS is returning via G, it removes G from Ant and adds G to Con,
   - when Ant becomes empty, all sets S from Con are cached.
   - If found X, $G \subseteq X$, return.



Optimisations: **Antichains + Simulation**

- Downward simulation
  - implies inclusion of (downward) tree languages of states,
  - usually quite rich.

Downward simulation $\preceq_D$



- In **Antichains**, instead of $\subseteq$ use $\preceq_D^{\forall\exists}$.
- further, minimise macrostates w.r.t. $\preceq_D$: $\{p, q, x\} \Rightarrow \{p, \quad x\}$

Experiments

- Comparison of different inclusion checking algorithms
  - `down` — downward, `up` — upward,
  - `+s` — using upward/downward simulation,
  - `-o` — with optimisation 3 (*Ant*, *Con*).

	`down`	`down+s`	`down-o`	`down-o+s`	`up`	`up+s`
Winner	36.35 %	4.15 %	32.20 %	3.15 %	24.14 %	0.00 %
Timeouts	32.51 %	18.27 %	32.51 %	18.27 %	0.00 %	0.00 %

VATA: A Tree Automata Library

VATA is a new tree automata library that

- supports non-deterministic tree automata,
- provides encodings suitable for different contexts:
  - explicit, and
  - semi-symbolic,
- is written in C++,
- is open source and free under GNU GPLv3,
  - `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`
  - or (shorter), `http://goo.gl/KNpMH`

Supported Operations

Supported operations:

- union,
- intersection,
- removing unreachable or useless states and transitions,
- testing language emptiness,

- computing downward and upward simulation,
- simulation-based reduction,
- testing language inclusion,

- import from file/export to file.

Simulations

Explicit:

- downward simulation $\preceq_D$,
- upward simulation $\preceq_U$.

Work by transforming automaton to labelled transition systems,

- computing simulation on the LTS, [Holík, Šimáček. MEMICS'09],
- which is an improvement of [Ranzato, Tapparo. LICS'07].

Semi-symbolic:

- downward simulation computation based on [Henzinger, Henzinger, Kopke. FOCS'95].

Reduction according to downward simulation.

Conclusion

- A new tree automata library available
  - containing various optimisations of the used algorithms,
  - particularly highly efficient inclusion checking algorithms.
- Support for working with non-deterministic automata.
- Easy to extend with own encoding/operations.
- The library is open source and free under GNU GPLv3.
- Available at

  `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`

Future work

- Add new representations of finite word/tree automata,
  - that address particular issues, such as
    - large number of states, or
    - fast checking of language inclusion.
- Add missing operations,
  - development is demand-driven,
  - if you miss something, write to us, the feature may appear soon.

Thank you for your attention.

Questions?